Analysis of Crosscutting across Software Development Phases based on Traceability

Klaas van den Berg
Software Engineering Group
University of Twente
P.O. Box 217
7500 AE Enschede
the Netherlands
+31 53 4893783
k.g.vandenberg@ewi.utwente.nl

José María Conejero
Quercus SEG
University of Extremadura
Avda. Universidad s/n
C.P. 10071 Cáceres
Spain
+34 927 257268
chemacm@unex.es

Juan Hernández
Quercus SEG
University of Extremadura
Avda. Universidad s/n
C.P. 10071 Cáceres
Spain
+34 927 257204
juanher@unex.es

ABSTRACT

Traceability of requirements and concerns enhances the quality of software development. We use trace relations to define crosscutting. As starting point, we set up a dependency matrix to capture the relationship between elements at two levels, e.g. concerns and representations of concerns. The definition of crosscutting is formalized in terms of linear algebra, and represented with matrices and matrix operations. In this way, crosscutting can be clearly distinguished from scattering and tangling. We apply this approach to the identification of crosscutting across early phases in the software life cycle, based on the transitivity of trace relations. We describe an illustrative case study to demonstrate the applicability of the analysis.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements Engineering – *Methodologies.*

General Terms

Theory.

Keywords

Aspect-Oriented Software Development, Traceability, Scattering, Tangling, Crosscutting, Crosscutting Concerns.

1. INTRODUCTION

Traceability is defined as 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 to one another [11]. In requirements engineering, these relationships describe trace dependencies between different artefacts such as requirements, stakeholder needs, design, system components, source code, etc. [17]. The trace dependencies can have different types, such as usage and abstraction dependencies (e.g. refinement and tracing [21]). By means of recording such dependencies, developers can improve software understanding and maintainability. Since a change in an early phase can be traced through the development process, traceability assists developers in quick evolving systems with new requirements or business domains' changes.

Traceability matrices [9] have been usually used to show such dependencies especially in early phases, because these matrices show the relationships between source and target elements both forward and backward. Adopting terminology from the Model Driven Architecture [15], we generically call *source* and *target* the two models or domains where trace dependencies are established. This situation is abstractly depicted in Figure 1, where there are trace dependencies among source and target elements. For simplicity, in this figure we only show two abstraction levels; however, multiple intermediate stages between source and target domains may exist.

In Aspect-Oriented Software Development (AOSD), crosscutting is usually described in terms of scattering and tangling. However, the distinction between these concepts is vague, sometimes leading to ambiguous statements and confusion, as stated in [12]:

.. the term "crosscutting concerns" is often misused in two ways: To talk about a single concern, and to talk about concerns rather than representations of concerns. Consider "synchronization is a crosscutting concern": we don't know that synchronization is crosscutting unless we know what it crosscuts. And there may be representations of the concerns involved that are not crosscutting.

We use these concepts based on our intuition and specific experience. For example, assume that the source elements of Figure 1 are concerns and the target elements are architectural components.

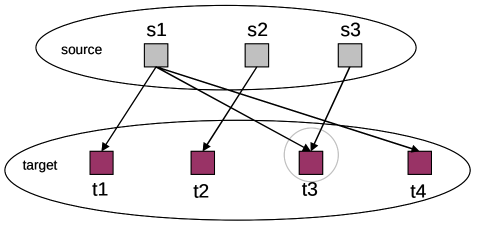

Figure 1. Trace relations between source and target elements

Intuitively, we would say that *s1* crosscuts *s3* for a given relation between source and target elements. However, vague definitions imply that it is not always possible to decide when a certain concept applies. Precise definitions are mandatory for the identification of crosscutting at any phase of the software life cycle, and to allow traceability of concerns from early phases.

In this paper we propose a definition of crosscutting based on an extension of traceability matrices, allowing developers both to identify crosscutting concerns in early phases [5] and to trace crosscutting concerns from early stages to subsequent phases of the software life cycle. Although there are other definitions of crosscutting in the literature, these definitions are usually very tied to the implementation level, such as [14]. A study of similarities and differences of such definitions and ours is out of scope of this paper. An extended description of our definition can be found in [6][7].

The rest of paper is structured as follows. In section 2, we introduce our definition of crosscutting. In section 3, we describe how to represent and visualize crosscutting in a matrix and how to derive this matrix from the dependency matrix using a scattering and tangling matrix. Examples of application in early phases and the transitivity of trace relations are shown in section 4. Then in section 5, we show a case study where we apply the concepts introduced in the previous sections. Finally in sections 6 and 7, we present related work and conclusions of the paper.

2. CROSSCUTTING FORMALIZATION

Our proposition is that crosscutting can only be defined in terms of 'one thing' with respect to 'another thing': at least two domains (or two levels or two phases) are related with each other in some way. For example:

- A *domain* refers for example to concerns in a concern model or to a decomposition of architectural elements.
- A *phase* refers to any phase in the software development life cycle (e.g. requirements, design, and so on)
- A *level* refers for example to models in the Model Driven Architecture [15] (e.g. CIM, PIM and PSM).

We use here the general terms *source* and *target* (as in [15]) to denote two consecutive domains, phases or levels.

We assume that elements in the source are related to elements in the target: there is a mapping between source and target elements. These mappings are captured in trace dependency relationships. The terms crosscutting, tangling and scattering are defined as specific cases of these mappings. We define *scattering* as the case where a source element is mapped to multiple target elements (and consequently has more than one trace relations to the target). We define *tangling* as the case where a target element is related to multiple source elements (and consequently has more than one trace relations to the source). We now define *crosscutting* as follows: *crosscutting occurs when, in a mapping between source and target, a source element is scattered over target elements and where in at least one of these target elements, one or more other source elements are tangled.* In other words, we say that source element s1 crosscuts source element s2 for a given mapping between source and target, if s1 is scattered over target elements and in at least one of these target elements, s1 is tangled with source element s2. These definitions will be explained in the following sections.

3. MATRIX REPRESENTATION

In this section we show how crosscutting can be represented and identified by means of an extension to traceability matrices. Trace relations are captured in a dependency matrix, representing the mapping between source and target. As an extension, we derive the crosscutting matrix from the dependency matrix. We describe how the crosscutting matrix can be constructed from the dependency matrix with some auxiliary matrices. This is illustrated with some examples.

3.1 Tracing from source to target.

Traceability matrices have been usually used to show the relationships between requirements elicitation and the representation of these requirements in a particular engineering approach (such as use cases [21] or viewpoints [10]).

In terms of linear algebra, traceability matrices show the mappings between source and target. We show these mappings in a special kind of traceability matrix that we called dependency matrix. *A dependency matrix (source x target) represents the dependency relation between source elements and target elements (inter-level relationship)*. In the rows, we have the source elements, and in the columns, we have the target elements. In this matrix, a cell with 1 denotes that the source element (in the row) is *mapped* to the target element (in the column). Reciprocally this means that the target element *depends on* the source element. Scattering and tangling can easily be visualized in this matrix (see the examples below).

We define a new auxiliary concept *crosscutpoint* used in the context of dependency diagrams, to denote *a matrix cell involved in both tangling and scattering*. If there are one or more crosscutpoints then we say we have crosscutting.

Crosscutting between source elements for a given mapping to target elements, as shown in a dependency matrix, can be represented in a crosscutting matrix. *A crosscutting matrix (source x source) represents the crosscutting relation between source elements, for a given source to target mapping (represented in a dependency matrix).* In the crosscutting matrix, a cell with 1 denotes that the source element in the row is crosscutting the source element in the column. In section 3.2 we explain how this crosscutting matrix can be derived from the dependency matrix.

A crosscutting matrix should not be confused with a coupling matrix. A *coupling matrix* shows coupling relations between elements at the same level (intra-level dependencies). In some sense, the coupling matrix is related to the design structure matrix [3]. On the other hand, a crosscutting matrix shows crosscutting relations between elements at one level with respect to a mapping onto elements at some other level (inter-level dependencies).

We now give an example and use the dependency matrix and crosscutting matrix to visualize the definitions (S denotes a scattered source element - a grey row; NS denotes a non-scattered source element; T denotes a tangled target element - a grey column; NT denotes a non-tangled target element). The example is shown in Table 1.

In this example, we have one scattered source element s[1] and one tangled target element t[3]. We apply our definition of crosscutting and arrive to the crosscutting matrix. Source element s[1] is crosscutting s[3] (because s[1] is scattered over [t[1], t[3], t[4]] and s[3] is in the tangled one of these elements, namely t[3]). The reverse is not true: the crosscutting relation is not symmetric.

Table 1. Example dependency and crosscutting matrix with tangling, scattering and one crosscutting

dependency matrix

		target				
		t[1]	t[2]	t[3]	t[4]	
source	s[1]	1	0	1	1	S
	s[2]	0	1	0	0	NS
	s[3]	0	0	1	0	NS
		NT	NT	T	NT	

crosscutting matrix

		source		
		s[1]	s[2]	s[3]
source	s[1]	0	0	1
	s[2]	0	0	0
	s[3]	0	0	0

3.2 Constructing crosscutting matrices

In this section, we describe how to derive the crosscutting matrix from the dependency matrix. We use a more extended example than the previous ones. We now show an example with more than one crosscutpoint, in this example 8 points (see Table 2; the dark grey cells).

Table 2. Example dependency matrix with tangling, scattering and several crosscuttings

dependency matrix

		target						
		t[1]	t[2]	t[3]	t[4]	t[5]	t[6]	
source	s[1]	1	0	0	1	0	0	S
	s[2]	1	0	1	0	1	1	S
	s[3]	1	0	0	0	0	0	NS
	s[4]	0	1	1	0	0	0	S
	s[5]	0	0	0	1	1	0	S
		T	NT	T	T	T	NT	

crosscutting matrix

		source				
		s[1]	s[2]	s[3]	s[4]	s[5]
source	s[1]	0	1	1	0	1
	s[2]	1	0	1	1	1
	s[3]	0	0	0	0	0
	s[4]	0	1	0	0	0
	s[5]	1	1	0	0	0

Based on the dependency matrix, we define some auxiliary matrices: the *scattering matrix* (source x target), and the *tangling matrix* (target x source).

These two matrices are defined as follows:

- In the scattering matrix a row contains only dependency relations from source to target elements if the source element in this row is scattered (mapped onto multiple target elements); otherwise the row contains just zero's (no scattering).

- In the tangling matrix a row contains only dependency relations from target to source elements if the target element in this row is tangled (mapped onto multiple source elements); otherwise the row contains just zero's (no tangling).

For our example in Table 2, these matrices are shown in Table 3.

We now define the crosscutting product matrix, showing the frequency of crosscutting relations. *A crosscutting product matrix (source x source) represents the frequency of crosscutting relations between source elements, for a given source to target mapping.* The crosscutting product matrix is not necessarily symmetric. The *crosscutting product matrix* ccpm can be obtained through the matrix multiplication of the scattering matrix sm and the tangling matrix tm: ccpm = sm . tm where $ccpm_{ik} = sm_{ij}\ tm_{jk}$

Table 3. Scattering and tangling matrices for dependency matrix in Table 2

scattering matrix

		target					
		t[1]	t[2]	t[3]	t[4]	t[5]	t[6]
source	s[1]	1	0	0	1	0	0
	s[2]	1	0	1	0	1	1
	s[3]	0	0	0	0	0	0
	s[4]	0	1	1	0	0	0
	s[5]	0	0	0	1	1	0

tangling matrix

		source				
		s[1]	s[2]	s[3]	s[4]	s[5]
target	t[1]	1	1	1	0	0
	t[2]	0	0	0	0	0
	t[3]	0	1	0	1	0
	t[4]	1	0	0	0	1
	t[5]	0	1	0	0	1
	t[6]	0	0	0	0	0

In this crosscutting product matrix, the cells denote the frequency of crosscutting. This can be used for quantification of crosscutting (crosscutting metrics). The frequency of crosscutting in this matrix should be seen as an upper bound. In actual situations, the frequency can be less than the frequency from this matrix analysis, because in the matrix we abstract from scattering and tangling specifics. In the crosscutting matrix, a matrix cell denotes the occurrence of crosscutting; it abstracts from the frequency of crosscutting.

The *crosscutting matrix* ccm can be derived from the crosscutting product matrix ccpm using a simple conversion: ccm_{ik} = if ($ccpm_{ik}$ > 0) $\wedge$ ($i \neq j$) then 1 else 0.

The crosscutting product matrix for the example is given in Table 4. From this crosscutting product matrix we derive the crosscutting matrix that we show in Table 2.

Table 4. Crosscutting product matrix for dependency matrix in Table 2

crosscutting product matrix

		source				
		s[1]	s[2]	s[3]	s[4]	s[5]
source	s[1]	2	1	1	0	1
	s[2]	1	3	1	1	1
	s[3]	0	0	0	0	0
	s[4]	0	1	0	1	0
	s[5]	1	1	0	0	2

In this example, there are no cells in the crosscutting product matrix larger than 1, except on the diagonal where it denotes a crosscutting relation with itself, which we disregard here. In the crosscutting matrix, we put the diagonal cells to 0. Obviously, this is because we interpret a source element can't crosscut itself.

As we can see in crosscutting matrix in Table 4, there are now 10 crosscutting relations between the source elements. The crosscutting matrix shows again that the crosscutting relation is not symmetric. For example, s[1] is crosscutting s[3], but s[3] is not crosscutting s[1] because s[3] is not scattered (scattering and tangling are necessary but not sufficient condition for crosscutting).

For convenience, these formulas can be calculated automatically by means of very simple mathematic tools. By filling in the cells of the dependency matrix, the other matrices are calculated automatically.

4. TRACEABILITY OF CROSSCUTTING

In this section, we apply our approach to the identification of crosscutting in the early phases of software development. Moreover, we consider properties of crosscutting across consecutive phases of the software lifecycle, based on transitivity of traceability relations.

4.1 Traceability in early phases

The extension to traceability matrices with a crosscutting matrix presented in this paper can be applied to any consecutive phases of the development process. In this section we show the application of our approach to early phases as a means to identify crosscutting in such phases.

Our approach abstracts from specific phases, such as concern modelling, requirements elicitation, architectural design and so on. The only proposition is that we define crosscutting for two phases (or levels), which we called source and target. This approach can be applied to early phases in software development, e.g. concerns and requirements, but also to other phases near implementation, e.g. a UML design and Java code. In each case, we have to define the trace relations between the respective source elements and target elements.

In Table 5, we show a table with examples of source and target elements. The result of the crosscutting analysis depends on the source and target. In section 5, we apply the approach in a case study.

Table 5. Crosscutting cases in Early Phases

	Source	Target	Result
Case 1	Concerns	Requirements	Crosscutting Concerns with respect to Requirements
Case 2	Requirem ents	Use Cases	Crosscutting Requirements with respect to Use Cases
Case 3	Concerns	Use Cases	Crosscutting Concerns with respect to Use Cases
Case 4	Requirem ents	Design Modules	Crosscutting Requirements with respect to Design Modules
Case 5	Concerns	Design Modules	Crosscutting Concerns with respect to Design Modules

4.2 Transitivity of trace relations

Usually we encounter a number of consecutive levels or phases in software development. From the perspective of software life cycle phases, we usually distinguish Domain Analysis, Concern Modelling, Requirement Analysis, Architectural Design, Detailed Design, and Implementation.

We consider here the cascading of two consecutive mappings: the target of the first mapping serves as source for the second one. For convenience, we call the first target our intermediate level, and second target just target (see Figure 2).

Each of these refinements can be described with a dependency matrix. We describe how to combine two consecutive dependency matrices, in an operation we call cascading. Cascading is an operation on two dependency matrices resulting in a new dependency matrix, which represents the dependency relation between source elements of the first matrix and target elements of the second matrix.

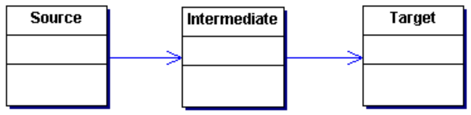

Figure 2. Cascading of consecutive levels

For cascading, it is essential to define the transitivity of dependency relations. Transitivity is defined as follows. Assume we have a source, an intermediate level, and a target as is shown in Figure 2. There is a dependency relation between an element in the source and an element in the target if there is some element at the intermediate level that has a dependence relation with this source element and a dependency relation with this target element. In other words, the transitivity dependency relation R for source s, intermediate level u and target t, and #u is the number of elements in u:

$$\exists\, k \in (1..\#u)\!: (s[i] \mathrel{R} u[k]) \wedge (u[k] \mathrel{R} t[m]) \Rightarrow (s[i] \mathrel{R} t[m])$$

We can also formalize this relation in terms of the dependency matrices. Assume we have three dependency matrices m1 :: s × u and m2 :: u × t and m3 :: s × t, where s is the source, u is some intermediate level, #u is the cardinality of u, and t is the target. The cascaded dependency matrix m3 = m1 cascade m2

Then, *transitivity* of the dependency relation is defined as follows:

$$\exists\, j \in (1..\#u)\!: m1[i,j] \wedge m2[j,k] \Rightarrow m3[i,k]$$

In terms of linear algebra, the dependency matrix is a relationship between two given domains, source and target (see section 3.1). Accordingly, the cascading operation can be generalized as a composition of relationships as follows. Let Dom_K, $k = 1..n$, be n domains, and let f_i be the relationship between domains Dom_i and Dom_{i+1}, $1 \le i < n$, denoted as $Dom_i \xrightarrow{f_i} Dom_{i+1}$. Let Source and Target be the domains Dom_1 and Dom_n, respectively. Consequently, we have the following relationship between the domains:

$$Source \xrightarrow{f_1} Dom_2 \xrightarrow{f_2} Dom_3 \xrightarrow{f_3} \ldots Dom_{n-1} \xrightarrow{f_{n-1}} Target$$

As a result, the dependency relationship between the Source and the Target is defined as $DM \equiv f_{n-1} \circ f_{n-2} \circ \ldots \circ f_1$. In this way, the dependence matrix between a source and target is obtained through matrix multiplication of the dependency matrices that represents each f_i, $1 \le i < n$.

Table 6. Two dependency matrices that will be cascaded

dependency matrix 1

	requirement			
concern	r[1]	r[2]	r[3]	r[4]
c[1]	1	0	0	1
c[2]	0	1	0	0
c[3]	0	0	1	1

dependency matrix 2

	module				
requirement	m[1]	m[2]	m[3]	m[4]	m[5]
r[1]	1	0	0	0	1
r[2]	0	1	0	0	0
r[3]	0	1	1	0	0
r[4]	0	0	0	1	1

As an example, we explain the cascading of two dependency matrices: one for concerns x requirements and one for requirements x modules. The two dependency matrices are shown in Table 6.

The first dependency matrix relates concerns with requirements. The second dependency matrix relates requirements with modules. The resulting dependency matrix relates concerns with modules (see Table 7). This matrix can be used to derive the crosscutting matrix for concern × concern with respect to modules.

The crosscutting matrix in Table 7 is not symmetric. Based on this matrix we conclude, for the given dependency relations between concerns and modules, that: concern c[1] is crosscutting concern c[3]; concern c[2] does not crosscut any other concern; concern c[3] is crosscutting concerns c[1] and c[2].

Table 7. The resulting dependency matrix and crosscutting matrix based on cascading of the matrices in Table 6

resulting dependency matrix

	module				
concern	m[1]	m[2]	m[3]	m[4]	m[5]
c[1]	1	0	0	1	2
c[2]	0	1	0	0	0
c[3]	0	1	1	1	1

crosscutting matrix

	concern		
concern	c[1]	c[2]	c[3]
c[1]	0	0	1
c[2]	0	0	0
c[3]	1	1	0

From this description, it is clear that cascading can be used for *traceability* analysis across multiple levels, e.g. from concerns to implementation elements, via requirements, architecture and design (c.f. [20]). We can trace concerns throughout the complete development process applying the crosscutting analysis in each level. Once the crosscutting concerns have been identified in a particular level, we can compare the results with the obtained in previous or subsequent levels. We can observe how crosscutting concerns can be identified in particular phases.

5. CASE STUDY

In this section, we show the application of our approach in a case study. This case has been used for some workshops, e.g. [22]. The case study implements a Conference Review System (CRS) [8]. For space reasons, we have used a simplification of this system. The general purpose of the system is to assist a conference's program committee to perform the review of papers and registration of participants of such conference.

There are four different user types in the system: PcChair, PcMembers, Authors and Participants. A PcChair is the main responsible of the review process. He has access to every paper and every review in the system. A PcMember takes over the reviews of the papers. A PcMember can see information of the papers but not reviews by other PcMembers. An Author can submit papers to the system. An Author can see only information about his own submission. A Participant must register in order to attend the conference. The register process is completely separated from the login process. However, once a user has registered he will need log whenever he accesses the system. This login process checks the role of the user in the system.

The use case model of the conference review system is shown in Figure 3. The complete requirements' analysis can be seen in [8].

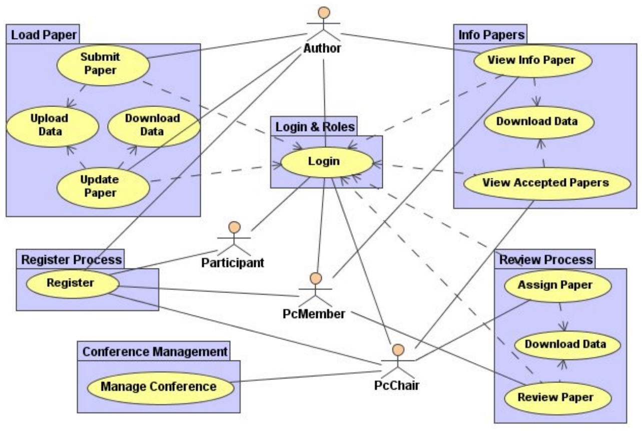

Figure 3. Use case model of the Conference Review System

We identified the following eight concerns: Papers Submission (PS), Papers Queries (PQ), Registration (Reg), Conference (C), Review (R), Information Retrieval/Supply (IRS), Login (L) and User Types (UT). Furthermore, we take the elements in the use case model (each package) shown in Figure 3 and the set of actors which take part in system as decomposition of requirements. We apply our approach to identify crosscutting in these domains. In Table 8a we show the dependency matrix with trace dependencies between concerns and requirements and in Table 8b the crosscutting matrix obtained from the former. Other decompositions of both concerns and requirements would be possible and the results obtained would be different.

Table 8. (a) Dependency matrix concerns x reqs and (b) crosscutting matrix for the Conference Review System

(a) dependency matrix (concerns x requirements)

		requirements							
		Register Process	Info Papers	Load Papers	Review Process	Conf. Manag.	Login& Roles	Actors	
concerns	PS	0	0	1	0	0	0	0	NS
	PQ	0	1	0	0	0	0	0	NS
	Reg	1	0	0	0	0	0	0	NS
	C	0	0	0	0	1	0	0	NS
	R	0	0	0	1	0	0	0	NS
	IRS	1	1	1	1	1	0	0	S
	L	0	1	1	1	1	1	0	S
	UT	0	0	0	0	0	0	1	NS
		T	T	T	T	T	NT	NT	

(b) crosscutting matrix (concerns x concerns) at requirements

		concerns							
		PS	PQ	Reg	C	R	IRS	L	UT
concerns	PS	0	0	0	0	0	0	0	0
	PQ	0	0	0	0	0	0	0	0
	Reg	0	0	0	0	0	0	0	0
	C	0	0	0	0	0	0	0	0
	R	0	0	0	0	0	0	0	0
	IRS	1	1	1	1	1	0	1	0
	L	1	1	0	1	1	1	0	0
	UT	0	0	0	0	0	0	0	0

As we can see in Table 8b, the Login concern crosscuts every concern where the user must authenticate and system must check the role of such user. Similarly, the Information Retrieval/Supply concern crosscuts the concerns which need an access to the correspondence information to perform their actions.

Once we have identified the crosscutting concerns with respect to the requirements domain, we can observe how the concerns are related to the design of the system. We show in Figure 4 a simple UML class diagram representing the static structure of the design.

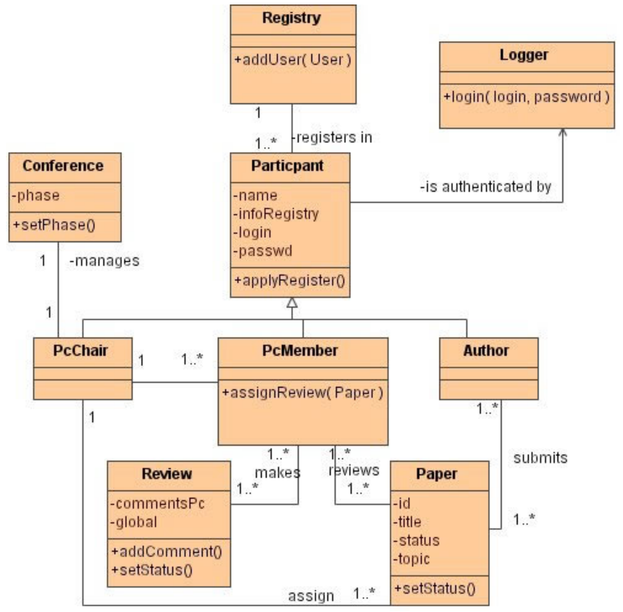

Figure 4. Structure diagram of Conference Review System

Now, we take the requirements as represented in the use case model as source elements, and the classes in the class diagram of the design as target elements. We can build the dependency matrix shown in Table 9 to show the trace dependencies between requirements and design elements.

Table 9. Dependency matrix requirements x design

		classes									
		Paper	Review	Conference	Pc Chair	Pc Member	Author	Participant	Logger	Registry	
Requirements	Register Process	0	0	0	0	0	0	1	0	1	S
	Info Papers	1	0	0	0	0	0	0	0	0	NS
	Load Papers	1	0	0	0	0	0	0	0	0	NS
	Review Process	0	1	0	0	0	0	0	0	0	NS
	Conf. Manag	0	0	1	0	0	0	0	0	0	NS
	Login& Roles	0	0	0	0	0	0	1	1	0	S
	Actors	0	0	0	1	1	1	1	0	0	S
		T	NT	NT	NT	NT	NT	T	NT	NT	

As we can see in Table 9, the trace dependencies between Requirements and classes are direct mappings except for *Register Process* and *Login&Roles* because of information added in the *Participant* class for such register and login purposes respectively (*infoRegister* and *login*, *passwd* attributes of Participant class). These requirements are tangled in such class with the own functionality of Participant class (User Type).

We apply the cascading operation (as defined in section 4.2) between the dependency matrix concerns x requirements (Table 8a) and the dependency matrix requirements x design (Table 9) to obtain trace dependencies between concerns and design elements. This derived dependency matrix concerns x design is shown in Table 10.

Table 10. Cascaded dependency matrix concerns x design

		classes									
		Paper	Review	Conference	Pc Chair	Pc Member	Author	Participant	Logger	Registry	
concerns	PS	1	0	0	0	0	0	0	0	0	NS
	PQ	1	0	0	0	0	0	0	0	0	NS
	Reg	0	0	0	0	0	0	1	0	1	S
	C	0	0	1	0	0	0	0	0	0	NS
	R	0	1	0	0	0	0	0	0	0	NS
	IRS	2	1	1	0	0	0	1	0	1	S
	L	2	1	1	0	0	0	1	1	0	S
	UT	0	0	0	1	1	1	1	0	0	S
		T	T	T	NT	NT	NT	T	NT	T	

Finally, applying our definition of crosscutting to last derived dependency matrix, we obtain the crosscutting matrix shown in Table 11.

Table 11. Crosscutting matrix based on cascaded dependency matrix in Table 10

		concerns							
		PS	PQ	Reg	C	R	IRS	L	UT
concerns	PS	0	0	0	0	0	0	0	0
	PQ	0	0	0	0	0	0	0	0
	Reg	0	0	0	0	0	1	1	1
	C	0	0	0	0	0	0	0	0
	R	0	0	0	0	0	0	0	0
	IRS	1	1	1	1	1	0	1	1
	L	1	1	1	1	1	1	0	1
	UT	0	0	1	0	0	1	1	0

From this matrix we can observe that - with respect to the design - we have obtained some new crosscutting concerns. The Reg concern crosscuts the IRS, L and UT. Similarly, the UT crosscuts the IRS, L and UT. As we showed in dependency matrix obtained by means of the cascading operation (see Table 10), all these concerns are scattered in several design modules and in at least one of these modules other concern is tangled.

Obviously, this conclusion about crosscutting very much depends on the decomposition at each level and the dependencies between elements at these levels. There are many alternatives, which could aim at avoiding crosscutting by using another modularization (e.g. aspect-oriented techniques such as [4]). Here, we showed how to analyse crosscutting across several phases in the software life cycle.

6. RELATED WORK

Several authors use matrices (design structure matrices, DSM) to analyze modularity in software design [3]. Lopes and Bajracharya [13] describe a method with clustering and partitioning of the design structure matrix for improving modularity of object-oriented designs. However, the design structure matrices represent intra-level dependencies (as coupling matrices in section 3.1) and not the inter-level dependencies as in the dependency matrices used for our analysis of crosscutting. In [18], a relationship matrix (concern x requirement) is described very similar to our dependency matrix, and used to identify crosscutting concerns. However, there is no formalized definition of crosscutting.

The approach presented in [2] allows the requirements engineer to identify crosscutting concerns. However, the identification of crosscutting functional concerns is not yet clear and it lacks explicit support (e.g. guidelines) to identify non-functional crosscutting concerns. In [19] the authors have improved this approach by means of a mechanism based on a natural language processor to identify functional and non-functional crosscutting concerns from requirements documents. However this approach is focused only on requirements phases while our approach can be applied throughout the software life cycle.

The papers described above lack the application of their definition of crosscutting to consecutive levels. We used our formalization to trace crosscutting concerns across levels of a software development process, as shown by the cascading operation.

A definition of crosscutting similar to ours can be found in [14] and [16]. Our definition is less restrictive as explained in [6]. Moreover, our definition can be applied to consecutive levels of abstractions in software development, such as requirements, design and implementation. This can be achieved through the cascading of dependency matrices as shown in section 4.

7. CONCLUSION

We proposed a definition of crosscutting based on an extension to traceability matrices. In a dependency matrix we show the mappings between source and target. As an extension, we used this matrix to derive the crosscutting matrix and to identify crosscutting. This can be applied to any phases in a software development process, also in early phases. The approach can be applied in systems where well known crosscutting concerns exist but also in systems where new crosscutting concerns are identified.

An interesting application is the cascading of crosscutting, which can be used to model crosscutting relations across several levels, for example from concern modelling, to requirements, architectural design to detailed design and implementation. As such, it provides an approach for traceability analysis. We showed the application of the approach in a case study to identify crosscutting. The operationalization of crosscutting with matrices constitutes a helpful means to analyse crosscutting in different scenarios or domains. Further research should show the scalability of this approach and provide support for different types of trace relations.

ACKNOWLEDGEMENT

This work has been carried out in conjunction with the AOSD-Europe Project IST-2-004349-NoE (see [1]) and also partially supported by CICYT under contract TIN2005-09405-C02-02.

REFERENCES

[1] AOSD-Europe (2005). *AOSD Ontology 1.0 - Public Ontology of Aspect-Orientation.* Retrieved May, 2005, from http://www.aosd-europe.net/documents/d9Ont.pdf.

[2] Araujo, J., Moreira, A., Brito, I. & Rashid, A. (2002), Aspect-Oriented Requirements with UML, In *Workshop on Aspect-Oriented Modelling with UML* at *International Conference on Unified Modelling Language*. Dresden, Germany.

[3] Baldwin, C.Y. & Clark, K.B. (2000). *Design Rules vol I, The Power of Modularity.* MIT Press.

[4] Baniassad, E. & Clarke, S. (2004). Theme : An Approach for Aspect-Oriented Analysis and Design. In *26th International Conference on Software Engineering*, Edinburgh, Scotland.

[5] Baniassad, E., Clements, P., Araújo, J., Moreira, A., Rashid, A.& Tekinerdogan, B. (2006). Discovering Early Aspects. In *IEEE Software*, vol. 23, nº 1, pp. 61-70.

[6] Berg, K. van den, & Conejero, J. M. (2005), A Conceptual Formalization of Crosscutting in AOSD. In *Iberian Workshop on Aspect Oriented Software Development*, TR-24/05 University of Extremadura (pp. 46-52). Granada, Spain.

[7] Berg, K. van den, & Conejero, J. M. (2005b). *Disentangling crosscutting in AOSD: A conceptual framework.* Paper presented at the EIWAS2005, Brussels

[8] Cachero. C., Gómez, J., Párraga, A. & Pastor, O. (2001). Conference Review System: A Case of Study. In [22].

[9] Davis, A. (1993). *Software Requirements: Objects, Functions and States.* Prentice–Hall, Second Edition.

[10] Finkelstein, A. & Sommerville, I. (1996). The Viewpoints FAQ. *BCS/IEE Software Engineering Journal*, 11(1)

[11] Institute of Electrical and Electronics Engineers (1990) *IEEE Standard Computer Dictionary.* New York.

[12] Kiczales, G. *Crosscutting*. AOSD.NET Glossary 2005. At http://aosd.net/wiki/index.php?title=Crosscutting.

[13] Lopes, C.V. & Bajracharya, S.K. (2005). An analysis of modularity in aspect oriented design. In 4th *International Conference on Aspect-Oriented Software Development*. Chicago, Illinois.

[14] Masuhara, H. & Kiczales, G. (2003). Modeling Crosscutting in Aspect-Oriented Mechanisms. In *17th European Conference on Object Oriented Programming*. Darmstadt.

[15] MDA (2003). MDA Guide Version 1.0.1, document number omg/2003-06-01.

[16] Mezini, M. & Ostermann, K. (2003). Modules for Crosscutting Models. In *8th International Conference on Reliable Software Technologies*. LNCS 2655. Toulouse, France.

[17] Ramesh, B. & Jarke, M. (2001). Toward reference models for requirements traceability. *IEEE Transactions on Software Engineering,* 27(4):58–93.

[18] Rashid, A., Moreira, A. & Araujo, J. (2003). Modularisation and Composition of Aspectual Requirements. In *Second Aspect Oriented Software Conference.* Boston, USA.

[19] Sampaio, A., Loughran, L., Rashid, A. & Rayson, P. (2005). Mining Aspects in Requirements. In *Early Aspects 2005 Workshop* at *Aspect Oriented Software Development Conference*. Chicago, USA.

[20] Tekinerdogan, B. (2004). ASAAM: Aspectual Software Architecture Analysis Method. In *4th Working IEEE/IFIP Conference on Software Architecture.*

[21] UML (2004). *Unified Modeling Language 2.0 Superstructure Specification*. Retrieved October, 2004 from http://www.omg.org/cgi-bin/doc?ptc/2004-10-02

[22] *First International Workshop on Web-Oriented Software Technology*. (2001). http://www.dsic.upv.es/~west/iwwost01/ . Valencia, Spain