Use Case Model Refactoring: Changes to UML´s Use Case Relationships

Pierre Metz[1], John O´Brien[1], Wolfgang Weber[2]

[1] Dept. of Mathematics & Computing, Cork Institute of Technology, Ireland,
{pmetz,jobrien}@cit.ie

[2] Dept. of Computer Science, Darmstadt University of Applied Sciences, Germany,
w.weber@fbi.fh-darmstadt.de

Abstract. *Use cases are a powerful and widely recognised tool for functional requirements elicitation. However, there still are major problems and gaps in UML´s use case foundation. One such issue is the way extend and include relationships are used in refactoring textual and graphical use case models. Experience shows that practitioners do not only use extend-relationships as defined by UML, but also to express both partial and fully parallel interaction courses and exceptional use case behaviour. However, UML´s current definition of the extend-relationship fails in this demand. Hence, this work presents significant and detailed UML changes to fully support these practical needs, based on a previous work that identified and defined types of alternative courses in use case interaction within the context of goal-driven requirement engineering. We include previous research findings on the notion of use case interleaving in use case relationships. As a result, this paper will close major and prevailing gaps between the UML use case metamodel and practical necessities leading to an improved understanding of how to apply use case relationships in practice.*

An initial review of use case basics is presented to help orientate the reader. Based on this, simple examples are presented that highlight the extend-relationship´s weaknesses. Finally, we will finish with UML change proposals to fill in these gaps; these proposals are illustrated by applying them to the examples presented.

1 Brief Review of Use Case Basics

A use case is the description of a software-supported part of an enterprise´s business processes. It is derived from the stakeholder´s and actor´s needs. Stakeholders and actors may be human, hardware, or component elements of the future system's IT environment. A use case must provide value to these business processes. Therefore, it is associated with a set of atomic measurable business results that is delivered to an actor or stakeholder [6]. The actor instances need to participate in a use case execution in order to deliver the use case business results. Consequently, a use case goal leads to an interaction between the system and actors. The description of use case interaction encompasses two parts [9]. A *Basic Course* describes the main sequence of interaction in which everything goes right ("happy path"). Any non-frequent alternative of the basic course of interaction, such as optional interaction parts and business error recovery, is called an *Alternative Course*. In this sense, the term a*lternative course* specifies a guarded variation of a part of another interaction course and, thus, is subject to a business guard [1], [6], [9], [10], [11], [17]. The following examples illustrate basic courses and alternative courses following the specification technique in [6]. These examples will further serve as a basis for understanding the reasoning presented in the subsequent sections.

Example 1: use case goal with the goal "Withdraw Cash" of an ATM

Basic course:

```
1. The customer inserts
   the card.
2. The system validates
   the card.
3. The Customer enters
   the PIN.
4. The system validates
   the PIN.
5. ...
```

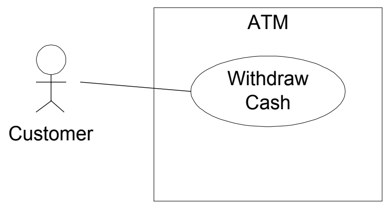

Fig. 1: Use Case Diagram for Example 1

Now consider the following alternative courses:

```
4a.    PIN has been entered incorrectly for the first or
       the second time:

       4a1. The system notifies the customer.
       4a2. The system logs the attempt.
       4a3. Rejoin at 3.

4b.    PIN has been entered incorrectly the third time:

       4b1. The system notifies the customer.
       4b2. The system logs the attempt.
       4b3. The system withholds the card.
       4b4. Use case is aborted.
```

Example 2: use case with the goal "Register New Student"

Basic course:

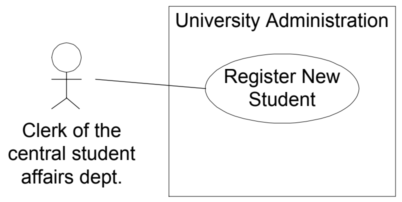
University Administration
Register New Student
Clerk of the central student affairs dept.

Fig. 2: Use Case Diagram for Example 2

```
1.  Clerk enters new student´s name, address, ...
2.  The system assigns a student ID.
3.  Clerk assigns the university department.
4.  System prints out a confirmation of registration.
5.  ...
```

Now consider the following alternative course:

```
3a.    Student is a foreign student with scholarship:

       3a1. The system shows the capacity of free campus
            apartments.
       3a2. The clerk assigns an apartment to the student ID.
       3a3. Rejoin at 3.
```

Example 3: use case with the goal “Register New Employee“

Basic Course:

```
1.   The personnel manager
     enters new employee´s
     name, address, and
     telephone number.
2.   The system displays the
     vacancies and job
     category.
3.   The personnel manager
     chooses the considered
     vacancy.
4.   The system shows the scale of wages.
5.   The personnel manager determines the hourly rate.
6.   The system assigns an employee number.
7.   The system assigns the company department based on the
     selected vacancy.
```

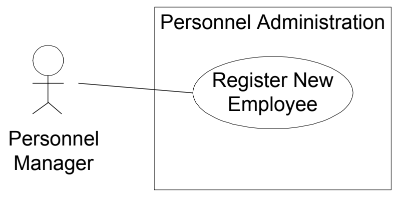

Fig. 3: Use Case Diagram for Example 3

Now consider the following alternative course:

```
4a.  The applicant will be paid outside the pay scale and
     will have flexible working hours:
     4a1. The personnel manager determines the new employee´s
          monthly salary.
     4a2. The personnel manager determines fringe benefits.
     4a3. Rejoin at 6.
```

The following section relates the above-highlighted use case fundamentals to the semantics of UML’s use case relationships.

2 UML´s Use Case Relationships

An extend-relationship is used for extracting alternative courses from a base use case into a new use case and attaching it to the base use case[1] [1], [5], [9], [10], [13], [15], [16], [17]. Therefore, when extracting an alternative course, an extend-relationship specifies a condition as well as an extension point which references the label of a single branching location within the base use case [13]. Fig. 4, Fig. 5 and Fig. 6 show examples of the application of extend-relationships based on the introductory examples in Section 1.

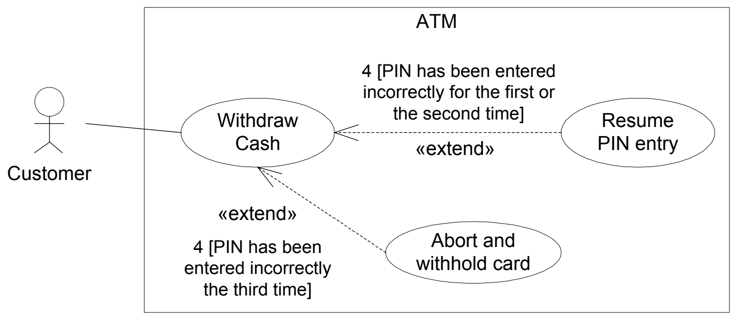
ATM
Customer
Withdraw Cash
4 [PIN has been entered incorrectly for the first or the second time]
«extend»
Resume PIN entry
«extend»
4 [PIN has been entered incorrectly the third time]
Abort and withhold card

Fig. 4: Extraction of the alternative courses in Example 1

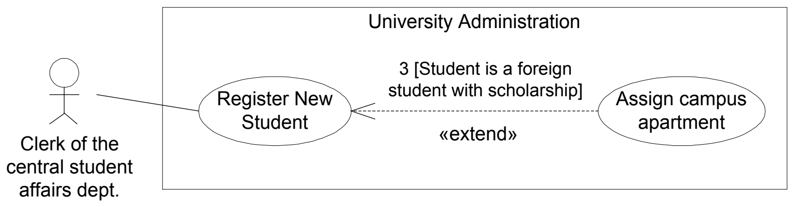
University Administration
Clerk of the central student affairs dept.
Register New Student
3 [Student is a foreign student with scholarship]
«extend»
Assign campus apartment

Fig. 5: Extraction of the alternative course in Example 2

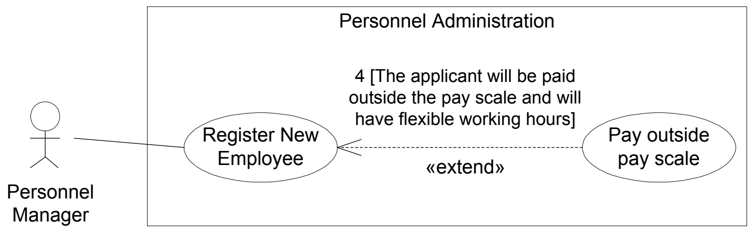

Fig. 6: Extraction of the alternative course in Example 3

Many software professionals disregard using extend-relationships when extracting alternative courses into new use cases because it means the arbitrary functional

[1] In fact, this is the only usage of the extend-relationship since the concept of use case interleaving for Include and Extend has been disproved [13].

decomposition of the use case model; they do prefer, instead, to keep alternative courses as local textual use case properties [6], [17]. Nevertheless, some authors consider the extraction of alternative courses when attempting to reduce the textual complexity within use cases [6]. Generally, however, an alternative course must remain extractable with an extend-relationship. This is necessary to remove redundancy from the use case model when an alternative course appears to be redundant in two or more use cases or within a single use case. This is evident from practical experience; it is also supported by the UML metamodel: "*Commonalities between use cases can be expressed in three different ways: with generalization, include relationships or extend relationships.*" [15] (p. 2-150) and "*An extend relationship defines that a use case may be extended with some additional behavior defined in another use case. One use case may extend several use cases and one use case may be extended by several use cases...*" [15] (p. 2-150), [16].

In contrast to the extend-relationship, UML´s include-relationship extracts unconditional use case interaction parts; an include-relationship is not subject to a business guard. Initially, the include-relationship was intended for removing redundancy from within two or more use cases [9], [10], [15], [17], [18]. An example for our ATM would be the authorisation of the customer by PIN entry within the use cases "Withdraw Cash" and "Transfer Money" of an ATM (see Fig. 7).

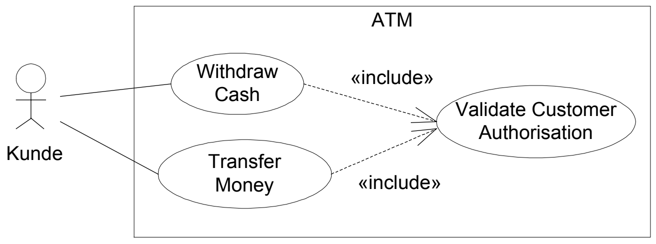

Fig. 7: Removing of redundant unconditional use case interaction with an include-relationship

3 The Semantics of the Extend-Relationship Are Insufficient

It is absolutely clear from the UML v1.4 metamodel that the extend-relationship metamodel supports conditional insertion only. This is evident in statements such as: "*Note that the condition is only evaluated once: at the ... extension point, and if it is fulfilled all of the extending use case is inserted in the original sequence.*" ([15], p. 2-151); these are further supported in the UML User Guide [5]. Accordingly, the UML Reference Guide confirms these semantics by stating that an extend-relationship is "*...specifying how the behavior defined for the extension use case can be inserted into the behavior defined for the base use case.*" ([16], p. 272) and "*When the performance of an extension ... is complete, the use case instance resumes performing the base use case at the location at which it left off.*" ([16], p. 274). This accurately shows that the extend-relationship solely follows conditional insertion semantics (see Fig. 8), a conclusion that is also shared in [1], [7], [17], [18]. The student registration example in Example 2 shows an alternative course that represents conditional insertion. Hence, in accordance with UML, we may extract this alternative course and attach it with an extend-relationship as correctly shown by the use case diagram in Fig. 5. However, apart from conditional insertion, an alternative course of use case interaction may further represent a use case exception, an alternative history, or an alternative interaction part as identified in [14] and as shown in the examples in Section 1:

1. Use Case Exception

   A use case exception branches the base interaction course and never rejoins. A use case exception means a non-recoverable use case goal failure, i.e. it aborts the overall use case not establishing the use case´s business results. This is indicated by the "earthing" in Fig. 8. The alternative course 4b in Example 1 shows a use case exception.

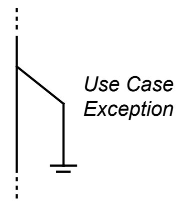

Fig. 8: Illustration of a se case exception

2. Alternative History

   An alternative history branches the base interaction sequence never to rejoin, i.e. an alternative history also represents a fully parallel interaction flow. An alternative history differs from a use case exception in that at the end of an alternative history the use case goal is achieved and its business results are delivered. This is indicated by the line going to the ground in Fig. 9.

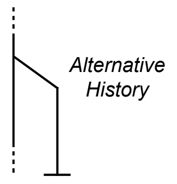

Fig. 9: Illustration of an alternative history

3. Alternative Part

   An alternative part encompasses conditional insertion (see Fig. 10a), interaction cycles (see Fig. 10b) and partially parallel interaction courses (see Fig. 10c). An alternative part is an alternative course that always rejoins the branched base sequence. The examples in Section 1 illustrate alternative parts: an interaction cycle is shown by the alternative course 4a in Example 1. The alternative course

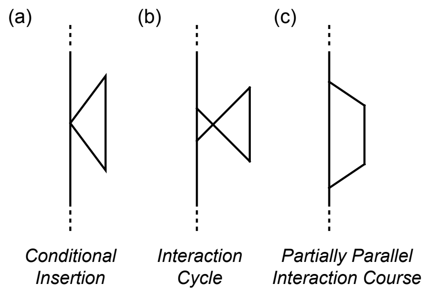

Fig. 10: Graphical illustration of the term Alternative Part

in Example 3 shows a partially parallel interaction course.
Practical experience shows that all such types of alternative courses are necessary [1], [6], [11], [12], [14], [17], [18]. Many authors and practitioners have already been adopting and practising these ideas [1], [6], [11], [12], [14]. UML neither defines use case documentation items, nor offers it a tailorable use case template [15]. Hence, as long as we do not extract alternative courses into another extend-attached use case, but keep alternative courses as local textual use case properties, no problems in exploiting all given types of alternative courses for capturing functional software requirements are encountered. However, with respect to the application of extend-relationships, the UML v1.4 metamodel fails to support use case exceptions, alternative histories, interaction cycles and partially parallel interaction courses [1], [15], [17], [18]. These types of alternative courses are not provable in the UML metamodel since, as has been identified above, that UML´s extend-relationship supports conditional insertion only. Consequently, the use case diagrams depicted in Fig. 4, Fig. 5 and Fig. 6 that are based on Example 1, Example 2 and Example 3 would be a violation of the UML semantics.

4 UML Metamodel Change Proposals

The following section proposes changes to UML's extend-relationship, with a view to addressing the above-identified inadequacies. To paraphrase Colin Atkinson [2], all suggestions for improving and advancing the UML specification need a sound and precise basic metamodel architecture. However, the UML architecture is still evolving and a number of competing proposals currently exists. Still, in our opinion, one of the most promising and thorough approaches is that of Atkinson/Kühne [3]; however, none of these approaches has yet been formally adopted. Consequently, our suggestions, based on the current UML architecture, recommend alterations of the abstract syntax of UML´s Behavioural Package Use Cases; we also recommend OCL constraints that ensure our intended semantics. Nevertheless, adaptations of our recommendations to future architectural advances are desired.

4.1 UML Abstract Syntax

Initially, we examine UML´s original use case metamodel part and consider recently recommended alterations. These are subsequently built on.

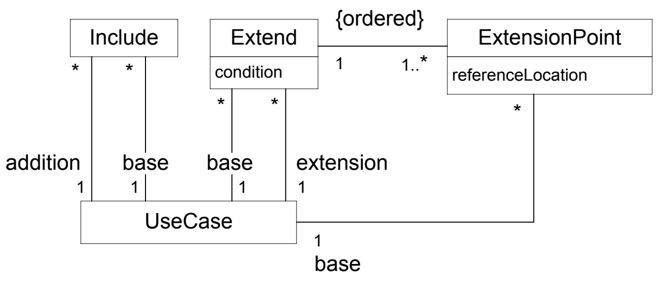

Fig. 11: Extraction from the UML v1.4 Abstract Syntax Behavioural Package Use Cases [15]

Originally, UML defined the use case metamodel part as shown in Fig. 11. Fig. 12, however, shows adaptations that have been made in order to remove inconsistencies caused by the notion of use case interleaving that has been disproved in [13]. As a result of these adaptations, the attribute *condition* of Extend has been removed and has been reintroduced as an explicit metamodel class *Condition* (see Fig. 11 and Fig. 12). Furthermore, the attribute *referenceLocation* has been removed (see Fig. 11 and Fig. 12). An instance of the UML metamodel element *ExtensionPoint* now specifies the reference to a published label of a *single* branching location within the base use case when an alternative course is extracted into another use case that is attached via «extend». Moreover, an extension can have more than one extension point (see Fig. 12). The reason is that within a single use case a certain conditional interaction may appear more than once. Hence, in order to remove such intra-use case redundancy, the alternative course is extracted and connected to more than one location within the base use case accordingly [16]. For further details see [13].

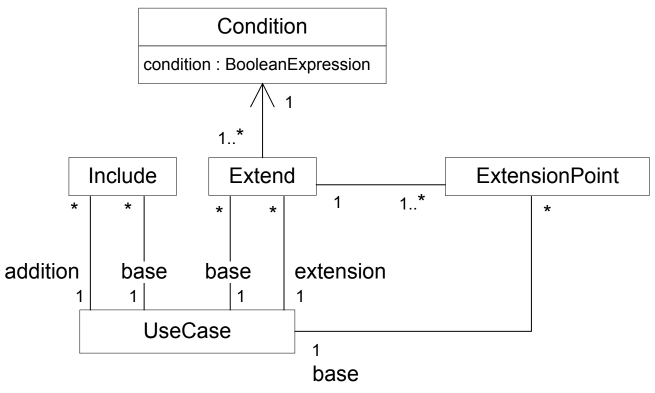

Fig. 12: Adapted UML Abstract Syntax Behavioural Package [13]

Let us now consider how the semantics of the extend-relationship can support alternative histories, use case exceptions and alternative parts. We need to introduce an abstract metamodel element *PointOfRejoin* (see Fig. 13) and two subtypes *RealRejoinPoint* and *NullRejoinPoint*. Like an ExtensionPoint, an instance of RealRejoinPoint specifies a reference to the published label of a single return location within the base interaction sequence, when an alternative course is extracted into an additional extend-attached use case. An extend-relationship will have an associated RealRejoinPoint if it represents an alternative part; correspondingly, an extend-relationship will have an associated NullRejoinPoint[2] if it represents either an alternative history or a use case exception. This solution approach allows the association from Extend to PointOfRejoin to have a lower bound of one which enables us to treat Include and Extend symmetrically (see below) and further simplifies the, subsequently introduced, OCL constraints.

[2] Following the idea of the Null-Object pattern in [20].

Since an extend-relationship may have more than one extension point, the higher bound of the association from Extend to PointOfRejoin is also many.

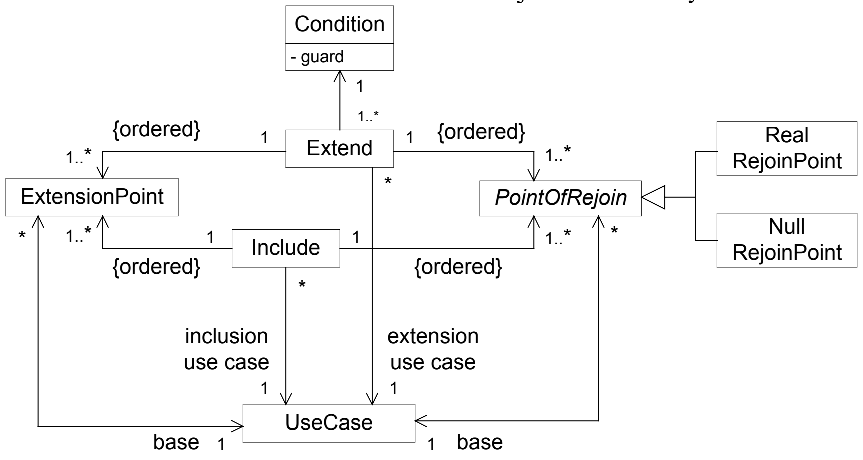

Fig. 13: Support for alternative histories, use case exceptions, and alternative parts - adapted UML Abstract Syntax Behavioural Package Use Cases (based on Fig. 12)

An ExtensionPoint and its corresponding PointOfRejoin belong together, i.e. semantically, they form pairs. In order to reflect this, we must find a mechanism for tying an ExtensionPoint and a PointOfRejoin together in our model. In this context we realise a further advantage of the PointOfRejoin hierarchy solution: since every extend-relationship now has a PointOfRejoin and the sizes of both associations are <always equal, because extension points and points of rejoin form pairs, the associations from Extend to ExtensionPoint and PointOfRejoin can be ordered. Consequently, these associations form a sequence and the positions within these sequences indicate the interrelation.
Fig. 14 shows how a point of rejoin is added to the extend arrow in a use case diagram; the absence of a point of rejoin specifies either an alternative history or a use case exception[3]. The exact type of alternative course is revealed by the extension use case textual specification (see [14]). If an extend-attached alternative course is redundant in a single use case, then the comma-separated pairs are separated with semicolons.

In the original UML definition of the include-relationship there is no extension point equivalent [15] (see Fig. 11). However, we believe that the include- and the extend-relationships should be treated symmetrically; other authors have also expressed similar views [4], [7], [17], [18]. In our opinion the only difference between Extend and Include is that Include is not subject to a condition. Therefore, include-attached extension use cases always contain mandatory interaction and adhere to insertion

[3] We do not introduce additional stereotypes to the extend-relationship to indicate the above-mentioned types of alternative courses since this would mean creating subtypes of Extend in the UML metamodel which we believe is not desirable; alternatively, our solution approach is to associate a PointOfRejoin with Extend as has been shown above.

semantics only [1], [5], [6], [11], [13], [16], [17], [18]. Extend, being guarded interaction, supports a richer set of interaction courses. In our view, an include-relationship also has extension points and corresponding points of rejoin.

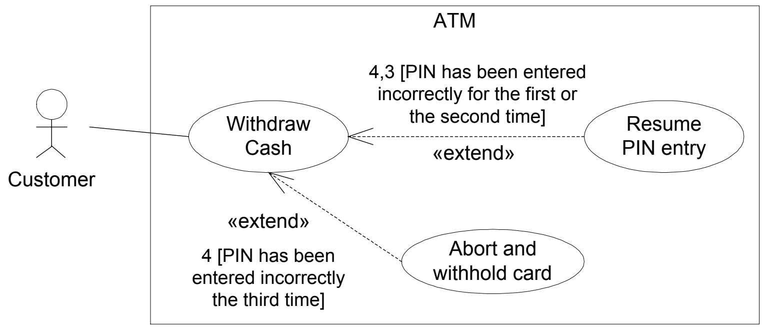
ATM
4,3 [PIN has been entered incorrectly for the first or the second time]
Withdraw Cash
Resume PIN entry
«extend»
Customer
«extend»
4 [PIN has been entered incorrectly the third time]
Abort and withhold card

Fig. 14: UML changes applied to Fig. 4

In order to establish symmetry, the metamodel element Include likewise gets ordered associations to ExtensionPoint and PointOfRejoin (see Fig. 13). We believe that there is no need to increase the metamodel complexity and number of terms required through the introduction of a metamodel element *InclusionPoint*; the semantics of such an inclusion point would not differ from those of ExtensionPoint. Of course, an include-relationship must never associate with a NullRejoinPoint which is guaranteed by OCL constraints presented below. The higher bound of Include´s associations to ExtensionPoint and PointOfRejoin is many for the same reason as for Extend: within a single use case a certain unconditional interaction, appearing more than once, is extracted in order to remove redundancy.
It should be noted that for an include-relationship the point of rejoin and the extension point are always consecutive within the base use case because it follows insertion semantics. Hence, for graphical convenience, we suggest allowing to omit extension points and points of rejoin for include-relationships in practical use case modelling (see Fig. 7). However, in the metamodel an include-relationship has a point of rejoin.

The associations between ExtensionPoint, PointOfRejoin, and UseCase are still considered bidirectional[4] as originally defined by UML [15] (see Fig. 11 and Fig. 13). The reason is that Classifier encapsulation [15], [13] must be observed: if extension points were invisible to the base use case and, thus, only the include- and extend-relationships were aware of the branching location within the base use case sequence, then the encapsulated features and state of the base use case would have to be exposed. Since use cases are Classifiers, this would result in a violation of UML´s

[4] Originally, UML states that an association without any navigation arrow implicitly imposes bidirectionality. However, we follow an approach that is preferred in practice and also suggested in [8]: no navigation arrow means the navigability being undefined, whereas one or two navigation arrows graphically express unidirectional or bidirectional navigability, respectively.

Classifier foundation (see also [13]). Hence, the base use case is aware of its extension points.

4.2 Additional UML Wellformedness-Rules

The following proposes additional UML Well-Formedness Rules to support the suggestions offered in Section 4.1. Following [19], the OCL operator "=" is considered as checking for object identity but not for object equality when applied to metamodel objects. All constraints specified in [15], that are not mentioned here, apply.

4.2.1 Constraints for the Metamodel Element Use Case

```
context UseCase inv:
```

[1] Each ExtensionPoint instance of a particular use case instance within the ExtensionPoint sequence is unique. This constraint substitutes the original constraint no. 4 for use cases in [15]:

```
self.extensionPoint->asSet()->size() =
                        self.extensionPoint ->size()
```

[2] Each PointOfRejoin instance of a particular use case instance within the PointOfRejoin sequence is unique:

```
self.pointOfRejoin->asSet()->size() =
                        self.pointOfRejoin->size()
```

4.2.2 Constraints for the Metamodel Element Extend

```
context Extend inv:
```

[3] Since an Extend-relationship instance is considered to have a set of pairs consisting of a single ExtensionPoint instance and a single PointOfRejoin instance, both sequences must be of the same size:

```
self.extensionPoint->size = self.pointOfRejoin->size
```

[4] Each ExtensionPoint instances of a particular Extend-relationship instance within the ExtensionPoint sequence is unique:

```
self.extensionPoint->asSet()->size() =
                        self.extensionPoint->size()
```

[5] Each PointOfRejoin instance of a particular Extend-relationship instance within the PointOfRejoin sequence is unique:

```
self.pointOfRejoin->asSet()->size() =
                    self.pointOfRejoin->size()
```

[6] An extension can never change its type, i.e. if one extension instance has associated a certain type of point of rejoin instance then all extension instances have that type of point of rejoin associated:

```
self.pointOfRejoin->forAll
(each | each.OCLType = self.PointOfRejoin->first.OCLType)
```

[7] All ExtensionPoint instances of a particular Extend-relationship instance refer to the same base use case instance:

```
self.extensionPoint->forAll
(each | each.base = self. extensionPoint->first.base)
```

[8] The base use case instance of the ExtensionPoint instances of a particular Extend-relationship instance must not be the same as the extension use case instance of this Extend instance:

```
self.extensionPoint->forAll
(each | each.base <> self.extensionUseCase)
```

[9] All PointsOfRejoin instances of a particular Extend-relationship instance refer to the same base use case instance:

```
self.pointOfRejoin->forAll
   (each | each.base =
      self.pointOfRejoin->first.pointOfRejoin.base)
```

[10] The base use case instance of the PointsOfRejoin instances of a particular Extend-relationship instance must not be the same as the extension use case instance of this Extend instance:

```
self.pointOfRejoin->forAll
(each | each.base <> self.extensionUseCase)
```

[11] The commonly referred base use case instance of the ExtensionPoint instances (see [7] above) and the commonly referred base use case instance of the PointsOfRejoin instances (see [9] above) of a particular Extend-relationship instance must be the same:

```
self.extensionPoint->first.base =
             self.pointOfRejoin->first.base
```

[12] The ExtensionPoint instances of a particular Extend instance must be contained in the sequence of ExtensionPoint instances of the base use case instance of this Extend instance:

```
self.extensionPoint->first.base.extensionPoint
    ->includesAll (self.extensionPoint)
```

[13] The PointOfRejoin instances of a particular Extend instance must be contained in the sequence of PointOfRejoin instances of the base use case instance of this Extend instance:

```
self.pointOfRejoin->first.base.pointOfRejoin->includesAll
        (self.pointOfRejoin)
```

4.2.3 Constraints for the Metamodel Element Include

```
context Include inv:
```

[14] Since an Include-relationship instance is considered to have a set of pairs consisting of a single ExtensionPoint instance and a single PointOfRejoin instance, both sequences must be of the same size:

```
self.ExtensionPoint->size = self.PointOfRejoin->size
```

[15] Each ExtensionPoint instance of a particular Include-relationship instance within the ExtensionPoint sequence is unique:

```
self.extensionPoint->asSet()->size() =
                       self.extensionPoint->size()
```

[16] Each PointOfRejoin instance of a particular Include-relationship instance within the PointOfRejoin sequence is unique:

```
self.pointOfRejoin->asSet()->size() =
                       self.pointOfRejoin->size()
```

[17] All ExtensionPoint instances of a particular Include-relationship instance refer to the same base use case instance:

```
self.extensionPoint->forAll
(each | each.base = self.extensionPoint->first.base)
```

[18] The base use case instance referenced by the ExtensionPoint instances of a particular Include-relationship instance must not be identical with the inclusion use case instance of this Include instance:

```
self.extensionPoint->forAll
        (each | each.base <> self.inclusionUseCase)
```

[19] All PointOfRejoin instances of a particular Include-relationship instance are instances of the type RealRejoinPoint:

```
self.pointOfRejoin->forAll
        (each | each.OCLType = RealRejoinPoint)
```

[20] All RealRejoinPoint instances of a particular Include-relationship instance refer to the same base use case instance:

```
self. pointOfRejoin->forAll
(each | each.base = self.pointOfRejoin->first.base)
```

[21] The base use case instance of the RealRejoinPoint instances of a particular include-relationship must not be the same as the inclusion use case instance of this Include instance:

```
self.pointOfRejoin->forAll
       (each | each.base <> self.inclusionUseCase)
```

[22] The commonly referenced base use case instance of the ExtensionPoint instances (see [17] above) and the commonly referenced base use case instance of the RealRejoinPoint instances (see [20] above) of a particular Include-relationship instance must be the same:

```
self.extensionPoint->first.base =
             self.PointOfRejoin->first.base
```

[23] The ExtensionPoint instances of a particular Include instance must be contained in the sequence of ExtensionPoint instances of the base use case instance of this Include instance:

```
self.extensionPoint->first.base.extensionPoint
   ->includesAll (self.extensionPoint)
```

[24] The RealRejoinPoint instances of a particular Extend instance must be contained in the sequence of PointOfRejoin instances of the base use case instance of this Extend instance:

```
self.pointOfRejoin->first.base.pointOfRejoin
   ->includesAll (self.pointOfRejoin)
```

Acknowledgements

We wish to express our gratitude to the following individuals who have helped a great deal to improve this work. These are Birgit Friedrich, Gonzalo Génova, and Oliver Nössler. Special thanks to Roderick Coleman, Alistair Cockburn and Tony Simons.

References

1. Armour F., Miller G. "Advanced Use Case Modeling", Addison-Wesley, 2001

2. Atkinson C., panel discussion on the architectural redesign of UML in v2.0, IEEE UML 2001 Conference, Toronto, Canada, October, 2001
3. Atkinson C., Kühne T. “The Essence of Multilevel Metamodelling”, Procs. of the 4th International Conference on the UML 2001, IEEE, Lecture Notes on Computer Science 2185, Springer Verlag, Germany, 2001
4. van der Berg K., Simons A. "Control-Flow Semantics of Use Cases in UML". Information and Software Technology, 41(10):651-659, July 1999.
5. Booch G., Rumbaugh J., Jacobson I., “The Unified Modeling Language User Guide”, Addison Wesley, Reading MA, 1999.
6. Cockburn A. "Writing Effective Use Cases", Addison-Wesley, 2001
7. Génova G., Llorens J., Quintana V. “Digging into Use Case Relationships“, Computer Science Department, Carlos III University of Madrid, 2002
8. Henderson-Sellers B. “Some Problems With The UML v1.3 Metamodel”, Procs. HICSS-34, IEEE Computer Society
9. Jacobson I. Christerson M., Jonsson P., Övergaard, G. “Object Oriented Software Engineering – A Use Case Driven Approach”, Addison-Wesley, 1992
10. Jacobson I. “The Road to the Unified Software Development Process”, Cambridge University Press, SIGS Reference Series, 2000
11. Kulak D., Guiney E. “Use Cases – Requirements In Context”, Addison-Wesley, ACM Press, 2000
12. Metz P., wibas Schulung und Beratung GmbH, “Requirements Engineering”, training course for software professionals, wibas Schulung und Beratung GmbH, Germany, 2001, http://www.wibas.de
13. Metz P., O´Brien J., Weber W. “Against Use Case Interleaving”, Procs. of the 4th International Conference on the UML 2001, IEEE, Lecture Notes on Computer Science 2185, Springer Verlag, Germany, 2001
14. Metz P., O´Brien J., Weber W. “Specifying Use Case Interaction: Types of Alternative Courses“, March, 2002, submitted for publication
15. OMG Unified Modeling Specification, Version 1.4, November 2000
16. Rumbaugh J., Jacobson I., Booch G. “The Unified Modeling Language Reference Manual”, Addison-Wesley, 1999
17. Simons A. "Use Cases Considered Harmful", Proceedings of TOOLS-29 Europe, IEEE, eds. R Mitchell, A C Wills, J Bosch and B Meyer, 1999, available on http://www.dcs.shef.ac.uk/~ajhs/abstracts.html#harmful
18. Simons A., Graham I. "30 Things That Go Wrong In Object Modelling With UML 1.3", Behavioral Specifications of Businesses and Systems, eds. H. Kilov, B. Rumpe, I. Simmonds, Kluwer Academic Publishers, 1999, available on http://www.dcs.shef.ac.uk/~ajhs/abstracts.html#uml30thg
19. Personal communication with J. Högström and M. Richters, co-authors of “Response to the UML 2.0 OCL RfP”, version 1.3, August, 2001, available on http://www.klasse.nl/ocl/index.html
20. Woolf B. “Null Object”, Pattern Languages of Program Design 3, eds. Robert C. Martin, Dirk Riehle, Frank Buschmann, Addison-Wesley, 1998