

# Hybrid OMDoc Documents in software development

Sönke Holsten

Guided Research in Mathematics

Spring 2008

Supervisor: Prof. Dr. Kohlhase

May 20, 2008

## Abstract

This report discusses how hybrid OMDoc documents facilitate change management in software development processes. It shows that the different data produced at different stages of the software development process can be represented in hybrid OMDoc documents using semantic markup. This enables explicit stating of relations between different data formats, which in turn allow for a granular change impact analysis throughout the whole software development process. Hybrid OMDoc documents integrate modeling languages, formal specification languages, programming languages and natural language in OMDoc.

## 1 Introduction

A typical large-scale software development process involves in each phase different people, working with different tools, using different representations of the implemented system. For instance during the requirements analysis and design phase the requirements are specified in natural language and in the Unified Modeling Language (UML) [16]. During the implementation the actual software is written using a programming language. For verification and validation formal system specifications are written using formal specification languages. When maintaining the software the maintenance of the documentation (including UML diagrams) and formal specifications is a time-consuming task that is vulnerable to human error as the different representations of an aspect of the system are only loosely linked, if at all.

To answer the need for a tighter integration of the different development phases I propose the integration of documentation, source code and formal specifications in OMDoc. More specifically I propose the use of *Hybrid OMDoc documents*, which can be regarded as self-contained structured compositions of information units hosted in various domains with OMDoc as the top-level information unit (see Fig. 1). In this report I present XML based languages to achieve a granular and standardized structure for these information units. Moreover I identify possible interrelations between them at different levels of granularity and discuss how changes in one can affect other parts of the document.

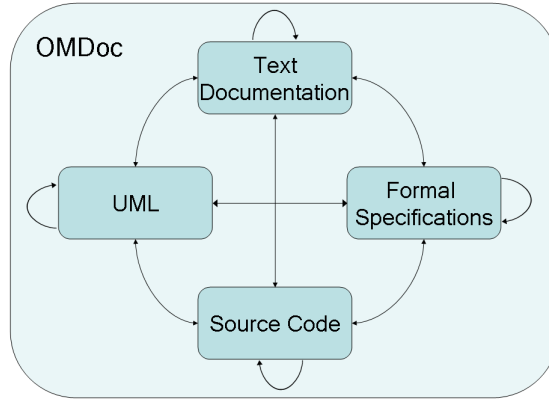


Figure 1: Integration in hybrid OMDOC documents

## 2 Preliminaries

One important precondition for being able to explicitly state relations in hybrid OMDOC documents is a sophisticated semantic markup of all parts of the hybrid OMDOC document. Such a markup should ideally be oriented at established standards, distinguish between presentation and content and be intuitive. In case of UML it should cover the most important UML diagrams (diagrams newly introduced in UML 2.0 are negligible).

In this section I briefly describe two different XML based languages which can be used in conjunction in hybrid OMDOC documents. First I explain the standards XML Metadata Interchange (XMI) [17] and Metaobject Facility (MOF) [15], which combined provide means for representing UML models in XML. Then I present the Code Markup Language (CODEML) [13], a markup for source code. These two markups already provide sufficient structure for the most important domains in hybrid OMDOC documents. Formal specifications can be represented in OMDOC, but the question of how to achieve this is out of the scope of this report.

### 2.1 XMI[UML]

The XML representation of UML models is based on XMI, which defines a mapping from MOF-compliant models to XML. UML is defined in MOF, so I will first explain MOF, then describe XMI and finally motivate my choice of using `ArgoUML` [1], a UML design tool with support for XMI export of UML models.

#### 2.1.1 MOF

MOF is a standard adopted by the Object Management Group (OMG) [18]. Its development was driven by the need for a metamodeling language for the popular modeling language UML, but in general can be used as a language to model metadata. E.g. the current specification of the Common Warehouse Metamodel [14] is defined as a MOF-compliant metamodel. In this paper I am only interested in MOF to define UML.

In its core MOF can be seen as a subset of UML for modeling classes, hence UML modeling tools can be used to design MOF metamodels in practice. The MOF specification comprises 4 layers of abstraction:

- M3 The *top layer* describing the metamodel of MOF, also referred to as the meta-metamodel.
- M2 The *second layer* describing metamodels of modeling languages, e.g. the UML metamodel.
- M1 The *third layer* describing models, e.g. UML models.
- M0 The *bottom layer* describing a specific instance of a modeled system.

The naming of the layers (M3 through M0) relates to the number of m's in the abstraction level, e.g. M2 stands for **metamodel**. Note that in the top layer MOF is used to define itself.

### 2.1.2 XMI

XMI is a standard released by the OMG with the aim to connect XML with MOF. It allows for a standardized XML representation of all models adhering to a MOF-compliant metamodel, hence also of UML models. Its main use is to facilitate data interchange.

It is important to note that XMI only specifies the mapping from MOF-compliant models to XML representations (in the XMI specification given as EBNF productions). XMI itself does not stand for a concrete XML representation, only its application yields such. Therefore when referring to the XML representation of UML one should use the term XMI[UML] (read XMI applied to UML) instead of just XMI, but since in this report I am only concerned with this specific application, I will not make this distinction in the rest of this paper. XMI in general describes two different aspects: (1) How XMI-valid validation mechanisms for a specific UML model look like — in earlier versions (1.x) this was done by specifying Document Type Definitions (DTDs), in the current versions (2.x) this is done by specifying XML schemas (2) How XMI-valid XML streams, i.e. specific instances of a UML model, look like. A typical XMI document as depicted in Fig. 2 has three top-level elements. The `XMI.header` element contains metadata about the model. Its most important child elements are the `XMI.documentation` element, which e.g. stores data about the specific exporter used to create the XMI file as well as some short textual descriptions, and the `XMI.metamodel` element, which specifies to which MOF compliant model the XMI was applied. Fig. 2 already shows the complete `XMI.header` element created by ArgoUML (cf. section 2.1.3).

The `XMI.content` element contains the UML model itself. Fig. 3 depicts a simplified version of a UML model. It consists of a class called `Class1` that has exactly one attribute named `Attribute1`.

The `XMI.extension` element contains the graphical representation of the model. The actual format of this representation is specific to the modeling tool used, e.g. ArgoUML does not use it at all. It can additionally be used to define constructs not included in the metamodel, i.e. extensions to the metamodel defined in the `XMI.header` element. Since I am only concerned with the core of UML and am not interested in tool specific details, I will not include the latter element in my further

---

```

1  <XMI>
    <XMI.header>
        <XMI.documentation>
            <XMI.exporter>
                ArgoUML (using Netbeans XMI Writer version 1.0)
6      </XMI.exporter>
        <XMI.exporterVersion>
            $0.24(5) revised on Date: 2006-11-06
            19:55:22 +0100 (Mon, 06 Nov 2006) $
        </XMI.exporterVersion>
11     </XMI.documentation>
        <XMI.metamodel xmi.name="UML" xmi.version="1.4"/>
    </XMI.header>
    <XMI.content>
        <UML:Model>
16     ...
        </UML:Model>
    </XMI.content>
    <XMI.extension>
        ...
21   </XMI.extension>
</XMI>

```

---

Figure 2: A sample XMI document.

---

```

<UML:Model>
    <UML:Namespace.ownedElement>
        <UML:Class name="Class1">
            <UML:Classifier.feature>
5          <UML:Attribute name="Attribute1">
                ...
                <UML:StructuralFeature.type>
                    <UML:DataType xmi.idref = '789' />
                </UML:StructuralFeature.type>
10         </UML:Attribute>
            </UML:Classifier.feature>
        </UML:Class>
        <UML:DataType xmi.id = '789' name = 'int' />
        </UML:Namespace.ownedElement>
15 </UML:Model>

```

---

Figure 3: A sample UML model in XMI.

discussion, but briefly explain `ArgoUML`'s representation of the UML diagrams in the next section.

For a more detailed introduction to XMI and MOF I refer the reader to [12].

### 2.1.3 ArgoUML

Although with XMI there exists a well-developed standard for UML model interchange, difficulties arise when looking at concrete tool support. Some tools do not support XMI at all, but only image formats, e.g. the free diagram creation software Dia [9]. Other tools such as Omondo [19] are based on the Ecore dialect of UML provided with the Eclipse Modeling Framework (EMF) [10]. Even though the latter is also a widely used metamodel, basing this theoretical work on the XMI[Ecore] format would require to be more concerned with tool and metamodel specific details, which would add unnecessary complexity to this discussion.

Another important criterion for my choice of a modeling tool is that for the purpose of tool development for hybrid OMDOC documents, e.g. for the presentation of such, I require the tool to be open source. All UML models presented in this report were created using ArgoUML, which is a freely available UML editor. The current version of ArgoUML (v0.24) is written to support UML 1.4. Note that I will be mainly concerned with three types of UML diagrams: (1) Class Diagrams, (2) Sequence Diagrams and (3) Statechart Diagrams. These diagrams already provide the main chunk of UML's functionality used in daily practice. This limitation also allows me to fully base my change management in UML models on the work presented in [5]. Therefore ArgoUML's lack of support for UML 2.x diagrams is negligible.

ArgoUML uses version 1.2 of XMI, which in contrast to versions 2.x uses DTDs instead of XML schemas as the underlying validation mechanisms. One important advantage of using XML schemas over DTDs is that it is easier to model inheritance relations in XML schemas. In DTDs the elements and attributes that are inherited have to be copied into the inheriting class, which results in redundancies. However this does not affect the general discussion of interrelations between UML and OMDOC elements. For the purpose of tool reuse different validation mechanisms can be useful, in particular RELAXNG schemas [23] are needed for `mdiff`, which can easily be created using the *Sun Relax NG converter* [24].

For storing the actual graphical representations of the UML diagrams ArgoUML uses the Precision Graphics Markup Language (PGML) [20]. PGML was developed by Adobe, IBM, Sun, and Netscape and proposed to the W3C as a standardized XML-based graphical representation format. It was combined with VML [30] and further developed to yield the eventually adopted standard: SVG [27].<sup>1</sup> As all semantics-related information of the UML model is already stored in the XMI element, the exact graphical representation of the diagrams does not matter for the impact analysis. What is important however is the fact that a possibility to store graphics-related information exists and it is possible to create viewable diagrams from these representations. ArgoUML uses GEF, the Graphical Editing Framework, to create image files, e.g. PNG, from the PGML representation. As GEF is free and open software, a tool that prepares hybrid OMDOC documents for presentation can be written based on GEF.

To sum up I conclude that ArgoUML's XML representation of UML models suits my requirements for such for the following reasons:

1. It is based on the OMG UML metamodel
2. It uses the OMG XMI standard to represent UML models
3. It supports all relevant diagram types

---

<sup>1</sup>The tool `uml2svg` [28] can convert XMI-compliant UML models into SVG.

4. It is easy to read by human readers
5. It provides sufficient means for graphical representations
6. Interchange with EMF-based tools is supported

## 2.2 CODEML

The Code Markup Language (CODEML) is a markup language for source code. CODEML provides means to represent the content of source code separately from its presentation. This concept can analogously be found in XMI, where the `XMI.content` element contains the content of the UML model and the `XMI.extension` element contains the exact graphical representation of the UML model. The presentation section of CODEML carries information about aspects such as indentation, line breaks, color and size of the source code as well as comments to it, while the content section carries information about the abstract structure. To illustrate this a small fragment of JAVA code represented in content CODEML is depicted in Fig. 4. It shows the return call returning the variable `x`. The `apply` element is used to represent a function application, in this case the return function represented by the `ccsym` element. This element has an attribute `cd` which indicates in which content dictionary the return function is defined, in this case in *java.proc*. The second child element of the `apply` element represents the value the return function is applied to, in this case it simply is the variable `x` represented by the `ccv` element.

---

```
<apply>
  <ccsym cd="java.proc" name="return"/>
  <ccv name="x"/>
</apply>
```

---

Figure 4: Content CODEML representation of `return x`

The presentation and content elements of CODEML can be combined using the `semantics` element. For a more granular integration the optional attributes `xlink` and `id` can be used, thus also providing means for a granular integration with other parts of the hybrid OMDoc documents.

## 3 A Taxonomy of Relations in Hybrid OMDoc Documents

In this section I look into how the different parts of hybrid OMDoc documents are related to each other and will give intuitive notions of how these relations can be understood. A mathematical formalizations of these relations is not provided at this stage of work, as I limit myself in this paper to giving the intuition about why these relations can facilitate change management. The question of exactly how this can be achieved is subject to future work.

The structure of this section is as follows: Subsection 3.1 gives a general overview of how a software system evolves during the software development process focussing on representations of the system itself. The next subsection describes how additional

additional documentation of a software system is related to the system representations, also focussing on testing. In Subsection 3.3 relations in UML models are described. In the last subsection I provide some examples for relations between UML models and / or source code and OMDOC elements.

### 3.1 Transformations

Fig. 5 shows a sketch of a hybrid OMDOC document illustrating the content produced during the most important stages of a typical software development process, which are: *requirements specification* (both in natural language and in UML) and *programming*. Intuitively all three child elements in the figure describe the same aspect of the system, just in different languages and at different levels of formalization. To capture this general notion I define a transformation relation as follows: a **transformation relation** holds between any two representation of the same system. I will refine this definition when introducing new interrelations as special cases of transformations.

One general observation about transformations can already be made at this stage: If an element  $A$  is related via the transformation-relation to an element  $B$ , which in turn is also related via a transformation-relation to an element  $C$ , I can conclude that there must exist a transformation-relation between elements  $A$  and  $C$ , which can intuitively be understood as a combination of the other two. For example in Fig. 5 the transformation relations not only holds between the textual description and the UML model and the UML model and the source code (as depicted), but also between the textual description and the source code. In mathematics this notion is referred to as transitivity, so in this report I will consider all transformation relations to be **transitive relations**. Also note that I consider transformations to be directional, i.e. when saying that a transformation relation holds between elements  $A$  and  $B$ , I will consider  $A$  to be the source and  $B$  the target of the transformation. Although intuitively a transformation relation should hold in both directions, its properties (which I explain below) might not be the same for both directions.

One approach to further concretize the notion of transformation relations is to look at how exactly the act of transforming one system representation to another is carried out. This can be done manually, semi-automatically or automatically. A **manual transformation** is a transformation carried out by the developer without any tool support. This is, for example, usually the case when a developer creates a UML model on the basis of written or oral requirements communicated by his customer. In Model Driven Development an approach to model transformations with tool support is taken using model transformation languages. The OMG adopted Query/Views/Transformations (QVT) [22] as the standard model transformation language for UML. It can only be used to define transformations between two UML diagrams, not between UML diagrams and source code. There exist other transformation languages such as the ATLAS TRANSFORMATION LANGUAGE (ATL) [2] that do support transformations from UML diagrams to source code. These transformations can be fully specified, then they are **automated transformations**, or they might require further input from the developer during the transformation process, then they are **semi-automated transformations**. In hybrid OMDOC documents transformations can be of any of these kinds. It is also possible to provide textual descriptions of how the transformation should be carried out, but they are considered to be manual transformations as well.

Looking at the three elements in Fig. 5 one can identify the different languages used as their main distinguishing feature. When a transformation relates elements of different languages I define the transformation relation to be a **translation relation**. In

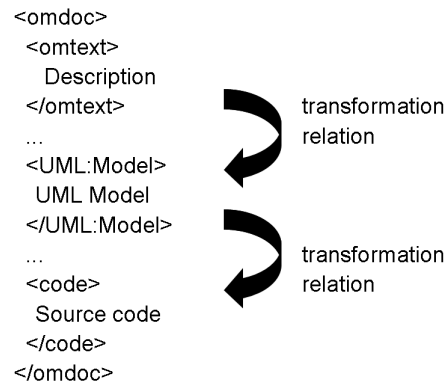


Figure 5: Software Development Process: overview

Fig. 5 the three different languages are: natural language, a modeling language (UML) and a programming language. The distinction of the different languages does not always have to be that apparent. E.g. UML 1.4 is considered to be a different language than UML 2.0. Or in case of texts written in german and english, one might not be able to distinguish the languages simply by looking at the element they are wrapped in. Note that not all transformations are also translations, e.g. two UML models representing the same system just at different levels of abstractions are transformations of each other, but not translations as both are written in the same version of UML. Note that in contrast to all other transformation relations the translation relation is not transitive.

Another aspect of transformations derives from the theory that underlies the transformations. For instance transformations between two different XML based languages can be described using graph theory. Transformations differ in their degree of formality depending on what (if at all) theory they are based on. If a transformation relation is based on some mathematical theory I will call it a **formal transformation relation**, otherwise it is an **informal transformation relation**.

When looking at model representations that are related via a transformation relation one can identify different transformations at different levels of granularity. Fig. 6 illustrates that a refinement cannot only hold between two model representations as a whole, but also between classes and attributes of the system. Lines 1-12 containing the textual representation of the class Class1 are refined in lines 14-23 containing the XMI representation of the class. On a finer level I can also identify a refinement between the two representations of the attribute Attribute1 in lines 7-11 and lines 16-21. This could be carried out even further with the type of the attribute or multiplicities. Note that the markup for text used in Fig. 6 is valid OMDOC. As can be seen the `omtext` element is not yet well-suited for a granular markup of textual descriptions and subject to future work.

A further classification of transformations derives from a semantic analysis of the source and the target of the transformation. If the target is semantically richer than the source, the transformation relation is considered to be a **refinement relation**. For instance, it is natural to view the UML model as being more formal and rigorous than a text written in natural language, i.e. I can consider the UML model to be a refinement



---

```

<omgroup id="gc1w">
  <omgroup id="gc1d">
    <omtext id="tc1d">
      This is a description of the class Class1 with
5    </omtext>
    </omgroup>
    <omgroup id="gc1a1">
      <omtext id="tc1a1">
        the attribute Attribute1 of some type.
10    </omtext>
    </omgroup>
  </omgroup>
  ...
  <UML:Class xmi.id = '851' name = 'Class1'>
15    <UML:Classifier.feature>
      <UML:Attribute xmi.id = '854' name = 'Attribute1'>
        ...
        <UML:StructuralFeature.type>
          <UML:DataType xmi.idref = '853' />
20        </UML:StructuralFeature.type>
        </UML:Attribute>
      </UML:Classifier.feature>
    </UML:Class>

```

---

Figure 6: Example for granular markup.

of the textual description. In the same way I can consider the source code to be a refinement of the UML model. Note that for a rigorous mathematical analysis the refinement relation has to be formal. I consider the inverse of refinement relation to be a **abstraction relation**.

It is also possible to have a transformation relation between two systems that are semantically equivalent. I will call such a transformation relation an **isomorphic relation**. Analogously to the refinement relation this relation can be formal, but does not have to be. In Model Driven Development one often defines several Platform-specific models (PSMs), which are refinements of a Platform-independent model (PIM), hence it is also sensible to define refinements within UML models. In [8] an approach to identifying refinement patterns in UML is given and thus provides a theoretical background for refinements in UML. The literature on formalizations of transformations and refinements in particular is vast and will not be further discussed in this report, for program transformations in particular I refer the reader to [21].

### 3.2 Documentation relations

During the software development process not only a textual description of the software is written but also many supplementary texts that document the process as a whole or give additional details, that cannot be expressed in a modeling language or in the programming language. I say that any text which provides information about a transformation or a certain element in our model is related to this transformation or element in a **documentation relation**. Analogously to the transformation relation I will refine

the documentation relation, i.e. consider all the following relations as refinements of the documentation relation. Note that in general the documentation relation is unidirectional, that is its inverse always carries different semantics.

We have already seen one instance of a documentation relation: the **description relation**. This relation holds between a text element and another model or model element when there exists a transformation relation between them. Fig. 6 shows an example of this relation, where the text in lines X-Y describes the UML class in lines X-Y. Analogously an UML element can also describe other model elements.

Some documentation is written not to exactly describe model elements, but to illustrate their behavior or their purpose. A method might be explained by giving an example of how it acts on certain inputs or a class might be explained by describing a sample instance of that class. Text elements providing such examples are related to the model elements for which they provide the examples by the **example relation**. When the text element contains a rigorously defined example for the purpose of testing I define the relation to be a refined instance of the example relation called **testcase relation**.

Another important aspect of documentation is planning. A text element can describe what still has to be done in a model or model element. Some examples for what such a text element can contain are: information about what parts of a model (classes, methods) are missing, which parts of source code have to be optimized in respect to performance or space constraints or what model elements still need to be documented. Such text elements are related to the model or the model element in a **todo relation**.

Todo statements that have been realized can be recorded in a changelog. I say that text elements of the above mentioned kind can be related to a model or model element in a **realized-change relation**. XMI also allows for representation of changes between two XMI elements. Accordingly the realized-change relation can also relate a change represented in XMI to the XMI element in which the change was realized, then I call it a formal realized-change relation. Conversely the change can be related to the XMI element to which it can be applied in a formal todo relation.

On a more detailed level planning can mean specifying how a model representation should be transformed into another, which usually is needed when implementing a model. Examples for this are algorithms that should be implemented by a method or constraints which hold for attributes or the applicability of a method. A text element specifying this can be related to the corresponding model or model element in a **howto-implement relation**.

Complementary to this relation there also exists a **how-implemented relation**, which relates a text element of the above mentioned kind to the actual part of the source code that implements a model element according to the contents of the text elements. Note that such text elements can also be referenced by informal transformations.

The howto-implement and how-implemented relations can also hold between elements containing transformations written in a formal transformation language such as QVT or ATL and the corresponding source and target elements. An example for such transformations was already given in section 3.1. In this case I define the howto-implement and how-implemented relations to be formal.

During testing and maintenance bugs are discovered and need to be fixed. Text elements containing bug reports can be related to the corresponding model element by the **occured-bug relation**. Again a bug that has been fixed can also be related to a model, in that case we say that they are related via the **fixed-bug relation**.

One might notice that all relations in the context of planning follow the same scheme, they either represent something that has to be done or fixed or implemented or

they represent something that has been done, has been fixed or has been implemented. For this reason we consider the howto-implement relation and the occurred-bug relation to be refined instances of the todo relation and the how-implemented and fixed-bug relation as refined instances of the realized-change relation.

Planning also includes specifying who is responsible for doing something, i.e. which programmer should fix the bug or realize the change. Therefore all instances of the todo relation can reference a text element containing the name of person or group and all instances. Of course the same holds for all instances of the realized-change relation. We call this relation the **maintainer relation**.

### 3.3 UML relations

One apparent advantage of UML diagrams is that most relations that exist in UML models are given explicitly. The **explicit UML relations** I am concerned with are:

- In Class Diagrams: **association, generalization, composition, aggregation, dependency, realization**
- In Sequence Diagrams: **message, association role**
- In Statechart Diagrams: **transition**

For a detailed discussion of how these relations are used I refer to [11]. In XMI the explicit relations are given as elements, e.g. an association is represented with the `UML:Association` element. Fig. 7 shows a simplified XMI representation of an

---

```

<UML:Association xmi.id='78F' name='Association1'>
  <UML:Association.connection>
    <UML:AssociationEnd>
      <UML:AssociationEnd.multiplicity>
5      <UML:Multiplicity>
        ...
        </UML:Multiplicity>
      </UML:AssociationEnd.multiplicity>
      <UML:AssociationEnd.participant>
10      <UML:Class xmi.idref='77C' />
      </UML:AssociationEnd.participant>
    </UML:AssociationEnd>
    <UML:AssociationEnd>
      ...
15    </UML:AssociationEnd>
  </UML:Association.connection>
</UML:Association>

```

---

Figure 7: Association in XMI.

association named *Association1*. It is important to note that id's are used to cross-reference the classes in the model, so besides the explicit relations modeled as XMI elements, there is a second kind of explicit relations, the **reference by id relation**. This relation is also commonly used in OMDOC and CODEML documents.

There are also relations that cannot be directly derived from XMI models, which relate to the consistency of the UML model. E.g. an operation that is invoked in a

message in a Sequence Diagram is related to an operation of a class in a Class Diagram. In [5] a detailed analysis of consistency rules for UML models can be found from which these **implicit relations** can be derived.

Both, the implicit and the explicit relations can also hold between different source code fragments or even different texts in natural language given that they are transformations of the corresponding concepts in UML. It is even possible for these relations to hold between different parts of hybrid OMDOC documents, e.g. an association can hold between a CODEML representation of a class and a XMI representation of another class.

### 3.4 Relations to OMDOC elements

In order to show that a software system has certain properties, it is often useful to specify the system with a formal specification language. Useful properties of software systems can in general be divided into two categories: liveness and safety properties. A liveness property ensures that the system exhibits a certain behaviour, i.e. 'good things' happen (e.g. termination analysis). A safety property ensures that certain events do not occur, i.e. 'bad things' do not happen (e.g. invariants are not violated). As mentioned in section 2 formal specifications can be represented in OMDOC. A mathematical proof showing that a safety or liveness property holds for a certain part of the system can be related to the XMI or CODEML representation of that part via a **property proof relation**, that can be further specified as a **safety** or **liveness** property. It is possible that properties of a software system are both, safety and liveness. Further refinements of the property proof relation are possible, but are not subject to this report.

One very specific possibility for a formalization in hybrid OMDOC documents that is also a transformation, is the refinement of types in the software specification. In OMDOC abstract data types are represented with the `adt` element, which provides means to capture all semantics of an abstract data type. In Fig. 3 the `UML:Datatype` element is depicted. Except for the name of the datatype it does not provide any further information. Thus a relation between the `adt` element and the `UML:Datatype` element allows for a sensible formalization of the `UML:Datatype` element, which can be described as a transformation relation or more specifically a refinement relation. On the other hand UML Class Diagrams can model abstract datatypes, thus be related to the `adt` element in a **visualization** relation.

## 4 Change Management

Having identified what kind of relations exist in hybrid OMDOC documents I now need to analyse how these relations help us when tracking changes throughout the whole software development process. To determine how a change propagates along a relation I first need to classify the changes as e.g. a simple reordering of classes in a UML model clearly has different effects than adding new classes to the model. In this section I first give an abstract description of change classes and then give a more detailed analysis of possible changes in UML models. I will end this section by sketching an analysis of how the identified changes propagate along the relations identified in section 3.

## 4.1 General discussion

I consider all changes to be of one of the following kinds: a **syntactic change**, which is a change in the model representation that does not change the model itself, and a **semantic change**, which is a change in the model representation that does change the model itself. Apparent examples for syntactic changes are inserting empty lines, reordering unordered lists or renaming classes. Semantic changes can for instance be adding or deleting classes, changing types of attributes or methods or introducing new relations within the model.

Depending on the language the model representation is written in changes can be of different kinds. E.g. changing the indentation in a programming language like `C` does not affect anything in the program (syntactic change), but in, for example, `Python` this might very well change the functioning of the whole program (semantic change). I have not yet introduced XML-based representations of supplementary documentations and formal methods, but — at this stage of this work — I assume that I have such representations. Changes that can be considered syntactic change for any XML-based language are e.g. the order of attributes in an element. Also modifying QNames might not affect the semantics of the system provided that the extended QNames stay the same. For a detailed discussion on syntactic changes in XML-based languages I refer to the specification of *canonical XML* [7]. Semantic changes can be refinements, isomorphisms or abstractions as introduced in section 3.1.

## 4.2 Impact analysis

I will now analyse whether a change propagates along the relations identified in section 3. For transformation and documentation relations I will present my own findings. For the UML relations I will briefly discuss the results presented in [5]. Relations to

**Transformations** The notion of a transformation itself is too general to exclude the propagation of changes of any kind. It depends on some of the identified properties whether semantic or syntactic changes propagate along transformation relations. Considering syntactic changes it is important to know whether the transformation is automated, semi-automated or manual as automated and semi-automated transformations might establish a relation between the structure of source and target, whereas manual transformations do not. If e.g. the transformation from a PIM to a PSM is specified in QVT transformation rules and a tool is specified that carries out the transformation, syntactic changes in the PIM can influence the syntactic structure of the PSM. As there are also automated transformations that do not take into account the syntactic structure of the source, the propagation of the syntactic change depends on the transformation language and the transformation tool. Syntactic changes are not propagated along manual transformations.

The degree of automation does not allow statements about the propagation of semantic changes. For these it is important to know whether the transformation relation is a refinement or an isomorphic relation. For isomorphic relations it is clear that semantic changes must propagate along them as they can only hold between elements that have the same semantics, independently of which element is source and which is target. For refinement relations it is necessary to discuss source and target separately. A semantic change in the source does not propagate if the change is also an abstraction, otherwise it does. Analogously a semantic change in the target does not propagate if the

change is also a refinement, otherwise it does. Conversely the same observations hold for abstraction relations with the roles of source and target being interchanged.

The other properties of the transformation relations, i.e. translation, formal and informal relation, are not affected by any change, since they only depend on the languages of source and target and not on what is described in that languages. Note that in this discussion we are only concerned with transformations that are translations as only those represent the interplay between different stages of the software development process.

**Documentation** Syntactic changes generally do not propagate along the documentation relation, semantic changes — depending on the type of documentation relation — might, so when talking about changes in the context of documentation relations I refer to semantic changes. The description relation is redundant to a transformation relation that holds between a textual description and any other system representation in the hybrid OMDOC document, i.e. change propagation along the description relation depends on the properties of the transformation relation that holds between the same two parts of the hybrid OMDOC document.

In an example relation the example element might need to be modified after a change in the system. The example element only has illustrative purposes, i.e. a change in the example element should not require a change in the class or method referred to (the example does not determine the behaviour of the class or the method). The same holds for the testcase relation.

In a todo relation the todo element might need to be modified after a change in the system. In the other direction it is necessary to further distinguish changes in a todo element. The addition of an item to the todo element does not propagate along the todo relation. The deletion of an item can occur for two reasons: (1) It has been dropped and will not be implemented, in that case the change does not propagate. (2) It has been implemented, in that case a change in the referenced part of the system is required. For the howtoimplement and the occurredbug relation the same statements hold.

In a realized-change relation the changelog element needs to be modified after a change in the system. This might depend on the kind of change as it is possible that certain types of changes might not require to be documented. A change in the changelog element cannot occur without a change in the system. For the howimplement and the fixedbugrelation the same statements hold.

**UML relations** An analysis of how changes in UML models propagate along the relations identified in subsection 3.3 can be found in [5], which is why I exclude these relations from my own impact analysis and solely present a short overview of the impact analysis in [5]. Fig. 8 shows an excerpt of the taxonomy of changes developed in [5]. It can be seen that the authors identified changes in Class, Sequence and State-chart Diagrams. In general the types of changes identified are either adding or deleting an element or changing a value. All in all the authors identify 97 different possible changes, thereby providing a very granular analysis. For each of these they describe the possibly impacted elements in the model and provide formal rules which are used in their tool to automatically propagate the changes. The OMDOC tool support provides all necessary means to implement the change detection and change propagation rules presented by the authors. The rules also hold for relations adopted from UML, i.e. can be applied to propagate changes in e.g. source code.

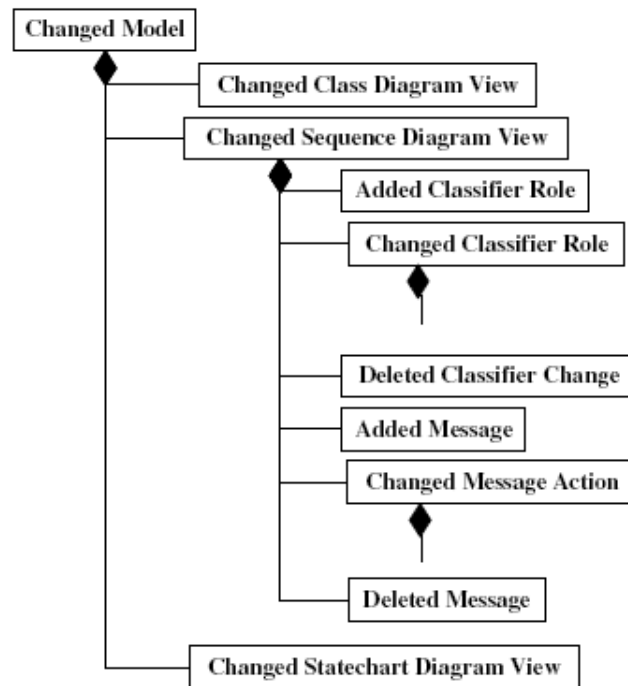


Figure 8: Taxonomy of change for UML diagrams

## 5 Case study: Phonebook

I now illustrate how changes propagate in an actual software development process by looking at the development of a program that manages phone numbers. At the beginning of the process the customer needs to specify what he expects of the software, a brief description that can be included in a hybrid OMDOC document might be phrased as follows:

*The software should be able to manage phone numbers, i.e. represent a phonebook. Each entry in the phonebook should consist of a phone number and a name. It should be possible to add entries, update phone numbers in existing entries, delete entries and query for entries by name.*

By looking at the description the software engineer models the software system. For the purpose of communicating with the customer he first designs a PIM, which is depicted in Fig. 9. According to the relations identified in section 3 the relation between the textual description and the PIM is a manual, informal transformation relation, that can be further described as a refinement and a translation. Additionally it is also a description relation.

In the next step the software engineer decides to implement the software in C++. He therefore modifies the UML Class Diagram such that it is tailored for implementation in C++. The relation between the PIM and the PSM is again a manual, informal transformation, which is also a refinement, but not a translation as both elements are written in UML. As XMI already provides a granular markup, it is possible to relate all attributes and methods that remain unchanged in the PSM to the corresponding el-

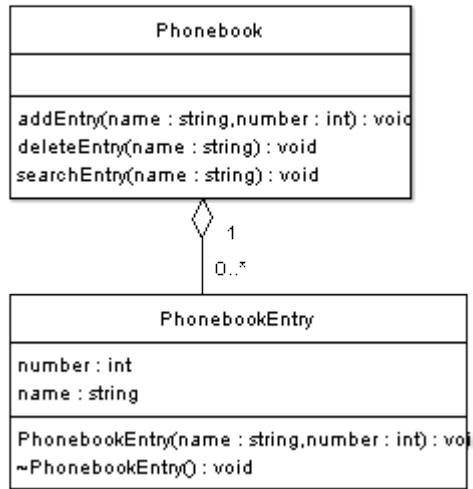


Figure 9: PIM for the phonebook program

elements in the PIM. These relations are manual, informal isomorphisms. The elements that are refined in the PSM can be related to the corresponding PIM elements with manual, informal refinements.

---

```

<ccdef export="addEntry">
  <ccsym cd="cpp.dec" name="public-type-function"/>
  <apply>
    <ccsym cd="cpp.types" name="funtype"/>
    <ccsym cd="cpp.types" name="void"/>
    <ccsym cd="cpp.types" name="string"/>
    <ccsym cd="cpp.types" name="int"/>
  </apply>
  <bind>
    <ccsym cd="cpp.proc" name="function"/>
    <bvar>
      <ccv name="name"/>
      <ccv name="number"/>
    </bvar>
    <rawcode format="cpp"><![CDATA[
      entryList .pushback(phonebookEntry(name, number))
    ]]>
  </bind>
</ccdef>
  
```

---

Figure 10: The CodeML element for the method addEntry().

During the implementation phase the C++ programmer writes the source code according to what is shown in the PSM. The relation between the PSM and the source code is again an informal, manual transformation relation that is also a translation and a refinement.



In our example the programmer makes a mistake in the implementation of the method `addEntry()`, which is depicted in Fig. 10. When he is done the next phase starts and the tester writes test cases to identify bugs in the software. One test case for the method `addEntry()` which fails is depicted in Fig. 11. Between the elements in Fig. 10 and Fig. 11 the testcase relation holds.

---

```
<rawcode format="cpp"><![CDATA[
pbook.addEntry("John Doe", 1234)
pbook.addEntry("John Doe", 5678)
]]>
```

---

Figure 11: Part of the test for the `addEntry()` method.

When the bugreport is added to the hybrid OMDOC document and related to the appropriate part of the source code, the person that is related to this part of the source code via the maintainer relation is notified. Now the programmer fixes the bug by modifying the method. He includes the error code in such a way that the `addEntry()` method now returns an integer, where the value -1 indicates that an error occurred. Now this small change propagates through our whole document:

- The changelog element for the class has to be updated by the programmer to document the change.
- The bugreport is now related to the source code via the `fixedbugrelation`.
- The code in the testcase element needs to be modified.
- It has to be determined whether the test now runs successfully.
- The return value of the method `addEntry()` in the PSM and the PIM has to be changed.
- Any part of the source code that calls the method `addEntry()` needs to be checked.

---

```
<Bugreport fixed="false">
The phonebook should not allow the addition of an entry with a name that already
exists . Therefore calling the method addEntry() twice with the same name should
result in an error .
</Testcase>
```

---

Figure 12: Bug report for the `addEntry()` method.

Even though the provided example is fairly simple, it can be seen that small changes to the source code can affect many different parts of the hybrid OMDOC document. The identification of all necessary actions becomes more vulnerable to human error the larger the software project that is dealt with becomes. Means to support this are necessary and can be implemented in tools acting on hybrid OMDOC documents.

## 6 Related work

In [25] the authors identify relations between JAVA source code given in JAVAML, a markup language for JAVA [3], and XMI and describe XSLT templates to reverse engineer JAVA to UML Class Diagrams. While their focus is on demonstrating that UML Class Diagrams can be automatically created from JAVAML by simply transforming the existing data (parsing of source code is not required), their work helps in identifying exactly what subset of the JAVA code is directly related to XMI elements and how implicit relations in JAVA code relate to explicit relations in XMI. Although in this report I presented CODEML as the markup language for representing source code, hybrid OMDOC documents might also contain JAVAML fragments, therefore it is possible to reuse the author's work within hybrid OMDOC documents.

In [4] a formal Validation and Verification framework is proposed where multiple formal specification languages (such as and VDM) are integrated into an existing modeling environment for UML, namely . The authors suggest specifying metamodels of the formal specification languages in UML, which can be mapped to UML models using model transformation languages, thereby providing automated and formal transformations. However manual transformations are not taken into account in their approach and the integration with supplementary documentation and source code is not discussed and appears to be too difficult to achieve in .

Another approach to integrate UML with formal specification languages can be found in [26]. They outline a close integration between UML and Z family languages (e.g. ) by using XSLT templates for automated conversion between ZML [29] and XMI. As the authors limit themselves to an integration of UML with Z family languages the problems I identified in the introduction arise, still their work can be integrated in hybrid OMDOC documents.

In [6] a taxonomy of software change is presented. It can be used as a framework to evaluate approaches to change management in software development, i.e. focusses on what should be provided by a sophisticated software change support tool rather than how such a tool should be designed. It is my belief that all aspects discussed in that paper can be taken into account when designing tools for hybrid OMDOC documents. In [4] an

## 7 Conclusion and Outlook

In this report I introduced hybrid OMDOC documents as a means to facilitate change management in software development processes. I discussed possibilities to represent source code and UML diagrams in such documents using semantic markup. I identified relations that hold between data produced at the different stages of the software development process and demonstrated how syntactic and semantic changes propagate along these relations.

As the taxonomy presented in section 3 was primarily developed to identify what relations there are in general in hybrid OMDOC documents, i.e. not with the direct intention to use them for change propagation, it is partly irrelevant to the impact analysis discussed in section 4 and needs to be further developed under this aspect. Furthermore a more thorough analysis of change relations in programming languages and supplementary documentation has to be developed. Also this report focussed on the intuition behind the relations, future work needs to concretize these notions and formalize them. On the basis of formalized relations a concrete representation for these relations in

hybride OMDOC documents needs to be developed.

The analysis of the integration of formal specification languages was only sketched in this report and needs to be examined in more detail. This means that possible representations of formal specifications in hybrid OMDOC documents need to be analysed and further elaboration on the relations to the other parts of hybrid OMDOC documents is necessary.

## References

- [1] ArgoUML - A UML design tool with cognitive support. <http://argouml.tigris.org>, seen at April 2008.
- [2] ATLAS transformation language. <http://www.eclipse.org/m2m/at1/>, seen at April 2008.
- [3] G. J. Badros. JavaML: a markup language for Java source code. *Computer Networks (Amsterdam, Netherlands: 1999)*, 33(1–6):159–177, 2000.
- [4] B. Baudry, C. Caston, and S. Ghosh, editors. *A formal VV framework for UML models based on Model transformation techniques*. Inria, France, 2005.
- [5] L. C. Briand, Y. Labiche, and L. O’Sullivan. Impact analysis and change management of uml models. *icsm*, 00:256, 2003.
- [6] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel. Towards a taxonomy of software change: Research articles. *J. Softw. Maint. Evol.*, 17(5):309–332, 2005.
- [7] Canonical XML. <http://www.w3.org/TR/xml-c14n>, seen at April 2008.
- [8] N. Correa and R. Giandini. A uml extension to specify model refinements, 2006.
- [9] Dia - A diagram creation program. <http://live.gnome.org/Dia>, seen at April 2008.
- [10] Eclipse Modeling Framework. <http://www.eclipse.org/modeling/emf/>, seen at April 2008.
- [11] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
- [12] T. J. Grose, G. C. Doney, and S. A. Brodsky. *Mastering XMI: Java Programming with XMI, XML and UML*. John Wiley & Sons, Inc., New York, NY, USA, 2001.
- [13] Michael Kohlhase. Code Markup Language. <http://www.omdoc.org/codeml>, seen at April 2008.
- [14] Object Management Group. Common Warehouse Metamodel. <http://www.omg.org/cwm>, seen at April 2008.
- [15] Object Management Group. Metaobject Facility. <http://www.omg.org/mof>, seen at April 2008.
- [16] Object Management Group. Unified Modeling Language. <http://www.uml.org>, seen at April 2008.

- [17] Object Management Group. XML Metadata Interchange. <http://www.omg.org/xmi>, seen at April 2008.
- [18] Object Management Group. <http://www.omg.org>, seen at April 2008.
- [19] Omondo - The live UML company. <http://www.omondo.de>, seen at April 2008.
- [20] Precision Graphics Markup Language. <http://www.w3.org/TR/1998/NOTE-PGML-19980410>, seen at April 2008.
- [21] Program Transformation Wiki. <http://www.program-transformation.org/>, seen at April 2008.
- [22] Query/Views/Transformations. <http://www.omg.org/cgi-bin/doc?ad/05-03-02>, seen at April 2008.
- [23] Relax NG Schema. <http://relaxng.org/>, seen at April 2008.
- [24] Sun Relax NG converter. <https://msv.dev.java.net/>, seen at April 2008.
- [25] C. Russell and R. Dewar. Xml encoded reverse engineering of java to uml, 2003.
- [26] J. Sun, J. S. Dong, J. Liu, and H. Wang. A formal object approach to the design of zml. *Annals of Software Engineering*, 13(1):329–356, 2002.
- [27] Scalable Vector Graphics. <http://www.w3.org/Graphics/SVG/>, seen at April 2008.
- [28] uml2svg - An XSLT-based tool for converting UML Diagrams to SVG. <http://uml2svg.sourceforge.net/>, seen at April 2008.
- [29] M. Utting, I. Toyn, J. Sun, A. Martin, J. S. Dong, N. Daley, and D. Currie. Zml: Xml support for standard z.
- [30] Vector Markup Language. <http://www.w3.org/TR/1998/NOTE-VML-19980513>, seen at April 2008.