Evolution of the UML Interactions Metamodel

Marc-Florian Wendland[1], Martin Schneider[1], and Øystein Haugen[2]

[1]Fraunhofer Institut FOKUS
Kaiserin-Augusta-Allee 31, 10589 Berlin, Germany
[2]SINTEF, Norway

{marc-florian.wendland,martin.schneider}@fokus.fraunhofer.de,
Oystein.haugen@sintef.no

Abstract. UML Interactions represent one of the three different behavior kinds of the UML. In general, they specify the exchange of messages among parts of a system. Although UML Interactions can reside on different level of abstractions, they seem to be sufficiently elaborated for a higher-level of abstraction where they are used for sketching the communication among parts. Its metamodel reveals some fuzziness and imprecision where definitions should be accurate and concise, though.
In this paper, we propose improvements to the UML Interactions' metamodel for Message arguments and Loop CombinedFragments that make them more versatile. We will justify the needs for the improvements by precisely showing the shortcomings of the related parts of the metamodel. We demonstrate the expressiveness of the improvements by applying them to examples that current Interactions definition handles awkwardly.

Keywords: UML, Interactions, Sequence Diagram, Messages, CombinedFragments

1 Introduction

UML Interactions are one of the three behavior kinds of UML 2 [1] and describe information exchange among parts of a system via messages. Graphically, UML Interactions are most commonly depicted as sequence diagrams.

UML 1 Interactions originated from a proprietary dialect of sequence charts which came from Siemens. When UML 2 was initiated in 1999 some of the driving forces from the telecom industry had already applied sequence diagrams for many years and were well acquainted with Message Sequence Charts (MSC) [2]. Ericsson, Motorola and Alcatel, supplemented also by tool vendor Telelogic, collaborated to formalize UML in the direction of MSC and SDL (Specification and Description Language, recommended in Z.100 by ITU). This resulted in trying to harmonize the MSC-2000 with UML 2 and still keep most of what had been in UML 1 sequence diagrams as well. While MSC was defined as a stand-alone language, Interactions of UML 2 should be well harmonized and integrated with the rest of UML. However, the telecom companies were not satisfied with informal relations between elements, but wanted a UML language that was as precise as what they were used to from SDL and MSC. Other stakeholders of UML were not convinced that UML should be that precise. A lot of compromises were made, though. The concept of *semantic variation points* was introduced and still remains central to the definition of UML. The overall metamodel, however, was supposed to tie the different parts of UML together and in some respects it did that, but in other respects the unification of different concepts was not done with rigor and the language became unnecessary complicated.

Since their advent sequence diagrams were used a lot, however, their use was mostly of descriptive nature. The communication between system parts was sketched rather than precisely defined. When the UML Testing Profile (UTP) ([3] and [4]) appeared, there was emphasis on being able to use sequence diagrams for defining test specifications. Even the data of the messages had to be defined more accurately. In Interactions, exchange of data is expressed as arguments of a message related to a certain element of the message's signature. Due to the compromises made in UML, several issues appear when message arguments need to be precisely specified.

This paper summarizes the most relevant issues for message arguments, explains how they manifest in the metamodel and suggests improvements to the relevant parts of the metamodel to overcome those issues. This paper does not question the general architecture of UML or the rigor of the integration of its parts (such as Activities and Interactions), but rather treat Interactions as a self-sufficient concept space with respect to its features for describing precise message exchange. The motivation for this work stems from the development of an UTP-based tool for model-based testing, called Fokus!MBT [20], and from the application of Interactions for test case specification in industrial and research projects. Thus, the presented work is not a mere theoretical consideration, but has been used for and proven its applicability to real use cases.

As typographical convention, all metaclasses of the UML metamodel are written in camel-case and start with a capital letter. Association ends and properties of metaclasses are written in camel-case, start with a lower case letter and are set to italic. For the sake of comprehensibility, the presented figures do not mention every aspect of the UML abstract syntax (e.g., names of non-navigable association ends are omitted). Introduced concepts are set italic the first time they are mentioned. In case the index of an ordered association ends is relevant for understanding, it is surrounded by square brackets (e.g., [1] indicating the first object). This notation is not standardized for UML object diagrams.

The remainder of this paper is structured as follows: Section 2 summarizes previous work in the area of Interactions. Section 3 presents the relevant parts of the metamodel regarding abstract syntax and semantics. Section 4 represents the main part of our contribution and describes metamodel improvement suggestions for Messages and CombinedFragments. Section 5 proposes two recommendations for the development of metamodels derived from the improvement suggestions presented in section 4. Finally, section 6 summarizes our work and provides an outlook on future considerations of the Interactions metamodel.

2 Related Work

Haugen compares UML Interactions and Message Sequence Charts [5] showing that Interactions and MSCs are similar down to small details.

Haugen, Stolen, Husa, and Runde have written a series of paper on the compositional development of UML Interactions supporting the specification of mandatory and potential behavior, called STAIRS approach ([6], [7], [8], and [9]). Although the compositional idea is reflected throughout the series, a special interest is dedicated to a fine-grained differentiation of event reception, consumption and timing [7] and the refinement of Interactions with regard to underspecification and nondeterminism [9]. Lund and Stolen have presented an operational semantics for UML sequence diagrams in the context of STAIRS [10].

Formal semantics of UML Interactions and sequence diagrams were several times discussed. Störrle presented a formal specification of UML Interactions and a comparison of UML 2.0 and UML 1.4 Interactions [11] and [12]). A similar work was done by Knapp and Cengarle ([13] and [14]), Li and Ruan [15] and Shen et al. [16]. Special attention was set to the semantics of assert and negative CombinedFragments ([17] and [18]), though.

An approach to model checking based upon a formal trace semantics of Interactions was described by Knapp and Wuttke [19].

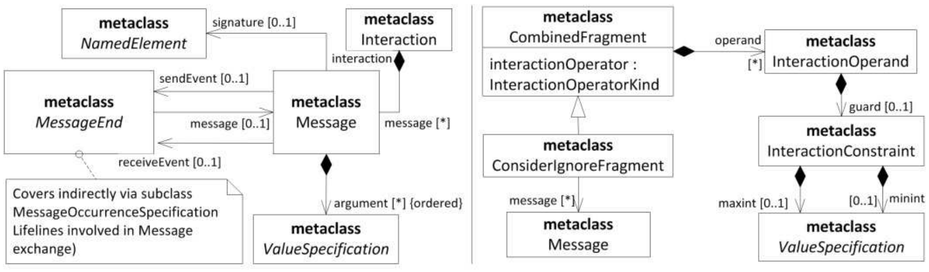
Our paper is different from the work described above. These publications were mostly dedicated to the trace semantics of Message reception and consumption within UML Interactions, but they did not focus on precisely specifying data transmitted by Messages. Furthermore, the complete metamodel of UML Interactions has not been considered and improved. Our work addresses the precise specification of Message arguments as well as revised parts of the UML Interactions metamodel to make them more robust and manageable by subsequent tooling.

3 Relevant Parts of the UML Interactions Metamodel

This chapter briefly summarizes those parts of the UML Interactions metamodel that are relevant for understanding the focal point of this paper. A full description of the semantics can be found in the current UML specification [1] our work is based on. For the sake of comprehensibility, the necessary parts of the metamodel are shown in Fig. 1. nevertheless. The left-hand side shows the relevant parts of Messages, the right hand side those of CombinedFragments.

Interactions describe the communication between (potentially loosely coupled) parts of a system. The most important building blocks of Interactions are Messages that constitute information exchange between different parts, and Lifelines that represent those communicating parts.

A *Message* represents either the invocation of an Operation or the sending and reception of a Signal. The first kind represents either an *asynchronous* or *synchronous* call, or a *reply* in case of a preceding synchronous call. The second kind (i.e., the sending of a Signal) is by definition always asynchronous. Messages commonly convey data in terms of its *actual arguments* to the receiver. The actual arguments of Message have to correspond to the elements determined by its *signature*. These *signature elements* can manifest as Parameters, in case of an Operation signature, or Properties, in case of a Signal signature. Consistency between actual argument and signature element requires that the actual argument (identified by its index in *Message.argument*) is type compliant with the corresponding signature element (identified by the very same index as the actual arguments, either in *Operation.ownedParameter* or *Signal.ownedAttribute*). The consistency definition implies that both lists must be of equal size.



metaclass NamedElement
signature [0..1]
metaclass Interaction
interaction
sendEvent [0..1]
metaclass MessageEnd
metaclass Message
message [0..1]
message [*]
receiveEvent [0..1]
Covers indirectly via subclass MessageOccurrenceSpecification Lifelines involved in Message exchange)
argument [*] {ordered}
metaclass ValueSpecification
metaclass CombinedFragment
interactionOperator : InteractionOperatorKind
operand
[*]
metaclass InteractionOperand
guard [0..1]
metaclass InteractionConstraint
metaclass ConsiderIgnoreFragment
message [*]
metaclass Message
maxint [0..1]
[0..1]
minint
metaclass ValueSpecification

Fig. 1. Relevant parts of the UML Interactions metamodel regarding Messages (left) and CombinedFragments (right)

CombinedFragments were introduced in UML 2 to enable more expressive Interactions. The semantics of a CombinedFragment is determined by its InteractionOperatorKind that also implies the number of InteractionOperands a CombinedFragment may possess. Each InteractionOperand may be guarded by an InteractionConstraint that defines what that must hold to activate the InteractionOperand. Some kinds of CombinedFragments are supplemented with additional information required in their semantic context. These are Loop-kind CombinedFragments (henceforth called Loops) and *ConsiderIgnoreFragments*. Loops represent repetitions of the events enclosed in its InteractionOperand. The number of repetitions can be omitted (any number of repetitions is valid), restricted to a single number of repetitions or specified as an interval for a minimally and maximally intended repetition.

4 Improving Messages and CombinedFragments

The following sections represent the main contribution of our work, i.e., improvement suggestions for the UML Interactions metamodel regarding a precise specification of Message arguments and CombinedFragments. UML is a language of compromises so there are most likely several opinions why the issues[1], being described subsequently, actually appear and how they ought to be resolved in the first place. Our improvements are strictly defined from an Interactions point of view. All suggested modifica-

[1] The issues we will discuss and mitigate are already filed in the OMG issue database (see http://www.omg.org/issues/uml2-rtf.open.html): #8786, #8899, #16569 and #16571.

tions are local to the Interactions metamodel to make them more robust and as expressive regarding the specification of arguments as Activities, for example. Resolving more fundamental and maybe philosophic or politic issues in the essence of UML is out of scope of this paper, though.

4.1 Precise and Robust Specification of Message Arguments

A Message's actual arguments and the signature elements they need to correspond to are implicitly related via their indices in two distinct lists. This is not problematic as long as the signature elements have just a single, non-optional multiplicity (i.e., lower and upper bounds equals 1) or only the last signature element is optional. In any other case, specifying actual arguments may lead to ambiguities due to UML's inability to model standalone collections of ValueSpecifications and the implicit relation of members of two independent lists based on the respective indices. A discussion whether ValueSpecification collections should be made available in UML is not in the scope of this paper.

For better illustration, we consider an Operation with a single Integer collection Parameter of an unbound size. Fig. 2 illustrates the corresponding object model for a scenario where a user specifies an actual argument list with the values (*1, 2, 3*).

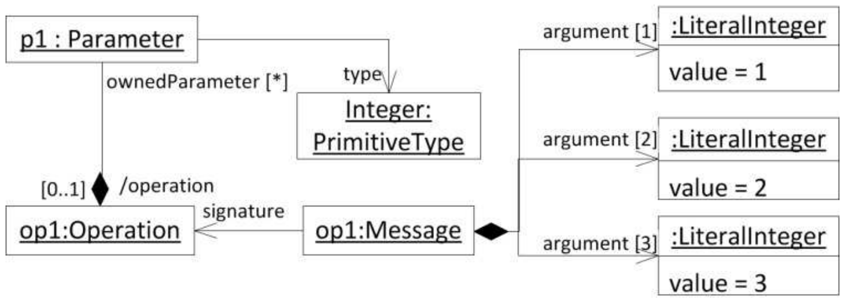


Fig. 2. Object model of ill-formed Message

The Message *op1* contains three actual arguments what would imply that its signature offers three signature elements as well. In fact, it just offers one (see Parameter *p1*), so referring to UML [1] the model presented above is invalid by definition. Activities, for example, can handle collections of actual arguments for a single signature element with the Pin metaclass and we believe Interactions should also provide a *native* concept to be able to handle actual arguments for collections. We emphasize the term *native*, because there are some metamodeling workarounds that misuse metaclasses to ensure syntactical correctness. The issue depicted in Fig. 2 might be solved by misusing the metaclass Expression as pseudo-collection of ValueSpecifications. As long as the metamodel of UML will not be enhanced with dedicated concepts for ValueSpecification collections, Expressions are actually the most elegant (but semantically disputable) way to specify them. Nevertheless, this is kind of a metamodeling trick, since Expressions are intended to specify expression trees in a sense of an Abstract Syntax Tree (AST). As an improvement, we suggest introducing a dedicated concept with clear semantics and syntax for the purpose of precise specification of a Message's actual arguments, called *MessageArgumentSpecification* (see Fig. 3).

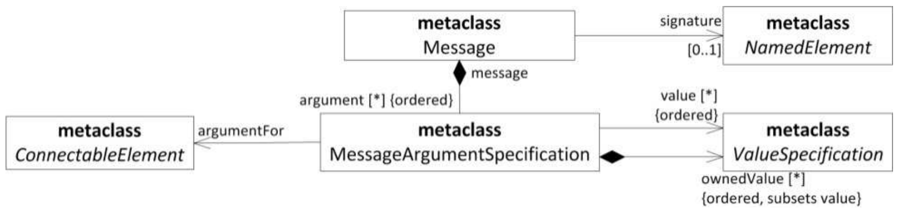


Fig. 3. Explicit relation between a Message's signature element and actual arguments

A MessageArgumentSpecification makes the correspondence of a set of actual arguments to its respective signature element explicit through the association end *argumentFor* that points to the related signature element (ConnectableElement represents the closest common metaclass of both possible signature elements Parameter and Property). The corresponding constraint expressed with the Object Constraint Language (OCL) for restricting what ConnectableElements can be addressed as signature element, is:

```
context MessageArgumentSpecification
inv: not self.message.oclIsUndefined() implies
if self.message.signature.oclIsTypeOf(Operation) then
self.message.signature.oclAsType(Operation).ownedParamete
r->exists(self.argumentFor)
else if self.message.signature.oclIsTypeOf(Signal) then
self.message.signature.oclAsType(Signal).attributes-
>exists(self.argumentFor)
else
false
end if
end if
```

Literally, the ConnectableElement referenced by MessageArgumentSpecification must either be a Parameter of an Operation or a Property owned by Signal. Both Operation and Signal are to be associated with the MessageArgumentSpecification's owning Message (association end *message*) through the association end *signature*. The explicit relation *argumentFor* between an actual argument and signature element eliminates the need for matching by indices of independent lists. Thus, there is no longer the need for collection ValueSpecifications, since the actual arguments for a certain signature element can be easily retrieved by gathering all MessageArgumentSpecifications that point to that signature element via *argumentFor* association end. This does not only simplify the processing of Messages, but also gives rise for more robust models in case of changes to the order of signature elements. As an example, we consider an Operation with two Parameters whose Types are non-compatible. If the user decides to alter the order of the Operation's Parameters, all Messages would have to reflect that change to not become invalid. If there is a large number of Messages that have set the Operation as their signature, and that already have correctly specified actual arguments, reflecting the changes might be a tedious task for the user. With the solution presented above, changing the order did not affect the validity of the Message at all due to the explicit coupling via *argumentFor*. Fig. 4 shows the relevant parts of the improved object model of Fig. 2.

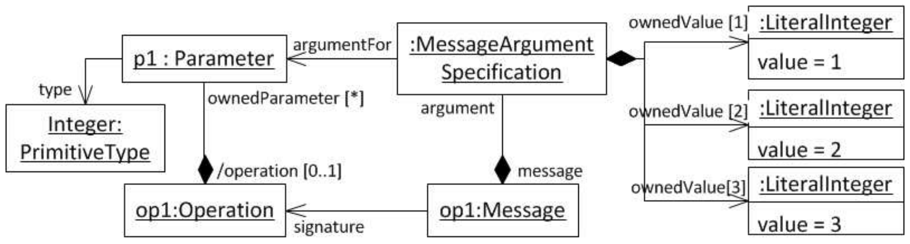


Fig. 4. Object model of well-formed model through improvements

4.2 Using References as Message Arguments

The sole use of ValueSpecifications as actual arguments is sufficient for expressing literal arguments or references to InstanceSpecifications. ValueSpecifications are, however, not capable to reference ConnectableElements (as superclass of Parameter and Property) directly. As a downside, it is not possible to reference values contained in data sources such as formal Parameters of the Interaction (or the corresponding BehavioralFeature the Interactions represents an implementation of) or Properties accessible to the sending Lifeline (such as local attributes of the Type the Lifeline represents, global attributes of the Classifier the Interaction is embedded in or local attributes of the Interaction itself). For the remainder of this paper, we call these values *reference arguments*. To motivate the improvement to the metamodel, the following Java code snippet shows a fundamental concept of using formal parameters of a surrounding Operation as actual parameter for a subsequent procedure call.

```
public class S { //context classifier of Interaction
  private C c; //offers op3(int i, String s)
  public void op2(int p1){
    c.op3(p1, “That works”); //realized as Interaction
  }
}
```

A realization of this snippet with the concepts offered by Interactions is only possible by either using an OCL navigation expression or again misusing other metaclasses like, e.g., OpaqueExpression (a subclass of ValueSpecification) as reference argument. Even though these workarounds would do the job, they are not satisfying because they impose additional parsing and execution facilities (e.g., in terms of OCL engine or any proprietary engine that evaluates the provided reference argument) being available. In Activities, there is a dedicated means to express data flow among actions (i.e., ObjectFlow and ObjectNode), for example. A native concept of Interactions is lacking, though. In preparation for this paper, we also checked the tools Rational Software Architect (RSA), MagicDraw and Enterprise Architect (EA). Except for the OCL variant, there is no mechanism offered to conveniently allow the user to specify reference arguments. OCL, however, is another language that needs to be learned by a user. Although OCL is highly recommend in the context of UML, for such fundamental concepts like referencing values in an accessible data source, we believe no additional language should be needed.

Unfortunately, the solution we presented in Fig. 3 suffers from the same deficiency as the current metamodel. A MessageArgumentSpecification still refers to ValueSpecifications solely, so consequently, we have to further elaborate our improvement to cope with the needs described above. Fig. 5 depicts our suggestion for such an improvement.

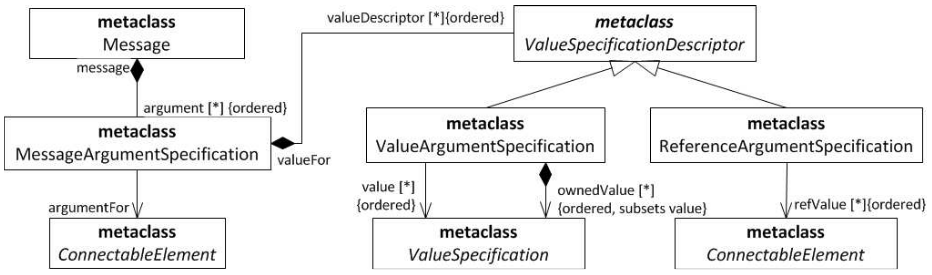


Fig. 5. Extended metamodel to cope with referenced arguments

The improved abstract syntax shown above introduces three new metaclasses. The abstract metaclass *ValueSpecificationDescriptor* replaces ValueSpecification as direct actual argument of a Message. ValueSpecificationDescriptor acts as a placeholder for the actual arguments, and knows two concrete subclasses *ValueArgumentSpecification* and *ReferenceArgumentSpecification*. The first one keeps the ability to use ValueSpecifications as actual arguments. The second one introduces the required facility to access reference arguments.

The extended metamodel now provides the required concepts to select reference arguments accessible from the sending Lifeline as actual arguments. The rules of what is actually accessible by a sending Lifeline are already defined in the current UML specification (see clause 5 of subsection Constraint of section 14.3.18) [1]. Furthermore, both ValueSpecificationDescriptor subclasses can be mixed with each other in a MessageArgumentSpecification. The Java snippet mentioned above stressed the need for mixing value and reference arguments.

A reference argument (*MessageArgument.valueDescriptor.refValue*) and its corresponding signature element (*MessageArgumentSpecification.argumentFor*) are interrelated by the fact that the reference argument needs to be type-compliant with and a subset of the multiplicity of the signature element. A multiplicity subset is defined as follows: Let M be the set of all MessageArgumentSpecifications in an Interaction. Furthermore, let s be a signature element, s_{low} its lower bound and s_{up} its upper bound. Let r be the reference argument corresponding to the signature element s, r_{low} the lower bound and r_{up} the upper bound of the reference argument, and $R_m(r,s)$ the relation of a concrete reference argument and signature element in the context of m (i.e., the concrete arguments are identified by the navigation expressions *m.valueDescriptor.refValue* and *m.argumentFor)*. Then the following must hold during runtime:

$$\forall m \in M : R_m(r, s) \rightarrow r_{low} \geq s_{low} \wedge r_{up} \leq s_{up} \tag{1}$$

In Fig. 6, the object model according to the Java code snippet is shown. The grey-shaded objects represent the parts of the specification of the Interaction. The bold-faced object is related to the reference argument concept. The association between MessageArgumentSpecification *ma1* and Parameter *i* as well as the association between ReferenceArgumentSpecification *vd1* and Parameter *p1* (marked by thick arrows) visualize how signature elements and reference elements belong together.

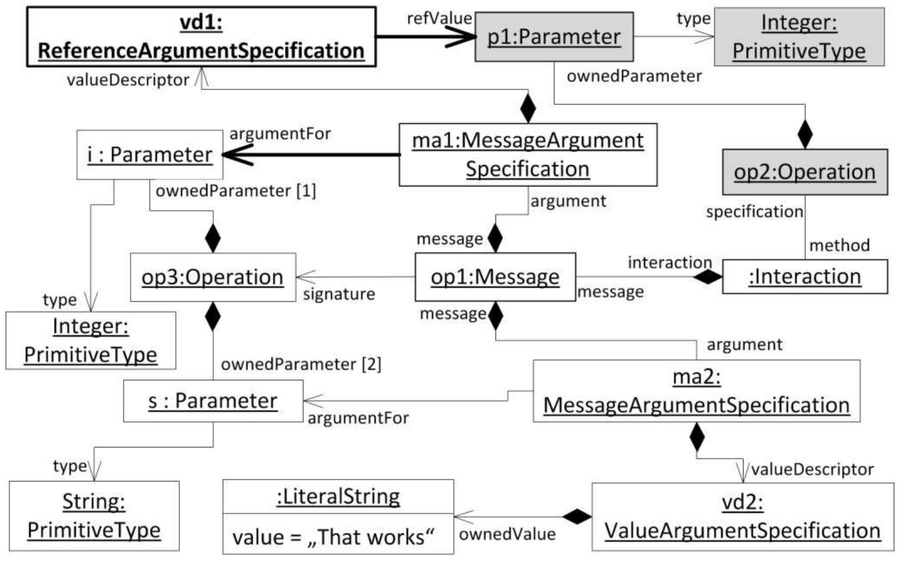


Fig. 6. Corresponding object model of improved Interactions metamodel

Still a problem appears in the solution, if the reference argument is a collection and has wider bounds than the corresponding signature element. There is currently no concept for extracting a subset of values from a reference argument collection. What is required is a facility for specifying such a subset of values that can be used by a reference argument. Therefore, the solution needs to be enhanced with a new metaclass *ReferenceValueSelector.* A ReferenceValueSelector is in charge of specifying that subset, if needed (see Fig. 7).

The subset of values for an actual argument is determined by one or more indices (expressed as Intervals) of the collection identified by ReferenceArgumentSpecification. An Interval allows specifying a minimal and maximal value. Since the association end *index* is unbound, it is possible to specify any number of subsets of elements, identified by their respective indices that shall be extracted from the reference argument collection. The flag *isIndexSetComplement* is a convenient way to specify what indices must not be taken over into the actual argument subset, whereas all indices which are not specified shall be actually considered. Runtime compliance of the index descriptions used in a ReferenceValueSelector cannot be ensured, of course.

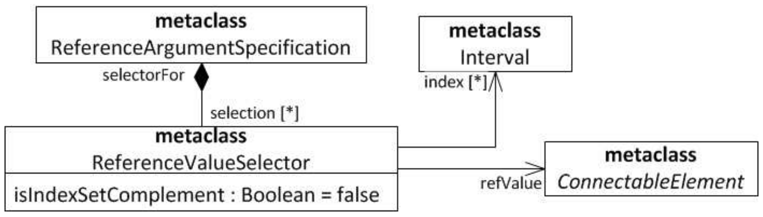


Fig. 7. Metamodel extended with ReferenceValueSelector metaclass

4.3 Assigning Values of a Message to Assignment Targets

Storing return values or parameters of a method call in appropriate assignment targets is rather natural in programming languages. A more complex (probably not meaningful) Java code snippet is presented below. The snippet is solely used for demonstration purposes of the *ArgumentAssignmentSpecification* metaclass we will introduce in this section. The code is supposed to represent parts of an operation body of the class *S*, which was already introduced in Section 4.2. *S* owns two Integer-typed lists (i.e., *piList1* and *piList2*) which are initialized. The actual content of the lists are not relevant for the example. The code simply selects a subset of a list retrieved by calling of `c2`'s operation `op4` and adds this subset to `piList1` and `piList2` of instance `s` of class S. In this section, we discuss the actual shortcomings of the current UML specification for such constructs and propose a solution.

```
List<Integer> list = c2.op4(); //actual size of list:999
List<Integer> tempList = new
List<Integer>(list.sublist(3,14)); //tempList size: 12
tempList.add(list.get(15)); //tempList size: 13
tempList.add(list.sublist(92,654)); //tempList size: 576
s.piList1.clear():
s.piList1.addAll(tempList);
s.piList2.addAll(tempList);
```

In a model, actual arguments of a Message shall be stored in assignment targets, which manifest in Properties or out-kind Parameters (i.e., Parameter with a ParameterDirectionKind *out, inout* or *reply*) of the surrounding Interaction accessible by the receiving Lifeline. Henceforth, we refer to an assignment target as data sink.

Even though argument assignment is reflected in the textual syntax of Messages in the current UML specification [1], there is no indication how this should be done with respect to the metamodel. The only statement in the notation subsection of Messages (see section 14.3.18) about assignment is that Actions are foreseen to describe the assignment. No further explanations or object model examples are given for clarification of how the connection between such an Action and an actual argument shall be established, nor what concrete Action to ultimately use. Furthermore, an Action needs to be integrated via an *ActionExecutionSpecification* covering the receiving Lifeline, but it is neither clear from the metamodel nor clarified in the textual specification how Message receptions and a set of conceptually related ActionExecutionSpecifications are linked with each other. In preparation for this paper, we investigated EA, RSA and MagicDraw. None of these most popular tools offered functionality for target assignment, though. Only the EA does have at least a notion for marking arguments for assignment, from the study of the resulting XMI, however, it was not clear to the authors how the assignment specification actually manifests.

Another rather conceptual shortcoming is that argument assignment is limited to the return Parameter of a Message solely, so that in-kind signature elements (i.e., either a Parameter with ParameterDirectionKind *in, inout*, or an attribute of a Signal) cannot be stored by a receiving Lifeline in a data sink. This ought to be possible, since in-kind signature elements represent information determined by the sending Lifeline and accessible by a receiving Lifeline. Therefore, actual arguments for in-kind signature elements should be further usable throughout the execution of the receiving Lifeline's behavior. This holds also true for out-kind signature elements of reply Messages, consequently, for sending Lifelines.

To cope with the needs for assigning actual arguments to data sinks accessible by Lifelines, we suggest introducing a similar concept as WriteStructuralFeatureAction from Activities (see clause 11.3.55 of UML [1]) for Interactions, called *ArgumentAssignmentSpecification* (see Fig. 8).

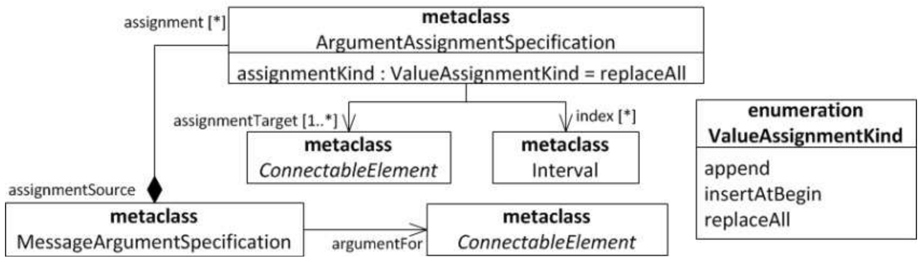


Fig. 8. Adding target assignment facilities to the metamodel

A MessageArgumentSpecification may contain a number of ArgumentAssignmentSpecifications, which, in turn, may specify a number of *assignment targets*. An assignment target represents a data sink that is intended to incorporate the actual arguments. In case the same actual arguments shall be assigned to several data sinks at the same time, the association end *assignmentTarget* is specified to be unbounded. Similar to ReferenceValueSelector, a number of Intervals can be used to specify what actual values at runtime shall be assigned to the assignment targets with respect to their indices, if the corresponding signature element represents a collection. However, the semantics in ArgumentAssignmentSpecification is converse, since it specifies what actual arguments shall be assigned to a data sink, in contrast to what reference arguments shall be taken from a data source as actual argument. However, as with ReferenceValueSelector, runtime compliance cannot be ensured at that point in time.

A *ValueAssignmentKind* specifies the treatment of already existing data in the assignment target in case the data sink represents a collection. Values of the actual argument at runtime will be either

- Added to existing contents of the data sink (*append*),
- Inserted at index 0 of data sink (*insertAtBegin*), or
- Replace all existing contents in the data sink *(replaceAll*).

Fig. 9 shows object model of the improved Interactions metamodel corresponding to the code snippet at the beginning of this section.

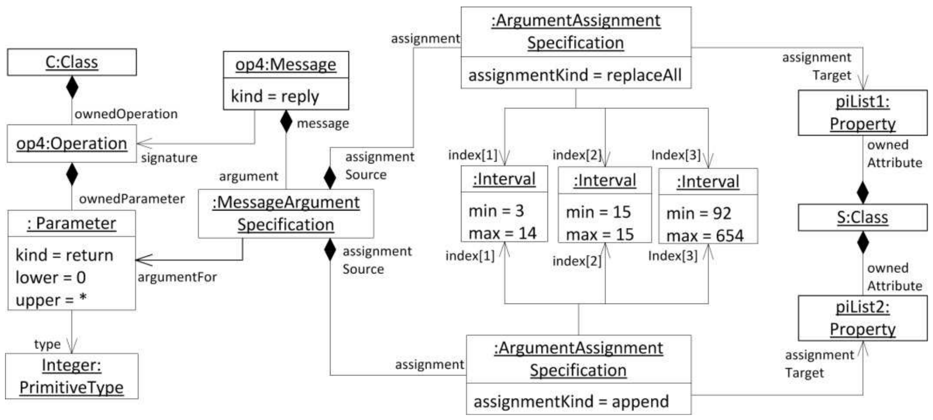


Fig. 9. Complex target assignment statements using collection indices

4.4 Improving Loop CombinedFragments

The semantics for CombinedFragments determined by their respective InteractionOperatorKind, but there are only two actual metaclasses for CombinedFragments in the Interaction’s metamodel: CombinedFragment and ConsiderIgnoreFragment, a specialization of CombinedFragment. The reason for a specialization of CombinedFragment by ConsiderIgnoreFragment is the additional information necessary to specify the messages to be considered or ignored. Additional information is also required for Loops to define the number of repetitions of the loop, however, in contrast to ConsiderIgnoreFragment, the repetition bounds have simply been added to the general CombinedFragment via the *InteractionConstraint* metaclass. It has two associations for specifying the bounds of a loop (*minint* and *maxint*). Anyway, it would be possible to specify meaningless combinations of CombinedFragments and repetition bounds, like Alternative CombinedFragment with explicit repetition bounds. To avoid these meaningless constructs, informal constraints were defined that disallow specifying repetition bounds in a different context than Loops. In the case of ConsiderIgnoreFragment the additional information is actually located in the metaclass that requires the information (i.e., ConsiderIgnoreFragment), for Loops, the information is located in the InteractionConstraint instead. This seems to be inconsistent when comparing Loop and ConsiderIgnoreFragment.

Our proposal treats Loops similar to ConsiderIgnoreFragment by introducing a new subclass of CombinedFragment called *LoopFragment* (see Fig. 10). This allows supplementing LoopFragment with the information required to specify the repetition bounds of the loop. Furthermore, the metaclass InteractionConstraint becomes obsolete, since the LoopFragment itself is now in charge of specifying the repetition bounds. By doing so, the only need for InteractionConstraint has vanished.

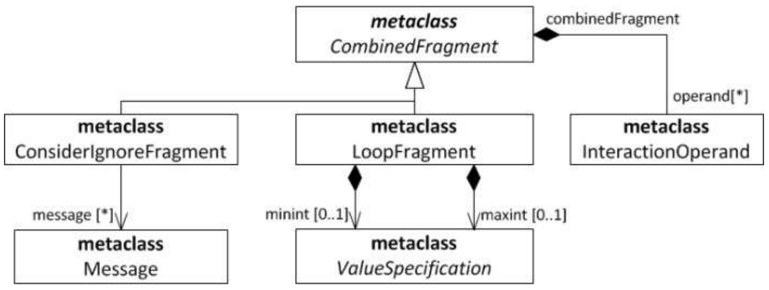


Fig. 10. Improved metamodel for *loop* Combined Fragments

Further considerations regarding CombinedFragments led to the conviction that the different kind of CombinedFragments, determined by the InteractionOperatorKind, should be resolved into concrete subclasses consequently. The reason for this lies in the too strong syntactical influence the InteractionOperatorKind impose on the structure of CombinedFragments. Applying a different InteractionOperatorKind to a CombinedFragment may enforce the removal of all but one InteractionOperand. For example, a CombinedFragment with two InteractionOperands and InteractionOperatorKind *alt* was defined and has been subsequently altered to *opt*, one of the InteractionOperands would have to be removed from the CombinedFragment. Therefore, we further refine the CombinedFragments metamodel in Fig. 11. Due to page limitations the figure does not show all specialized CombinedFragments that would ultimately result. The …*Fragment* metaclasses are placeholder for all remaining CombinedFragments with one or multiple InteractionOperands.

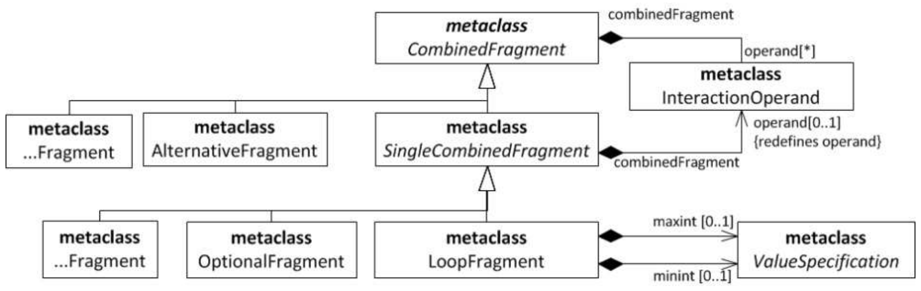


Fig. 11. Further refined CombinedFragment metamodel

5 Lessons Learned

The work presented led to two guidelines infor metamodel development activities. The first one refers to avoiding implicitly related elements; the second one provides an indicator when to use enumerations and when to use multiple metaclasses instead.

5.1 Avoid Implicitly Related Elements

This recommendation is accompanied by Einstein's famous simplicity principle: "Everything should be made as simple as possible, but not simpler." The UML Interactions metamodel counteracted this principle by simply reusing ValueSpecifications for a Message's arguments, instead of introducing a new metaclass that should have actually established a unidirectional link to the signature element. This gave rise to a situation where the list members of two semantically related lists were just implicitly related with each other via their respective indices. A new metaclass *MessageArgumentSpecification*, as we have suggested it, would have made the relation explicit and, the metamodel itself more robust regarding changes done by the user. The problem of implicitly related elements holds also true for other parts of the UML metamodel, though. InvocationAction, for example, exhibits the same issue as Messages in the relation of actual arguments and signature elements.

Our guideline for the creation of more robust metamodels is: Avoid implicitly related elements. The assumed benefits of saving the metaclass that formalizes the relation are paid off by increased efforts for future maintenance, comprehension and metamodel processing.

5.2 Enumeration vs. Metaclass

A question that is still not sufficiently answered, at least to the knowledge of the authors, is when to use enumerations and when to use several specialized metaclasses? Doubtlessly, the underlying semantics will not be influenced either way. Enumerations allow reducing the actual number of metaclasses in a metamodel. For example, every NamedElement defines a visibility within the Namespace it is contained in. The possible visibilities a NamedElement can declare are defined in the enumeration VisibilityKind as *public*, *private*, *protected* and *package*. Each subclass of NamedElement inherits the visibility feature and its semantics, thus, the design of visibility throughout the entire inheritance hierarchy of NamedElement was well chosen. Specialized metaclasses instead (e.g., NamedElementPublic, NamedElementPrivate etc.) would have resulted in an unnecessarily complex metamodel.

So, using enumerations seems to be adequate and accurate if the EnumerationLiterals merely affect the semantics of the metaclass they are referenced from. Furthermore, enumerations can keep the inheritance hierarchy of the metaclass concise.

With respect to the CombinedFragment's *interactionOperator* (and in few other metaclasses in UML such as Pseudostate), the situation is different. The various literals of InteractionOperatorKind do affect not only the semantics, but the syntactical structure of CombinedFragments as well. In this case, changing the enumeration may require changing the instance of the metaclass as well. The problem of varying syntax due to enumerations is that the understanding of the metamodel becomes unnecessarily complicated and its maintenance prone to errors. Even though the solution we presented in Fig. 11 results in a larger number of similar metaclasses, the metamodel becomes more comprehensible and the actual syntactical differences of the specialized metaclasses become obvious.

Our guideline for metamodels regarding enumerations or specialized subclasses is: If different literals of an Enumeration may turn the model into a syntactically ill-formed model, one should use specialized metaclasses instead.

6 Conclusion and Outlook

In this paper, we have presented improvement suggestions for parts of the UML Interactions metamodel regarding Message arguments and CombinedFragments. We stressed that the current metamodel of Message arguments reveals some issues of precise specification of actual arguments, usage of reference arguments as actual arguments and assignment of actual arguments to data sinks accessible by the receiving Lifeline. Whether these issues originate from the UML Interactions metamodel or ought to be solved by general concepts of the UML metamodel is not in scope of this paper. We assumed the view of a user of UML who is wondering that actual argument handling is possible in UML Activities, but only inconveniently (if ever) supported by Interactions. From that perspective, we suggested improvements limited to the Interactions' Message metamodel to overcome these issues. The improved metamodel was the result of the development of a tool for test modeling, called Fokus!MBT that relies on the UML Testing Profile and leverages UML Interactions as test case behavior [20]. In the scope of Fokus!MBT, a minimalistic profile was created that realizes the metamodel improvements we described with stereotypes. So, the metamodel improvements have been applied to real situations and are not just theoretical considerations.

Finally, we extracted two guidelines to metamodeling for more robust metamodels.

The fact that UML Activities and Interactions do provide different approaches for the very same logical concept gives rise to the considerations that these behavior kinds should be more tightly integrated with each other in future. There is actually an issue submitted for this[2] need. We support that need, which would result in a more concise and comprehensible metamodel for UML. As a result, it might turn out that the issues discussed in the paper rather belong to the fundamental parts of the UML metamodel. However, as long as Activities and Interactions are treated as separate parts, the improvements we presented are most minimalistic, since they do not affect any other part of the UML metamodel. An integration of both behavior kinds is not a trivial task, though, and not in scope of this paper.

[2] http://www.omg.org/issues/uml2-rtf.open.html#Issue6441

Acknowledgements. This work was partially funded by the EU FP 7 projects REMICS (no. 257793) and MIDAS (no. 318786) and ARTEMIS project VARIES.

References

1. OMG UML: OMGT Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1, #formal/2011-08-06, http://www.omg.org/spec/UML/2.4.1/, 2011.
2. Grabowski, J., and Rudolph, E.: Message Sequence Chart (MSC) - A Survey of the new CCITT Language for the Description of Traces within Communication Systems. CCITT SDL Newsletter, No 16, pp. 30–48, 1993.
3. OMG UTP: OMG UML Testing Profile (UTP), Version 1.2, #ptc/2012-09-13, http://www.omg.org/spec/UTP, 2012.
4. Baker, P., Dai, Z.R., Grabowski, J., Haugen, Ø., Schieferdecker, I. Williams, C.: Model-driven testing – using the UML testing profile. Springer (2007)
5. Haugen, Ø: Comparing UML 2.0 Interactions and MSC-2000. 4th International SDL and MSC Workshop, pp. 65–79, SAM 2004, Ottawa, Canada, 2004.
6. Haugen, Ø. and Stølen, K.: STAIRS — Steps to analyze interactions with refinement semantics. In Proc. International Conference on UML, Volume 2863 of LNCS. Springer, pp. 388–402, 2003.
7. Haugen, Ø., Husa, K. E., Runde, R. K., and Stølen, K.: Why timed sequence diagrams require three-event semantics. In Scenarios: Models, Transformations and Tools, Volume 3466 of LNCS. Springer, pp. 1–25, 2005.
8. Haugen, Ø., Husa, K.E., Runde, R.K., and Stølen, K.: STAIRS towards formal design with sequence diagrams. Journal of Software and Systems Modeling, pp. 349–458, 2005.
9. Runde, R. K., Haugen, Ø., Stølen, K.: Refining UML interactions with underspecification and nondeterminism. In: Nordic Journal of Computing, Volume 12, Issue 2, pp. 157–188, 2005.
10. Lund, M. S., and Stølen, K.: A fully general operational semantics for UML 2.0 sequence diagrams with potential and mandatory choice. In: Proceedings of the 14th international conference on Formal Methods (FM'06), Pages 380-395, 2006.
11. Störrle, H.: Semantics of interactions in UML 2.0. In: Proceedings of IEEE Symposium on Human Centric Computing Languages and Environments, 2003.
12. Störrle, H.: Trace Semantics of UML 2.0 Interactions. Technical report, University of Munich, 2004.
13. Knapp, A.: A Formal Semantics for UML Interactions. In: R. France and B. Rumpe (eds.): Proc. 2nd Int. Conf. Unified Modeling Language (UML'99), LNCS volume 1723, pp. 116–130. Springer, Berlin, 1999.
14. Cengarle, M., Knapp, A.: UML 2.0 Interactions: Semantics and Refinement. In: J. Jürjens, E. B. Fernàndez, R. France, B. Rumpe (eds.): 3rd Int. Workshop on Critical Systems Development with UML (CSDUML'04), pp.85–99, 2004.
15. Li, M., and Ruan Y.: Approach to Formalizing UML Sequence Diagrams. In: Proc. 3rd International Workshop on Intelligent Systems and Applications (ISA), 2011, pp. 28–29, 2011.
16. Shen, H., Virani, A.; Niu, J.: Formalize UML 2 Sequence Diagrams. In: Proc. 11th IEEE High Assurance Systems Engineering Symposium (HASE) 2008, pp. 437–440, 2008.
17. Störrle, H.: Assert, Negate and Refinement in UML-22 Interactions. In: J. Jürjens, B. Rumpe, R. France, and E. B. Fernandez, Proc. Wsh. Critical Systems Development with UML (CSDUML'03), San Francisco, 2003.

18. Harel, D., and Maoz, S.: Assert and negate revisited: modal semantics for UML sequence diagrams. In: Proc. International workshop on Scenarios and state machines: models, algorithms, and tools (SCESM '06), 2006.
19. Knapp, A., and Wuttke, J.: Model Checking of UML 2.0 Interactions. In; Proc. of the 2006 International conference on Models in Software Engineering (MoDELS'06), pp. 42–51, Springer, Heidelberg, 2006.
20. Wendland, M.-F., Hoffmann, A., and Schieferdecker, I.: Fokus!MBT – A Multi-Paradigmatic Test Modeling Environment. To appear in proceedings of: Academics Tooling with Eclipse Workshop (ACME) 2013, in conjunction with the joint conferences ECMFA/ECSA/ECOOP, Montpellier, France, ISBN 978-1-4503-2036-8, 2013