The Whole-Part Relationship in the Unified Modeling Language: A New Approach

Franck Barbier
University of Pau
France
Franck.Barbier@univ-pau.fr

Brian Henderson-Sellers
University of Technology, Sydney
Australia
brian@it.uts.edu.au

Andreas L. Opdahl
University of Bergen
Norway
andreas@ifi.uib.no

Martin Gogolla
University of Bremen
Germany
gogolla@informatik.uni-bremen.de

Abstract. This study of the semantics of the Whole-Part relationship in OO modeling is based upon, extends and, specifically, formalizes earlier analyses of the semantics of UML's Aggregation and Composition (white and black diamonds, also called shared aggregation and composite aggregation). Although UML is nowadays regarded as a standard and is widely used as an OO modeling language, the way the Whole-Part relationship is formalized is unsatisfactory. Here, we provide a rigorous specification of various forms of the Whole-Part relationship using OCL (Object Constraint Language). The first part of the specification is based on the differentiation between primary characteristics (applicable to *all* Whole-Part relationships) assigned to a new Whole-Part metatype in the UML, and secondary features, which are possessed by subtypes of this metatype and permit the representation of several "flavors" of the Whole-Part relationship. This UML-compliant style of specification, based on the use of OCL as well as metamodeling, allows us to directly incorporate our results into the UML metamodel, in particular revising UML's definition of Composition.
Keywords: object-oriented modeling, whole-part relationship, aggregation, composition.

INTRODUCTION

The Whole-Part (WP for short) relationship in OO modeling has been a subject of keen interest since, among others, its appearance in OMT (Rumbaugh, Blaha, Premerlani, Eddy & Lorensen, 1991). Named "aggregation" in OMT, this relationship is considered to be important for object modeling, although neither OMT nor its successor UML (OMG, 1997, 1999) provide a well-grounded semantics. As it happens, UML supports two inconsistent kinds of WP relationship despite the fact that, both inside and outside the world of OO software engineering, numerous high-quality contributions exist on this very old research theme. This chapter is aimed at rectifying the current ambiguities and confusion in UML's white and black diamonds (shared aggregation and composite aggregation, respectively) and thus hope to influence the next versions of the UML standard. We purposefully use a UML-compliant style of specification, i.e. the use of OCL (Warmer & Kleppe, 1998) which is part of UML, complementing the metamodel, which is the primary way in which the UML's semantics is currently described.

The second section of this chapter, named "Background", is a concise overview of the WP relationship in OO modeling. In the third section called "Foundation", we present and formalize a minimal set of characteristics for the WP relationship. In particular, we analyze this set according to three viewpoints: ontological, mathematical and software engineering-based considerations. In the fourth section named "Properties of the WP Relationship in OO modeling", we specify in OCL the key features (e.g. separability, existence dependency) for possible subtypes of the WP relationship. We finally conclude in the fifth and last section ("Conclusion and Future Trends") by showing how this work can be used to modify the next version of the UML.

BACKGROUND

A WP relationship is a binary relationship from a set called **Whole** to a set called **Part**. A tuple is then `(w, p)` with `w` being an instance of **Whole** and `p` an instance of **Part**. These two sets are not necessarily disjoint. For

instance, the **Programming statement** object type can be linked to itself by using a WP relationship. In OO modeling, a given object type **T** in a given object model **M** corresponds to a set of potential instances that, by definition, conform to **T**. **T** plays the role of either **Whole** or **Part** or both depending upon which **T** is involved in one or more WP relationships. Throughout this chapter, we deal with the generic terms **Whole** and **Part** in order to talk about "the" WP relationship as a metatype (called **Whole-Part** on the left side of Figure 1) from which "a" WP relationship in a specification model **M** is known to be its instance ("model level" on the right side of Figure 1). One major challenge of this chapter is then to supply robust criteria for distinguishing *without ambiguity* a WP relationship from an ordinary binary relationship (also called binary association in UML (OMG, 1999) or referential relationship in OML (Firesmith, Henderson-Sellers & Graham, 1997)) or, more accurately, to distinguish it from a *non*-WP relationship. In the UML 1.3 documents, it is written that: "An association may represent an aggregation, i.e. a whole/part relationship." and "Only binary associations may be aggregations." (OMG, 1999, pp.2-57). Therefore, in Figure 1, depicted using the UML's style of notation (white diamond or Aggregation for convenience to mean simply WP relationship), we show a WP relationship between **X** and **Y** as an instance of the **Whole-Part** metatype. As a result, a WP linkage (tuple) between **x** (an instance of **X**) and **y** (an instance of **Y**) is also an instance of the WP relationship between **X** and **Y**.

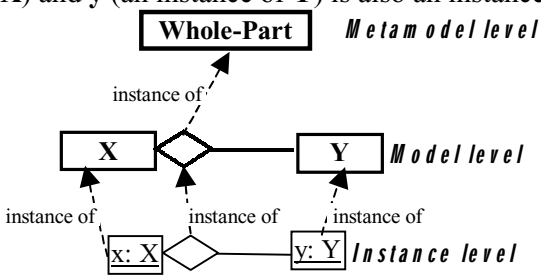

Figure 1. OO metamodeling for the WP relationship

Overview of the WP relationship in object modeling

Until now, the OO modeling domain has not really taken advantage of all the available knowledge related to the WP relationship. As a result, the semantics of the WP relationship remains inevitably fuzzy and convoluted in many object modeling languages, notably UML. In order to correct this, we focus our investigation on what constitutes an adequate subset of the WP characteristics for OO modeling. Such a definition has been sketched in (Henderson-Sellers & Barbier, 1999a) and applied to UML in (Henderson-Sellers & Barbier, 1999b, 1999c). In this chapter, the novelty is that we provide an OCL specification in addition to that initiated in (Gogolla & Richters, 1998), such that our research results become a candidate and credible proposition for the next versions of UML.

In order to formalize an appropriate subset of the WP problematics for OO modeling, concordant with the literature, we classify the properties generally assigned to the WP relationship. In (Saksena, Larrondo-Petrie, France & Evett, 1998; Henderson-Sellers & Barbier, 1999a), it is shown that some of these are recurrent and unanimously recognized as key properties, which are vital for *any* true WP relationship. We call them "primary" properties by arguing that no well-grounded definition of the WP relationship is acceptable if it does not possess these primary characteristics (Table 1, line 1). The rest form a set of "secondary" properties (Table 1, line 2) which permit specialized types of the WP relationship to be characterized. There are also a number of consequent properties (Table 1, line 3) that are, by definition, deduced from the primary and/or secondary properties.

Table 1. Primary and secondary characteristics (there are other secondary characteristics, not shown here, related to implementation)

Primary Characteristics	Whole-part, emergent property, resultant property, irreflexivity at instance level, antisymmetry at instance and type level; therefore asymmetry at instance level.

Secondary Characteristics	Encapsulation (visibility), overlapping lifetimes (9 cases), transitivity, shareability, configurationality, separability, mutability.
Consequent Properties	Propagation of operation(s), ownership, abstraction, existential dependency, propagation of destruction operation.

In metamodeling, we necessarily view primary properties as axiomatic characteristics of the WP relationship. As a result, primary properties are assigned to an abstract **Whole-Part** metatype which itself needs to be added to the current (version 1.3) UML metamodel. By use of inheritance, subtypes of the WP relationship can be derived from this metatype. Subtypes differ by the secondary properties that they possess. For example in Figure 2, one may say that the four WP relationships conform to the **Whole-Part** metatype while each is an instance of a distinct subtype determined by its secondary properties. For instance, **Room-Wall** is an instance of a subtype that possesses the secondary characteristic (Table 1) of shareability, i.e. the same wall can be a component of more than one room. This is clearly not true of the **Egg-Yolk** WP relationship. In the third example, canceling an order leads to the destruction of its order lines (coincident lifetimes) while breaking an egg does not result in destroying the yolk. So, parts sometimes outlive wholes, sometimes not. As a final example here, a given paragraph can be separated from a given section in order to dynamically become part of another section while it would be an error to do the same for a yolk, a wall or an order line because they participate in frozen structures.

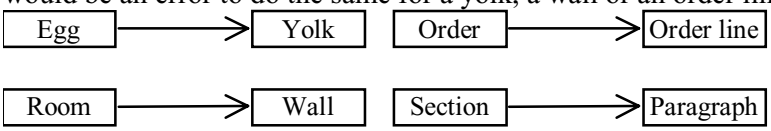

Figure 2. Four special sorts of the WP relationship: the arrow points to the **Part** type, thus replacing the semantically vague UML notation (white diamond) in Figure 1

Scope and Aim of the Study

The scope of our study on the WP relationship is confined to OO modeling with the underlying motivation of seeking high quality in the OO models that are thus built. In fact, OO analysts/designers use the WP relationship not only to precisely describe the requirements that their applications must meet, but they also rely on the WP relationship as a tool for arranging/organizing their models better. This especially introduces a higher level abstraction: the **Whole**, and allows the modeler to cope with various granularities. Indeed, these two facets have been noted previously: "These two goals are difficult to separate by looking at a finished model, as this not only attempts to reflect the structure of a problem domain, but is also "designed" to be understandable, manageable, reusable, resilient to change (…)" (Civello, 1993). In the same line of reasoning, Artale, Franconi, Guarino & Pazzi (1996) underline that the accepted and unified GEM (General Extensional Mereology) theory is "(…) inappropriate when considering real domains of application of the theory." They next state that any OO framework for the WP relationship will only be suitable provided that *reusability*, *understandability* and *extensibility* are direct consequences of the use of the WP relationship in the models that are built in this way. Thus, in a model such as in Figure 2, an OO modeler expects to find recurrent forms of analysis and design or modeling patterns and idioms whose implementation is similar from one application to another. For instance, detecting that **Section-Paragraph** is an occurrence of a special WP relationship that has separability as a secondary characteristic can benefit from acquired knowledge on a previously encountered modeling pattern. The high frequency of a property like separability in real-world situations may then require a dedicated notational element in an OO modeling language, for instance as proposed by Saksena, Larrondo-Petri, France & Evett (1998). A complete and consistent subtype of the **Whole-Part** metatype can embody a modeling pattern and, provided that it is implemented, be (re)used to code the structure of this pattern. So, understandability and reusability naturally increase, subject to the constraint that only a small number of subtypes of the WP

relationship are separately identified and rigorously specified. As for extensibility , any subtype may itself be extended/restricted by inheritance.
Odell (1994) and Henderson-Sellers (1997) have both echoed the reputable work of Winston, Chaffin & Herrmann (1987) and, moreover, showed the context in which the WP relationship shall be used for OO modeling. Saksena, France & Larrondo-Petrie (1998) proposed a necessary and sufficient set of properties for the WP relationship in OO modeling. This set of papers has been reviewed and even completed by Henderson-Sellers & Barbier (1999a) by using earlier discussions on conceptual/formal OO modeling, including (Kilov & Ross, 1994; Dong & Duke, 1995; Lano, 1995; Firesmith & Henderson-Sellers, 1998). Concurrently, implementation of models was also discussed by Kolp & Pirotte (1997) and Barbier (1998). Regarding the continuing lack of solution in the latest UML reviews (Booch, Rumbaugh & Jacobson, 1999; Rumbaugh, Jacobson & Booch, 1999), this chapter attempts to bring to a closure the continuing debate on WP relationships and the notions of "aggregation".

Formalism

The specification of candidate properties for the **Whole-Part** metatype on the left side of Figure 1 is done in this chapter using the latest version of OCL (OMG, 1999). OCL is named by its authors a "lightweight" specification language. It can be used to write logical expressions on types in models but, as we do in this chapter, on metatypes in metamodels too. However, as observed by Henderson-Sellers & Barbier (1999b, 1999c), the current UML metamodel fragment relating to the WP relationship is defined by means of **AssociationEnd** and its metaattributes. In fact, there is currently no distinct metatype that embodies the WP relationship. Another detrimental factor here is the lack of OCL constraints on the "aggregation" and "changeability" attributes of the **AssociationEnd** metaclass. Indeed, we have demonstrated in (Barbier & Henderson-Sellers, 1999; Barbier & Henderson-Sellers, 2000) that such a situation allows the creation and use of inconsistent forms of the WP relationship. Next, writing OCL is not sufficient in itself because the **Whole-Part** metatype does not exist at this time in the UML metamodel, being simulated via attributes on **AssociationEnd** (as noted earlier). Therefore, we show in this chapter all the benefits related to the possible introduction into the UML of a WP relationship metatype – perhaps along the lines suggested in Figure 3. That is the reason why we need a modified version of the UML metamodel fragment relating to the WP relationship as a basis for our proposed revision. Moreover, we provide the **Whole** and **Part** types in Figure 3 with specific OCL capabilities (see the section called "Axioms") in order to underpin this proposition. Indeed, **Whole** and **Part** both inherit from **Classifier** (Figure 3). As an existing metatype in the UML metamodel, **Classifier** allows us to use **Whole** and **Part** in OCL expressions (OMG, 1999, pp.7-3).

OCL

An OCL expression (see appendix for more details) begins with the word `context` followed by the name of a type, **T** for example, as well as an invariant symbolized by `inv`. This symbol is optionally followed by an invariant name, as well as an invariant content (mandatory). The rest of the expression (invariant content) then describes an anonymous instance of **T**. Formulae are based on properties borne by **T**. Traditional logical operators (`forAll` which is equivalent to ∀, `exists` which is equivalent to ∃, etc.) are used as well as other constructs like those related to object collections (`iterate`, `select`…), those based on typing (`oclIsTypeOf`, `oclIsKindOf`…) and so on. Here is a sample:

```
context Egg inv An invariant related to Figure 2:

     yolk→size = 1

-- "self.yolk→size = 1" is an equivalent expression

-- "yolk" is a navigation and yields a singleton set for any

-- instance of Egg via the "→" operator which acts on sets
```

Another possible expression archetype in OCL refers to pre-conditions and post-conditions for user-defined operations on types. This applies only for observer functions, i.e. those which are free from side effects. Here is an illustration:

```
context Egg::freshness() : Boolean

    post: result = yolk.freshness() and white.freshness()

-- We assume here the existence of another WP relationship, i.e. Egg-White
```

The great peculiarity of OCL is that it basically relies on textual material. In particular, this generates a suitable object-oriented "flavor", close to OO programming languages, in order to write more comprehensive formal specification expressions.

Starting Basis

Regarding a common WP relationship as in the middle of Figure 1 or as may be drawn in a model (Figure 2), we introduce the following invariant:

```
context Whole-Part inv:

    WPWhole.instance→forAll(w | w.part→oclIsTypeOf(Set(Part)) = true)
```

This invariant means that for any instance of the **Whole** type in the context of a given WP relationship (i.e. a given instance of the **Whole-Part** metatype), we get the set of **Part** instances linked to this instance of **Whole** in using the `part` navigation.
We also make the assumption of the existence of navigation in the opposite direction in order to obtain another useful invariant:

```
context Whole-Part inv:

    WPPart.instance→forAll(p | p.whole→oclIsTypeOf(Set(Whole)) = true)
```

Nevertheless, **Whole** and **Part** are just roles played by types in models. So, a given type **T** in a given model **M** may act as a **Whole** in one WP relationship, a **Part** in another one, a **Part** in a third one and so on.

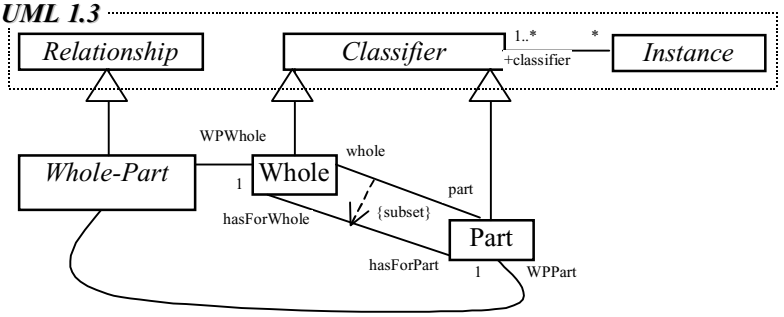

Figure 3. A revised version of the UML metamodel fragment related to the WP relationship – a possible contribution to UML version 2.0

Axioms

We need three axioms in order to build our specification. The first one, called *Snapshot*, is the consistent relation between the `part` and the `whole` navigation directions:

```
context Whole-Part inv Snapshot:

    WPWhole.instance→forAll(w | w.part→forAll(p | p.whole→includes(w)))

    WPPart.instance→forAll(p | p.whole→forAll(w | w.part→includes(p)))
```

The second axiom named *History* introduces the `hasForPart` and `hasForWhole` navigation directions as well as their mutually consistent relation. As was the case for the `part` and `whole` navigation directions, we

axiomatically view `hasForPart` and `hasForWhole` as predefined navigation directions assigned to the **Whole** and **Part** types, respectively, with the following restrictions:

```
context Whole-Part inv History:

    WPWhole.instance→forAll(w |

        w.hasForPart→forAll(p | p.hasForWhole→includes(w)))

    WPPart.instance→forAll(p |

        p.hasForWhole→forAll(w | w.hasForPart→includes(p)))
```

Finally, the relation between the two kinds of navigation is required:

```
context Whole-Part inv Inclusion:

    -- {subset} constraint in Figure 3

    WPWhole.instance→forAll(w | w.hasForPart→includesAll(w.part))

    WPPart.instance→forAll(p | p.hasForWhole→includesAll(p.whole))
```

This third and last axiom shows that, at any time, the parts of a given whole make up a set which is by definition included in the set of all the parts linked to this whole during its life cycle. In other words, the evaluation result of the `w.part` expression depends upon the moment the evaluation occurs. However, computing `w.hasForPart` is free from temporal concerns: it yields all the part objects connected with `w`, even those separated from `w` before its death. Note that in OCL, the `c.instance` expression in which `c` is an instance of **Classifier** is based on the existing metarelationship between **Classifier** and **Instance** in the UML metamodel (Figure 3). This yields the set of all instances of `c`. In contrast, the predefined `allInstances` operator in OCL is only directly applicable to types and not to instances such as `c`.

FOUNDATION

We may identify three facets of the WP relationship, as represented in the literature. The first one is the ontological point of view. The second one comes from mathematics while the third one adds software engineering-based constraining features to the WP relationship. It seems reasonable to consider that all the listed features discussed hereafter constitute the set of primary properties for the WP relationship in OO modeling. We simply try to gather together a minimal set of characteristics which does not contradict the numerous opinions found in the literature. Moreover, we reuse some of the results coming from (Henderson-Sellers & Barbier, 1999a) in which the choice of a set of primary characteristics is argued. In (Henderson-Sellers & Barbier, 1999a), we outline the existing widespread confusion in the semantics of the WP relationship. Indeed, opinions diverge widely on the true nature of the WP relationship due to a misuse of terminology as well as the use of secondary properties (rather than primary properties), as discussed in the section called "Properties of the WP Relationship in OO Modeling".

Ontology

In his paper, Varzi (1996) states the historical foundation of what he calls the theory of Parthood or Mereology as well as the theory of Wholeness or Topology. A very wide range of disciplines has contributed to create the existing knowledge base on the WP relationship, including ontology/philosophy, linguistics, mathematics and cognitive science. Opdahl, Henderson-Sellers & Barbier (2001) re-analyze the impact of ontology and apply it to the OO framework elaborated in (Henderson-Sellers & Barbier, 1999a).

In ontology, the two most reputable features of the WP relationship are *emergent* and *resultant*. In the case of an egg for instance, its freshness is emergent in the sense that it depends upon the laying date while its taste is resultant because it directly depends upon its yolk and its white.

Expressing these notions in a rigorous manner leads to the following:

Emergent: for several WP relationships owning the same **Whole** type, this type offers *at least one* emergent property. In other words, for any instance of the **Whole** type, the value of this property is not computed in any way from the properties of the **Part** types.

Resultant: for several WP relationships owning the same **Whole** type, this type offers *at least one* resultant property. In other words, for any instance of the **Whole** type, the value of this property is necessarily computed from some of the properties of the **Part** types.
It is important to note that *emergent* and *resultant* are not defined according to a single WP relationship but to a bundle of WP relationships. For example, we must consider two WP relationships (i.e. **Egg-Yolk** and **Egg-White**) in order to establish the taste of an egg. Hence, a bundle of WP relationships is such that the **Whole** type is the same for each WP relationship in the bundle. Formally, we then have:

```
context Whole-Part inv:

    let bundle = Whole-Part.allInstances→select(wp |
        self.WPWhole = wp.WPWhole) in …
```

Next, within OCL, by definition, the attributes plus the operations plus the structural relationships possessed by a type make up its properties. We can then complete the last OCL expression as follows:

```
context Whole-Part inv Emergent:

    let bundle = Whole-Part.allInstances→select(wp |
        self.WPWhole = wp.WPWhole) in

    bundle→forAll(wp |
        (self.WPWhole.attributes→union(self.WPWhole.operations→union(self.W
        PWhole.associationEnds)))→exists(e |
        not
        (wp.WPPart.attributes→union(wp.WPPart.operations→(wp.WPPart.associa
        tionEnds)))→includes(e)))
```

Due to the current limitations of OCL, note that we just provide a syntactical definition of *emergent*. Indeed, we only ensure that a given property (at least one) of the **Whole** type does not match any property among the sum of properties for each **Part** type in the bundle. A semantic definition must in fact verify that the *value* of any emergent property is not computed by means of properties of parts. The reflection capabilities of OCL do not allow the expression of such a semantic constraint at this time.
Finally, we provide a similar syntactical characterization of *resultant*:

```
context Whole-Part inv Resultant:

    let bundle = Whole-Part.allInstances→select(wp |
        self.WPWhole = wp.WPWhole) in

    bundle→forAll(wp |
        (self.WPWhole.attributes→union(self.WPWhole.operations→union(self.W
        PWhole.associationEnds)))→exists(r |
        (wp.WPPart.attributes→union(wp.WPPart.operations→(wp.WPPart.associa
        tionEnds)))→includes(r)))
```

Mathematics

An agreed characterization of the WP relationship is based on the binary nature of the WP relationship. In an equivalent style, one may also use predicates as in (Bourdeau & Cheng, 1995) so that the signature of the WP relationship is: WP: Whole, Part → Boolean. In fact, this mathematical foundation must remain as simple as possible, as occurs for relational theory. In this respect, we take up the pseudo-binary approach from Kent (1990). This approach may be illustrated by the fact that a synergetic relation between `n` sorts of thing in the real world (e.g. **Chassis**, **Engine**, **Wheel**…) can be modeled by `n` binary relationships from each sort to a type which acts as a controller. In the previous example, **Car** is obviously the controller type.

We may note that in (Kilov & Ross, 1994), although a binary approach is chosen, the flavor of the WP relationship proposed and called "composition" is formalized in quite a different way, i.e. a binary relationship from the set **Car** to the set corresponding to the Cartesian product of three types: **Chassis**, **Engine** and **Wheel**. In such an approach, a tuple is then as follows: (aCar, (aChassis, anEngine, aWheel)). In (Barbier & Henderson-Sellers, 2000), this approach is recognized as problematic at implementation time and, more generally, does not seem natural and direct for OO modelers.
Therefore, we choose a more comprehensive approach in first analyzing the mathematical properties of the WP relationship between instances of **Whole** and **Part**. Next, at the type level, we raise some problems related to software engineering concerns. Thus, at the instance level, we notice that the WP relationship is asymmetric (asymmetry ⇔ irreflexivity ∧ antisymmetry). We clearly reject transitivity as a primary characteristic since it has been clearly shown on many occasions that only very specific cases of the WP relationship are transitive (Markowitz, Nutter & Evens, 1992; Motschnig-Pitrik, 1996), despite that fact that transitivity has been considered as axiomatic in Mereology theory – see also discussion in (Opdahl, Henderson-Sellers & Barbier, 2001). For irreflexivity, we have:

```
context Whole-Part inv Irreflexivity:
    WPWhole.instance→forAll(w |
        w.oclIsKindOf(Part) implies not w.part→includes(w))
    WPPart.instance→forAll(p |
        p.oclIsKindOf(Whole) implies not p.whole→includes(p))
```

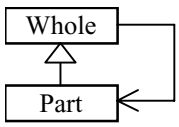

Figure 4. **Part** inherits from **Whole**

This means that if **Part** directly (Figure 4) or indirectly derives from **Whole**, **Part** is associated with itself through a WP relationship, i.e. **Part** inherits this WP relationship from **Whole**. Although this case is rare, it is one of the cases in which non-irreflexivity may occur. Thus, in the OCL constraint above, if the hypothesis is true (i.e. `p.oclIsKindOf(Whole)`) then one ensures that `p` is never linked to itself. We do the same (other possible case) whenever **Whole** inherits from **Part**. An illustration for the template in Figure 4 is **Car door-Car part** whereas **Car door** inherits from **Car part**. We then simply ensure that:

```
Car door.instance→forAll(cd | cd.oclIsKindOf(Car part) implies not
cd.part→includes(cd))
```

Another typical example is **Tree branch-Tree branch** which leads to:

```
Tree branch.instance→forAll(tb | not tb.whole→includes(tb))
```

From the *Snapshot* axiom, we easily deduce that:

```
Tree branch.instance→forAll(tb | not tb.part→includes(tb))
```

For antisymmetry, we have:

```
context Whole-Part inv Antisymmetry at the instance level:
    WPWhole.instance→forAll(w |
        w.oclAsType(Part).whole→forAll(x |
            w.part→includes(x) implies w = x))
    WPPart.instance→forAll(p |
        p.oclAsType(Whole).part→forAll(x |
            p.whole→includes(x) implies p = x))
```

In this OCL constraint, `oclAsType` is a partial function that returns an object whose type is the argument, or is otherwise `Undefined` in the case of non-conformance. Therefore `p.oclAsType(Whole)` transforms `p` as an instance of **Whole** if and only if `p.oclIsKindOf(Whole)` is true. This allows us to deal with tricky situations as in Figure 5 where non-irreflexivity and/or non-antisymmetry might occur.

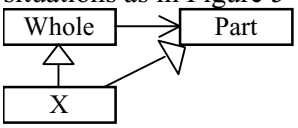

Figure 5. Special possible case of non-irreflexivity and/or non-antisymmetry in which **X** inherits from **Whole** and from **Part**

An example for the template in Figure 5 may be the **State union-State** WP relationship. Therefore **X** may be **Federal state** with USA as well as Switzerland being two instances of $\mathbf{X}^{i}$. So, as **Whole**, `USA.oclAsType(State).whole→forAll(x | USA.part→includes(x)` **`implies`** `USA = x)` is true because `USA.oclAsType(State).whole→isEmpty` is true. As **Part**, `USA.oclAsType(State union).part→forAll(x | USA.whole→includes(x)` **`implies`** `USA = x)` is true because once again `USA.whole→isEmpty` is true. A more or less utopian example is that whether in the future Switzerland becomes part of the USA. The specification above precludes then that USA will not become at the same time part of Switzerland. This corresponds to asymmetry, i.e. if WP(USA, Switzerland) is true, WP(Switzerland, USA) is false.

Software Engineering

Despite some authors such as Halper, Geller & Perl (1998) who prefer to stress the part-whole relationship, most authors consider the dominant position of the **Whole** type compared to the **Part** type (see use of arrows in Figure 2). This is especially true and obvious in the OO field because one accurately wants the **Whole** type to play an extra role. Indeed, it limits or prevents any interaction between *anonymous* clients of **Part** instances. In contrast, clients of these **Part** objects request the whole(s) with which they are connected. Thus one delegates/propagates the request and acts as a proxy for its **Part** objects (Lewandowski, 1998). In OMT, this property is defined as follows: "*Propagation* (also called *triggering*) is the automatic application of an operation to a network of objects when the operation is applied to some starting object (…) The propagation behavior is bound to an association (or aggregation), direction, and operation." (Rumbaugh, Blaha, Premerlani, Eddy & Lorensen, 1991, p.60). This last sentence gives us to understand that the propagation of operations is not a discriminator for the aggregation due to it also being related to an association.

In UML 1.3, this property of propagation is more strongly connected with Composition: "(…) a composite implies propagation semantics (i.e. some of the dynamic semantics of the whole is propagated to its parts). For example, if the whole is copied or destroyed, then so are the parts as well (because a part may belong to at most one composite)." (OMG, 1999, pp.2-57). More generally, in component based client/server computing (Lewandowski, 1998), one customarily views the WP relationship as offering strict support for the delegation of operations. In other words, a **Whole** object must possess an interface so that potential requesting clients are able to call services available on its **Part** objects. All client requests are treated by the whole and forwarded (propagated) to its parts.

In addition, **Part** objects coming from different types can interact intensively (Bock & Odell, 1998a, 1998b). For instance, **Geographical Site, Business Unit, Department, Employee** and so on may be all declared as part of **Company**. While they, together, develop an intra-communication, **Company** is responsible for offering and managing a complete interface for any service demand that originates from outside.

From a design viewpoint, one intuitively seeks to use the WP relationship as a mean for isolating clear, reasonably independent and robust modules – of increasing importance for the evolving component industry. This principle is also known as that of high cohesion. Furthermore, one looks for low coupling in particular and modularity in general. In fact, based on these software quality considerations, the WP relationship is not only a

semantic relationship as it is sometimes treated in the ER approach or in OODBMS (Kim, Banerjee, Chou, Garza & Woelk, 1987) but it also has architectural and behavioral consequences.
This software engineering influence has led us to specific secondary properties, for example, configurationality in OML (Henderson-Sellers, 1997). Configurationality occurs when parts not only relate to the whole but also to each other – either functionally or structurally. It aims to represent the synergetic relation between complementary (in terms of supplied functions) parts of the same whole and is fairly typical of the OO modeling area. More generally, we thus focus on the restricted directionality of the WP relationship, as it happens from **Whole** to **Part**, in order to use the WP relationship in a special way. In this respect, Henderson-Sellers & Barbier (1999a) promote antisymmetry at the type level as far as the WP relationship is a coupling factor. This first-class rule simply avoids the building of tortuous models, which are, first, difficult to understand and, next, not easily implementable. So, we have:

```
context wp1,wp2 : Whole-Part inv Antisymmetry at the type level:
     wp1.WPWhole = wp2.WPPart and wp1.WPPart = wp2.WPWhole implies wp1= wp2
```

PROPERTIES OF THE WP RELATIONSHIP IN OO MODELING

We highlight in this section, properties of the WP relationship immediately useful in object modeling. The relative importance of these properties has already been discussed by Winston, Chaffin & Herrmann (1987) and re-studied and extended for operational concerns in the OO field by Barbier & Henderson-Sellers (2000). We here offer a more concise framework in OCL.

Abstraction/Encapsulation/Visibility

Tightly linked in the field of object technology, these three keywords (abstraction, encapsulation and visibility) are often recognized as important characteristics for differentiating a non-WP relationship from a WP relationship (Civello, 1993; Henderson-Sellers & Barbier, 1999a). Roughly speaking, a **Whole** object may sometimes be considered as a juxtaposition of other objects. Called its parts, these last objects are agglomerated together in order to form an object which has no strict mapping in the real world but can play the role of a proxy for its parts.
For instance, an ATM is made up a user interface kit, a card reader, a cash dispenser, a deposit drawer and a receipt printer without being anything else (physically) than the sum of five part objects. Nevertheless, all banking transactions are checked and processed through the ATM, which corresponds to an abstraction of its five linked devices. Thus, the bank requires that the ATM accomplishes the effective delivery of money in ignoring its functional dependency with the cash dispenser. This neglect is valid in object modeling because it permits the construction of isolated and robust packages. In an object model, such an absence of visibility of the **Bank** type on the **Cash dispenser** type can possibly be depicted by means of a WP relationship from **ATM** to **Cash dispenser**.
Beyond this example, we basically want to state that the abstraction/encapsulation/visibility trio can or cannot be used as a first-class criterion to characterize the WP relationship. Because encapsulation is a foundation of object orientation, its impact and reciprocal influence on the WP relationship is high. We may see the WP relationship as a tool for encapsulation. We may then use it on purpose to avoid any side effect resulting from the absence of protection in the models thus built. On the other hand, we may observe that the WP relationship induces some encapsulation differences in the models based on whether it is used or not. As a result, the use of the WP relationship may occur in conjunction with very specific concerns, as for example, security preoccupations in (Fernandez, Gudes & Song, 1994).
More generally, Cook & Daniels (1994, pp.38-39) deal with this same idea as follows: "Encapsulation. The idea here is that the aggregate encapsulates its parts in some way. (…) the abstraction provided by encapsulation is a vital part of object theory. The idea that one object is composed of others, and that the components are not known to clients of the whole, is a powerful structuring principle in object technology." Because of the wide range of interpretations and uses of encapsulation regarding the WP relationship, we consider that the abstraction/encapsulation/visibility trio must remain a secondary property. This is confirmed by other authors

who also reject it as a primary characteristic or axiom. This is the case for instance for Cook & Daniels (1994) despite their observation above.

Shareability

Shareability is, in essence, distinct from sharing in the sense that it defines the notion of a potential sharing rather than mandated actual sharing (as in the current UML Aggregation or white diamond). It is always studied for a **Whole** object which may possibly share the same **Part** object with another **Whole** object. An archetype is the **Scrabble suggestion-Letter** WP relationship in which a **Letter** object may be linked to two **Scrabble suggestion** objects. On the other hand, that a **Whole** object is linked to more than one **Part** object is a very common situation and does not lead to any specific analysis.

Most of the papers in the literature discuss this property but few take into account its true dimension and, as a result, its significant and direct impact on the WP relationship characterization. Indeed, the use of this notion in conjunction with other candidate characteristics is a source of conflict and inconsistency, as for instance for the white diamond in UML (Henderson-Sellers & Barbier, 1999b, 1999c). In this respect, we take advantage of the research results in (Kolp & Pirotte, 1997; Gogolla & Richters, 1998) as well as in (Halper, Geller & Perl, 1998) which carefully deal with shareability in association with exclusiveness.

Kolp & Pirotte (1997) talk about *local/global sharing* as well as *local/global exclusiveness*. Local exclusiveness is the contrary of local sharing – also called "class-exclusive" by Halper, Geller & Perl (1998). Local exclusiveness means that, for a WP relationship, the maximum cardinality near the **Whole** type is less or equal to 1, in other words:

```
context Whole-Part inv Local exclusiveness:

    WPPart.instance→forAll(p | p.whole→size <= 1)
```

We can then state that if this predicate holds for a WP relationship, it is said to own the local exclusiveness property much like one interpretation of the black diamond in UML (OMG, 1997). Global exclusiveness, which is the contrary of global sharing and is also called "conceptual sharing" by Saksena, France & Larrondo-Petrie (1998) or "strong form of forbidden sharing" by Gogolla & Richters (1998), is related to the fact that a **Part** type can be involved in more than one WP relationship in a model **M** (Figure 6). In OCL, this is specified by:

```
context wp1,wp2 : Whole-Part inv Global exclusiveness:

    wp1.WPPart = wp2.WPPart implies

    wp1.WPPart.instance→forAll(p |

        wp1.WPWhole.instance→exists(w | w.part→includes(p)) implies

        wp2.WPWhole.instance→forAll(w | not w.part→includes(p)))
```

An archetype that illustrates global exclusiveness is the coexistence in a model **M** of the **Car-Engine** and **Truck-Engine** WP relationships. The same engine object cannot be at the same time part of a car instance and a truck instance. In contrast in Figure 6, global sharing holds since the same article (assuming articles are conceptual not physical entities) might be shared by a journal and a compilation (example appearing in (Kolp & Pirotte, 1997)).

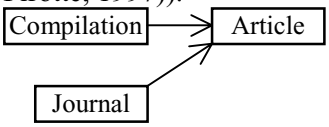

Figure 6. Global sharing

We may also note that a WP relationship can, at the same time, possess the local exclusiveness as well as the global exclusiveness properties. For instance, a chassis cannot be part of a car and a truck at the same time (global exclusiveness) neither can it be part of two distinct cars or two distinct trucks at the same time (local exclusiveness). Thus, we suppose in the rest of the chapter that unsharing is equivalent to local exclusiveness plus global exclusiveness and that shareability is the opposite of unsharing. Regarding our definition, we may

nevertheless observe that some other characterizations are clearly different. Indeed, Civello (1993, p.385) writes "*Sharing*. An object is shared if two or more objects hold references to it." Unlike Civello, most authors *implicitly* analyze shareability within the scope of the WP relationship. In fact, must we extend the scope of the study for non-WP relationships? Indeed, a **Part** object can be referenced by its whole(s) as well as other objects resulting from the existence of ordinary associations involving the **Part** type in a model **M**. This problem is discussed further in the section called "Lifetime Dependency".

Separability/Mutability

Separability (similar to replaceability in (Lano, 1995) or to variance in (Odell, 1994)) is viewed as a key notion in the classification proposed by Winston, Chaffin & Herrmann (1987). By definition, in an ordinary binary association, the instances of the two types involved in this association are dynamically linked and separated. In other words, this association is not a frozen structure in the sense that it may evolve. In this line of reasoning, some kinds of WP relationship exhibit separability while some do not. For instance, a **Sailboat-Sail** WP relationship owns separability because we obviously need to be able to replace a ripped sail of sailboat by another one. In contrast, a **Book copy-Sheet** WP relationship is such that new manufactured sheets cannot be substituted for damaged sheets. More precisely, a copy of, say, "OPEN Modeling Language (OML) Reference Manual" by Firesmith, Henderson-Sellers & Graham is no longer scientifically *the same book* if some pages are missing. Could we nevertheless say that this book copy minus some pages has lost its identity? – a topic which needs further analysis.

This raises the problem of mutability, whereby we have to pay attention to when a **Whole** object continues or not to be the same object if any change occurs regarding its linked **Part** objects. As an illustration, Smalltalk uses the `becomes:` method to manage the mutation of objects. Returning to modeling, a WP relationship that supports inseparability does not prevent the growth of the number of the parts of a whole (e.g. **Human egg-Cell**). We thus consider that the most interesting formal property is immutability which then permits us to characterize a very commonly occuring type of WP relationship, called **Component-Integral Object** by Winston, Chaffin & Herrmann (1987). We then specify immutability as follows:

```
context Whole-Part inv Immutability (first part):

    WPPart.instance→forAll(p | p.hasForWhole = p.whole)
```

Starting from this OCL expression expressing immutability, we easily prove that:

```
context Whole-Part inv Immutability (second part):

    WPWhole.instance→forAll(w | w.hasForPart = w.part)
```

Indeed, let us suppose the following assumption:

```
WPWhole.instance→exists(w |

    WPPart.instance→exists(x |

        w.hasForPart→includes(x) and not w.part→includes(x)))
```

From the *History* axiom in the section named "Axioms", we have:

```
WPWhole.instance→exists(w |

    WPPart.instance→exists(x |

        w.hasForPart→includes(x) implies x.hasForWhole→includes(w)))
```

From the *Immutability (first part)* assertion above, we have:

```
WPWhole.instance→exists(w |

    WPPart.instance→exists(x |

        x.hasForWhole→includes(w) implies x.whole→includes(w)))
```

Finally, from the *Snapshot* axiom in the section called "Axioms", we have:

```
WPWhole.instance→exists(w |
    WPPart.instance→exists(x |
        x.whole→includes(w) implies w.part→includes(x)))
```

The contradiction between `w.part→includes(x)` and our assumption above, i.e. **not** `w.part→includes(x)`, demonstrates that this assumption is unrealistic. Thus, immutability states that the set of **Part** objects linked to any **Whole** object is the same at any time, and vice versa. Moreover, all the **Part** objects bound to a **Whole** object remain the same from its birth until its death. Referring once again to the **Book copy-Sheet** example, we can now assume it to be immutable because destroying some pages leads not to a different book but to a thing which cannot be named a book.
In order to develop our investigation further, we nevertheless note that nothing in our specification prevents the birth of all or some of the parts of a whole to precede the birth of this whole. This is the same for the wholes of a part.

Existential Dependency

The use of existential dependency may lead to confusion since there is a potential overlap between this concept and the lifetime dependency characteristic of the WP relationship (Motschnig-Pitrik & Kaasboll, 1999). While most authors deliberately blend the notion of existential dependency with that of lifetime dependency (e.g. Halper, Geller & Perl, 1998; Motschnig-Pitrik & Kaasboll, 1999), it is, in fact, possible to make a clear distinction. This is inspired by the specification proposed by Gogolla & Richters (1998) and called essentiality by Motschnig-Pitrik & Kaasboll (1999). Therefore, the existential dependency for the part is defined in a precise manner as follows:

```
context Whole-Part inv Existential dependency for the part:
    WPPart.instance→forAll(p |
        WPWhole.instance→exists(w | w.part→includes(p)))
```

A typical example is the **Order-Order line** WP relationship in which an **Order line** object is meaningless without a pre-existing link to an **Order** object. This case is even more restrictive in the sense that an order line belongs to one and only one order. In fact, it owns the local exclusiveness plus the global exclusiveness features.
A symmetrical OCL expression can be easily introduced to describe the existential dependency for the whole:

```
context Whole-Part inv Existential dependency for the whole:
    WPWhole.instance→forAll(w |
        WPPart.instance→exists(p | p.whole→includes(w)))
```

We can, however, simply note that the existential dependency for the whole is equivalent to a cardinality which is greater or equal to 1 on the part role of the WP relationship shown in Figure 7.

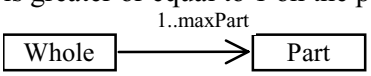

Figure 7. A **Whole** object cannot live without at least one **Part** object

Furthermore, if we want to ensure that any **Part** object can only exist with at least one connection to a **Whole** object, i.e. the existential dependency for the part, we then redraw Figure 7 as shown in Figure 8.

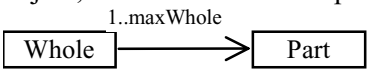

Figure 8. A **Part** object cannot live without at least one **Whole** object

It is essential to observe that these two characterizations are not peculiar to the WP relationship but can also be applicable to a regular association. This is confirmed by Fowler & Scott (2000) who reject this as a discriminator for the WP relationship. They indeed consider that a "1..1" cardinality is indicative of some

lifetime dependency. Moreover, Kilov & Ross (1994, pp.90-91) define "mandatory participation" as: "(…) the existence of an instance of the entity implies the existence of a corresponding instance of the association (…) The invariant for an entity with mandatory participation in an association implies that both the entity instance and the first corresponding association instance must be created in the same business process." Finally in (Henderson-Sellers & Barbier, 1999a), it is rigorously shown that existential dependency cannot be part of an axiomatic definition of the WP relationship due to it being inferred from other dominant primary features.
We thus propose that the existential dependency for the part according to Gogolla & Richters (1998) is equivalent to the mandatory participation for the part according to Kilov & Ross (1994). The same equivalence symmetrically applies for the existential dependency for whole. However, a deeper analysis of the idea of existence dependency is done in (Snoeck & Dedene, 1998). From the discussion in (Snoeck & Dedene, 1998), existence dependency is seen to be strictly correlated with life span constraints – we thus now move to consideration of this important characteristic of the WP relationship.

Lifetime Dependency

Snoeck & Dedene (1998) supply a definition of existence dependency. This definition is such that some kinds of WP relationship possess this property while others do not. The generic form of this definition is: "if each object of a class A always refers to minimum one, maximum one and always the same occurrence of class B, then A is existence dependent of B." (Snoeck & Dedene, 1998, p.234). Let us consider **A** as **Part** and **B** as **Whole**. It becomes then clear that this definition implies what is called existential dependency for the part in the previous section. Nevertheless, it is more restrictive in the sense that the cardinality in Figure 8 should be "1..1" instead of "1..maxWhole" and immutability applies. Indeed, the definition says that we cannot substitute a distinct instance of **B** for the currently referenced instance of **B**. Furthermore, Snoeck & Dedene conclude that the life span of an existence-dependent object is contained within the life span of the object upon which it depends. That is the reason why, as for many others authors, Snoeck & Dedene provide a very constrained definition that is based on lifetime dependency.
Obviously, Snoeck & Dedene restrict the definition of existential dependency for the part from (Gogolla & Richters, 1998) in first assigning the local exclusiveness value ("maximum one"). They also consider the unchangeability of the whole, which, by definition, always exists and is unique. This leads to the following necessary but non-sufficient expression (see from Figure 10 below for the completion of the characterization):

```
context Whole-Part inv Part is existence-dependent on Whole:
    -- Existential dependency for the part
    -- Local exclusiveness
    -- Global exclusiveness: omission in (Snoeck & Dedene, 1998)?
    -- Immutability (first part)
    -- Immutability (second part): inferred from Immutability (first part)
```

It is important to notice that their definition does not reject the global sharing value to be set to true for a WP relationship. Indeed, they say nothing about the fact that **A** may possibly be involved (as the **Part** type) in another WP relationship. In Figure 9, this seems realistic and coherent in the sense that an ill cell can be shared both by a brain and a tumor while its life span is obviously contained within the life cycle of its two composites.

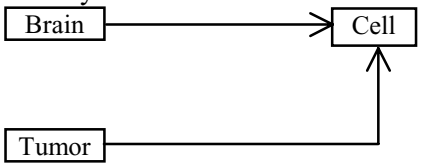

Figure 9. Global sharing plus *Part is existence-dependent on Whole*

This encourages us to scan all the possible dependency configurations between the birth and the death of a **Part** object according to those of a **Whole** object. In Figure 10, we sum up the cases in which the life cycle of the

part is included (not necessarily strictly) within the life cycle of the whole (top of figure) as well as the remaining cases (bottom of figure).

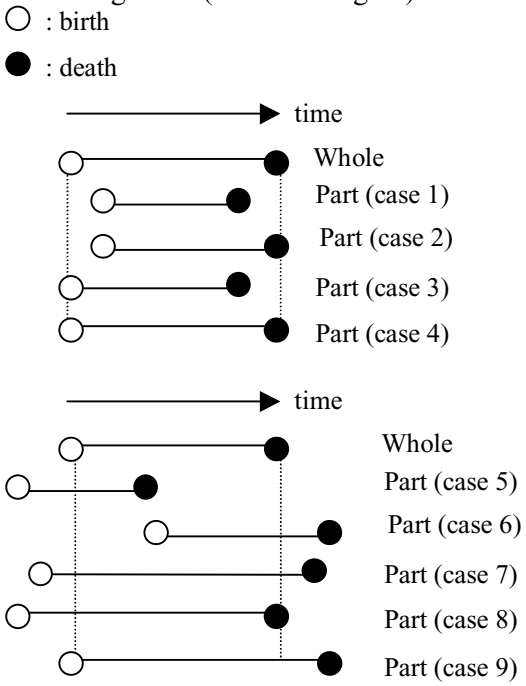

Figure 10. The nine cases of lifetime dependency

We observe that cases 1, 2 and 6 share the antecedent of the whole birth. This corresponds to the fact that any instance of **Part** is such that the commencement of its life coincides with or follows the life beginning of *any* instance of **Whole** linked to it (scenario in Figure 11). This is specified as follows:

```
context Whole-Part.WPWhole::creation() : Whole

    post: result.hasForPart→forAll(p | p.oclIsNew or p@pre.oclUndefined)
    -- this means that any part of "result" is created within its own
    -- creation process or does not exist at the start of the operation
    -- (use of the "@pre" operator)
```

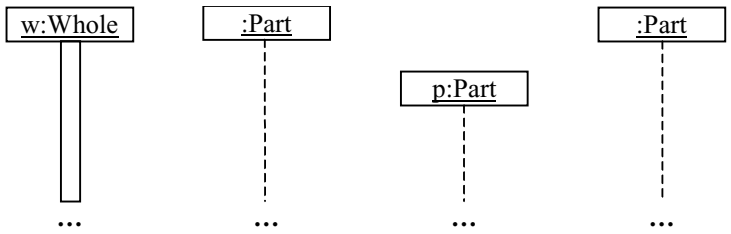

Figure 11. The ellipses mean that the life goes on, i.e. no special assumption is made on the object's death

Compared to cases 1, 2 and 6, cases 3, 4 and 9 (Figure 10) share the coincidence of birth. This corresponds to the fact that any instance of **Part** is such that its life beginning coincides with the life beginning of *any* instance of **Whole** linked to it. This is specified as follows:

```
context Whole-Part.WPWhole::creation() : Whole

    post: result.hasForPart→forAll(p | p.oclIsNew)
```

Finally, cases 5, 7 and 8 of Figure 10 are such that they share the precedence of the part birth:

```
context Whole-Part.WPWhole::creation() : Whole
```

```
post: result.hasForPart→forAll(p |
        p.oclIsNew or not p@pre.oclUndefined)
```

Symmetrically, we have to analyze the relation between the death of a **Part** object relative to that of a **Whole** object. In this respect, Figure 12 embodies cases 1, 3 and 5.

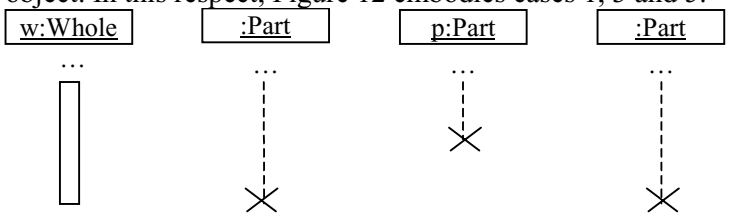

Figure 12. The crosses indicate the end of life and no assumption is made on the object's birth

Thus, cases 1, 3 and 5 share the antecedent of the part death. In other words, any instance of **Part** is such that its life end coincides with or precedes the life end of *any* instance of **Whole** linked to it (scenario in Figure 12). This is specified as follows:

```
context Whole-Part.WPWhole::deletion()

    pre: self.hasForPart→exists(p | p.oclUndefined)

    post: self.oclUndefined and

            self@pre.hasForPart→forAll(p | p.oclUndefined)
```

Following the specification process, we have for cases 2, 4 and 8:

```
context Whole-Part.WPWhole::deletion()

    pre: self.hasForPart→forAll(p | not p.oclUndefined)

    post: self.oclUndefined and

            self@pre.hasForPart→forAll(p | p.oclUndefined)
```

For cases 6, 7 and 9, we have:

```
context Whole-Part.WPWhole::deletion()

    pre: self.hasForPart→forAll(p | not p.oclUndefined)

    post: self.oclUndefined and

            self@pre.hasForPart→exists(p | not p.oclUndefined)
```

Finally, there is now no difficulty in formalizing each case individually. To conclude, we merely provide an illustration, i.e. the synthesis case in which the life span of the part is contained (not strictly) within the life span of the whole (cases 1 to 4 in Figure 10):

```
context Whole-Part.WPWhole::creation() : Whole

    post: result.hasForPart→forAll(p | p@pre.oclUndefined)

context Whole-Part.WPWhole::deletion()

    post: self.oclUndefined and

            self@pre.hasForPart→forAll(p | p.oclUndefined)
```

Note that this specification permits us to complete what Snoeck & Dedene (1998) name "existence dependency". Moreover, it seems probable that the sum of cases 1 to 4 may be what was conceived of as Composition in UML.

CONCLUSION AND FUTURE TRENDS

This chapter provides, in a UML-compliant style, the precise specification of most of the unanimously reputable characteristics of the WP relationship in OO modeling. A set of primary properties can be assigned to an abstract **Whole-Part** metatype which is not currently included in the UML metamodel, but can easily be viewed as a subtype of the existing abstract metatype called **Relationship** (this only applies for version 1.3, not for version 1.1). Furthermore, a set of secondary features can be used to derive concrete (i.e. non-abstract) subtypes from **Whole-Part**. Since Composition in UML is very ill-defined, this allows us to fully revise this modeling construct (and propose it for inclusion in version 2.0) in order to obtain something reliable and, as a result, increase the quality of UML models which benefit from a (now accurate) use of the ideas of a WP relationship. Concerning the perspectives of the work presented in this chapter, two ways can be explored. Further investigation relies on the mixing of several secondary properties to build direct (re)usable and implementable subtypes of the **Whole-Part** metatype. Indeed, secondary properties cannot be combined together without taking care of some possible contradictions raised by the combination. Because of lack of space, we use proofs only incidentally in this chapter (see the section named "Separability/Mutability" as an example of such a proof) but any attempt to mix properties such as shareability, separability/mutability and so on, will lead to new results and perhaps improved definitions of the kinds of WP relationship adequate for OO modeling. The other axis of exploration is the introduction of temporal logic into OCL in order to enhance the formalization of properties, especially lifetime dependency, which is recognized as a key and relevant notion for the WP relationship.
This is Contribution number 00/1 of the Centre for Object Technology Applications and Research.

ACKNOWLEDGEMENTS

We would thank Jos Warmer (father of OCL, member of the UML core team), Steve Cook (member of the UML core team) and Jean-Michel Bruel (member of the pUML group) for their advice about our use of OCL.

REFERENCES

Artale, A., Franconi, E., Guarino, N., & Pazzi, L. (1996). Part-whole relations in object-centered systems: An overview. Data and Knowledge Engineering. 20(3), 347-383.

Barbier, F. (1998). Systematic construction of UML associations and aggregations using cOlOr framework. Proceedings of ECOOP'98 Workshop Reader (pp.480-482). Heidelberg, DE: Springer.

Barbier, F., & Henderson-Sellers, B. (1999). Object metamodelling of the whole-part relationship. Proceedings of TOOLS PACIFIC'99 (pp.127-138). Washington, DC: IEEE Computer Society Press.

Barbier, F., & Henderson-Sellers, B. (2000). The whole-part relationship in object modelling: A definition in cOlOr. Information and Software Technology. In press.

Bock, C., & Odell, J. (1998). A more complete model of relations and their implementation: Roles. Journal of Object-Oriented Programming. 11 (2).

Bock, C., & Odell, J. (1998). A more complete model of relations and their implementation: Aggregation. Journal of Object-Oriented Programming. 11 (5).

Booch, G., Rumbaugh, J., & Jacobson, I. (1999). The unified modeling language user guide. Reading, MA: Addison-Wesley.

Bourdeau, R., & Cheng, B. (1995). A formal semantics for object model diagrams. IEEE Transactions on Software Engineering. 21 (10), 799-821.

Cook, S., & Daniels, J. (1994). Designing object systems, object-oriented modelling with Syntropy. Upper Saddle River, NJ: Prentice Hall.

Civello, F. (1993). Roles for composite objects in object-oriented analysis and design. ACM Sigplan Notices. 28 (10), 376-393.

Dong, J., & Duke, R. (1995). The geometry of object containment. Object Oriented Systems. 2(1), 41-63.

Fernandez, E., Gudes, E., & Song, H. (1994). A model for evaluation and administration of security in object-oriented databases. IEEE Transactions on Knowledge and Data Engineering. 6(2), 275-292.

Firesmith, D., Henderson-Sellers, B., & Graham, I. (1997). OPEN modeling language (OML) reference manual. Cambridge, UK: Cambridge University Press.

Firesmith, D., & Henderson-Sellers, B. (1998). Clarifying specialized forms of association in UML and OML. Journal of Object-Oriented Programming. 11(2), 47-50.

Fowler, M., & Scott, K. (2000).). UML distilled: a brief guide to the standard object modeling language (2nd ed.). Reading, MA: Addison-Wesley.

Gogolla, M., & Richters, M. (1998). Transformation rules for UML class diagrams. Proceedings of «UML»'98 (pp.92-106). Heidelberg, DE: Springer.

Halper, M., Geller, J., & Perl, Y. (1998). An OODB part-whole model: Semantics, notation, and implementation. Data and Knowledge Engineering. 27(1), 59-95.

Henderson-Sellers, B. (1997). OPEN relationships – compositions and containments. Journal of Object-Oriented Programming. 10(7), 51-55, 72.

Henderson-Sellers, B., & Barbier, F. (1999). What is this thing called aggregation. Proceedings of TOOLS EUROPE'99 (pp.236-250). Washington, DC: IEEE Computer Society Press.

Henderson-Sellers, B., & Barbier, F. (1999). Black and white diamonds. Proceedings of «UML»'99 (pp.550-565). Heidelberg, DE: Springer.

Henderson-Sellers, B., & Barbier, F. (1999). A survey of the UML's aggregation and composition relationships. L'objet. 5(3-4), 339-366.

Kent, W. (1990). Data and reality (5th reproduction). Amsterdam, NL: Elsevier Science.

Kilov, H., & Ross, J. (1994). Information modeling, an object-oriented approach. Upper Saddle River, NJ: Prentice-Hall.

Kim, W., Banerjee, J., Chou, H., Garza, J., & Woelk, D. (1987). Composite object support in an object-oriented database system. Proceedings of OOPSLA'87 (pp.118-125). New-York, NY: ACM Press.

Kolp, M., & Pirotte, A. (1997). An aggregation model and its C++ implementation. Proceedings of OOIS'97 (pp.211-221). Heidelberg, DE: Springer.

Lano, K. (1995). Formal object-oriented development. Heidelberg, DE: Springer.

Lewandowski, S. (1998). Frameworks for component-based client/server computing. ACM Computing Surveys. 30(1), 3-27.

Markowitz, J., Nutter, J., & Evens, M. (1992). Beyond is-a and part-whole: More semantic network links. Computer Mathematics with Applications. 23(6-9), 377-390.

Motschnig-Pitrik, R. (1996). Analyzing the notions of attribute, aggregate part and member in data/knowledge modeling. Journal of Systems and Software. 33(2), 113-122.

Motschnig-Pitrik, R., & Kaasboll, J. (1999). Part-whole relationship categories and their application in object-oriented analysis. IEEE Transactions on Knowledge and Data Engineering. 11(5), 779-797.

Odell, J. (1994). Six different kinds of composition. Journal of Object-Oriented Programming. 7(8), 10-15.

OMG (1997). UML Summary, Semantics and Notation Guide, version 1.1.

OMG (1999). OMG Unified Modeling Language Specification, version 1.3.

Opdahl, A. L., Henderson-Sellers, B., & Barbier, F. (2001). Ontological analysis of whole-part relationships in OO-models. Information and Software Technology. In press.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., & Lorensen, W. (1991). Object-oriented modeling and design. Upper Saddle River, NJ: Prentice-Hall.

Rumbaugh, J., Jacobson, I., & Booch, G. (1999). The unified modeling language reference manual. Reading, MA: Addison-Wesley.

Saksena, M., France, R., & Larrondo-Petrie, M. (1998). A characterization of aggregation. Proceedings of OOIS'98 (pp.11-19). Heidelberg, DE: Springer.

Saksena, M., Larrondo-Petrie, M., France, R., Evett, M. (1998). Extending aggregation constructs in UML. Proceedings of «UML»'98 (pp.435-441). Heidelberg, DE: Springer.

Snoeck, M., & Dedene, G. (1998). Existence dependency: The key to semantic integrity between structural and behavioral aspects of object types. IEEE Transactions on Software Engineering. 24(4), 233-251.

Varzi, A. (1996). Parts, wholes, and part-whole relations: The prospects of mereotopology. Data and Knowledge Engineering. 20(3), 259-286.
Warmer, J., & Kleppe, A. (1998). The object constraint language. Reading, MA: Addison-Wesley.
Winston, M., Chaffin, R., & Herrmann, D. (1987). A taxonomy of part-whole relations. Cognitive Science. 11, 417-444.

APPENDIX: A BRIEF INTRODUCTION TO OCL

Following the UML documentation, OCL was introduced in UML for a variety of reasons. First, a pure graphical model is in general not enough for a precise and unambiguous specification of a system. Often one needs to describe additional constraints about the objects, for example using a textual specification language. However, the disadvantage of traditional formal specification languages is that they are difficult to learn and to use for average business or system modeler. OCL was developed to fill this gap with a strong emphasis on user-friendliness. OCL is a pure expression language which means that every OCL expression is free from any side effect. Therefore OCL is not a programming language. It is not possible to write program logic or flow of control in OCL. On the other hand OCL is a typed language, so that each OCL expression has a type and types within OCL can be any kind of **Classifier** (one of the more important metaclasses in the UML metamodel). There is a variety of places where one can use OCL in models. One may specify invariants on classes and types in class diagrams. It is also possible to formalize user-defined stereotypes by means of OCL. The description of pre-conditions and post-conditions on operations is supported by OCL as well as the specification of guards in statechart diagrams. Furthermore the language can be used as a navigation (or query) language. Regarding the semantics of UML, OCL is used to express the well-formedness rules related to the metaclasses in the metamodel.

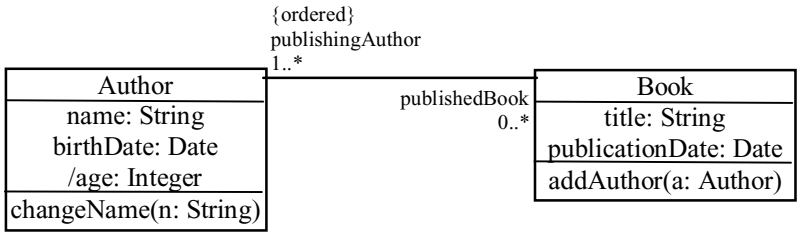

Figure 13. A simple UML model

Let us underpin these statements by considering a class diagram (Figure 13). This class diagram introduces two classes, **Author** and **Book**, and an association between them. On the **Author** side, we find the multiplicity "1..*" indicating that a **Book** has at least one and possible many more authors represented as an ordered list with the role name **publishingAuthor**. On the **Book** side, we see the multiplicity "0..*" allowing authors without books to exist and the role name **publishedBook**. As claimed above we can use OCL for a number of purposes. An appropriate *invariant* in the class diagram is that an author's date of birth must precede the publication of her/his book:

```
context Book inv:

    self.publishingAuthor→forAll(a | a.birthDate < self.publicationDate)
```

This OCL expression is formulated within the context of the class **Book**. Therefore `self` refers to a book. The expression `self.publishingAuthor` navigates through the association and yields the set of authors for the fixed book. On this set, we use the universal quantification `forAll` and require that each author's `birthDate` must be less than the `publicationDate`. OCL is close to first-order predicate calculus in the sense that it allows universal and existential quantification as well as the usual logical connectives **`not`**, **`and`**, **`or`**, **`implies`** to be used in an expression.

The following *derivation* rule for the `age` attribute of **Author** relies on the existence of a class **Date** providing an operation `today() : Date` that returns the current date. The **Date** class is also assumed to possess an

operation `yearDifference(d: Date) : Integer` determining the number of years remaining between two dates.

```
context Author inv:

    self.age = d.today().yearDifference(self.birthDate)

    -- this is a comment: d is an arbitrary instance of Date
```

The pre-conditions and post-conditions for the operations `changeName` and `addAuthor` describe the manipulations done by these operations.

```
context Author::changeName(n: String)

    pre: n <> name

    post: name = n
```

The pre-condition for `changeName` requires that the parameter `n` is different from the current `name`, whereas the post-condition states that after the completion of the operation, `name` will hold the value of the parameter `n`.

```
context Book::addAuthor(a: Author)

    pre: not self.publishingAuthor→includes(a)

    post: publishingAuthor = publishingAuthor@pre→append(a)

    -- this a comment: note that we omit "self" in the post-condition
```

The pre-condition for `addAuthor` requires that the author to be added is not already present in the book's author list. The post-condition states that the value of `publishingAuthor` after the operation is equal to the value of `publishingAuthor` before the operation (use of the "`@pre`" operator) plus the parameter `a` appended.

To demonstrate the navigation possibility, the following expression computes for a given author, the names of her/his co-authors:

```
context Author inv:

    publishedBook.publishingAuthor→reject(a | a = self)→collect(a | a.name)
```

The term `publishedBook.publishingAuthor` has the type **Set(Person)** although one might expect the type **Set(Set(Person))** because `publishedBook` has the type **Set(Book)**. Nevertheless, nested collections are automatically flattened according to the OCL documentation. The `reject` expression filters the author set such that only authors different from the current author remain. The `collect` expression applies the attribute `name` to each author in the author set.

[i] USA and Switzerland are officially confederations instead of federations but in practice they both behave as federations due the dominant position of the state union compared to each state in the union. In contrast, EU is currently a confederation and does not then conform to the “State” type.