Metamodel for Tracing Concerns across the Life Cycle

Bedir Tekinerdogan, Christian Hofmann, Mehmet Aksit, Jethro Bakker

Department of Computer Science,
University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands
{bedir, c.hofmann, m.aksit, j.bakker}@cs.utwente.nl

Abstract Several aspect-oriented approaches have been proposed to specify aspects at different phases in the software life cycle. Aspects can appear within a phase, be refined or mapped to other aspects in later phases, or even disappear. Tracing aspects is necessary to support understandability and maintainability of software systems. Although several approaches have been introduced to address traceability of aspects, two important limitations can be observed. First, tracing is not yet tackled for the entire life cycle. Second, the traceability model that is applied usually refers to elements of specific aspect languages, thereby limiting the reusability of the adopted traceability model. We propose the concern traceability metamodel (CTM) that enables traceability of concerns throughout the life cycle, and which is independent from the aspect languages that are used. *CTM* can be enhanced to provide additional properties for tracing, and be instantiated to define customized traceability models with respect to the required aspect languages. We have implemented *CTM* in the tool *M-Trace*, that uses XML-based representations of the models and XQuery queries to represent tracing information. *CTM* and *M-Trace* are illustrated for a Concurrent Versioning System to trace aspects from the requirements level to architecture design level and the implementation.

1 Introduction

Several aspect-oriented approaches have been proposed to specify aspects at different phases in the software life cycle. At the programming level it appears that almost for every popular programming language there is now an aspect-oriented version in which crosscutting concerns are represented using dedicated language constructs. The early aspects domain has focused on defining approaches for modeling aspects at the level of requirements engineering and architecture design. Several design notations for representing aspects have been proposed in the context of aspect-oriented modeling using, for example, UML-based approaches. Aspects rarely occur in isolation. They are related to other artifacts within a phase or across multiple phases. Aspects can appear within a phase, be refined or mapped to other aspects in later phases, or may even disappear. Changes to an aspect can have consequences for other artifacts, which are directly or indirectly related to it. To assess the impact of a change to an aspect before it is made, it is necessary to have tool support for the storage and analysis of dependency relationships. Tracing aspects is necessary to support understandability and maintainability of software systems.

The concept of tracing implies usually following dependency relationships between artifacts. When developing large systems it is hard to identify the dependency relations.

Improving the traceability of artifacts supports not only the understandability but also is important for maintainability, adaptability and managing complexity. Traceability is a topic that is considered relevant and discussed in various domains. In requirements engineering lots of work has been done on tracing requirements from the stakeholders and in the design process [1–4]. In the model-driven engineering approach [5,6] traceability is considered important for tracing model elements. A reference model for requirements traceability is provided in [3]. Here a simple metamodel for traceability models is defined and elaborated for requirements traceability. In [7] a UML-based metamodel for requirements traceability is presented that is translated into a UML profile. In [6] a traceability model and a UML profile is presented that include both requirements and model elements. In [5] a metamodel is defined for traceability of models in model-driven development. Hereby, tracing is defined in the transformation specification.

The problem of traceability has recently also been addressed by the AOSD community [8]. The AOSD community encompasses the adoption of aspects throughout the lifecycle and for each phase aspects are specified. Although several initial approaches for traceability have been introduced to address traceability of aspects, two important limitations can be observed. *First*, tracing is tackled within a phase or does not cover the entire life cycle yet. For example, tracing of aspects has been defined within the requirements analysis phase [9], from requirements to architecture [10] and from architecture to design [11]. *Second*, the traceability model that is applied usually is focusing on elements of specific aspect languages. The selection of tracing properties, however, might be dependent on the corresponding project requirements. The traceability model must be therefore sufficiently generic to cope with the different aspect-oriented approaches.

We propose the concern traceability metamodel (CTM) that enables traceability of aspects throughout the life cycle, and which is independent from the aspect languages that are used. CTM can be enhanced to provide additional properties for tracing, and instantiated to define customized traceability models with respect to the required aspect languages. We have implemented CTM in the tool M-Trace, that uses XML-based representations of the models and XQuery queries to represent tracing information. CTM and M-Trace are illustrated for a Concurrent Versioning System to trace aspects from the requirements level to architecture design level and the implementation.

The remainder of the paper is organized as follows. In section 2 we will shortly discuss the background on traceability. In section 3 we present as an example a concurrent versioning systems (CVS) and illustrate on it the need for tracing crosscutting concerns. In section 4 we define the requirements for concern traceability. Based on these requirements we will propose the concern traceability metamodel that will be explained in section 5. The CTM can be implemented in various ways. In section 5.3 we will present an implementation using XML. We will discuss then the application of CTM to trace aspects in and accross phases of the software development lifecycle in section 7. Section 8 will finalize the paper with the conclusions.

2 Terminology

In the following we describe the basic terms that we adopt throughout the paper.

- *Artifacts* – We adopt here the view from [12] in which artifacts are defined as workproducts of the software development lifecycle, such as models, source code or other documents related to the development of a software system.
- *Units* – Units are used to represent artifacts and can have different granularity, ranging from, for example, a sourcecode-file to a single statement. In essence, the structure of the artifacts can also be reflected in units.
- *Concerns* – Concerns can be generally defined as a matter of interest, or design intentions of corresponding stakeholders [13]. Concerns are usually implemented in artifacts.
- *Crosscutting concern* – Concerns that cannot be easily mapped to a single artifact and are scattered over multiple artifacts are called crosscutting concerns. Typical examples for crosscutting concerns are security, reliability and concurrency concerns.
- *Aspect* – An aspect is an artifact that implements a crosscutting concern.
- *Dependency relation* – Artifacts are conceptually dependent on the concerns. As such a change to a concern, for example, will impact also the artifacts that are dependent on the concern.
- *Traceability Link* – To enhance traceability of artifacts the dependency relations among them need to be recorded. We call a recorded dependency relation a traceability link.

3 Example: Concurrent Versioning System (CVS)

To analyze the impact of concerns a Software Configuration Management (SCM) example will be used as a case study. The SCM deals with control of software changes, proper documentation of changes, the issuing of new software versions and releases, the registration and recording of approved software versions. An important functionality in SCM forms the concurrent version control system (CVS), which keeps a history of the changes made to a set of files that can be concurrently accessed.

3.1 The Life Cycle Phases of CVS

To illustrate the traceability throughout the life cycle we have defined (1) requirements model, (2) architecture design, (3) design, and (4) implementation of the concurrent version control system. Figure 1(a) shows the requirements for the CVS and Figure 1(b) the layered architecture. This architecture consists of three major layers: client's layer, session layer, and data layer. The client layer represents the programmer's environment and provides a set of programming tools, such as compilers, interpreters and debuggers, editors and tools for integrating program modules into a consistent program. When a programmer wants to edit a file which is stored in the project repository, a request is made to the session manager in the session layer. The session manager associates a timestamp with it, and initiates an editing session by calling on the request handler in the data layer. The session layer also includes administration functions providing a set of management tools. The administrator further includes a performance monitoring module that is used to generate reports on the average time of accesses, the effect of

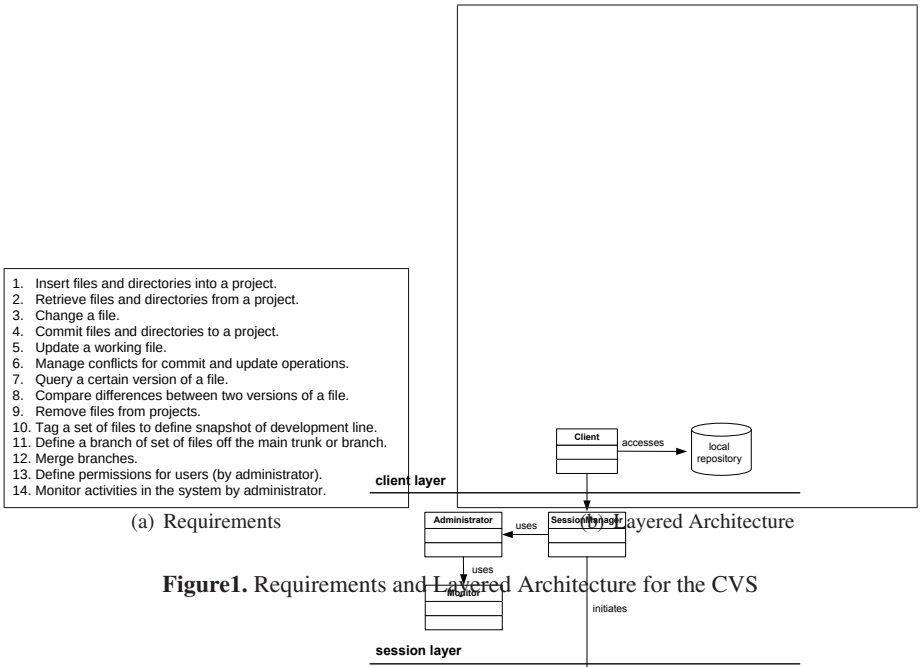


Figure1. Requirements and Layered Architecture for the CVS

data size and simultaneous accesses to performance, number of aborts, etc.

The request handler in the data layer *checks out* the requested file and passes it to the programmer's environment. When files are checked out they can be edited and compiled and *check in* the modifications to the file. Checking out a file does not give a programmer exclusive rights to that file. Other programmers can also check it out, make their own modifications, and check it back in. The concurrency control module administrates all the simultaneous accesses to the same file. This module is also responsible in identify-

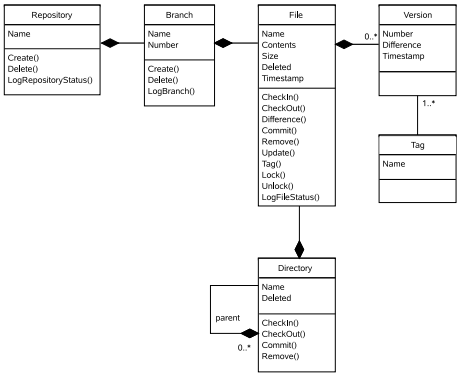


Figure2. Object-Oriented Design of the CVS System

ing the read/write conflicts in accessing the files. If a conflict is detected, the integration manager is called. The integration manager provides a set of functions to resolve the conflicting accesses. The version manager generates version ID's, compares versions of files and notifies if there are inconsistent versions in the same configuration. Based on this architecture (part of) an object-oriented design of the CVS system is shown in Figure 2. Besides of the requirements specification, architecture design and the design we have also an implementation of the system in AspectJ [14]. The program includes nine classes and two aspects implementing the concerns versioning and monitoring.

4 Requirements for Concern Traceability

Based on the work on the literature on traceability and the concern modeling in AOSD we provide a set of requirements for traceability of concerns. To illustrate the problem we will first define a set of change scenarios for the CVS in section 4.1. In sections 4.2 to 4.4 we list the requirements for traceability of concerns throughout the lifecycle.

4.1 Change Scenarios for CVS and Observed Traceability Problems

We consider the following change scenarios for the CVS system we described in the last sections:

- *Update Versioning*
  In the current design, versioning is only implemented for files but this should be enhanced to support versioning for directories as well. With file versioning, the system allows you to track every change made to any file by saving a copy (version) of a file each time that file is saved. However, versioning on a directory merely represents the default versioning setting for all files created within that directory. This scenario will impact the artifacts in the different phases. For the requirements analysis this will impact at least the requirements 4, 7, 8 as listed in Figure 1(a). In the architecture in Figure 1(b) it will impact the *VersionManager* module but also other modules that are directly or indirectly related to versioning. In the object-oriented design and the implementation this will require changing several classes.
- *Add Security*
  In the initial design the session manager starts an editing session by calling on the request handler in the data layer. To ensure that data is only accessed by the corresponding authorized person security needs to be added to the CVS. This means, that for example, before the session manager initiates a session it should authorize the request of the client. All the artifacts in the phases that depend on the security concern, that is, authentication and authorization in this case, will need to be enhanced to implement the required concern.

The above scenarios indicate that in the given (object-oriented) design it is hard to follow all the dependency relations within and across phases. The following sections define the requirements for achieving traceability of concerns in general and aspects in particular.

4.2 Explicit Modeling of Concerns

In order to explicitly reason about traceability of the concerns it is necessary that the corresponding concerns are explicitly modeled as first class abstractions. The detail of concern model could range from just a description of its name to a full semantic model including attributes such as stakeholder, the domain of the concern, the date it was raised at, the impact that it has, etc.

Harrison et al. [12] define the following requirements for concern modeling: (a) providing modeling concepts for *concerns* and their organization (b) Neutrality and open-endedness with respect to the kind of artifacts, (c) and specification that captures intended structure of material rather than simply reflecting existing structure.

If we decide to explicitly model concerns then the question arises whether to provide a uniform model for both the concerns and artifacts, or explicitly separate these using dedicated language constructs. In general these two different approaches are identified as symmetric and asymmetric approach [15]. In the symmetric approach one adopts a single concern model to represent both the concerns and the artifacts. Note that hereby concerns are still represented explicitly, and this is different from a language which only provides uniform modeling notations for artifacts but which does not consider concerns explicitly. In the asymmetric approach separate from the artifacts, concerns are modeled using their own abstraction mechanisms.

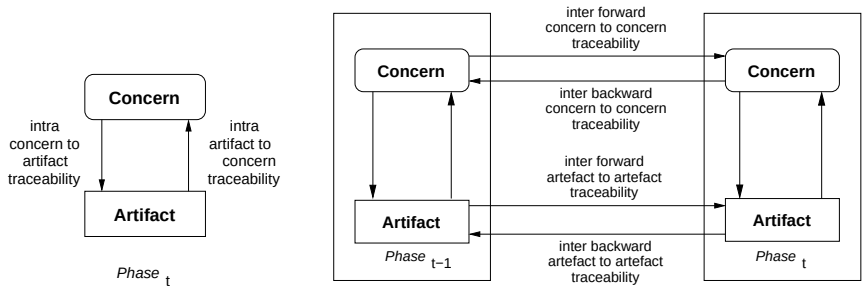
4.3 Explicit Modeling of Dependency Relations

Assuming that concerns are related to concerns or artifacts, it is necessary to make these relations explicit. This can be only done when dependency relations are recorded as traceability links. For this traceability should be specified as first class abstractions in the adopted traceability model. The choice for a symmetric or asymmetric approach has also an impact on the traceability links. In the asymmetric model, the traceability links will need to be established for both artifacts and concerns. On the other hand, in the symmetric approach the traceability links need to refer to one type of concern. This simplifies the traceability specification but could reduce understandability because the user has to explicitly distinguish between concerns and artifacts.

4.4 Support for Traceability Within and Across Phases

Obviously, concerns can occur in various phases of the life cycle such as requirements analysis, architecture design, design or programming. Tracing should be supported within and across life cycle phases, as we will explain in the following.

Intra-phase Traceability To understand the relations among the concerns and artifacts within the same phase it is necessary to model traceability for the given phase. Figure 3(a) shows the abstract model for traceability within a phase *t*. Here we have shown the case of an asymmetric model and distinguished between an artifact and a concern. We define two types of intra-phase traceability: *intra concern to artifact traceability* and *intra artifact to concern traceability*.



(a) Within Phases (b) Across Phases

Figure3. Traceability Relationships

Inter-phase Traceability Figure 3(b) presents the abstract model for traceability across software development lifecycle phases. Four types of relations are defined that we think are necessary. In alignment with the literature on traceability we make here a distinction between *forward* and *backward traceability*. Forward traceability defines traceability relationships between a concern or artifact in one phase to another concern or artifact in a *later* phase. Backward traceability defines the traceability relationships between a concern or artifact in one phase to another concern or artifact in a *previous* phase. To distinguish from intra phase traceability we use the term *inter* referring to relations across different phases.

Traceability relationships accross different phases can also be distinguished into forward and backward traceability relationships. These relations are defined from a concern to an artifact, or from an artifact to a concern. It should be noted that, for example, *inter forward concern to artifact* traceability relationships can be traced, even if corresponding trace-links are not explicitly specified. The trace-link can be inferred, for example, from the existence of an *intra concern to artifact* trace-link and an *inter forward artifact to artifact* trace-link for an artifact in a given phase. From this perspective we can state that the traceability relations of Figure 3 represent the primitive traceability relations.

4.5 Support for Automated Tracing Using Queries

The explicit models for concerns and traceability links help to automatically determine relationships between the different concerns and artifacts. Although relationships among concerns can only be determined in an implicit manner by following the trace links between concerns and artifacts. Following the traceability links manually might not be trivial for a complex system, even though the traceability links are made explicit. The management of models and dependencies, as well as tracing the dependencies between model elements and concerns should be automated. In order to minimize the amount of trace data presented to the user and to select only relevant concerns, it should be possible to define generic queries that determine the set of elements for which a trace is calculated.

5 CTM: Concern Traceability Metamodel

In the following we present the concern traceability metamodel for tracing concerns throughout the life cycle. CTM is symmetric with respect to making no distinction between crosscutting and non-crosscutting concerns, but asymmetric in distinguishing between artifacts and concerns, though artifacts may introduce new concerns, as we will discuss later.

5.1 The Traceability Metamodel

The concern traceability metamodel CTM, which is shown in Figure 4, models concerns as parts of a concern model. The concern model is represented in our metamodel by the meta-class *ConcernModel* that consists of one or more instances of *ConcernGroup* and *UnitModel*, as shown on the left. An instance of *ConcernModel* would be, for example, a concern model like the one used in the Concern Modeling Environment (CME), which is described in [12]. We assume in our metamodel that concerns, which are modeled by the meta-class *Concern*, are grouped into concern groups. A concern group corresponds in AspectJ, for example, to a package that contains a set of aspects. We consider aspects in our metamodel as a kind of concern. *CrosscuttingConcern* is therefore a subclass of *Concern*. The source of concerns, as we required for traceability, is explicit in our model. Each concern is associated with one or more stakeholders, represented by the meta-class *Stakeholder,* and a stakeholder has one or more concerns.

The right hand side of Figure 4 shows the part of our metamodel that is used to trace units and concerns. Before we explain this part of the metamodel, however, we need to take a closer look at the unit model. An instance of meta-class *UnitModel* is a model used in a phase of the lifecycle. For example structured text documents, for the requirements engineering phase, Architecture Description Languages (ADLs) and UML-class diagrams, for the architectural design phase, and Java sourcecode files, for the implementation phase. Several such models may be used simultaneously in one phase, like in the Rational Unified Process, where UML-Class Diagrams, UML-Sequence Diagrams, and so on, are used to model the units interesting for architectural design.

The unit model consists of one or more instances of the meta-class *Unit,* like use-case, connector, class, and import-statement, to name only one example for each lifecycle phase. As we already mentioned, are units representations of artifacts. Therefore has the meta-class *Unit* the attributes *reference* and *name* that allow referring to the artifact. Sub-units may be associated through the *parent* relationship. Examples for units and their corresponding sub-units in an object-oriented design document are: class, class-variables, instance-variables, methods and so on.

5.2 Traceability of Crosscutting Concerns

Supporting traceability of crosscutting concerns in our metamodel requires, besides explicit trace-links, also an explicit model of aspects. Our metamodel should be independent of the implementation of aspects and the particular aspect model (symmetric

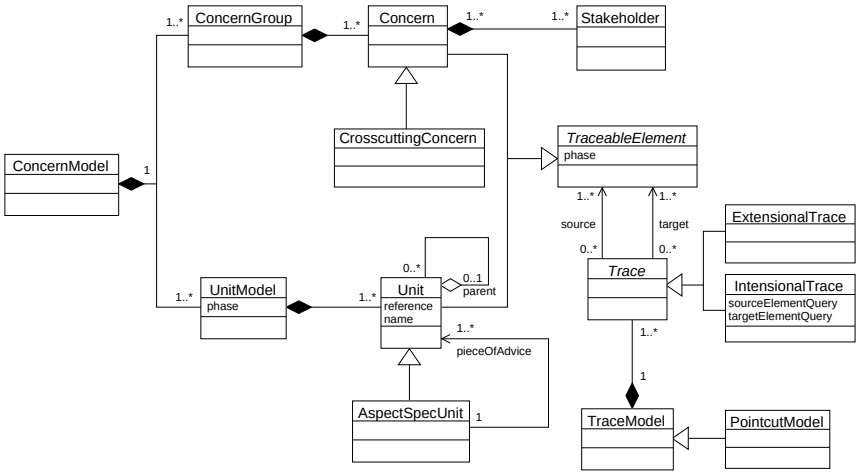


Figure4. Concern Traceability Metamodel

or asymmetric) that was chosen. Therefore, we distinguish crosscutting concern from aspect specification [1] and pointcut model.

An aspect specification is a unit of the artifact model and is represented in Figure 4 by the meta-class *AspectSpecUnit*, which is a subclass of *Unit*. An instance of an *AspectSpecUnit* would be, for example, a piece of Java code containing an AspectJ aspect declaration, or an aspect class in an AODM model [16] of a system's design. The relationship *pieceOfAdvice* represents a piece of advice (which is also a unit in terms of our metamodel) that implements a crosscutting concern of the system or model of the system.

We stated in Section 4 that traceability of concerns throughout the lifecycle requires an explicit representation of traceable elements and trace links. We represent these in our metamodel by the abstract meta-classes *TraceableElement* and *Trace*. Each instance of the meta-classes *Unit*, *AspectSpecUnit*, *Concern* or *CrosscuttingConcern* is a kind of traceable element. We explicitly model the phase in the software development lifecycle the traceable element belongs to. Also the trace relation is modeled explicitly by the meta-class *Trace* and the relations *source* and *target*. These relate one or more traceable elements that are in the domain of the relation to one or more traceable elements that belong to the codomain of the relation. Traces can be specified extensionally by listing all source-target mappings between traceable elements, or intensionally by a query. A query can compute, for example, the set of target elements associated with a source element. Extensional and intensional traces are represented in the metamodel by the meta-classes *ExtensionalTrace* and *IntensionalTrace*, which are sub-classes of *Trace*. The queries associated with an instance of meta-class *IntensionalTrace* are represented by the attributes *sourceElementQuery* and *targetElementQuery.* This allows us to effectively represent *m:n* traceability relationships by queries that compute the source and target elements. Still, *1:n* and *n:1* traceability relationships can be represented easily in

[1] The way the crosscutting concern is specified in a specific aspect language like AspectJ.

our meta-model, without having to write queries for single source- or target elements. For example, the target elements of a *1:n* relationship can be calculated by a query and the reference to the single source element can be simply listed.

Traces are modeled as parts of a trace model, represented by the meta-class *TraceModel*, which makes it possible to specialize it. This can be used to explicitly represent the different traceability relationships we identified in Section 4.4. We also regard pointcut models, represented by the meta-class *PointcutModel* in Figure 4, as a kind of trace model. A pointcut model is a model of a pointcut designator that allows identifying the units to which the piece of advice from an aspect specification unit applies. Part of the aspect declaration is a pointcut designator expression, formulated in some pointcut language. The semantics of the pointcut in a specific pointcut language is modeled by instantiating the meta-class *PointcutModel*. Because both units and concerns are traceable elements, the metamodel poses no restrictions on the type of elements that are composable. It is therefore element symmetry neutral [15] and therefore all kinds of compositions of components, aspects, or aspects and components can be modeled. The meta model is also neutral with respect to join-point symmetry, because *any element* of a model can be related with a trace link.

Selecting any model element with a query allows also to establish a mapping between the aspect specification and the units into which the piece of advice is woven. As a result remains the relation between an aspect specification and the woven artifacts, like model or code, traceable. Together with the trace link from the crosscutting concern to the aspect specification unit it is possible to determine the units to which a crosscutting concern applies. Crosscutting concerns remain therefore traceable; even in later lifecycle stages, where they would have normally become invisible after weaving. The pointcut model is not explicitly part of an aspect specification in our metamodel. Instead, remains the element where the pointcut specification resides implicit. Our meta model is therefore also neutral with respect to the symmetry of the composition relationships. This allows to instantiate the meta model for aspect languages where composition relationships can be part of another element. For example Hyper/J, where composition rules are at least conceptually not part of the crosscutting concern. They are, however, part of the aspect specification. In a truly composition relationship symmetric aspect language, aspect specification and pointcut designator declaration would be separate. Furthermore would be pointcut designators needed that can compute join-points for arbitrary element types. These can also be expressed in our meta-model by using the meta-class *IntensionalTrace*. The queries associated with this meta-class are formulated for instances of type *TraceableElement*, which can be concerns and arbitrary units of some artifact of the software development lifecycle.

5.3 Utilizing CTM

Figure 5 represents the process for utilizing the CTM. Traceability models can be instantiated directly from the CTM without restrictions to a specific syntax. It is also possible to first enhance CTM, resulting in a customized meta-model that is then later on instantiated. New types of dependencies between traceable-elements, for example, can be added by specializing the meta-class trace-link. This way we can explicitly representat relations like, specializes, includes, eliminates or supports in the traceability

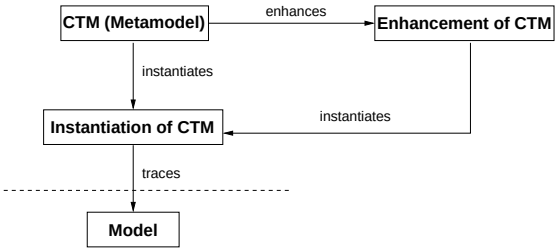


Figure5. The Process for Utilizing CTM

meta-model. If such a restriction is not wanted, then all these relations can be introduced at the model level by first instantiating CTM and enhancing the resulting model.

6 M-TRACE: Implementation of CTM

We have implemented CTM in our tool M-TRACE that uses XML models to instantiate CTM and represent concerns, artifacts and trace links. Using the eXtensible Markup Language (XML) to instantiate CTM has the advantage that the instantiated model can be easily used together with standard tools for creating, editing and transforming XML files. Such tools can also be used to transform the representation into other models. We instantiate CTM by defining the XML-document structure according to the meta-model in the Document Type Definitions (DTD) shown in Figure 6-9. These document type definitions also contain enhancements with respect to the meta-model. They introduce, for example, comments to ease readibility, unit types to distinguish between different kinds of artifacts, and unique identifiers that are needed to represent units and concerns. We will now explain the instantiation and representation of CTM using XML, as well as the enhancements we added.

A concern model element (Figure 6) consists of an optional description, one or more unit models, and concern groups. The comma identifier, called a sequence, specifies the

```
1 <!ELEMENT concernmodel (description?, concerngroup+, unitmodel+)>
2 <!ELEMENT description (#PCDATA)>
```

Figure6. Concernmodel Structure

order in which the elements must occur. The plus indicates that one or more elements are required. We declare that several unit models may be used, because an architectural model of the system, for example, may be comprised of several architectural views.

6.1 (Crosscutting) Concerns

The element type declaration for the element *concerngroup* is shown in Figure 7, Lines 1-4. A *concerngroup* element consists of one or more crosscutting- or non-crosscutting

```
1  <!ELEMENT concerngroup ((concern | crosscutting-concern)+)>
2  <!ATTLIST concerngroup
3    name CDATA #IMPLIED>
4
5  <!ELEMENT concern (description, stakeholder+)>
6  <!ATTLIST concern
7    id CDATA #REQUIRED
8    phase CDATA #REQUIRED
9    name CDATA #REQUIRED>
10
11 <!ELEMENT crosscutting-concern (description, stakeholder+)>
12 <!ATTLIST crosscutting-concern
13   id CDATA #REQUIRED
14   phase CDATA #REQUIRED
15   name CDATA #REQUIRED>
16
17 <!ELEMENT stakeholder (#PCDATA)>
```

Figure7. Concerngroup Structure

concerns. To make the XML-model more readable, an optional name (Lines 3-4) may be given to the concern group. This is actually an extension of CTM. We will also extend the instances of the meta-classes *Concern* and *CrosscuttingConcern*; namely the element type definitions *concern* and *crosscutting-concern* in Lines 6-17. The definition states that a concern element consists of a description and one or more stakeholder elements. The description is the first extension with respect to the meta-model. Another extension is adding an *id* attribute (Line 8). We need this attribute, because it is not possible to specify an XML model in an object-oriented manner. Thus, we have to use an explicit identifier to mimic the object identities of the traceable elements. Explicit modeling of the inheritance relationship is also not possible. We therefore just duplicate inherited relationships, and all inherited attributes like *phase* in the definition of the element attributes in Line 8-10 and Line 14-17. The fact that a crosscutting concern is a kind of concern remains implicit.

6.2 Unitmodel

The element type definition for the element *unitmodel*, which is an instance of the correspondingly named CTM meta-class, is shown in Figure 8 in Line 1. It states that the unit model element must contain one or more *unit* elements or *aspect-specunit* elements. Because a unit model is used in a specific phase of the software lifecycle, for example, ADLs in the software architecture design phase, we have to define a *phase* attribute (Lines 2-3) in the XML model as well. Lines 5-11 describe the *unit* element and Lines 13-20 the element for defining an aspect specification unit. Both may have child units, which are defined in the *child-units* element definition in Line 22 and 23. *Child-unit* elements consist of one or more *unit* or *aspect-specunit* elements. Again, it becomes not apparent that an aspect specification unit is a kind of unit.

This means, we also have to mimic object identities with an *id* attribute. We define it for *unit* and *aspect-specunit* elements in the Lines 7 and 16 respectively. The element type definitions define phase, reference and name attributes, as required in the meta-model. We extend CTM again, by adding an attribute *type* to the element *unit* and *aspect-specunit*. The element *aspect-specunit* may also contain child-elements.

```
1  <!ELEMENT unitmodel ((unit | aspect-specunit)+)>
2  <!ATTLIST unitmodel
3    phase CDATA #REQUIRED>
4
5  <!ELEMENT unit (child-units?)>
6  <!ATTLIST unit
7    id CDATA #REQUIRED
8    phase CDATA #REQUIRED
9    reference CDATA #REQUIRED
10   name CDATA #REQUIRED
11   type CDATA #REQUIRED>
12
13 <!ELEMENT aspect-specunit (child-units?, piece-of-advice+)>
14 <!ATTLIST aspect-specunit
15   id CDATA #REQUIRED
16   phase CDATA #REQUIRED
17   reference CDATA #REQUIRED
18   name CDATA #REQUIRED
29   type CDATA #REQUIRED>
20
21 <!ELEMENT child-units (unit | aspect-specunit)+)>
22
23 <!ELEMENT piece-of-advice (unit | aspect-specunit)>
```

Figure8. Unitmodel Structure

As presented so far, do *aspect-specunit* elements not differ much from the *unit* elements according to their definition. The only differences are, that *aspect-specunit* elements may contain one or more pieces of advice represented by a *piece-of-advice* element. The relation between pieces of advice and the units that it applies to is represented as a traceability-relationship. We will explain this later when we define the XML-representation of the trace model.

A *piece-of-advice* element (Lines 25-26) can consist of a unit or an aspect specification unit. As mentioned before, allows this the specification of arbitrary compositions and makes therefore also the instantiation of CTM as XML model, element-symmetric.

6.3 Queries

One of our goals was to model traceability of aspects independent of the aspect model used. This requires to separate the pointcut model from the aspect specification units, and to model traces explicitly and independent of the traceable elements. The trace model is defined in Figure 9. A *tracemodel* element consists of at least one *intensional-trace* or *extensional-trace* element, as we define in line 3. The *ExtensionalTrace* meta-class in CTM has references, called source and target, to the meta-class *TraceableElement*. We define the corresponding element definitions in Line 6, where we define that an *extensional-trace* element consists of a source and target element. The source and target elements contain one or more elements *traceable-element*. We can identify the traceable elements referenced by the trace using the attribute *id*, as defined in Line 13. To make manual editing easier we allow an optional description that can be added to a traceable element. Intensional trace-links, where we calculate the elements belonging to the trace-link, are modelled in XML as defined in Line 6. An *intensional-trace* element consists of a *source* or *source-query* and a *target* or *target-query* element. The source-

```
3  <!ELEMENT tracemodel ((extensional-trace | intensional-trace)+)>
4
5  <!ELEMENT extensional-trace (source, target)>
6  <!ELEMENT intensional-trace ((source | source-query), (target | target-query))>
7
8  <!ELEMENT source (traceable-element+)>
9  <!ELEMENT target (traceable-element+)>
10
11 <!ELEMENT traceable-element (description?)>
12 <!ATTLIST traceable-element
13   id CDATA #REQUIRED>
14
15 <!ELEMENT description (#PCDATA)>
16
17 <!ELEMENT source-query (#PCDATA)>
18 <!ELEMENT target-query (#PCDATA)>
19
20 <!ELEMENT pointcutmodel ((extensional-trace | intensional-trace)+)>
```

Figure9. Trace Model DTD

query and target-query elements contain the text that describes the XQuery, which calculates the source- and target elements. XQuery is a technology under development by the W3C, that is designed to query collections of XML-data.

Intensional- and extensional trace are also part of the *pointcutmodel* element and are used there as a kind of pointcut mechanism. The query is then used to select the units out of the unitmodel to which a piece of advice should be applied. We will usually use queries as pointcut designators, although, the pointcut could be designated by enumerating all the elements. Queries are a very powerful mechanism, since XQuery is a full-blown functional programming language with strong typing. The evaluation of the query expression reads a sequence of XML fragments or atomic values and returns a sequence of XML fragments or atomic values that are the query result. The simplest kind of query is an XPath expression.

For example, *'/concernmodel[@phase=RA]/unitmodel/unit'* selects all units of the concern model from the requirements analysis phase, which we will introduce in Section 7. To compute the trace links, we have defined queries that can be used to select parts of the unit and concern model. Figure 10 shows in Line 1, for example, a query

```
1  f:getUnitByName("RA"," ","Insert File")
2
3  declare function f:getUnitByName($phase,$root,$name) as element()* {
4    let $units := f:check($phase,$root)
5     for $i in $units
6      return  if($i[@name=$name]) then $i else f:getUnitByName($phase,$i,$name)
7  };
8
9  declare function f:check($phase,$root) as element()* {
10    if($root instance of element())
11     then $root/unit union $root/aspect-specunit
12     else /concernmodel/unitmodel[@phase=$phase]/unit union
13            /concernmodel/unitmodel[@phase=$phase]/aspect-specunit
14  };
```

Figure10. Query of an Intensional Trace

statement that returns all units corresponding to the requirement statements about the insertion of files into the CVS. The parameter *"RA"* indicates that we want to search the unitmodel of the requirements analysis phase. A unit can correspond to a section in the requirements document, which describes the insertion of files for example, because it can reference an artifact. To query all units with a name equal to *"Insert File"*, we have to specify the function: *'f:getUnitByName'* as depicted in Lines 3 to 11. This function recurses over all units (Line 6) until a unit with the same name as the parameter *'$name'* is found. It returns then the selected unit. The function *check* (Lines 9-13) is used to test if the parameter *$root* is an XML-element or just plain text. If the root element has been determined, all unit-elements starting from the element *$root* are returned. If no valid XML-element was passed, all unit-elements and all aspect-specunit elements in the unitmodel are returned that were defined for the phase *$phase*.

7 Application of CTM

After defining the structure of the XML-documents, we will show in the following how they can be used to model concerns and artifacts of the requirements analysis phase. Furthermore, we will show how they can be used to define intra-phase traceability links for the CVS case we introduced in Section 3. Architecture design, aspect-oriented design, and aspect-oriented implementation can be modeled accordingly. Due to space limitations we will only show an excerpt of the whole XML-model that was made for the first change scenario, i.e. updating versioning. In Section 7.3 a short summary is given, of how the presented models can be used to trace the impact of change scenarios.

7.1 Concern Modeling and Tracing within a Phase

Figure 11(a) shows the concern model for the requirements of CVS. We structured the concern model using concern groups. The example shows only two concerns of that concern group: *retrieve file or directory*, and the crosscutting concern *synchronization*. These are examples for concerns that the requirements related to information stored in files and directories depend upon. The unitmodel that follows the concern group, shows an example of a unit for a requirement from Figure 1(a). The dependencies between concerns and requirements are specified by the trace links in Figure 11(b). We use XQuery statements or explicitly list the units that are related to each other. The first intensional trace link in 11(b) specifies , for example, that the concern *"Retrieve file or dir"* with identity *id="r2"*, is addressed by the requirement corresponding to the unit with *id="2"*.

7.2 Tracing Across Lifecycle Phases

To trace concerns across the lifecycle phases, units and concerns have to be defined for each phase in the same way as shown for the requirements phase. The architectural module *VersionManager* from the architectural model of the CVS (Figure 1(b)), for example, can be represented with our XML model as shown in Figure 12(a). The reference to an XMI representation of the UML model depicting this architecture, could

```
<concernmodel phase="RA">
<concerngroup>
 <concern id="r2"
   phase="RA" name="Retrieve file or dir">
   <stakeholder> User </stakeholder>
 </concern>
 <crosscutting-concern id="r16"
   phase="RA" name="Synchronization">
   <description>Synchr. activities </description>
   <stakeholder> Admin </stakeholder>
   <stakeholder> User </stakeholder>
 </crosscutting-concern>
...
</concerngroup>
<unitmodel phase="RA">
 <unit id="2" phase="RA"
   reference="-//RequFileRetrieval.doc"
   name="Retrieve file"
   type="requirements statement">
 </unit>
...
</unitmodel>
...
</concernmodel>
```

(a) Concern Model

```
<tracemodel>
 <intensional-trace>
  <source>
   <traceable-element id="r2">
   </traceable-element>
  </source>
  <target-query>
   f:getUnitStartsWith("RA","statement",
     "Retrieve")
  </target-query>
 </intensional-trace>
...
 <intensional-trace>
  <source>
   <traceable-element id="r16">
   </traceable-element>
  </source>
  <target-query>
   f:getUnitStartsWith("RA","statement",
     "Concurrent usage")
  </target-query>
 </intensional-trace>
...
</trace-model>
```

(b) Intra-Phase Trace Model

Figure11. Concern Model and Trace Model for the Requirements Analysis Phase

also contain the full path to the *VersionManager* element. Due to space limitations the example only shows a link to the whole document.

```
<unit id="a12"
  phase="AD"
  reference="-//CVSArchModel.xmi"
  name="VersionManager"
  type="architectural element">
</unit>
```

(a) Architectural Module

```
f:getUnitContains("RA", "statement", "Commit")
  union f:getUnitContains("RA", "statement", "Branch")
```

(b) Intensional Trace Link

Figure12. XML and XPath Representations

Figure 12(b) shows an XPath expression that defines a trace link between the requirements analysis phase and the architecture design phase. The query defines a backward traceability link that states that *VersionManager* realizes the requirements related to committing and branching. Trace links and units can be defined in the same way for the remaining phases of the software development lifecycle. Modeling artifacts using unit elements allows us to decouple the trace model from a specific implementation language or modeling formalism.

7.3 Application of CTM for Impact Analysis

The information represented in the concern models and trace models for the CVS can be used to trace the impact of the change scenarios listed in Section 4.1. We use the

XML database ‘eXist’ [17] to execute queries over our models. The queries calculate a trace containing all elements that are impacted by the change. In the following we will describe shortly how to trace the impact of the first change scenario *Update Versioning*.

Determining the Initial Set of Changed Elements The change scenario *Update Versioning* adds new requirements that impact the concerns and requirements that we defined in the requirements analysis phase. Apparently all requirements making statements about files or directories will be influenced by the new requirement that the version mechanism should be extended to directories as well. This can be formulated as a query over concerns and units from our trace model. Figure 13 shows the corresponding queries.

```
f:getUnitContains("RA", "statement", "File")
  union
  f:getUnitContains("RA", "statement", "Dir")
```

(a) Changed Requirements

```
f:getConcernContains("RA", "statement", "File")
  union
  f:getConcernContains("RA", "statement", "Dir")
```

(b) Changed Concerns

Figure13. Queries for Impact Analysis

The union of the elements calculated by these queries, is the initial set of elements that are impacted by the changed requirements. This first step is the only step that has to be done individually for each change scenario and cannot be re-used for other change scenarios.

Calculating the Impact The impact on the other lifecycle phases can be automatically calculated by following the forward-traceability links that are defined in the trace-model. If a trace-link exists for an element of the initial element set, then all the target elements listed in the trace-link are indirectly impacted by the change scenario. These elements are added to the result set of the trace. All elements in the next phase are then determined for which a trace link exists that has one of the elements in the result set as source. This proces is repeated for all models from each phase. All query definitions that were used to follow the trace-links can be re-used for other change scenarios.

7.4 Enhancing the Metamodel

As we depicted in Figure 5 can CTM be (1) directly instantiated or (2) first enhanced and then instantiated, to customize it for a particular context. The context can be defined by, for example, tracing through different phases, application of different modeling approaches, using different joinpoint models, etc. CTM is generic and can be applied for at least the following cases:

Symmetric vs. asymmetric paradigms Although an explicit distinction is made between unit and concern, CTM can express both symmetric and asymmetric composition

paradigms. Since units can represent any type of artifact, both paradigms can be represented by defining the DTD's accordingly. It does not matter whether this distinction is made or not in the model. Only the mapping between the model elements represented by the units of the concern model and the artifact that contains these model elements will differ. If concerns are explicit in the models then exists a mapping between the concern model and these concerns. Otherwise, are the traceability links between model elements and concerns only defined in the concern model that was derived from CTM.

Adoption of different aspect languages CTM abstracts from the languages applied. Concerns and units are represented in a declarative way independent from the artifacts. No assumptions are made for using a particular aspect-oriented language. In fact, it even does not make a difference whether we are dealing with an object-oriented language or an aspect-oriented language.

Adoption for different phases and life cycle models CTM abstracts from phases in the life cycle or even the adoption of a particular life cycle model such as the waterfall model, iterative model or agile model. In many cases the instantiation of CTM will be sufficient to define the concerns and the mappings of these to the artifacts. The element *UnitModel* in the metamodel in Figure 4 can be enhanced to represent models of different phases.

Expression of existing traceability models using CTM The recent work on traceability of aspects have resulted in several traceability models [8,9,11]. Since CTM abstracts from phases and the adopted aspect specification languages these models can be represented either through instantiation or through first enhancing the metamodel. The latter approach can be used if, for example, different models (enhancement of *UnitModel*) or different joinpoint models (enhancement of *PointcutModel*) are required.

8 Conclusions

Traceability is an important quality factor that has been addressed in various domains to improve other quality factors such as understandability, maintenance and adaptability. Recently traceability has recently also been addressed by the AOSD community [8] to improve the traceability of aspects and as such support the maintenance throughout the life cycle. Although several initial approaches for traceability have been introduced to address traceability of aspects, we can observe that still lots of work needs to be done to achieve a complete traceability approach. First of all it appears that there is no tracing approach for the entire software life cycle yet. It is beneficial to enhance and complement existing work to include tracing for the entire life cycle. Second, it is important to provide a generic traceability model that is independent of the various approaches for specifying concerns and the way aspects are declared in a specific language.

In this paper we have built on the general literature on traceability, concern modeling and the recent work on traceability of aspects. We have proposed the concern traceability metamodel (CTM) that enables traceability of concerns throughout the life

cycle, and which is independent from the aspect languages that are used. We have defined a case study on Concurrent Versioning System and defined a set of scenarios to illustrate the problem of traceability of concerns. Based on our observations and the literature on traceability we have defined a set of requirements that are realized by CTM. CTM has been implemented in our tool M-Trace, that uses XML-based representations of the models and XQuery queries to represent tracing information. It is possible to enhance the meta-model CTM to define various traceability models.

References

1. O. Gotel and A. Finkelstein, "An analysis of the requirements traceability problem," in *First International Conference on Requirements Engineering (ICRE'94)*, pp. 94–101, Apr. 1994.
2. F. A. C. Pinheiro and J. A. Goguen, "An object-oriented tool for tracing requirements," *IEEE Softw.*, vol. 13, no. 2, pp. 52–64, 1996.
3. B. Ramesh and M. Jarke, "Toward Reference Models for Requirements Traceability," *IEEE Trans. Softw. Eng.*, vol. 27, pp. 58–93, January 2001.
4. B. Ramesh, C. Stubbs, T. Powers, and M. Edwards, "Requirements traceability: Theory and practice," 1997.
5. L. Bondé, P. Boulet, and J.-L. Dekeyser, "Traceability and interoperability at different levels of abstraction in model transformations," in *Forum on Specification and Design Languages, FDL'05*, (Lausanne, Switzerland), Sept. 2005.
6. S. Gérard, J.-P. Babau, and F. J. Champeau, eds., *Model Driven Engineering for Distributed Real-time Embedded Systems*. ISTE Ltd, 2005.
7. P. L. Torres, "A framework for requirements traceability in uml-based projects," (Edinburgh, U.K.), pp. 32–41, Sept. 2002.
8. *Workshop on Early Aspects: Traceability of Aspects in the Early Life Cycle (held with AOSD '06)*, (Bonn, Germany).
9. R. Chitchyan and A. Rashid, "Tracing requirements interdependency semantics," in *Workshop on Early Aspects (held with AOSD '06)*, (Bonn, Germany), 2006.
10. S. Katz and A. Rashid, "From aspectual requirements to proof obligations for aspect-oriented systems," in *Requirements Engineering Conference, 2004. Proceedings. 12th IEEE International*, pp. 48–57, 2004.
11. A. Jackson, P. Sanchéz, L. Fuentes, and S. Clarke, "Towards traceability between ao architecture and ao design," in *Workshop on Early Aspects (held with ASOD '06)*, (Bonn, Germany), 2006.
12. W. Harrison, H. Ossher, S. M. Sutton Jr., and P. Tarr, "Concern modeling in the concern manipulation environment," in *MACS '05: Proceedings of the 2005 workshop on Modeling and analysis of concerns in software*, (New York, NY, USA), pp. 1–5, ACM Press, 2005.
13. M. M. Kandé, *A Concern-Oriented Approach to Software Architecture*. PhD thesis, Ecole polytechnique fédérale de Lausanne, 2003.
14. R. Laddad, *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., 2003.
15. W. H. Harrison, H. L. Ossher, and P. L. Tarr, "Asymmetrically vs. symmetrically organized paradigms for software composition," Tech. Rep. RC22685, IBM Research, 2002.
16. D. Stein, S. Hanenberg, and R. Unland, "A uml-based aspect-oriented design notation for aspectj," in *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, (New York, NY, USA), pp. 106–112, ACM Press, 2002.
17. "http://www.exist-db.org."