Embedding Defect and Traceability Information in CIM- and PIM-level Software Models

Jörg Rech and Mario Schmitt

Fraunhofer IESE, Fraunhofer Platz 1, 67663 Kaiserslautern, Germany
+49 (0) 631 6800 2210, Joerg.Rech@iese.fraunhofer.de
+49 (0) 631 6800 2215, Mario.Schmitt@iese.fraunhofer.de

Abstract. Additional information about software models comes in different forms such as detected defects, used design patterns, traceability information to other abstraction levels, etc. In this paper, we presented how additional information about defects, context, traceability, etc. can be embedded into a UML- or BPMN-based software model. Furthermore, we presented a tool that uses the information about quality defects within a PIM to visualize defects directly in the software model diagrams.

Keywords: Quality Defects, PIM, CIM, Traceability, Defect Annotation, Traceability Annotation, Embedded Information, Software Models, MDSD

1 Introduction

Business users together with business analysts and architects generate the basic characteristics of a software system that results in computational independent models (CIM) including, e.g., role, product, or process models. In order to support the traceability of elements and of decisions made on the CIM-level to the PIM-level as well as the traceability of problems identified at the PIM-level to the CIM-level, we have to store additional information about the software model along with the software model.

Additional information about elements in a software model such as a PIM or CIM comes in different forms such as detected quality defects, used patterns (roles), traceability information to other abstraction levels, etc. This information is documented by users or automated mechanisms and has to be visualized in standard or special views of a modeling tool, made available to other systems for further analysis (e.g. impact analyses), or persisted over a long period of time.

Kolovos et al. [7] differentiate between external and embedded traceability information and made a decision towards the external approach. They argue against the embedded approach (based on stereotypes) as it does not support inter-model relations, pollutes the models, and degrading uniformity.

However, while several other options beside stereotypes are possible, storing additional complex information in a metamodel such as the UML [11] at PIM level is not straight-forward. An extension of the UML metamodel would result in non-standard models that are not exchangeable between tools. Besides, in order to apply similar mechanisms to models at different abstraction levels based on different (or previously unknown) metamodels, we need a generic approach that can be easily adapted to and complies with a broad range of metamodels.

In our context, the de-facto metamodel on PIM-level is the UML [11] which is built using the OMG's Meta Object Facility (MOF) meta-metamodel. MOF is a common modeling language kernel providing a unified basis for all OMG metamodels. On CIM-level modeling focuses on business process modeling using mostly the Business Process Modeling Notation BPMN [2] and the Business Process Definition Metamodel BPDM [1]. While BPMN provides just a graphical notation for process orchestration, BPDM, however, is a CIM-level metamodel for business process modeling, using BPMN as the graphical notation. Similar to UML, BPDM is based on the MOF [8] meta-metamodel.

Several solutions to the abovementioned problem of embedding information in software models are possible. In order to store the information in an Eclipse-based IME for PIMs, such as Topcased [10], and CIMs such as the BPMN-Editor of the SOA Tools Platform (STP) [4], we can persist our information as:

- *Markers/Properties* to the software model that are stored by the tool, but cannot easily be shared between users or across a versioning system (e.g., CVS)
- *MOF Annotations***,** a construct in MOF (a.k.a. MOF::Tag) that enable the multiple "tagging" of MOF elements with attribute-value pairs and can be shared across a versioning system,
- *Comments*, an element in the UML and many other metamodels (e.g., "Text Annotations" in BPMN) that can store one comment (i.e., text field) per element, but, typically, is used for comments by the modelers,
- *UML Stereotypes/Profiles*, an extension mechanism in UML that can be used to integrate additional elements into the UML. However, the information might be confused with, for example, domain-specific stereotypes and could floods the user with too much information, not necessary in the day-to-day work, or
- *External files*, similar to diagram interchange [3] files in Topcased, which use a semantic bridge to refer to elements of the software model(s). However, these files need to be used by the tools at work in order to synchronize changes to the model elements.

In order to enable the annotation of elements in a MOF-based software model, with respect to provide easily synchronizable and versionable information, we selected MOF Annotations to persist information about detected defects, context factors, and traceability information. Furthermore, the annotation mechanism allows embedding complex information within a software model using a XML schema to describe and structure the specific content for every specific annotation. The XML schema represents a metamodel that allowed us to define the substructures for the information on defects, traceability, etc.

2 A Metamodel for Defect and Traceability Annotations

While traceability and context information has to be annotated manually (for now), defects are identified by diagnostic mechanisms that analyze the system and find typical recurring problems, which have a negative effect on a quality aspect (e.g., maintainability, portability, or usability).

The five types of defect-related information embedded are: *Defect Annotations* (with information about the diagnosed quality defects), *Context Annotations* (with context information on used design pattern roles or special stereotypes, which is used to differentiate the diagnosis), *Decision Annotations* (with decisions such as “ignore” for individual diagnosed quality defects in case they are wrongly diagnosed or not removable in this specific location), *Symptom Annotations* (with information on the identified symptoms), and *Treatment Annotations* (which are used to store the treatments applicable for removing the diagnosed quality defects).

Traceability information uses just one type of annotation (*Trace Annotations*) that realizes traces from one element to one or more other elements (e.g., from one CIM element to multiple PIM elements (downward), from one PIM to multiple CIM elements (upward), or from one PIM to multiple other PIM elements (sideward)). Multiple types of references can be used *between abstractions* and *within abstractions*.

Furthermore, while single-location defects are enclosed within one abstraction at one element, *multi-location defects* do refer to other elements (resp. annotations) within the same model, and all defects might refer to elements in another abstraction level to document rationales for not removing a defect or to pinpoint a cause or (design) decision (e.g., in a CIM).

Figure 1 shows five aspects. On the specification-side (right) it outlines the generic approach as proposed by MOF for annotating model elements with additional information (metainformation) using the *Tag* entity. It introduces the base model elements of BPMN-based CIMs (*BPDM::Element*) and UML-based PIMs (*UML::Element*) both deriving from *MOF::Element* as common model element abstraction. The realization in eclipse either implements or maps (center of Figure 1) these concepts. For MOF, the element *MOF::Element* is mapped to the Ecore element *Ecore::EModelElement* of the Eclipse Modeling Framework (EMF) and *MOF::Tag* is mapped to *Ecore::EAnnotation*. Similarily, for CIM-modeling, the BPMN element *BPDM::Element* is mapped to the SOA Tools Platform (STP) project object *STP::BPMN::NamedBpmnObjects*. Finally, the element *UML::Element* of OMG’s UML are implemented (a one-to-one representation) by the Model Development Tools (MDT) project’s UML2 *MDT::UML2::Element*.

Furthermore, Figure 1 presents a XML-based metamodel (left) for defect- and traceability-oriented model metainformation and shows how actual metainformation is embedded using the tagging mechanism/within annotations. A metamodel similar to the traceability information metamodel is used by Feng et al. [6] for external traceability models.

A model element may have multiple annotations (*EAnnotations*) associated with it, each consisting of a *source* URI denoting an annotation’s type and an arbitrary number of *key/value* pairs. Following the structure of annotations, we bundle all model metainformation in a single annotation element, but distributing information internally to multiple *key/value* pairs according to the information’s scope/type (e.g. «Traceablity» or «Quality»). That is, *key* determines the type/scope of the XML-based metainformation assigned to *value*.

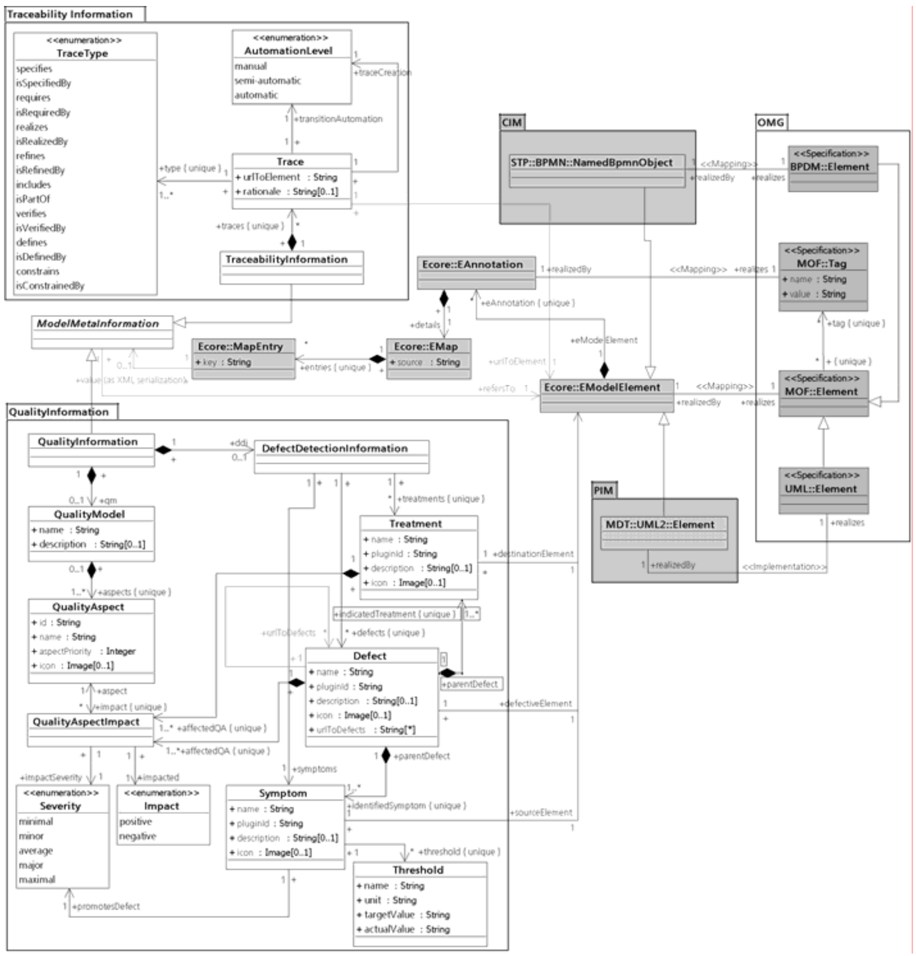

Figure 1 Metamodel for Quality and Traceability Information in CIM and PIM Models

Figure 2 gives a simplified exemplary XMI-serialization of a UML-based PIM model having a model element annotated with *quality information*. Quality information consists of *quality model* and *defect detection information*. According to the (non-functional) requirements a software system has to meet, a quality model defines and prioritizes mandatory *quality aspects* and thus, is the basis for interpreting/verifying the quality of a software model. In the context of VIDE-DD, determining the quality of a software model focuses on detecting *quality defects*. A quality defect represents a system-independent defect at one or more model elements with a negative *impact* on certain *quality aspects.* Defects are diagnosed on the basis of one or more quantifiable characteristics of a model or its model elements, so-called *symptoms*. The intensity to which symptoms promote related defects differs and depends, amongst others, largely on the characteristic's deviation from previously defined *threshold*(s). For removing a defect or mitigating a defect's (negative) impact on certain quality aspects, *treatments* refer to available techniques (e.g. refactorings). The exemplary annotation in Figure 2 illustrates the concept of *defect detection in-*

formation: A *Lazy Class* defect has been diagnosed for the PIM-level class *Opportunity* based on the *Number of Operations*. Hence, a negative impact on the declared quality aspect *Maintainability* is expected, treatable by applying an *Inline Class* refactoring.

```
<uml:Model>
 ...
 <packagedElement xmi:type="uml:Class" xmi:id="_CyIsaF-fEdySHqlLXw_Tew"
  name="Opportunity">
  <eAnnotations source="http://www.iese.fraunhofer.de/ModelMetaInformation">
   <details key="QualityInformation" value="
    <!-- BEGIN: Embedded QualityInformation XML-string -->
    <QualityInformation>
     <DefectDetectionInformation>
      <Defects>
       <Defect name="Lazy Class" description="Class Opportunity provides
                     not enough functionality to justify its existence."
        pluginId="de.fhg.iese.modeldefectdetection.diagnosis.lazyclass"
        defectiveElement="_CyIsaF-fEdySHqlLXw_Tew">
        <IdentifiedSymptoms>
         <Symptom name="Number of Operations"
          description="Number of operations is below threshold"
          pluginId="analysis.noo"
          sourceElement="_CyIsaF-fEdySHqlLXw_Tew"
          parentDefect="diagnosis.lazyclass" promotesDefect="major">
          <Thresholds>
           <Threshold name="Lower Threshold"
            unit="Integer" targetValue="6" actualValue="2"/>
          </Thresholds>
         </Symptom>
        </IdentifiedSymptoms>
        <AffectedQualityAspects>
         <QualityAspectImpact id="ISO9126_Maintainability"
          impact="negative" severity="major"/>
        </AffectedQualityAspects>
        <IndicatedTreatments>
         <Treatment name="Inline Class"
          description="Move all features of Opportunity into another class and delete it."
          pluginId=" refactoring.inlineclass"
          destinationElement="_CyIsaF-fEdySHqlLXw_Tew"
          parentDefect="diagnosis.lazyclass"/>
        </IndicatedTreatments>
       </Defect>
      </Defects>
     </DefectDetectionInformation>
     <QualityModel name="" description="">
      <QualityAspect id="ISO9126_Maintainability"
       name="Maintainability"
       description="The ease with which a software system or component can be modified…"
       aspectPriority="2"/>
     </QualityModel>
    </QualityInformation>
    <!-- END: Embedded QualityInformation XML-string -->
   "/>
  </eAnnotations>
 </packagedElement>
 ...
</uml:Model>
```

Figure 2 Serialization of Quality Information Annotation

Furthermore, we distinguish single- from multi-location defects [9]. A single-location defect (e.g. Lazy Class) affects one model element (e.g., a class), whereas multi-location defects apply to more than one element within the same model. For example a *Shotgun Surgery* defect is present, when, due to strong coupling of classes, a change in one class requires many subsequent changes in other classes. As each concerned class is annotated with defect information, it is necessary to interrelate this information, e.g. for eliciting and applying adequate treatments. Thus, *urlToDefects* (cf. Figure 1) allows for referencing related defects at other model elements.

A model element's *traceability information* comprises one to many *traces* to elements at both, different as well as same abstraction level. As presented in Figure 3, the key component of a trace is *urlToElement* for identifying related elements using a URL reference. The URL syntax is a path to the containing model repository, followed by a model identifier (the model's name) and the XMI-id of the model element. To qualify the relation of two elements linked by a trace, different types of references can be assigned to a) *traces between abstractions*, such as "realizes / is realizes by", "refines / is refined by", "specifies / is specified by", "requires / is required by", etc. and b) *traces within abstractions*, such as "includes / is part of", "verifies / is verified by", "defines / is defined by", "constrains / is constrained by", etc. (see [12] or [5]).

```
<uml:Model>
 ...
 <packagedElement xmi:type="uml:Class" xmi:id="_CyIsaF-fEdySHqlLXw_Tew"
  name="Opportunity">
  <eAnnotations source="http://www.iese.fraunhofer.de/ModelMetaInformation">
   <details key="TraceabilityInformation" value="
    <!-- BEGIN: Embedded Traceability Information XML-string -->
    <TraceabilityInformation>
     <Trace
      urlToElement="http://iese.fhg.de/SalesOpportunity_CIM.bpmn#_TG7coT3iEd2hQ-HeytPXvA"
      type="realizes"
      rationale="Implementation of Opportunity data object"
      traceCreation="automatic" transitionAutomation="automatic"/>
    </TraceabilityInformation>
    <!-- END: Embedded Traceability Information XML-string -->
   "/>
  </eAnnotations>
 </packagedElement>
</uml:Model>
```

Figure 3 Serialization of Traceability Information Annotation in PIM

As generative model-driven development relies on model transformations between abstraction levels, the information if a *trace creation* or *transition* between two related elements has been carried out manually, semi-automatically, or automatically is of interest for e.g. evaluating the quality of model transformations/model generators or determining the overall automation-level. The XMI-serialization of traceability information between an Opportunity data object at CIM-level and its implementation class at PIM-level is exemplified in Figure 3 (PIM-to-CIM) and Figure 4 (CIM-to-PIM).

```
<bpmn:BpmnDiagram>
 ...
 <artifacts xmi:type="bpmn:DataObject" xmi:id="_TG7coT3iEd2hQ-HeytPXvA"
  name="Opportunity">
  <eAnnotations source="http://www.iese.fraunhofer.de/ModelMetaInformation">
   <details key="TraceabilityInformation" value="
    <!-- BEGIN: Embedded Traceability Information XML-string -->
     <TraceabilityInformation>
      <Trace
       urlToElement="http://iese.fhg.de/SalesOpportunity_PIM.uml#_CyIsaF-fEdySHqlLXw_Tew"
       type="isRealizedBy"
       rationale="Implementation of Opportunity data object"
       traceCreation="automatic" transitionAutomation="automatic"/>
     </TraceabilityInformation>
     <!-- END: Embedded Traceability Information XML-string -->
   "/>
  </eAnnotations>
 </artifacts>
 ...
 </bpmn:BpmnDiagram>
```

Figure 4 Serialization of Traceability Information Annotation in CIM

3 Using (Defect) Annotations in Modeling Environments

The information stored within the annotations can be used, for example, by the diagram visualizer to enrich the standard UML diagrams with information about the defects. As presented in Figure 5 the VIDE Defect Detector (VIDE-DD) extends the Topcased modeling environment [10] and decodes the information within the annotation to decorate an element (e.g., a class) with a defect icon or to list all annotations to the user (see ⑥). This tool is aimed at enriching the visualization of the models in order to inform designers and maintainers about potential threats to model quality.

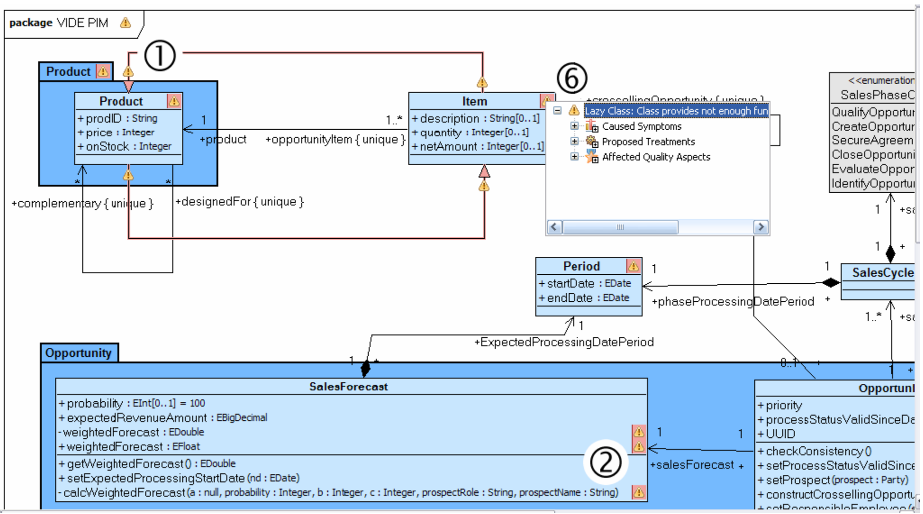

Figure 5 The VIDE Quality Defect Detector

4 Conclusion

We presented how additional information about defects, context, or traceability can be embedded in a UML- or BPMN-based software model (PIM or CIM) using Annotations. To structure the information within these annotations, we used a XML-based metamodel that supports single- and multi-location annotations from CIM-to-PIM, within PIM, and from PIM-to-CIM. Furthermore, we presented a tool that integrates quality defect diagnosis into the contemporary modeling environment Topcased and uses the annotations to present them in standard diagrams.

In the future, more tools for defect diagnosis and traceability support will be developed and integrated into software development tools that have to overcome the synchronization and versioning challenges. This is especially important for tools on the model level, as these have to support quality assurance in and traceability between multiple software and transformation models.

References

1. BPDM-Beta1, Business Process Definition MetaModel (BPDM), Beta 1, OMG Adopted Specification, OMG, dtc/07-07-01, 2007.
2. BPMN-1.1, Business Process Modeling Notation, V1.1, OMG, 2008.
3. DI-1.0, UML Diagram Interchange Specification, version 1.0, Specification, Object Management Group, Inc. (OMG), Needham, MA, USA, 2006.
4. EMF, "Eclipse Modeling Framework (EMF)," http://www.eclipse.org/modeling/emf/, last accessed on 1. April 2008.
5. A. Espinoza, P. P. Alarcon, and J. Garbajosa, Analyzing and Systematizing Current Traceability Schemas, In Annual IEEE/NASA Software Engineering Workshop (SEW), pp. 21-32, 2006.
6. Y. Feng, G. Huang, J. Yang, and H. M. Mei, Traceability between Software Architecture Models, In 30th Annual International Computer Software and Applications Conference (COMPSAC), pp. 41-44, 2006.
7. D. S. Kolovos, R. F. Paige, and F. A. C. Polack, On-Demand Merging of Traceability Links with Models, In 2nd EC-MDA Workshop on Traceability, 2006.
8. MOF-2.0, Meta Object Facility (MOF) Core Specification, version 2.0, Specification, Object Management Group, Inc. (OMG), Needham, MA, USA, formal/06-01-01, 2006.
9. J. Rech and A. Spriestersbach, Quality Defects in Model-driven Software Development, Deliverable, Fraunhofer Institute for Experimental Software Engineering (IESE), Kaiserslautern, D4.1, 2007.
10. TopCased, "Topcased IDE," http://www.topcased.org/, last accessed on 27 November 2007.
11. UML-2.1.1, Unified Modeling Language (UML), version 2.1.1, Object Management Group, Inc. (OMG), Needham, MA, USA, 2007.
12. A. von Knethen, Change-Oriented Requirements Traceability. Support for Evolution of Embedded Systems PhD Thesis. Kaiserslautern: University of Kaiserslautern, Department of Computer Science, 2002.