Modeling Traceability of Concerns in Architectural Views

Bedir Tekinerdogan, Christian Hofmann, Mehmet Aksit

Department of Computer Science,
University of Twente
P.O. Box 217 7500 AE Enschede

{bedir|c.hofmann, m.aksit}@cs.utwente.nl

ABSTRACT

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. For this 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 trace 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.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques. D.2.11 [**Software Engineering**]: Software Architectures; K.6.3 [**Software Management**]: Software Maintenance

General Terms

Design, Languages, Management

Keywords

Software Architecture Modeling, Traceability of Concerns, Concern Modeling

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 architecture 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 [7]. 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 [13]. 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 [7].

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 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 [6][2], 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. In requirements engineering lots of

work has been done on tracing requirements from the stakeholders and in the design process [14][21][22]. In the model-driven engineering approach [4][5] traceability is considered important for tracing model elements. The problem of traceability has recently also been addressed by the AOSD community [12], encompassing the adoption of aspects throughout the lifecycle. In each of these domains different definitions for traceability are given [16].

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 will shortly discuss the background on architectural views. In section 3 we present the example on Climate Control System (CCS) and illustrate the need for tracing crosscutting concerns within and across views. In section 4 we define the requirements for architectural concern traceability. In section 5 we provide the CTM which aims to meet these requirements. Section 6 will discuss the application of CTM to trace aspects within and across architectural views in the example case. Section 7 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 in [7] and present the so-called module view, C&C view and deployment view[1]. We define a set of concerns that can be identified within each view and across views.

2.1 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 *Car* and provides on request

---

[1] These views should only be considered as an example to motivate the problem. In principle, the approach for tracing concerns is independent of the different views.

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*.
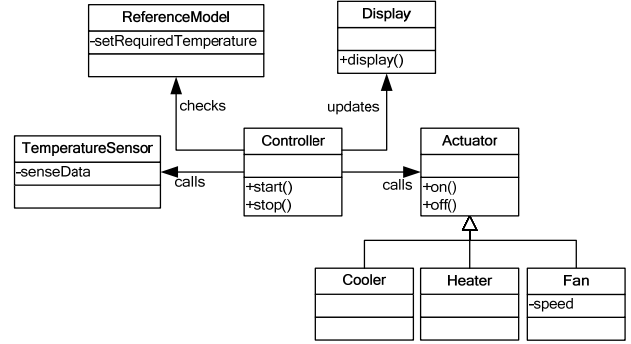

Figure 1. *Module View of CCS*

2.2 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. Figure 2 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.
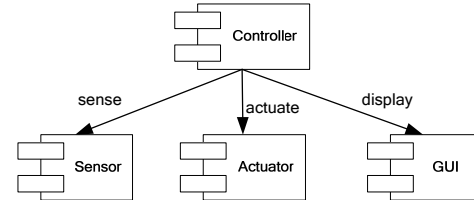

Figure 2. *C&C View of CCS*

2.3 Deployment View of CCS

The deployment view represents the allocation of software elements to hardware nodes. Figure 3 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.
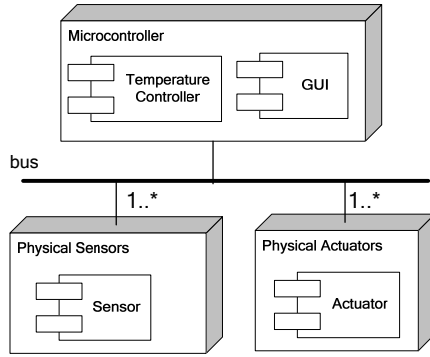
Figure 3. *Deployment View of CCS*

2.4 Change Scenarios

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. What is important here is that, without tracing support, it is very difficult to assess the impact of these scenarios and enhance the architectural views appropriately. For the impact analysis, we need to know to which architectural elements the concerns in the scenarios apply. Since the architecture description does not provide explicit links to the concerns, it is very difficult to understand the relation among these concerns and the architectural elements in the architectural views. Usually, the tracing is done implicitly by iterating over each element and interpreting whether the element relates to a given concern or not. It should be clear that this is not easy and also cumbersome due to the potential subjective interpretations.

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.

3.1 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 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 approaches [15].

3.2 Explicit Modeling of Dependency Relations

In principle, every architectural element implements one ore more concerns. To support tracing these relations among architectural elements and concern explicit needs to made explicit. This can be achieved when dependency relations are recorded as trace 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 traceability specification but could reduce understandability because the user has to explicitly distinguish between concerns and architectural elements.

3.3 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. Here we have shown the case of an asymmetric model and distinguished between an architectural element and a

concern. 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.
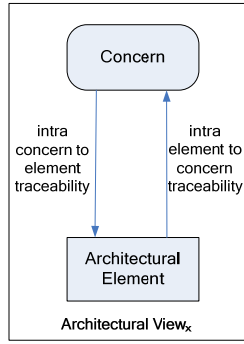

Figure 4. *Traceability Relationships within a View*

3.4 Inter-View tracing of concern

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.
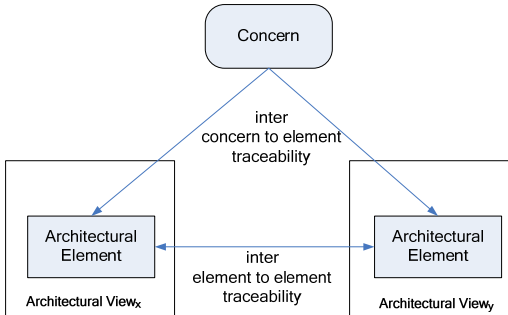

Figure 5. *Traceability Relationships across Architectural Views*

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*.

3.5 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 trace links. For simple, small scale systems tracing could be done in this way. However, for a complex system following the traceability links manually might not be trivial. Even though the traceability links are made explicit 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 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: the architecture, the concerns, and the tracing. 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 *ArchitectureView* which consists of one or more *ArchitecturalElement*. *ArchitecturalElement* includes in the metamodel is 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. *Architectural Aspect* 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.
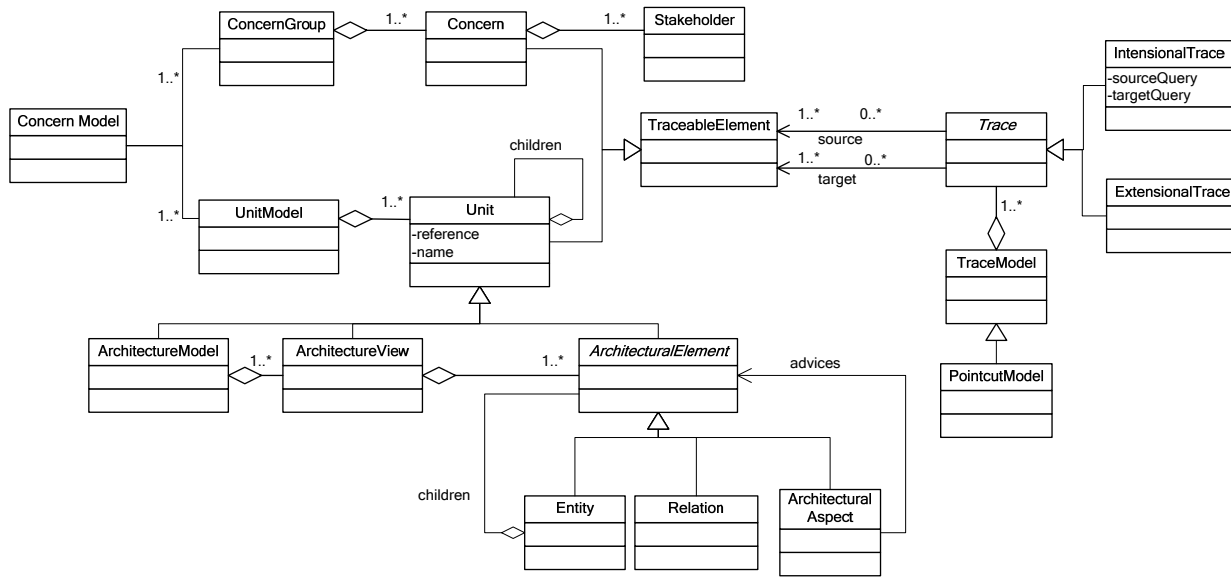
Figure 6. *Architectural Concern Traceability Metamodel (CTM)*

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. Since the source and target can be an architectural element or a concern within any view, and for any tracing direction, the metamodel can express the traceability relations as defined in section 3.

Traces can be specified extensionally by just enumerating all source-target mappings between traceable elements, or intensionally queries are applied to specify the source and target elements. *ExtensionalTrace* and *IntensionalTrace* represent the two different alternatives. Finally, traces are modeled as parts of a trace model, represented by *TraceModel*, which makes it possible to specialize it. Pointcut models are represented by *PointcutModel*.

5. APPLICATION OF CTM
The CTM is a metamodel that can be instantiated in different ways. We follow the subsequent steps to instantiate and use the metamodel for supporting traceability of concerns in architectural views.

- *Instantiating CTM*
CTM is implemented by defining XML Document Type Definition (DTD) files for the metamodel elements. This includes DTDs for concern model, unitmodel, architectural views and trace links.

- *Modeling concerns and architecture model for case*
By instantiating the DTDs the specific concern model and architecture model of the corresponding case are defined. This is explained in section 5.1.

- *Defining trace links (mappings) of concerns to architectural elements in the views*
To support traceability, the concerns are mapped to the corresponding architectural elements. This could be within or across architectural views. This is explained in section 5.2.

- *Tracing of concerns*
Once the models and the mappings are defined we can trace any concerns in the architectural views. In this paper we show traceability for impact analysis of evolving concerns. This is explained in section 5.4.

5.1 Modeling Concerns and Architecture
Figure 7 shows the concern model for the CCS. It consists of *ConcernGroup* and *UnitModel* elements that are used to organize *Units* and *Concerns*, as we have defined in the metamodel. In the example we have defined three concerns *controlling, sensing,* and *displaying.* The unitmodel defines one architecture model with identifier `AM1`. The architectural views are specified in a separate file.

```
1.  <concernmodel>
2.   <concerngroup>
3.    <concern id="c1" name="controlling"></concern>
4.    <concern id="c2" name="sensing"></concern>
5.    <concern id="c3" name="displaying"></concern>
6.    …
7.   </concerngroup>
8.   <unitmodel>
9.    <unit id="am1" reference="CCS-AM.xml"
10.    name="CCS" type="architectural model"></unit>
11.  </unitmodel>
12. </concernmodel>
```

Figure 7. Concern Model for CCS

The XML document shown in Figure 8 represents the architectural model of CCS consisting of three views. The components and connectors view, for example, is shown in line 5. 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* (Line 5). The module view and deployment views with the identifiers *mv1* and *dv1* are defined similarly.

```
1.  <arch-model id="am1"
2.    reference="CCS-AM.xml" name="CCS">
3.  <view id="mv1" reference="ccs-mv.xml"
4.    name="CCS Module View" type="module view"/>
5.  <view id="cc1" reference="ccs-cc.xml"
6.    name="CCS CC View" type="cc view"/>
7.  <view id="dv1" reference="ccs-dv.xml"
8.    name="CCS Deployment View"/>
9.  </arch-model>
```

Figure 8. Architectural Model for CCS

Figure 9 shows, for example, 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* attributes of *relation* and *entity*. Since in Figure 9 we are representing the C&C view the 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 in a similar sense.

```
1.  <view id="cc1" reference="ccS-cc.xml"
2.       name="CCS CC View" type="cc view">
3.  <relation id="r1" name="sense" type="connector">
4.   <from><entity id="e1" name="Controller"
5.       type="component"></entity></from>
6.   <to><entity id="e2" name="Sensor"
7.       type="component"></entity></to>
8.  </relation>
9.  <relation id="r2" name="actuate" type="connector">
10.  …
11. </relation>
12. …
13. </view>
```

Figure 9. C&C View of CCS

5.2 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.

In the implementation of CTM we use a separate DTD to describe the structure of the trace model. As indicated before, the dependency relations are defined through trace models including traces. Traces define the dependency between a source and a target. The dependency is either directly enumerated (extensional) or indirectly defined through 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.

5.2.1 Defining Trace links within Views

Figure 10 shows an example of a query that can be used to define the trace links between a concern and architectural elements in the component and connector view. Note that trace links can be defined either in an intensional or extensional manner. Figure 10 shows a query in which the source is enumerated (extensional) and the target element to which it should be mapped is defined using a query defined as an XQuery expression. The query is a union of the results returned by the predefined function *getElementFromView()*. This function takes as parameter an *id* string that identifies the view, the element type it searches and a sub-string it should match with the *name* attribute of the element. It travels the tree of view elements within the architectural model, until it finds a view element that *contains* the given *id* string. From there on, all relation and entity elements are returned where the name attribute contains the given sub-string. This means for the example in Figure 10, that the concern "controlling" is related to all architectural elements from the C&C view that have a name that is related to controlling.

```
<tracemodel>
 <intensional-trace>
  <source>
   <traceable-element id="c1"> </traceable-element>
  </source>
  <target-query>
   local:getElementFromView("cc1","","control") union
   local:getElementFromView("cc1","","sens") union
   local:getElementFromView("cc1","","actua")
  </target-query>
 </intensional-trace>
 …
 </trace-model>
```

Figure 10. Query defining an Intensional Trace within C&C View

The set returned by the query contains the components *Controller*, *Sensor* and *Actuator,* as well as the connector *sense* and *actuate*.

Using XQuery we can define easily other matches like *getElementFromViewStartingWith(), getElementFrom-ViewEndingWith()* or *getElementFromViewWithName()* that select architectural elements of a given type within a view, that start, end or exactly match a given string. The mappings between the concerns and the architectural elements in the module- and deployment view are defined in a similar way.

5.2.2 *Defining Trace links Across Views*

Some concerns might not be limited to a single view but appear in different views. To define the trace links of these concerns the similar queries as defined before can be used. The only difference will be the reference to multiple views. Figure 11 specifies, for example, a trace using a query across multiple views. The concern "controlling" with *id* "c1" (Line 4) that was defined in Figure 7, Line 5, is the source of the trace link. It is related to all kinds of units (denoted by the empty String "" as 2nd parameter in Line 7) in all architectural views (1st parameter) named "controller". To refer to specific set of views, the first argument of *getElementFromViewWithName* can include a set of views separated by a comma.

```
1.<tracemodel>
2. <intensional-trace>
3.  <source>
4.   <traceable-element id="c1"> </traceable-element>
5.  </source>
6.  <target-query>
7.local:getElementFromViewWithName("","","controller")
8.  </target-query>
9. </intensional-trace></tracemodel>
```

Figure 11. *Trace Model across Views*

5.3 Tracing Concerns for Impact Analysis

So far we have defined the concern model, the architecture 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. Tracing concerns can support several goals. In this paper we will focus on 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* that trace concerns to architectural elements, and *backward tracing queries* that trace architectural elements to concerns. We use the XML database 'exist'[2] to execute the queries over our models and calculate the traces. For example, the function *traceForward()* is presented in Figure 12. The function determines first the trace links where the source of the link is one of the elements in the start set. Then all the target elements are determined for these trace links. The target elements in trace links are recursively calculated

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

for all elements in the start set and the identified target elements.

```
declare function local:traceForward( $startSet ){
 let $targetElements :=
  local:targetElementsForElements( $startSet )
 return
  if( empty( $targetElements ) )
   then $startSet
   else $startSet union
        local:traceForward( $sourceElements )};
```

Figure 12. *Forward Traceability*

This function can be called to trace the impact of concerns. For example to analyze the impact of the scenario *sensing* (id=c2) we can invoke the following call:

```
traceForward(concern[@id="c2"])
```

This results in a XML file that includes all the elements that are related to the concern *sensing*. We transform the XML files to human readable documents that list all the elements. Obviously it is not always possible to directly trace the concerns by providing specific names (like *sensing*). In this case we have to apply domain knowledge to characterize the concerns. For example, to analyze the impact of adding humidity control we need to identify all the elements that use cooling, heating, sensing or fan. The start set of the query is then determined by the following expression:

```
let startSet :=
f:getElementFromViewStartingWith("","","sens") union
f:getElementFromViewStartingWith("","","cool") union
f:getElementFromViewStartingWith("","","heat") union
f:getElementFromViewStartingWith("","","fan")
```

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 defined a case study on Climate Control System and defined a set of change scenarios to illustrate the problem of traceability of concerns in 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 represent tracing information.

The metamodel has been applied to trace concerns for impact analysis of selected concerns for the case. It was not difficult to explicitly model the concerns and the architectural elements, and the mapping of the concerns to the elements. By defining expressive queries we could also easily realize both forward and backward traceability of concerns. However, for some concerns it appeared that sufficient domain knowledge was necessary to define the

appropriate queries. Our future work will include a systematic application of domain knowledge to provide more expressive queries. Another issue is the visualization of the tracing. In our tool the impact analysis results in a set of XML files. We are working on enhancing the M-Trace tool for better visualization.

ACKNOWLEDGMENTS

REFERENCES

[1] J. Bakker, *Traceability of Concerns*, MSc thesis, Faculty of Electrical Engineering, Mathematics and Computer Science, University of Twente, Enschede, NL, April 2005.

[2] E. Baniassad, P. Clements, J. Araujo, A. Moreira, A. Rashid, B. Tekinerdogan, *Discovering Early Aspects*, IEEE Software, Vol. 23, No. 1, pp. 61-70, January, 2006.

[3] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 2nd ed., Addison-Wesley, 2003.

[4] L. Bonde, P. Boulet, J.-L. Dekeyser. *Traceability and interoperability at different levels of abstraction in model transformations*, in: Proceedings of FDL'05, Lausanne, Switzerland, September 2005.

[5] J. Champeau, J. and E. Rochefort. *Model Engineering and Traceability*. in UML 2003. SIVOES-MDA Workshop. 2003. San Francisco, California, USA.

[6] 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*, Network of Excellence AOSD-Europe, 2005.

[7] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, J. Stafford, *Documenting Software Architectures*. Addison-Wesley, 2002.

[8] P. Clements, R. Kazman, M. Klein. *Evaluating Software Architectures*, Addison-Wesley, 2002

[9] R. Chitchyan and A. Rashid. *Tracing Requirements Interdependency Semantics*. in Proc. of workshop on Early Aspects (held with ASOD 06), Bonn, Germany, 2006.

[10] L.Dobrica & E.Niemela. *A survey on software architecture analysis methods*. IEEE Trans. on Software Engineering, Vol. 28, No. 7, pp.638-654, July 2002.

[11] Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design web site: http://www.early-aspects.net/, 2003.

[12] Early Aspects Workshop: Traceability of Aspects in the Early Life Cycle, in conjunction with the fifth international Conference on Aspect-Oriented Software Development (AOSD), March 21, 2006, Bonn, Germany.

[13] IEEE P1471, *IEEE Recommended Practice for Architectural Description of Software-intensive Systems--Std. 1471-2000*. 2000.

[14] 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.

[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] IEEE Std. 610.12-1990. Standard Glossary of Software Engineering Terminology, 1990.

[17] A, Jackson, P. Sanchez, L. Fuentes, S. Clarke. *Towards Traceability* between AO Architecture and AO Design, in Proc. of workshop on Early Aspects (held with ASOD 06), Bonn, Germany, 2006

[18] M. Kandé, *A Concern-Oriented Approach to Software Architecture*. PhD thesis, Ecole polytechnique fédérale de Lausanne, 2003.

[19] P. Letelier. *A Framework for Requirements Traceability in UML-based Projects*. In *Proceedings of the1st International Workshop on Traceability, co-located with ASE 2002*, Edinburgh, Scotland, UK, 2002.

[20] N.Medvidovic & R.N. Taylor. *A classification and comparison framework for Software Architecture Description Languages,* IEEE Trans. on Software Engineering, Vol. 26, No.1 pp. 70-93, 2000..

[21] F. Pinheiro, J. Goguen, *An Object-Oriented Tool for Tracing Requirements*, IEEE Software, v.13 n.2, p.52-64, March 1996.

[22] B. Ramesh & M. Jarke. *Towards Reference Models for Requirements Traceability*. IEEE Trans. On Software Engineering, Vol. 27., No. 1. January, 2001.

[23] A. Rashid, A. Moreira & J. Araujo, *Modularisation and Composition of Aspectual Requirements*, in: Proc. of 2nd AOSD Conference, pp. 11-21, Boston, US, 2003.

[24] B. Tekinerdogan. *ASAAM: Aspectual Software Architecture Analysis Method*. Proc. 4th Working IEEE/IFIP Conf. Software Architecture (WICSA 04), IEEE CS Press, pp. 5–14, 2004.