JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2004

Vol. 6, No. 7, Special Issue: Aspect-Oriented Modeling, August 2007

Modeling Traceability of Concerns for Synchronizing Architectural Views

Bedir Tekinerdoğan, University of Twente, The Netherlands
Christian Hofmann, University of Twente, The Netherlands
Mehmet Akşit, University of Twente, The Netherlands

Software architecture modeling includes the description of different views that represent the architectural concerns from different stakeholder perspectives. In case of evolution of the software system the related architectural views need to be adapted accordingly. To synchronize the architectural views it is necessary that the dependency links among the architectural concerns in the architectural views can be easily traced. Unfortunately, despite the ongoing efforts for modeling concerns in architectural views, the traceability of concerns remains a challenging issue in architecture design. We propose the concern traceability metamodel (CTM) that enables traceability of concerns within and across architectural views. The metamodel can be used for modeling the concerns, the architectural elements and the traceability links among the elements in architectural views. 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 analyzing the impact of concerns of a Climate Control System case and synchronizing the architectural views.

1 INTRODUCTION

Software architecture design aims to identify the key concerns at an early stage of the software development lifecycle and modularize the concerns in an architectural model. A software architecture for a program or computing system consists of the structure or structures of that system, which comprise elements, the externally visible properties of those elements, and the relationships among them [9]. This definition implies that software architecture does not consist of a single structure but is represented using more than one architectural views. An architectural view is a representation of a set of system elements and relations associated with them [3]. Different views may include different type of elements, relations and constraints. Several approaches for organizing architecture around views have been proposed in the literature. These include, for example, the traditional Kruchten's 4+1 view approach, the views in the Rational's Unified Process, the Siemens Four Views model, and others as described in [9].

Concerns in the system are rarely stable and need to evolve in accordance with the changing requirements. To cope with the evolution at the architecture design level it is necessary that the dependency links between the architectural concerns in

Cite this article as follows: : *Modeling Traceability of Concerns for Synchronizing Architectural Views*, in Journal of Object Technology, vol. 6, no. 7, Special Issue: Aspect-Oriented Modeling, August 2007, pages 7–25,
http://www.jot.fm/issues/issues_2007_08/article1

the architectural views can be easily traced. This is because changes to concerns as such can have consequences for other architectural elements, which are directly or indirectly related to it.

Unfortunately, despite the ongoing efforts for identification and modeling of concerns in architectural views, the traceability of concerns remains a challenging issue in architecture design. In the aspect-oriented software development community the interest is in particular on crosscutting concerns which cannot be easily localized and are scattered over multiple implementation units. Several approaches have already been proposed to model crosscutting concerns at the architecture design level [8, 5], and focused on mapping aspect-oriented models through the life cycle. However, traceability of concerns in AOSD, whether crosscutting or not, has not yet been tackled broadly.

The topic of traceability is not new and has been discussed in various domains. The IEEE provides the following definition of traceability [2]: "Traceability is the degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship one another; for example, the degree to which the requirements and design of a given software component match." In requirements engineering lots of work has been done on tracing requirements from the stakeholders and in the design process [10, 12, 13]. In the model-driven engineering approach [6, 7] traceability is considered important for tracing model elements. The problem of traceability has recently also been addressed by the AOSD community [4], encompassing the adoption of aspects throughout the lifecycle. In each of these domains different definitions for traceability are given [2].

In this paper we build on existing work on traceability to trace aspects in architectural views. For this, we propose the Concern Traceability Metamodel (CTM) that can be used for modeling the concerns, modeling the architectural elements and the traceability links among the elements within and across the architectural views. 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 Climate Control System.

The remainder of the paper is organized as follows. In section 2 we present the example on Climate Control System (CCS) and illustrate the need for tracing crosscutting concerns within and across views. In section 3 we define the requirements for architectural concern traceability. In section 4 we provide the CTM which aims to meet these requirements. Section 5 will discuss the application of CTM to trace aspects within and across architectural views in the example case. Section 6 will finalize the paper with the conclusions.

2 EXAMPLE: CLIMATE CONTROL SYSTEM (CCS)

In the following we will define the case study that we will apply throughout the paper. The case study involves the architecture design of a climate control system (CCS) in cars. A CCS includes functions for heating, ventilating and air-conditioning. For the representation of architectural views we adopt the approach as defined by Clements et al [] and present the so-called module view, C&C view and deployment view. We define a set of concerns that can be identified within each view and across views.

Module View of CCS

The module view represents the structuring of implementation units, or modules. The module view of CCS is illustrated in Figure 1. *Controller* is the module that defines the main control loop. It uses *ReferenceModel* that defines the preferences of the user. *TemperatureSensor* senses the temperature of the car and provides on request sensor data to *Controller*. *Controller* sends current state of the car to *Display* and determines the action climate control action based on the difference between *ReferenceModel* and sensor data. The actions are defined by *Cooler*, *Heater* or *Fan*.

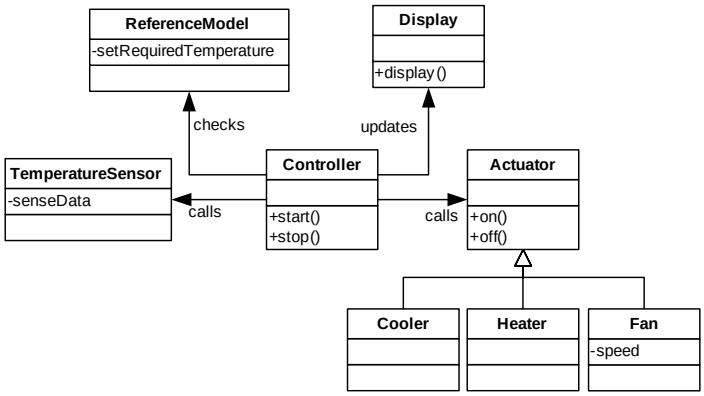

Figure 1: Module View of CCS

Component and Connector View of CCS

The Component and Connector (C&C) view represents the structuring of elements which have run-time behavior, which are usually components and connectors. shows the C&C view consisting of four components: *Controller*, *Sensor*, *Actuator* and *GUI*.

The *GUI* component controls user inputs and transfer the information to the *Controller* component. The *GUI* component will also present information from the *Controller* component to the user. The *Sensor* component senses the car information, while *Actuator* consists of the invoking the implementations of the actuator classes.

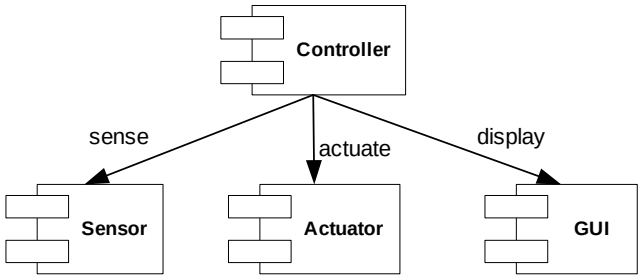

Figure 2: C&C View of CCS

Deployment View of CCS

The deployment view represents the allocation of software elements to hardware nodes. shows the deployment view of the CCS in which components are mapped to physical nodes in the system. We have identified three nodes: Microcontroller, Physical Sensors and Physical Actuator. The MicroController includes the components Controller and GUI. Physical Sensor executes the Sensor. Physical Actuator includes the Actuator component.

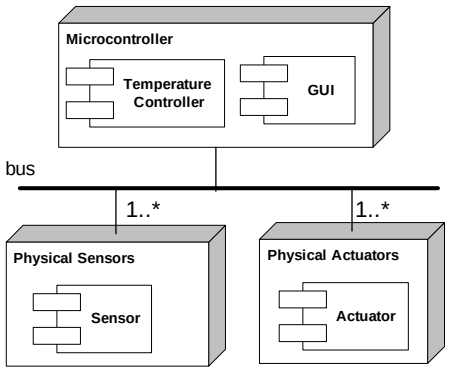

Figure 3: Deployment View of CCS

Change Scenarios and Problems

To illustrate the problem of traceability within and across architectural views we define a set of change scenarios. Each of these change scenarios refers to a particular concern. The scenarios are the following:

- *Adapt data format*

  The common data format that is used in the CCS for representing the sensor data needs to be adapted.

- *Adapt UI*

  The CCS will be deployed in cars that require different UI platforms. As such the display must be adaptable and be changed to the corresponding context.

- *Add humidity concern*

  The current design includes only the control of temperature in the car. The system needs to be enhanced to control the humidity in the car.

- *Add diagnostics*

  To cope with failures in the system it is required that the climate control elements provide mechanisms for failure detection and failure correction.

The above scenarios are selected examples that could be required in a CCS and we could easily identify several other scenarios that implement different concerns. In general we encounter the following two problems in realizing such kind of concern changes:

Impact analysis of concern changes Each change of a concern requires the identification of the architectural elements which implement the concern. For example, for realizing the scenario *Adapt data format* we need to identify all the architectural elements that are related to the *data formatting* concern. For the scenario *Add humidity* we need to identify the architectural elements that implement the concern *humidity*. In some cases we could derive from the names of the architectural elements which concerns are implemented, however, like in this case this is usually not that straightforward. Moreover, each concern might also map to more than one architectural element.

Synchronizing architectural views In case of a change of a concern the required adaptations will not be limited to a single view but need to be adapted in all the affected architectural views to preserve the consistency among the views. That is, the architectural views need to be *synchronized*. Without the maintenance and synchronization of the architectural views the architectural model becomes outdated and fails to provide its goal of communication, guidance and organization.

Unfortunately, the architectural views do not model concerns explicitly and do not provide explicit tracing links from concerns to architectural elements that implement these concerns. The tracing from concerns to architectural elements and vice versa, can then only be done implicitly and manually by iterating over each element in different architectural views and interpreting whether the element relates to a given concern or not. To support the impact analysis of concerns and synchronization of architectural views, traceability appears to be an important challenge.

3 REQUIREMENTS FOR CONCERN TRACEABILITY IN ARCHITECTURAL VIEWS

Based on the work in the literature on traceability and the concern modeling in AOSD we provide a set of requirements for traceability of concerns in architectural views.

Explicit Modeling of Concerns

In order to explicitly reason about traceability of the concerns in architectural views 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, the impact that it has, etc. Harrison et. al [14] 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 the design intentions 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 approaches [11].

Explicit Modeling of Dependency Relations

In principle, every architectural element implements one ore more concerns. To support traceability, the architectural elements and the relations with the corresponding concerns need to be modeled explicitly. This can be achieved when dependency relations are recorded as traceability links. For this, like concerns, traceability links should also be specified as first class abstractions in the adopted traceability model. The choice for a symmetric or asymmetric approach seems also to have an impact on the traceability links. In the asymmetric model the traceability links will need to be

established for both architectural elements and concerns. On the other hand, in the symmetric approach the traceability links need to refer to one type of concern. This simplifies the specification of trace-links but could reduce understandability because the user has to explicitly distinguish between concerns and architectural elements.

Intra-View Traceability of Concerns

To understand the relations among the concerns and architectural elements within the same view it is necessary to model traceability for the given view. Figure 4 shows the abstract model for tracing within a given view. We define here two types of traceability: (1) intra concern to element traceability and (2) intra element to concern traceability. Note that *Architectural Element* can be either an architectural relation or architectural entity. In this way concerns can be both linked to architectural relations and architectural entities. Further, since architectural entities may be composed of other sub-entities a single concern can then be attached to a composition of architectural entities.

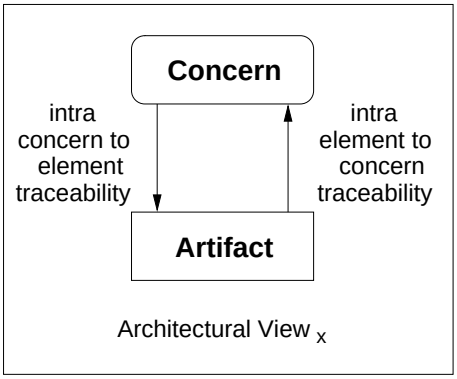

Figure 4: Traceability Relationship within a View

Inter-View Traceability of Concerns

Besides tracing concerns within an architectural view it is important to trace concerns that cut across views. Figure 5 presents the abstract model for traceability relationships across architectural views.

To distinguish from the previous intra-view traceability we use the term inter referring to traceability relations across different views. In principle, there are two kinds of relations. First, architectural elements in different views might be related, this is called, inter element to element traceability. Second, a common concern might be related to architectural elements in different views, which is termed as inter concern to element traceability.

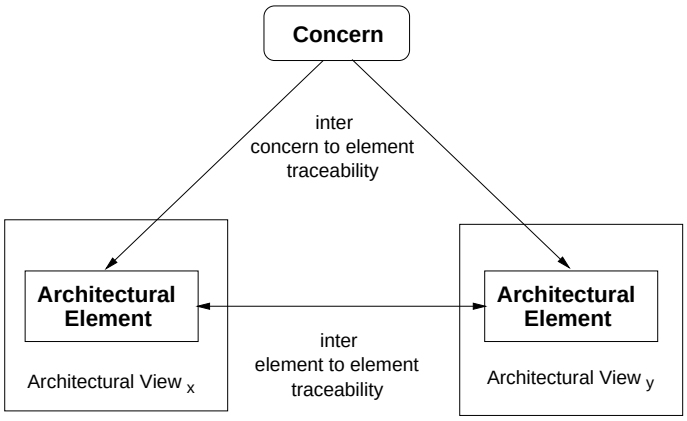

Figure 5: Traceability Relationship Accross Architectural Views

Support for Automated Tracing

Explicit models for concerns and the traceability will help to define the links between the different concerns and the architectural elements. By providing the traceability links, concerns can be more easily traced by just following the traceability links. This may work for simple, small scale systems. However, following the traceability links of a complex system manually might not be trivial. Even though the traceability links are made explic, it it may be hard to expose the required traceability links. To support tracing, the system should provide automated support for defining generic and user-defined queries to identify and trace the concerns. This is in particular important for the architectural models that consist of a broad set of concerns and architectural views.

4 CTM: ARCHITECTURAL CONCERN TRACEABILITY METAMODEL

In the following we present the concern traceability metamodel (CTM) for tracing concerns in architectural views as depicted in Figure 6. The metamodel represents three key issues: concern modeling, architecture modeling and tracing modeling. The metamodel should be preferably read from the left to the right. On the left, *ConcernModel* consists of *ConcernGroup* and *UnitModel*. *ConcernGroup* groups a set of *Concerns*. *Concerns* can be either crosscutting or not, the metamodel does not make an explicit distinction. *Concern* is defined for one or more *Stakeholder*. *UnitModel* represents the *Units* to which the concerns apply. A unit refers to an artifact in the software life cycle. Here we focus on the architecture design phase. *ArchitectureModel* is a subclass of *Unit* and consists of one or more *Architecture-View* which consists of one or more *ArchitecturalElement*. *ArchitecturalElement* in the metamodel represents in fact a representation of the actual architectural elements. To refer to the actual elements *ArchitecturalElement* includes the attributes reference and name. *Element* can be *Relation*, *Entity* and *Aspect*. *Relation* represents an architectural relation such as uses, depends on and calls. *Entity* represents

an architectural entity such as a *Module*, *Component* or *Node*. The specific elements will be different for different views, and if necessary, the metamodel can be extended for this purpose. Entities may have sub-elements that are represented by children relationship. *ArchitecturalAspect* represents a specification of an architectural aspect, which is associated to one or more entities. The relationship advices represents the dependency of an aspect with the architectural elements.

Traceability of architectural elements and concerns is represented by *TraceableElement* and *Trace*. A traceable element is either a *Unit* or a *Concern*. The trace relation is modeled explicitly by *Trace*, which relates one or more source elements to one or more target elements with the respective source and target relations. Source and target can be architectural elements or concerns that can be chosen from any view. The metamodel can therefore express all the traceability relations that we defined in Section 3.

Traces can be enumerated or specified succinctly using abstract queries. In the first case *ExtensionalTrace* is used and all source-target mappings between traceable elements are defined explicitly. In the latter case *IntensionalTrace* is used, whereby queries are applied that select source and target elements from the architectural views. It is possible to combine these trace specifications to define traces. For example, by listing the source elements explicitly and defining a query to select the target elements. Traces are part of a *TraceModel*, which can be specialized to represent different kinds of tracing specifications. For instance, *PointcutModel* is a specialization of *TraceModel* that is used to represent pointcut specifications as relations between a piece of advice and (several) elements of architectural views.

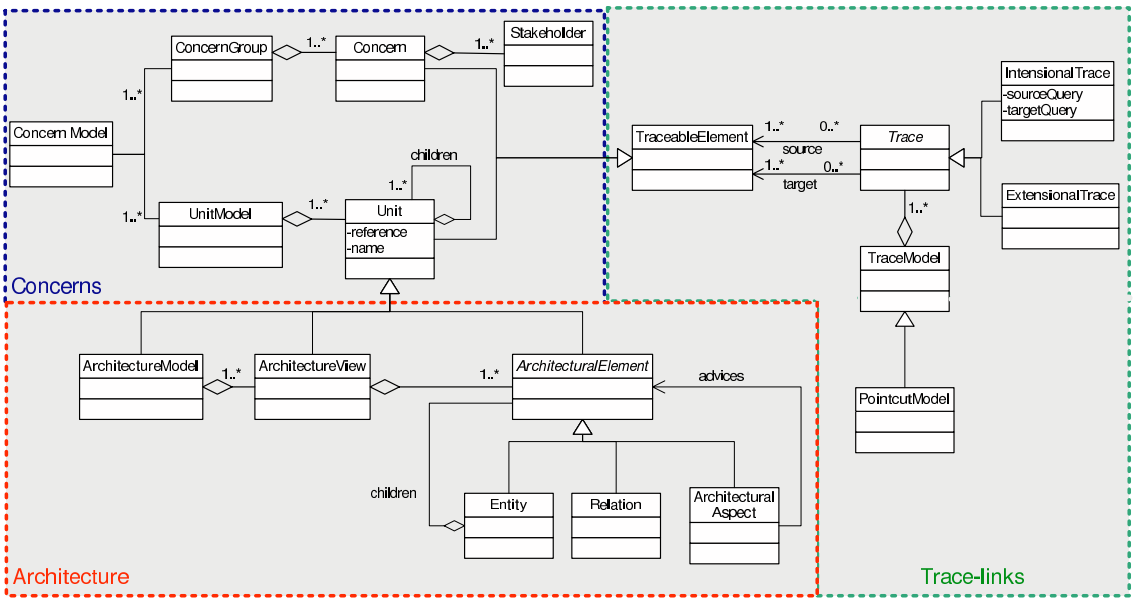

Figure 6: Architectural Concern Traceability Metamodel (CTM)

5 APPLICATION OF CTM

The CTM is a metamodel that can be instantiated in different ways. We follow the steps as described in the activity diagram in 7 to instantiate and use the metamodel for supporting traceability of concerns in architectural views.

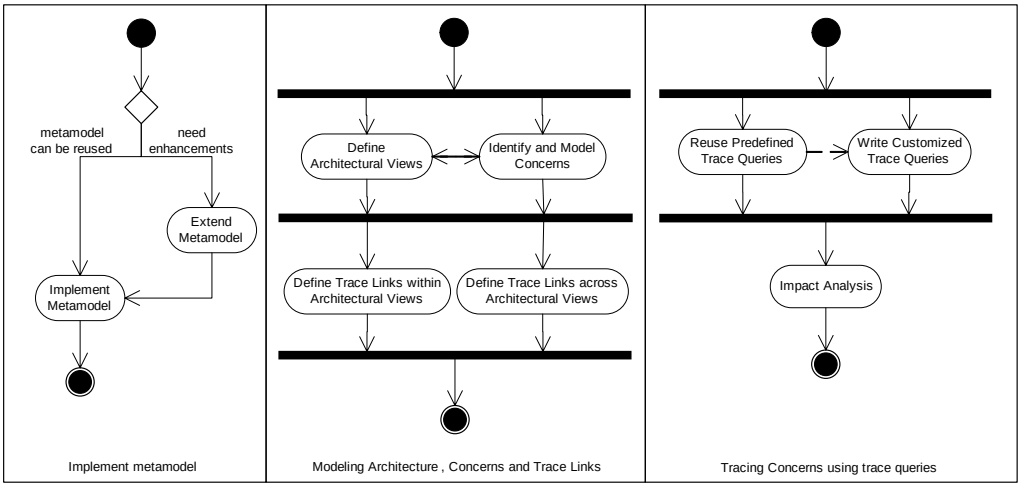

Figure 7: Process for utilizing the Metamodel and Tracing of Concerns

The process for utilizing the metamodel can be summarized in three steps:

- *Implementing CTM*

  This step is shown in the left part of 7. To support traceability CTM needs to be implemented. In our case we have implemented CTM by defining XML Document Type Definitions (DTD) for the metamodel elements including concern model, unit model and trace model. CTM is quite general but like any metamodel it can be first extended, if necessary, and then implemented.

- *Modeling architecture, concerns and trace-links*

  This step is shown in the middle part of 7. After implementation of CTM we need to provide instantiations to support tracing for the corresponding architectural views. In our implementation the DTDs representing the metamodel elements, are instantiated to define the architecture views, the concerns and the trace-links of the corresponding case.

- *Tracing of concerns*

  This is shown in the right part of 7. Once the models and the mappings are defined we can trace any concerns in the architectural views. For this purpose we can use either predefined reusable queries for tracing concerns or if needed write customized queries. In this paper the result of the tracing is used for impact analysis of evolving concerns.

In the following we will provide examples of the application of CTM.

Implementing CTM

Figure 8 provides an implementation of CTM using DTDs in XML. left column provides the DTD for architectural modeling. Here we can see that each architectural model consists of one or more views. Each view includes the attribute id, phase, reference, name and type. A view can consist of subviews, entity, relation or arch-aspect. The right part presents the DTDs for concern modeling and traceability modeling. The concernmodel follows the metamodel but in addition provides an explicit definition for crosscutting concerns. The tracemodel also is defined in accordance with the metamodel and consists of an extensional trace or an intensional trace. An extensional trace includes a source and target. An intensional trace can include also source-query or target-query. Note the DTD also implements a pointcutmodel, which is a specific trace model that can also include an extensional or intensional trace.

ARCHITECTURE MODELING

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!ENTITY % architecturalview SYSTEM
   "ArchitecturalView.dtd">
%architecturalview;

<!ELEMENT arch-model (view)+>
<!ATTLIST arch-model
id CDATA #REQUIRED
phase CDATA #IMPLIED
reference CDATA #IMPLIED
name CDATA #REQUIRED
type CDATA "architectural model">

<!ELEMENT view
   ((view)* ,(entity | relation | arch-aspect)*)>
<!ATTLIST view
id CDATA #REQUIRED
phase CDATA #IMPLIED
reference CDATA #IMPLIED
name CDATA #REQUIRED
type CDATA #REQUIRED>

<!ELEMENT entity (entity | relation)*>
<!ATTLIST entity
id CDATA #REQUIRED
phase CDATA #IMPLIED
reference CDATA #IMPLIED
name CDATA #REQUIRED
type CDATA #REQUIRED>

<!ELEMENT relation ((from)+, (to)+)>
<!ATTLIST relation
id CDATA #REQUIRED
phase CDATA #IMPLIED
reference CDATA #IMPLIED
name CDATA #REQUIRED
type CDATA #REQUIRED>

<!ELEMENT from (entity)+>
<!ELEMENT to   (entity)+>

<!ELEMENT arch-aspect (arch-advice)>
<!ATTLIST arch-aspect
id CDATA #REQUIRED
phase CDATA #IMPLIED
reference CDATA #IMPLIED
name CDATA #REQUIRED
type CDATA #REQUIRED>

<!ELEMENT arch-advice(entity |relation)+>
```

CONCERN MODELING

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

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

<!ELEMENT concerngroup
   ((concern | crosscutting-concern)+)>
<!ATTLIST concerngroup name CDATA #IMPLIED>

<!ELEMENT concern (description, stakeholder+)>
<!ATTLIST concern
id    CDATA #REQUIRED
phase CDATA #REQUIRED
name  CDATA #REQUIRED>

!ELEMENT crosscutting-concern (description, stakeholder+)>
<!ATTLIST crosscutting-concern
id    CDATA #REQUIRED
phase CDATA #REQUIRED
name  CDATA #REQUIRED>
<!ELEMENT stakeholder (#PCDATA)>
```

TRACING MODELING

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!ELEMENT tracelink-definition
   (tracemodel | pointcutmodel)+>

<!ELEMENT tracemodel
(extensional-trace | intensional-trace)+>
<!ELEMENT extensional-trace (source, target)>
<!ELEMENT intensional-trace
   ((source | source-query),(target | target-query))>

<!ELEMENT source (traceable-element)+>
<!ELEMENT target (traceable-element)+>

<!ELEMENT traceable-element (description?)>
<!ATTLIST traceable-element
id CDATA #REQUIRED
type CDATA #REQUIRED>
<!ELEMENT description (#PCDATA)>

<!ELEMENT source-query (#PCDATA)>
<!ELEMENT target-query (#PCDATA)>

<!ELEMENT pointcutmodel
   (extensional-trace | intensional-trace)+>
```

Figure 8: DTDs used to implement CTM

Modeling Concerns and Architecture

Once the CTM has been implemented as a set of DTDs we can provide instantiations, XML models, based on these DTDs. For example, Figure 9 shows the concern model for the CCS example. It consists of *ConcernGroup* and *UnitModel* elements that are used to organize units and concerns, as we have defined in the metamodel. The example shows six concerns *controlling*, *sensing*, *status display*, *actuate*, *observe* and *compare*. The unitmodel defines one architectural model with identifier *AM1*. We use a separate file to specify the architectural views of the architectural model.

```
<concernmodel><?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concernmodel PUBLIC "-//ConcernModel.dtd" "ConcernModel.dtd">

<concernmodel>
  <concerngroup>
    <concern id="c1" name="controlling"></concern>
    <concern id="c2" name="sensing"></concern>
    <concern id="c3" name="status display"></concern>
    <concern id="c4" name="actuate"></concern>
    <concern id="c5" name="observe"></concern>
    <concern id="c6" name="compare"></concern>
  </concerngroup>

<unitmodel>
  <unit id="am1" reference="ccs-am.xml" name="CCS"
      type="architectural model"></unit>
  </unitmodel>
</concernmodel>
```

Figure 9: Concern Model for CCS

The XML document shown in Figure 10 represents the architectural model of CCS consisting of three views. The components and connectors view, for example, is shown in the fifth line. This view with the identifier *cc1* is defined in the XML document "ccs-cc.xml", which can be found under the link given by the parameter *reference*. The module view and deployment views with the identifiers *mv1* and *dv1* are defined similarly.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE arch-model PUBLIC "-//ArchitecturalModel.dtd" "ArchitecturalModel.dtd">

<arch-model id="am1" reference="ccs-am.xml" name="CCS">
  <view id="cc1" reference="ccs-cc.xml" name="CCS CC" type="cc view"></view>
  <view id="mv1" reference="ccs-mv.xml" name="CCS MV" type="module view"></view>
  <view id="dv1" reference="ccs-dv.xml" name="CCS DV" type="deployment view"></view>
  <view id="dv2" reference="ccs-dv.xml" name="CCS DV" type="deployment view"></view>
</arch-model>
```

Figure 10: Architectural Model for CCS

Figure 11 shows the XML representation of the components and connectors view of the CCS. The elements *relation* and *entity* refer to the corresponding elements in the metamodel. The type of the element is defined in the *type* attribute of *relation* and *entity*. Since the XML document shown Figure 11 represents the C&C view, values for the *type* attribute of relation and entity are *connector* and *component*,

respectively. Relations include the *from* and *to* attributes to denote the architectural entities that are connected by the relation. The module view and the deployment view are represented similarly.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE view PUBLIC "-//ArchitecturalView.dtd" "ArchitecturalView.dtd">

<view id="cc1" reference="-//ccs-cc.xml" name="CCS CC" type="cc view">

 <relation id="cc1.r1" name="sense" type="connector">
  <from><entity id="cc1.e1" name="Controller" type="component"></entity></from>
  <to><entity id="cc1.e2" name="Sensor" type="component"></entity></to>
 </relation>

 <relation id="cc1.r2" name="actuate" type="connector">
   <from><entity id="cc1.e1" name="Controller" type="component"></entity></from>
   <to><entity id="cc1.e3" name="Actuator" type="component"></entity></to>
 </relation>

  <relation id="cc1.r3" name="display" type="connector">
   <from><entity id="cc1.e1" name="Controller" type="component"></entity></from>
   <to><entity id="cc1.e4" name="GUI" type="component"></entity></to>
 </relation>
</view>
```

Figure 11: C&C View of CCS

Defining Trace-Links

After explicit modeling of the concerns we will now define the trace-links among the concerns and the architecture elements within a view. This is again done by instantiating from a DTD that represents tracing model. In the trace model traces define the dependency between a source and a target. Dependencies are defined by either an enumeration of the separate dependencies (extensional) or specified more abstractly using queries (intensional). Queries are written using XQuery, which is a technology developed by the W3C that is designed to query collections of XML-data. Trace-links can be defined within or across architectural views. In the following we will explain these separately.

Defining Trace-links within Views

Figure 12 shows an example of a query for defining trace-links within the C&C view. Hereby the source is defined explicitly (enumeration) denoting *c3* (concern *Status Display*). The target is defined using a an XQuery expression (intensional) that uses the predefined function *getElementsContaining()*. This function takes as parameter a string that identifies the view (its *id*), the name of the element type it searches, and a string it should match (case insensitive) with the name attribute of the element. Here the first argument is "*mv*" denoting that elements in the module view will be addressed. The second argument includes a wildcard (.*) that denotes that any type of element (i.e. module or relation) in the module view can be matched. The

third argument includes the name that should appear in the matched elements. As as result of query in Figure 12 concern *Status Display* (*id=c3*) is related to all architectural elements from the module view that have "display" in its name.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE tracelink-definition PUBLIC "-//TraceModel.dtd" "TraceModel.dtd">
</tracemodel>
  <intensional-trace>
    <source> <traceable-element id="c3"> </traceable-element>  </source>
    <target-query>
       f:getElementsContaining("mv",".*","display")
    </target-query>
  </intensional-trace>
   ...
</tracemodel>
  …
</tracelink-definition>
```

Figure 12: Example for Defining Trace-Links Within the C&C View

The function *getElementFromView()* is one example of a predefined query that can be used to support tracing. Figure 13 lists some other functions that we have predefined and that can be used these to define the mappings between the concerns and the architectural elements. In general these queries seem to be generic enough for tracing concerns. However, in case more complex relationships are needed that cannot be expressed by these queries, it is quite easy to implement new queries or enhance existing ones.

`f:getElementsStartingWith($view, $elements, $name)`	Matches elements whose names start with a given sub-string
`f:getElementsEndingWith($view, $elements, $name)`	Match elements whose name ends with a given substring
`f:getElementsWithName($view, $elements, $name)`	Match case insensitive names exactly matching a given string
`f:getElementsContaining($view, $elements, $name)`	Matches elements whose names start with a given sub-string

Figure 13: Functions for Calculating Traceability Links through Element Matching

Defining Trace links Across Views

Similar to defining trace-links within views we can easily define trace-links across views. The only difference here is that we have to refer to more than one view in the queries for the source or target elements. Figure 14, for example, specifies a query across multiple views that calculates the traceability link. The concern "*controlling*" with id "*c1*" is related by the trace-link to all units named "*controller*" in all architectural views (1st parameter in line 3) regardless of their type (2nd parameter). Note that we have used here the wildcard ".*" that matches any view.

```
<intensional-trace>
 <sourc>
   <traceable-element id="c1" type="concern"> </traceable-element></source>
   <target-query>f:getElementFromViewWithName(".*",".*","controller")</target-query>
</intensional-trace>
```

Figure 14: Intensional Definition of a Traceability Link Across Views

Besides using the ".*" wildcard, we can also list the views explicitly; using a comma operator to select specific views. For example, the query

```
f:getElementsWithName(("cc1", "dv1"),".*","Sensor")
```

matches the component and or connector elements called Sensor in the component and connector view (cc1) or the deployment view (dv1). The result is provided again as an XML file as it is depicted in Figure 15.

```
<exist:result xmlns:exist="http://exist.sourceforge.net/NS/exist" hitCount="2">
  <traceable-element id="cc1.e2" name="Sensor" type="component"/>
   <traceable-element id="dv1.e7" name="Sensor"  type="process"/>
</exist:result>" type="processor"/>
```

Figure 15: Query Result for the Element "Sensor"

Tracing Concerns for Impact Analysis and View Synchronization

So far we have defined the concern model, the architectural model including the views, and the mappings between the concerns and the architectural elements. In principle we can now trace any concern to the architectural elements in the views. In this paper we will focus on tracing concerns to support the impact analysis of evolution of concerns.

We have implemented a set of queries that can trace concerns to elements and vice versa. There are two types of queries: *forward tracing queries* and *backward tracing queries*. Forward tracing queries can be used to trace the architectural elements starting from a given set of concerns. Backward tracing queries determine the concerns for a given set of architectural elements. To assess the impact of a given concern, we can thus use forward tracing queries. Backward queries can be used, for example, to inspect the set of concerns that an architectural element is related to. We use the XML database "eXist" [1] to execute queries, calculate traceability links and store the models.

`traceForward($view, $concernId )`	Matches elements for given concerns
`traceBackward($view, $elementId)`	Match concerns for given elements

Figure 16: Tracing Functions

Again, we predefined functions that can be utilized to perform these tracing activities. Figure 16 shows to of them.

We can use *traceForward()*, for example, to identify the elements that are impacted if the concern *sensing* (*c2*) is changed. The corresponding query statement is:

```
f:traceForward(".*", "c2")
```

The result of the query is an XML file, as shown in Figure 17, that lists all the elements related to the concern sensing. The relationship between the concern and the returned elements that we calculated is an inter concern-to-element traceability relationship.

```
<exist:result xmlns:exist="http://exist.sourceforge.net/NS/exist" hitCount="6">
  <relation id="cc1.r1" name="sense" type="connector">
    <from> <entity id="cc1.e1" name="Controller" type="component"/> </from>
    <to> <entity id="cc1.e2" name="Sensor" type="component"/> </to>
  </relation>
  <entity id="cc1.e2" name="Sensor" type="component"/>
  <entity id="mv1.e1.1" name="setRequiredTemperature" type="interface"/>
  <entity id="mv1.e3" name="TemperatureSensor" type="module">
    <entity id="mv1.e1.1" name="senseData" type="interface"/>
  </entity>
  <entity id="dv1.e7" name="Sensor" type="process"/>
  <entity id="dv1.e2" name="Physical Sensors" type="processor"/>
</exist:result>
```

Figure 17: Query Result of Following the Forward Trace-Links Defined for the Concern Sensing

Similarly, we can trace the concerns given an architectural element. The query

```
f:traceBackward("cc1", "Sensor")
```

calculates all the concerns that are related to the element *Sensor* from the component and connector view. In principle, once all the tracing links have been defined we can query both the elements to the concerns and concerns to the elements. Tracing can be done both within a view as well as across multiple views. The result of the tracing queries is currently provided as an XML file. In our future work we are planning to visualize the results of the XML file.

6 CONCLUSION

In this paper we have built on the general literature on traceability, concern modeling and the recent work on traceability of aspects. We have used a *Climate Control System* as case to illustrate the impact of a set of change scenarios on the concerns and the architectural views. Based on our observations and the literature on traceability we have defined a set of requirements for tracing concerns in architectural views. We have proposed the concern traceability metamodel (CTM) that enables traceability of concerns in architectural views. CTM has been implemented in our tool M-Trace, that uses XML-based representations of the models and XQuery queries to calculate relationships, like traceability links, between the elements of these models.

The metamodel has been applied to trace concerns and for the impact analysis of

changes to these concerns. It was not difficult to explicitly model the concerns and the architectural elements of the case, and define the mapping of concerns to the elements of the architectural views. By defining expressive queries we could easily realize forward and backward traceability of concerns.

Our future work will include a systematic application of domain knowledge to provide more expressive queries. Another issue is the visualization of the traceability relationships. Also the results of the impact analysis are represented XML. We are therefore working on enhancing the M-Trace tool with a more intuitive representation of the results to the user.

7 ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for the valuable feedback on earlier versions of this paper. This work is supported by the European Network of Excellence on AOSD project, and the Aspect-Oriented Software Architecture Design project funded by the Dutch Scientific Organisation in the Jacquard Software Engineering Program.

REFERENCES

[1] http://www.exist-db.org.

[2] Glossary of software engineering terminology. IEEE Standard 610.12-1990, IEEE, 1990.

[3] IEEE recommended practice for architectural description of software-intensive systems. IEEE Standard 1471-2000, IEEE, 2000.

[4] *Workshop on Early Aspects: Traceability of Aspects in the Early Life Cycle (held with AOSD '06)*, Bonn, Germany, 2006.

[5] Elisa Baniassad, P.C. Clements, J. Araujo, A. Moreira, A. Rashid, and B. Tekinerdogan. Discovering early aspects. *Software, IEEE*, 23(1):61–70, 2006.

[6] Joel Champeau and Emmanuel Rochefort. Model engineering and traceability. In *Workshop SIVOES-MDA, UML'03*, October 2003.

[7] R. Chitchyan, A. Rashid, P. Sawyer, A. Garcia, M. Pinto, J. Bakker, B. Tekinerdogan, S. Clarke, and A. Jackson. Survey of analysis and design approaches. AOSD-Europe Deliverable D11, Network of Excellence AOSD-Europe, 2005.

[8] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. The SEI Series in Software Engineering. Addison Wesley Professional, 2002.

[9] Paul Clements, Rick Kazman, and Mark Klein. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley, 2001.

[10] Orlena Gotel and Anthony Finkelstein. An analysis of the requirements traceability problem. In *First International Conference on Requirements Engineering (ICRE'94)*, pages 94–101, April 1994.

[11] William H. Harrison, Harold L. Ossher, and Peri L. Tarr. Asymmetrically vs. symmetrically organized paradigms for software composition. Technical Report RC22685, IBM Research, 2002.

[12] Francisco A. C. Pinheiro and Joseph A. Goguen. An object-oriented tool for tracing requirements. *IEEE Softw.*, 13(2):52–64, 1996.

[13] Balasubramaniam Ramesh, Curtis Stubbs, Timothy Powers, and Michael Edwards. Requirements traceability: Theory and practice, 1997.

[14] Stanley M. Sutton Jr. and Isabelle Rouvellou. Modeling of software concerns in Cosmos. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 127–133, New York, NY, USA, 2002. ACM Press.

Bedir Tekinerdoğan is an assistant Professor at the University of Twente in The Netherlands. Contact him at Univ. of Twente, Dept. of Computer Science, Software Eng., PO Box 217 7500 AE, Enschede, NL; bedir@cs.utwente.nl.

Christian Hofmann is a PhD student at the the University of Twente in The Netherlands. He can be reached at c.hofmann@utwente.nl

Mehmet Akşit is a full Professor at the University of Twente in The Netherlands. He is the head of the Software Engineering chair and the leader of the Twente Research and Education on Software Engineering (TRESE) Group. Contact him at Univ. of Twente, Dept. of Computer Science, Software Eng., PO Box 217 7500 AE, Enschede, NL; aksit@ewi.utwente.nl.