Science of Computer Programming 75 (2010) 1113–1141

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Mining early aspects based on syntactical and dependency analyses

José M. Conejero [a,*], Juan Hernández [a], Elena Jurado [a], Klaas van den Berg [b]

[a] Quercus Software Engineering Group, University of Extremadura, Avda. de la Universidad, s/n, 10071, Spain
[b] Software Engineering Group, University of Twente, 7500 AE Enschede, The Netherlands

ARTICLE INFO

Article history:
Received 15 July 2009
Received in revised form 15 April 2010
Accepted 16 April 2010
Available online 12 May 2010

Keywords:
Requirements engineering
Aspect mining
Crosscutting concerns
Concern-oriented metrics

ABSTRACT

Aspect-Oriented Requirements Engineering focuses on the identification and modularisation of crosscutting concerns at early stages. There are different approaches in the requirements engineering community to deal with crosscutting concerns, introducing the benefits of the application of aspect-oriented approaches at these early stages of development. However, most of these approaches rely on the use of Natural Language Processing techniques for aspect identification in textual documents and thus, they lack a unified process that generalises its application to other requirements artefacts such as use case diagrams or viewpoints. In this paper, we propose a process for mining early aspects, i.e. identifying crosscutting concerns at the requirements level. This process is based on a crosscutting pattern where two different domains are related. These two different domains may represent different artefacts of the requirements analysis such as text and use cases or concerns and use cases. The process uses syntactical and dependency based analyses to automatically identify crosscutting concerns at the requirements level. Validation of the process is illustrated by applying it to several systems and showing a comparison with other early aspects tools. A set of aspect-oriented metrics is also used to show this validation.

1. Introduction

Enhancing business performance in contemporary domains (e.g. e-commerce, financial environments, home automation) requires systems whose size and intricacy challenge most of the current software engineering methods and tools. From early stages in the development of enterprise computing systems to their maintenance and evolution, a wide spectrum of methodologies, models, languages, tools and platforms are adopted. Aspect-Oriented Software Development (AOSD) is one of these methodologies that has emerged as a strong alternative to tackle the design and development of complex software systems [1]. Modularity and abstraction are essential techniques for managing such complexity and aspect-orientation has appeared with the goal of supporting improved modularity of software systems, emphasising modular structures that cut across traditional abstraction boundaries [2].

AOSD has the *separation of concerns* principle and their further integration as key factors to obtain high-quality and evolvable large software systems. This principle refers to the process of partitioning the software into different features that address the functionality of a system [3]. However, and because of the complexity of modern software systems, *separation of concerns* inevitably leads to the problem of crosscutting concerns [1], which is usually described in terms of scattering and tangling [4]. Scattering occurs when the realisation of a concern is spread over the software modules resulting in the decomposition of the system, whilst tangling occurs when the concern realisation is mixed with other concerns in a module. AOSD has just focused on providing new abstractions for modelling crosscutting concerns allowing their separate design and implementation, and further integration (weaving) with the components of the system [1].

* Corresponding author. Tel.: +34 927 25 7195; fax: +34 927 25 7202.
E-mail addresses: chemacm@unex.es, chemacm@gmail.com (J.M. Conejero), juanher@unex.es (J. Hernández), elenajur@unex.es (E. Jurado), k.g.vandenberg@ewi.utwente.nl (K. van den Berg).

0167-6423/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2010.04.013

One of the main challenges in aspect-orientation relies on aspect identification. AOSD is meaningless unless crosscutting concerns are properly identified in software systems. Logging, tracing and security are known to be crosscutting concerns but, certainly, as Gregor Kickzales states in [5]: "*we don't know that they are crosscutting unless we know what they crosscut*". Aspect mining refers to the process of identifying crosscutting concerns throughout an existing software system which can then be refactored using aspect-oriented techniques [6]. Most of the systematic studies of aspect mining (e.g. [7–12]) concentrate on the analysis of source code, when architectural decisions have already been made. However, crosscutting concerns manifest in early development artefacts, such as requirements descriptions [4] and architectural models [13,14], due to their widely-scoped influence in software decompositions. They can be observed in every kind of requirements and design representations, such as use cases and component models [15,4,13,14]. In that sense, the Early Aspects community has focused on dealing with crosscutting properties at early phases [16], and have coined the term "early aspect" to refer to crosscutting concerns at early development stages. An early aspect is defined as: "*a concern that crosscuts an artefact dominant decomposition, or base modules derived from the dominant separation of concerns criterion, in the early stages of the software life cycle*" [15]. Thus, for example, an early aspect in requirements is a concern that crosscuts requirements artefacts [15].

As at the implementation level, aspect mining techniques have also been introduced at early phases to be able to identify and modularise crosscutting concerns earlier, incorporating the benefits of aspect-orientation from early stages of software development. EA-Miner [17] and Theme/DOC [18] are two of these approaches. However, on the one hand these approaches lack a formal definition of crosscutting to be based on. In some cases, precise definitions are mandatory to allow tool support. On the other hand, these approaches rely on the use of Natural Language Processing techniques to identify crosscutting concerns at this level. Thus, although these proposals contribute to aspect mining at the requirements level, they cannot be applied to requirements artefacts other than text. Nevertheless, an important application area of mining early aspect is the refactoring of legacy systems [19], which are usually described using other requirements artefacts, such as UML use cases or viewpoints.

In this context, the major contributions of this paper are threefold. First, it presents an aspect mining process based on syntactical and dependency analyses at the requirements level. Unlike other previous works, our aspect mining process is based on a conceptual framework [4] that is independent of specific requirements artefacts. The conceptual framework provides a formal definition of crosscutting based on the trace relations or mappings that exist between two different domains, source and target (e.g. concerns and requirement statements or concerns and use cases). The syntactical and dependency analyses allow the process to be automated since the existing mappings between the two domains are automatically obtained. This is a new and important contribution with respect to the framework presented in [4]. Second, early aspect refactoring is given for UML use cases diagrams. This refactoring allows early aspects to be properly modularised from the requirements level, improving the modularity of the system since the crosscutting concerns are isolated. Moreover, the system may be easily evolved by just using simple composition rules which allow the weaving of base and crosscutting concerns. Third, our process is both validated by the utilisation of a set of concern-oriented metrics at requirements level and compared with other early aspect mining proposals. The comparative study is particularly useful as a benchmark for other aspect mining approaches. The addition of an empirical analysis (based on the metrics) to the process also supposes a new contribution with respect to our previous work in [4] which lacks this empirical analysis. Although the metrics used here were also introduced in [20], the aspect mining process presented here was not previously used to calculate the metrics.

The rest of the paper is structured as follows. In Section 2 we summarise the conceptual framework that we presented in [4]. Section 3 describes our aspect mining process, including all the steps that must be performed to identify the crosscutting concerns at requirements. This section also illustrates the process of refactoring UML use cases diagrams once early aspects have been identified. Section 4 presents the validation of the aspect mining process and the comparison with other approaches. Finally, Sections 5 and 6 discuss related works and conclude this paper.

2. Characterising and identifying crosscutting concerns

When talking about aspect-orientation, we use concepts for which we have some intuition based on our specific experience. We share these concepts with others who may have a similar intuition usually based on another experience. However, the definitions of the concepts are sometimes not consistent with other concepts. Vague definitions imply that it is not always possible to decide when a certain concept applies. When do we have just scattering? When do we have just tangling? When do we have crosscutting and when not? Whatever the stage of software development, precise definitions of these concepts are mandatory, especially in the aspect mining area for providing automatic identification of crosscutting concerns.

Our proposed early aspect mining approach is based on a previously-defined conceptual framework [4], where concern properties, such as scattering, tangling and crosscutting are formally defined. This conceptual framework supports the characterisation and identification of crosscutting, and it is briefly summarised in this section. Section 2.1 describes the key definitions of this framework. Section 2.2 illustrates how traceability matrices can be used for the identification of crosscutting concerns.

2.1. A conceptual framework for analysing crosscutting dependencies

Our previous work [4] presented a conceptual framework where a formal definition of the aforementioned concern properties, scattering, tangling and crosscutting were provided. This framework is based on the study of the trace

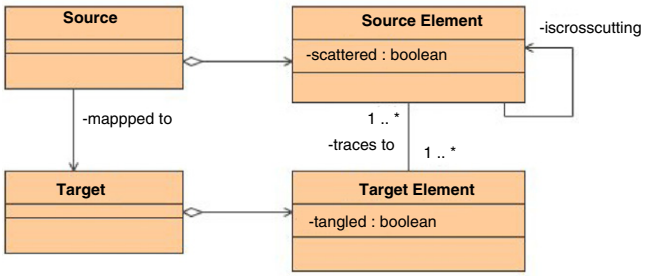


Fig. 1. The crosscutting pattern.

dependencies that exist between two different domains. These domains, generically called source and target, could be, for example, concerns and requirement statements respectively, or concerns and use cases or, in a different situation, design modules and programming artefacts. We use the term crosscutting pattern (Fig. 1) to denote the situation where source and target are related to each other through trace dependencies.

From a mathematical point of view this means that the domains source and target are related to each other through a mapping or trace relationship. The relationship between source and target can be represented through traceability matrices, which can be formalised as outlined below.

According to Fig. 1, there exists a multivalued function f' from source to target domains such that if $f'(s) = t$, then there exists a trace relation between s and t. Analogously, we can define another multivalued function g' from target to source that can be considered as a *special* inverse of f'. If f' is not a surjection, we consider that target is the range of f'. Obviously, f' and g' can be represented as single-valued functions considering that the codomains are the set of non-empty subsets of target and source, respectively.

Let f: *Source* $\longrightarrow \mathscr{P}$ *(Target)* and g: *Target* $\longrightarrow \mathscr{P}$ *(Source)* be these new functions defined by:
$\forall s \in Source, f(s) = \{t \in Target : f'(s) = t\}$
$\forall t \in Target, g(t) = \{s \in Source : g'(t) = s\}$.

The concepts of scattering, tangling and crosscutting are defined as specific cases of these functions.

Definition 1 (*Scattering*)**.** We say that an element $s \in Source$ is scattered if $card(f(s)) > 1$, where $card(f(s))$ refers to cardinality of $f(s)$. In other words: *scattering occurs when, in a mapping between source and target, a source element is related to multiple target elements.*

Definition 2 (*Tangling*)**.** We say that an element $t \in Target$ is tangled if $card(g(t)) > 1$. Then: *tangling occurs when, in a mapping between source and target, a target element is related to multiple source elements.*

There is a specific combination of scattering and tangling which we call crosscutting.

Definition 3 (*Crosscutting*)**.** Let $s1, s2 \in Source$, $s1 \neq s2$, we say that $s1$ crosscuts $s2$ if $card(f(s1)) > 1$ *and* $\exists t \in f(s1)$: $s2 \in g(t)$. In other words: *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, some other source element is tangled.*

According to the previous definitions, the following result is obvious.

Lemma. *Let* $s1, s2 \in Source$, $s1 \neq s2$, *then* $s1$ *crosscuts* $s2$ *if* $card(f(s1)) > 1$ *and* $f(s1) \cap f(s2) \neq \emptyset$.

2.2. Identification of crosscutting

In [4], we defined a special kind of traceability matrix that we called a *dependency matrix* to represent function f. An example of a dependency matrix with five source and six target elements respectively is shown in Table 1. In the rows, we have the source elements, and in the columns, we have the target elements. A1 in a cell denotes that the target element of the corresponding column contributes or addresses the source element of the corresponding row (in Table 1, $s[1]$ is mapped onto $t[1]$ and $t[4]$, that means that $t[1]$ and $t[4]$ address $s[1]$). Based on this matrix, two different matrices called scattering matrix and tangling matrix are derived, which show the scattered and tangled elements in a system respectively (see Table 2):

- In a 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 zeros (no scattering).
- In a 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 zeros (no tangling).

The crosscutting product matrix is obtained through the multiplication of the scattering matrix and tangling matrix. The crosscutting product matrix shows the quantity of crosscutting relations and is used to derive the final crosscutting matrix. Table 3 shows the crosscutting product and crosscutting matrices for the example. Note that the crosscutting product matrix is not a binary matrix since it contains the result of scattering and tangling matrices product. We will show in Section 3.5 how these values are used to define different modularity metrics. On the other hand, the crosscutting matrix is obtained by

Table 1
Example dependency matrix.

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

Table 2
Scattering and tangling matrices for dependency matrix shown in Table 1.

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

Table 3
Crosscutting product and crosscutting matrix for dependency matrix in Table 1.

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

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

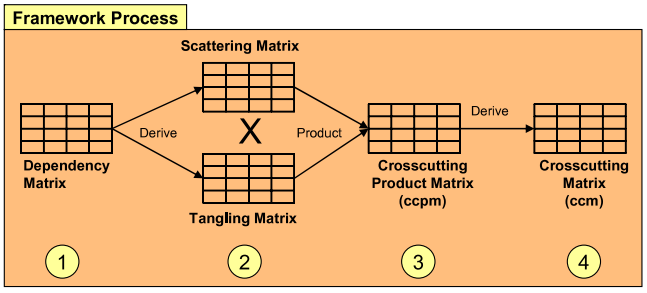


Fig. 2. Overview of steps in the framework.

converting the crosscutting product matrix into a binary matrix. In the crosscutting matrix, a matrix cell just denotes the occurrence of crosscutting; it abstracts from the quantity of crosscutting. The crosscutting matrix *ccm* can be derived from the crosscutting product matrix *ccpm* using a simple conversion: $ccm[i][k] = if\ (ccpm[i][k] > 0) \wedge (i \neq j)\ then\ 1\ else\ 0$. Using this simple conversion, any value higher than one is converted into one and the values of the diagonal are set to zero (obviously we consider that a source element cannot crosscut itself).

Fig. 2 shows the whole process to obtain the final crosscutting matrix. More details about the conceptual framework and the matrix operations can be found in [4].

The crosscutting pattern summarised in this section has different application areas, such as the identification of crosscutting or the definition of aspect-oriented metrics. However, its generic property is one of its main contributions. Since the crosscutting pattern is not defined in terms of any specific development artefact, it is not tied to any abstraction level. That implies that it may be used at any development phase just selecting the corresponding source and target domains. As an example, a first analysis of crosscutting was presented in [21,4] at the design and requirements levels, respectively. However, in this paper this analysis is extended by providing an automatic methodology to obtain the crosscutting concerns (including an empirical analysis). Note that the main purpose of the process is to assess modularity in software systems. Modularity is not restricted to any abstraction levels, e.g. class diagrams at design. Any modelling language introduces constructs for grouping entities as a way of modularity. For instance, use case models group functionalities into use cases. This is why

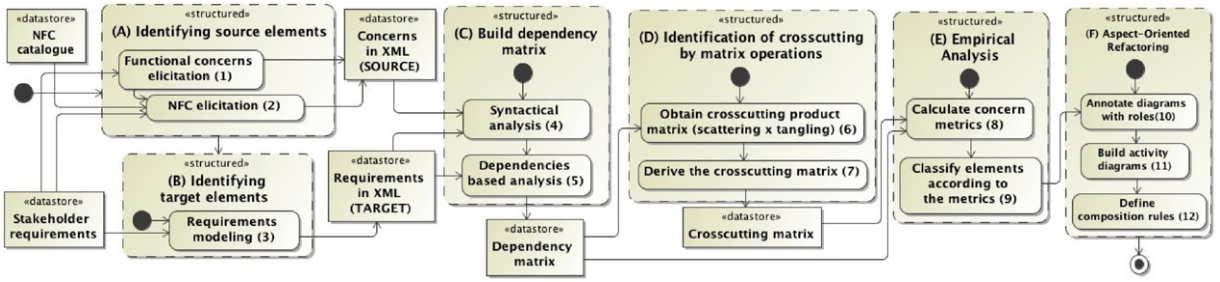


Fig. 3. Main phases of the aspect mining process.

we generalise the concept of crosscutting using source and target domain. Thus, considering the different models that may compose a system (e.g. concern models, use case models, structural models, behavioural models) we analyse trace relations between them and discover concerns that crosscut to any of these models.

3. The early aspect mining process

In this section, we propose an early aspect mining process based on the framework presented in Section 2. We have extended our framework with syntactical and dependency based analyses for identifying and managing crosscutting concerns at the requirements level. These analyses allow us to automatically correlate elements of source domain to elements of target domains, providing, thus, an automated traceability method between these domains. Note that automated traceability methods at requirements aim to decrease the effort needed to construct and maintain traceability links and provide traceability across a much broader set of documents [46].

As mentioned previously, the crosscutting pattern is not defined in terms of specific abstraction levels so that it may be applied in any development stage. Nevertheless, in this paper a specific instantiation of the process to be applied at the requirements level has been provided. In that sense, the process has been applied using use case diagrams as the target domain. Note that use cases usually specify functionality belonging to different concerns, so that these concerns are tangled in them. Then, we consider that this is a clear situation where aspect mining comes into play. In this setting, the aspect mining process presented in this paper aims to identify crosscutting situations based on concerns scattered over different use cases and where other concerns are tangled.

Our approach in a nutshell.

The main steps of our approach are outlined in Fig. 3 and summarised as follows:

(A) **Identifying source elements**. Requirements are usually represented in several documents and they are provided from different interviews with stakeholders. We analyse these requirements to identify the main concerns: functional (Fig. 3(1)) and non-functional (Fig. 3(2)). To identify the non-functional concerns (NFCs), we use a catalogue where common NFCs are defined. Both functional and NFCs are represented in XML format.

(B) **Identifying target elements.** In this phase, requirements are modelled using use cases (Fig. 3(3)). As with concerns, requirements are also represented in an XML format, exporting the use case diagrams to XMI [28].

(C) **Build the dependency matrix.** Taking concerns and requirements as source and target respectively we establish the trace relations between them; this is the function f defined in Section 2.1. These trace relations are automatically established by means of syntactical (Fig. 3(4)) and dependencies based (Fig. 3(5)) analyses so that the dependency matrix is automatically obtained.

(D) **Identification of crosscutting by matrix operations**. The next step consists of the application of several simple matrix operations (Fig. 3(6) and (7)), shown in Fig. 2, to obtain the crosscutting concerns at the requirements level. Both, the crosscutting product and crosscutting matrices may be used for assessing the degree of crosscutting in the system (we use them to establish several concern-oriented metrics, summarised in Section 3.5).

(E) **Empirical analysis**. Based on the matrices obtained in the previous step, a set of concern metrics that allow the quantification of crosscutting properties are calculated (Fig. 3(8)). These metrics are also based on the crosscutting pattern so that they are not tied to any specific deployment artefact. They were introduced in [20]. However, in this paper, they are incorporated into the aspect mining process presented. Based on the values obtained for the different metrics, the source elements are classified according to their degree of scattering or crosscutting whilst target elements are classified according to their degree of tangling (Fig. 3(9)).

(F) **Aspect-oriented refactoring**. Finally, the crosscutting concerns identified are modelled using aspect-oriented techniques (Fig. 3(10) and (11)). By means of this refactoring, these crosscutting concerns are isolated and encapsulated in separated entities, improving modularity and the reusability of the system. Using simple composition rules, the system may be composed later by weaving the crosscutting concerns identified with the base system (Fig. 3(12)).

The example: a Concurrent File Versioning System (CFVS)

To illustrate the process, we apply our approach to a simplification of the case study presented in [23], the CFVS. The CFVS allows different versions of files of a project to be maintained. It also allows a group of developers to work on the same

Table 4
Requirements of the CFVS system.

	Description
r1.	Users should be able to insert files into a project. A message is stored with the inserted files.
r2.	Users can retrieve files from a project, these files are a copy and are saved on the computer of the user.
r3.	Users are able to change a file. This is an action that occurs outside the version system but it is necessary because a versioning system is useless without change.
r4.	Users can commit files to a project. This means that the file is updated with the working copy of the user. A message is stored with the committed files and the version number is updated.
r5.	Users can update a working file, it updates the working copy with the latest version available in the project.
r6.	Both commit and update actions have to perform some form of conflict management, in the case that a file has been changed in the central system while the user made some changes to its working copy.
r7.	A user can ask the system to show the message that is stored with a certain version of a file.
r8.	Users can ask the system to show differences between two versions of a file in a certain project.
r9.	Users can remove files from projects.
r10.	Users can undo a file, which means that a file is restored.
r11.	A user can label (tag) a specific set of files (file versions), thus taking a static snapshot of those files. After the release and continued development, a tagged set of files can be retrieved as if they were never changed.
r12.	A user can branch a set of files off the main trunk or other branch. A branch can only be constructed when the set of files has been tagged. A branch does not disturb the main trunk or any other branch, thus making them useful for testing bug fixes or new features.
r13.	A user can merge a branch back into the branch where it originated from. The merge action is similar to the update action;
	it only merges two branches and has to also do some conflict management in the same manner as the action merge and update.
r14.	The administrator is responsible for project management (placeholder for the different branches) and for assigning permissions to different users.
r15.	The administrator also has the possibility to monitor different activities that occur in the system (especially check-in operations).
r16.	The system should also be able to store the different users and their permissions, restricting the project users' access.
r17.	The system should support concurrent usage of the versioning system.

project and to modify the same files at the same time. In this running example, a developer may download a version of a software project (**check-out**). After changing the current version of the project, the developer may upload a new version of the project to the CFVS system (**check-in**). Table 4 shows the main requirements that the system must fulfill (extracted from [23]). Based on these requirements we perform the whole analysis explained in the following sections.

3.1. Identifying source elements

The first step of the process is to decide the main concerns that the system must address. There are some works about finding concerns on a system (e.g. [24]). However they are usually focused on programming phases. In our process, we use the requirements documents to obtain the concerns of the system. We distinguish two different kinds of concerns, functional and NFCs, related to functional and non-functional requirements, respectively.

3.1.1. Elicitation of functional concerns

To identify the functional concerns (Fig. 3(1)), the requirements of the system are analysed. Each requirement may address one or more concerns. The ideal situation would be to have a one-to-one relationship between concerns and requirements. However, this situation is not always possible in real systems, and concerns are usually scattered over the requirements and tangled with other concerns, so that crosscutting concerns emerge. Concern scoping is one of the major issues in aspect-orientation. Sometimes the task of discovering concerns is really difficult and the decision of what is a concern and what is not a concern is left to the developers' expertise. In our running example we have used the same concerns that were identified by the original authors in [23]. These concerns are described in Table 5. The identification of the functional concerns is beyond the scope of this paper since it focuses mainly on the automatic identification of the mappings between source and target domains. Based on these mappings we automate the identification of crosscutting. However, some techniques to identify the functional concerns could be used to provide a higher degree of automation in the process. As an example, concerns are usually extracted by analysing the results of other requirements elicitation techniques, e.g. stakeholders' interviews transcripts. Other techniques tackle the semi-automatic identification of these concerns using different heuristics, usually based on the semantic analysis of the text in the requirement documents. Examples of concern modelling techniques at an early abstraction level are COSMOS [25] or EA-Miner [17]. Nevertheless, most of the techniques to automatically identify concerns are based on the analysis of source code at the programming level, e.g. FEAT [24] (a deeper comparison of these techniques may be found in [26]).

Once the functional concerns are identified, they are represented in a XML file. This file will be automatically processed in later steps of the process. We use a XML file with simple ⟨*concern*⟩ tags. Each ⟨*concern*⟩ tag may have three sub-elements: ⟨*description*⟩, ⟨*stakeholder*⟩ and ⟨*keyword*⟩. The tag ⟨*stakeholder*⟩ is used to identify who is interested in the concern. This tag may contain three different children: ⟨*user*⟩, ⟨*administrator*⟩ or ⟨*developer*⟩. The ⟨*keyword*⟩ tag represents the word that

Table 5
Functional concerns in the CFVS system.

Concern	Description
c1	Insert File
c2	Retrieve File
c3	Commit File
c4	Update Working Files
c5	Remove Files
c6	Restore File
c7	Store Message
c8	Retrieve Message
c9	Difference
c10	Tag a Set of Files
c11	Branch a Set of Files
c12	Merge Set of Files

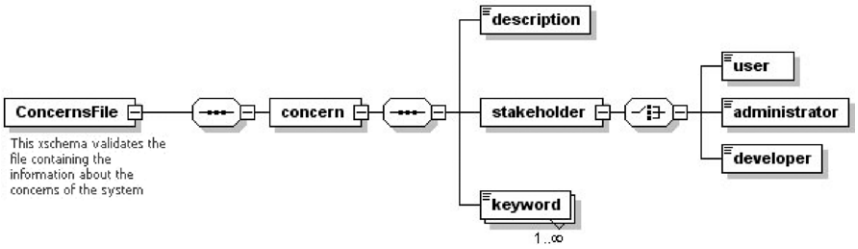


Fig. 4. XML-Schema to validate the concerns file.

```
<?xml version="1.0" encoding="UTF-8"?>
 <concern id="c1" name="Insert file">
   <description>Concern related to… </description>
   <stakeholder>
      <user>Final User</user>
   </stakeholder>
   <keyword>Insert</keyword>
 </concern>
  ...
 <concern id="c13" name="Persistence">
   <description>Concern related to… </description>
   <stakeholder>
      <developer>Analyst</developer>
   </stakeholder>
   <keyword>store</keyword>
   <keyword>storage</keyword>
   <keyword>data</keyword>
 </concern>
</ConcernsFile>
```

Fig. 5. Example of functional and NFCs in XML format.

we use to relate this concern with elements of the target domain (use cases). We explain later how to use this tag. In Figs. 4 and 5 we show a representation of the XML-Schema defined to validate this format and an example of several concerns represented in XML, respectively.

In [46], Cleland-Huang et al. introduced a set of best practices to automate traceability in requirements stages. One of the best practices presented by the authors consists of the building of a project glossary where the keywords of the project and specific domain are defined. In that sense, the definition of the concerns file presented in this section is extremely related to this technique since the main functionalities (concerns) and the keywords related to them are also defined. Moreover, the authors demonstrated the benefits obtained in projects where this glossary was defined at the very beginning of the development process, in contrast to the projects where it was created at the end of the development.

3.1.2. Elicitation of non-functional concerns

We apply a syntactic analysis based on identifiers or keywords to elicit the NFCs (Fig. 3(2)). To apply this analysis, a NFC catalogue is used. The catalogue consists of an XML file where common NFCs are presented and related to different words that usually appear in requirements documents. The catalogue is an extension of the one used by the EA-Miner tool since it was completed with new words related to NFCs. NFC decomposition is considered since each concern is related to several different words (which may represent different granularity levels of the NFCs). These words are used to analyse the stakeholder requirements so that NFCs are identified when one of these words appears in the text. In Fig. 6 we may observe an example which relates the words authorise and permission to the security concern defined in the catalogue.

```
<?xml version='1.0' encoding='utf-8'?>
<lexicon>
   <word content="authorise" nfr="security"/>
   <word content="permission" nfr="security"/>
</lexicon>
```

Fig. 6. Part of the catalogue of NFCs.

Table 6
Finding NFCs in the CFVS requirements.

Persistence	<word content="**stored**" nfr="**Persistence**"> </word>
r1.	A message is **stored** with the inserted files.
r4.	A message is **stored** with the committed files ...
r7.	… show the message that is **stored** with a certain version ....
Persistence	<word content="**saved**" nfr="Persistence"> </word>
r2.	… these files are a copy and are **saved** on the computer ...
Visual representation	<word content="**show**" nfr="Data representation" > </word>
r7.	… ask the system to **show** the message that …
r8.	… ask the system to **show** differences …
Security	<word content="**permission**" nfr="**Security**"> </word>
r14.	… and for assigning **permissions** to different users.
r16.	… users and their **permissions**, restricting ...
Logging	<word content="**monitor**" nfr="**Logging**"> </word>
r15.	… the possibility to **monitor** different ...
Persistence	<word content="**store**" nfr="**Persistence**"> </word>
r16.	… also be able to **store** the different ...
Security	<word content="**access**" nfr="**Security**"> </word>
r16.	… possibilities to **access** parts of the projects.
Concurrency	<word content="**concurrent**" nfr="concurrency" > </word>
r17.	… should support **concurrent** usage ….
Concurrency	<word content="**conflict**" nfr= "Concurrency"> </word>
r6.	… some form of **conflict** management, ...
r13.	… to do also some **conflict** management in ...

Table 6 shows an example of some occurrences of words from the catalogue in the requirements. As an example, the requirements $r1$, $r4$ and $r7$ contain the word "stored". This word appears in the catalogue related to the persistence concern. Then we relate these requirements to the persistence concern. The first column of Table 6 also shows other concerns of the CFVS.

After the analysis of the requirements, we have completed Table 5 with the NFCs identified: persistence, visual representation, concurrency, logging and security. These NFCs are shown in the lower part of Table 7. To automatically process these concerns in later phases, we also represent them using XML. In particular, we use the same concerns file as for the functional concerns completing it with the identified NFCs. The keyword tags for the NFCs are completed with the words presented in the catalogue for the concerns identified. In Fig. 5 we show the persistence concern represented in the lower part of the XML file.

3.2. Identifying target elements

In this activity we select the target elements that we use to build the dependency matrix later on. In particular, use case diagrams are used as a target domain so that the target elements are the use case artefacts. These use case diagrams are derived from the concerns that the system must implement. In other words, target elements are derived from the source elements as the former represent the first implementation or representation of the latter.

3.2.1. Requirements modelling

In this activity the requirements engineer must build the first representation of the system using some requirements language or notation (Fig. 3(3)). In our example, we have selected UML [27] as the modelling notation to represent the requirements and in particular, we utilise use case diagrams. The use case diagram that represents the requirements described in Table 4 can be observed in Fig. 7.

As we have achieved with concerns in previous activity in Fig. 3(3), we need to represent the elements of target in an XML format. In this case, we use XMI [28] to describe the use case diagram. The use of XMI ensures that we may perform the same analysis in other phases of the development life cycle. For instance, we may identify crosscutting concerns at design,

Table 7
Functional and NFCs of the CFVS.

Concern	Description	
c1	Insert File	Functional concerns derived from the analysis of the stakeholders' requirements (1).
c2	Retrieve File	
c3	Commit File	
c4	Update Working Files	
c5	Remove Files	
c6	Restore File	
c7	Store Message	
c8	Retrieve Message	
c9	Difference	
c10	Tag a Set of Files	
c11	Branch a Set of Files	
c12	Merge Set of Files	
c13	Persistence	NFCs automatically derived by means of keywords analysis using a catalogue (2).
c14	Visual Representation	
c15	Concurrency	
c16	Logging	
c17	Security	

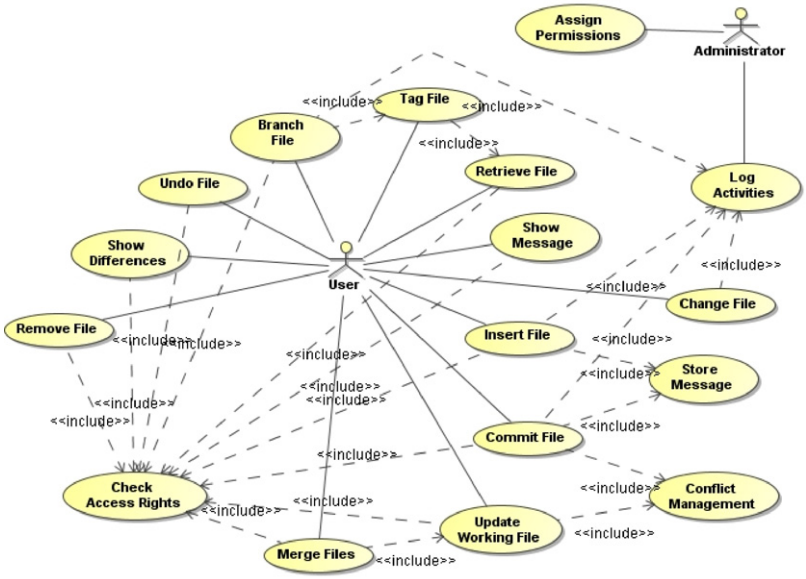


Fig. 7. Use case diagram of the CFVS system.

```
<packagedElement xmi:type="uml:Actor"
    xmi:id="_11_0_55e01df_1189682768937_411313_23"
    name="Administrator"/>
<packagedElement xmi:type="uml:UseCase"
   xmi:id="_11_0_55e01df_1189682802703_797484_41"
   name="Insert File">
   <include xmi:id="_11_0_55e01df_1189682942328_645326_149"
     addition="_11_0_55e01df_1189682927984_574791_137"/>
     ...
</packagedElement>
<packagedElement xmi:type="uml:UseCase"
   xmi:id="_11_0_55e01df_1189683228359_463673_281"
   name="Check Access Rights"/>
</packagedElement>
```

Fig. 8. Part of the XMI file for use case diagram of Fig. 7.

taking concerns as source and UML class diagrams as target respectively (see [21]). In Fig. 8 we show part of the XMI file which represents the use case diagram of Fig. 7.

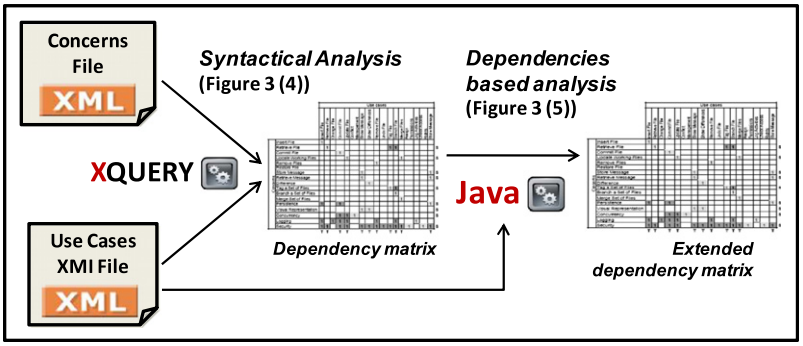


Fig. 9. Building the dependency matrix.

```
for $b in doc("useCasesXMI.xml")//packagedElement
 where some $p in doc("concerns.xml")//concern
   satisfies (contains ($b/@name,$p/keyword))
   return $b
```

Fig. 10. Example of a query in XQuery.

3.3. Building the dependency matrix

The trace relations between elements of source and target domains are represented by means of a simple traceability matrix, called the dependency matrix. This matrix represents the starting point for the analysis of crosscutting. In our example, the matrix shows the relations between concerns (source elements) and artefacts of the use case diagram (target elements). Source elements are represented in rows and target elements in columns. A cell with digit one denotes that the target element (in general, the artefact) of this column contributes to the source element of the corresponding row. In this section we explain how to automatically obtain the dependency matrix. Basically, we establish the values of the function f (defined in Section 2.1). This phase is divided into two main activities: syntactical analysis based on keywords (Fig. 3(4)); and dependencies based analysis (Fig. 3(5)). The inputs of this phase are the XML files generated in previous phases: the concerns file and the XMI file which represents the use case diagram. The output of the activity is the dependency matrix built. In Fig. 9 we show a graphical representation of the different analyses performed for building the dependency matrix.

3.3.1. Syntactical analysis based on keywords

In this activity, the trace relations between concerns (source elements) and artefacts of the use case diagram (target elements) are derived. Since both functional and NFCs are represented in the same XML file, we take into account both types of concerns in the analysis. To relate these two set of elements, an analysis based on the similarities between the identifiers of both concerns and artefacts of the use case diagram (Fig. 3(4)) is performed. We use the ⟨*keyword*⟩ and ⟨*packagedElement*⟩ tags of concerns and XMI file respectively to establish the matching between the identifiers. In particular, we use the attribute name of the ⟨*packagedElement*⟩ tag. We perform a syntactical analysis where the values of ⟨*keyword*⟩ tag and *name* attribute of ⟨*packagedElement*⟩ are totally or partially compared. That means that we can use either the whole word of the identifier to perform the analysis or the morpheme of the word to compare the identifiers. For instance, we can relate a concern with the ⟨*keyword*⟩ "Insert" to a ⟨*packagedElement*⟩ with the *name* "Insert file" but also "Insertion of files".

To compare the keywords of each concern and the identifiers of the elements defined in the XMI file, we use the XQuery language [29] (a recommendation of the W3C) to search all the matching words (see Fig. 9). In this language, we can write functions to obtain all the ⟨*packagedElement*⟩ tags in the XMI file which contain a name with a specific word. In Fig. 10 we show an example of a query which provides all the use cases whose names match with the keyword of a particular concern.

Applying the same analysis with all the use cases, we obtain the mappings shown in the dependency matrix presented in Table 8. As we can see in this matrix, we have related both functional and NFCs with the use cases that contribute to them (zeros are not shown to make the table clearer). The mappings corresponding to functional concerns are shown in light grey whilst the corresponding NFCs are shown in dark grey. In case some mappings were missed or wrongly added, the user could use the use case template descriptions to correct the results obtained by the analysis. However, as we show in Section 3.5, the process is also completed by the addition of a metrics suite that allows the developer to identify possible errors in the process and to avoid these situations.

3.3.2. Dependencies based analysis

In this activity we search for relations between elements of the target domain (intra-level relations) to complete the dependency matrix (Fig. 3(5)). The relations that we take into account are dependencies. We use these dependencies to derive a new mapping or indirect relation between elements of source and target. In particular, we use the ⟪*include*⟫ relationships of the use case diagrams to relate an element of source domain with an element of target domain. Observe that,

Table 8
Dependency matrix after the syntactical analysis.

		Use cases																
		Insert File	Retrieve File	Change File	Commit File	Update File	Conflict Management	Show Message	Show Differences	Remove File	Undo File	Tag File	Branch File	Merge Files	Assign Permissions	Log Activities	Check Access Rights	Store Message
Functional Concerns	Insert File	1																
	Retrieve File		1															
	Commit File				1													
	Update Working Files					1								1				
	Remove Files									1								
	Restore File																	
	Store Message							1										1
	Retrieve Message							1										1
	Difference								1									
	Tag a Set of Files											1						
	Branch a Set of Files												1					
	Merge Set of Files													1				
NFCs	Persistence																	1
	Visual Representation							1	1									
	Concurrency						1											
	Logging															1		
	Security														1		1	

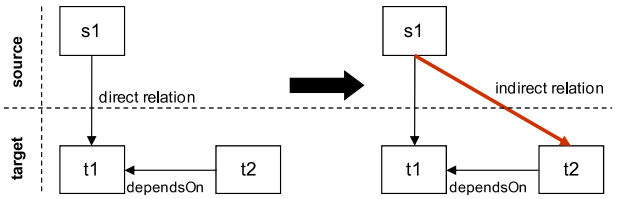


Fig. 11. Indirect relation derived between $s1$ and $t2$.

although including relationships allows the division of the included functionality into separated entities, they imply a dependency between the two use cases. This dependency is translated into coupling relations in later stages of development. Then, they are an indication of crosscutting relations between the functionality coupled. In that sense, in Section 3.6 we show how to refactor the use cases to avoid these kinds of dependencies when they involve crosscutting. In particular ⟨⟨*crosscut*⟩⟩ relations are used instead of ⟨⟨*include*⟩⟩. These relations allow the complete removal of the dependencies from the base use cases (use cases implementing base non crosscutting concerns) whilst the dependencies are added to the crosscutting use cases (those that implement crosscutting concerns). Then, the AOSD obliviousness principle [30] is fulfilled since the base artefacts are completely unaware of the crosscutting dependencies.

In Fig. 11 we show an example of the derivation of an indirect relation. We establish a special kind of transitivity relation between elements of source and target so that *if* $s1$ *is related to* $t1$ *and* $t2$ *depends on* $t1$ *then* $s1$ *is related to* $t2$. So, in the use case diagrams, we consider that *if use case* $t2$ *includes use case* $t1$*, then* $t2$ *depends on* $t1$. Note that ⟨⟨*extend*⟩⟩ relations are not used because they represent a specialisation and not a dependency (the extended use case does not really depend on the use case which extends it).

We can see in the use case diagram of Fig. 7 that there are different ⟨⟨*include*⟩⟩ relationships. In particular, there are several use cases that include the functionality of the check access rights use case. Since the check access rights use case contributes to the security concern (see Fig. 7), we relate all the use cases which include the check access rights use case to the security concern. The extended dependency matrix with the new relations (in dark grey) is shown in Table 9. We added a column and a row showing the concerns scattered (marked as S) and the use cases where concerns are tangled (marked as T) respectively.

The application of the dependencies based analysis is also automatically done by means of analysing the XMI file that represents the use case diagram of Fig. 7. As we can see in the XMI file of the example (see Fig. 12), the ⟨⟨*include*⟩⟩ relations are represented as sub-elements (⟨*include*⟩ tag) of the corresponding ⟨*packagedElement*⟩ tags. The ⟨*include*⟩ tag appears in

Table 9
Extended matrix after dependencies analysis.

		Use cases																		
		Insert File	Retrieve File	Change File	Commit File	Update File	Conflict Management	Show Message	Show Differences	Remove File	Undo File	Tag File	Branch File	Merge Files	Assign Permissions	Log Activities	Check Access Rights	Store Message	
Functional concerns	Insert File	1																	
	Retrieve File		1									1	1						S
	Commit File				1														
	Update Working Files					1								1					S
	Remove Files									1									
	Restore File																		
	Store Message							1										1	S
	Retrieve Message							1										1	S
	Difference								1										
	Tag a Set of Files											1	1						S
	Branch a Set of Files												1						
	Merge Set of Files													1					
NFCs	Persistence	1			1													1	S
	Visual Representation							1	1										S
	Concurrency				1	1	1												S
	Logging	1		1	1	1							1			1			S
	Security	1	1		1	1		1	1	1	1	1	1	1	1		1	1	S
		T	T		T	T		T	T	T		T	T	T				T	

```
<packagedElement xmi:type="uml:UseCase" xmi:id="_11_0_55e01df_1189682802703_797484_41"
    name="Insert File">
   ...
   <include xmi:id="_11_0_55e01df_1189683262343_262295_293" name=""
     visibility="public" addition="_11_0_55e01df_1189683228359_463673_281"/>
   </packagedElement>
   <packagedElement xmi:type="uml:UseCase" xmi:id="_11_0_55e01df_1189683228359_463673_281"
     name="Check Access Rights"/>
</packagedElement>
```

Fig. 12. Include relations in the XMI file.

the use case source of the relationships (where the arrow starts). It has an attribute called *addition* that indicates the use case target of the ⟨⟨*include*⟩⟩ relationship, this is the use case where the arrow (of the include relation) ends. This use case is indicated by means of an alphanumeric identifier. As we can see in Fig. 12 we just need to search the identifier in the rest of ⟨*packagedElement*⟩ tags. For instance, the insert file use case has an include relation with the addition attribute pointing out the check access rights use case. We use a simple Java programme to localise the corresponding elements in the XMI file (see Fig. 9).

3.4. Identification of crosscutting by matrix operations

In this last phase, we perform some simple matrix operations to identify crosscutting concerns. From the dependency matrix build in the previous step, we derive two different matrices: a scattering matrix and a tangling matrix (refer to Section 2.2 to see more details on how to derive these two matrices).

Once the scattering and tangling matrices have been derived, we perform the product of scattering and tangling to obtain a new matrix called the crosscutting product matrix (Fig. 3(6)) where crosscutting concerns may be identified and quantified. Finally, the crosscutting product matrix is used to derive the final crosscutting matrix (Fig. 3(7)). In the crosscutting matrix, a matrix cell denotes the occurrence of crosscutting; it abstracts from the quantity of crosscutting (since it is a binary matrix). Nevertheless, all these matrices may be used to quantify crosscutting and to establish different metrics about modularity. In particular, in [20] we have presented a set of aspect-oriented metrics which are automatically calculated using the matrices. These metrics allow the developer to avoid the presence of false crosscutting concerns and to have a more realistic measurement of the values obtained by the matrix operations. In addition, they may be used to anticipate important decisions about software quality (such as stability) at early phases of development [20]. These metrics have been

Table 10
Crosscutting product matrix for the CFVS example.

		Concerns											NFC's				
		Insert File	Retrieve File	Commit File	Update Working Files	Remove File	Restore File	Store Message	Retrieve Message	Difference	Tag a Set of Files	Branch a Set of Files	Merge Set of Files	Persistence	Visual Representation	Concurrency	Logging	Security
Functional concerns	Insert File																	
	Retrieve File		3								2	1					1	3
	Commit File																	
	Update Working Files				2								1			1	1	2
	Remove Files																	
	Restore File																	
	Store Message							2	2					1	1			2
	Retrieve Message							2	2					1	1			2
	Difference																	
	Tag a Set of Files		2								2	1					1	2
	Branch a Set of Files																	
	Merge Set of Files																	
NFCs	Persistence	1		1				1	1					3		1	2	3
	Visual Representation							1	1	1					2			2
	Concurrency			1	1									1		2	2	2
	Logging	1	1	1	1	1					1	1		2		2	5	5
	Security	1	3	1	2	1		2	2	1	2	1	1	3	2	2	5	11

integrated in the process (shown in the next step). Moreover, in Section 4, we discuss the utilisation of these metrics for the validation of the process.

Tables 10 and 11 show the crosscutting product matrix and crosscutting matrix for the CFVS example, respectively. In the crosscutting matrix, a cell with one denotes that the concern of this row is crosscutting the concern of the corresponding column. As we can see in the crosscutting matrix shown in Table 11, there are several concerns which can be considered as candidate concern. We can also observe how the candidate crosscutting concerns may be both functional and NFCs. In particular, Table 11 firstly confirms what intuition perceives: the NFCs are the elements which crosscut more concerns. In particular, logging and security concerns crosscut 10 and 15 concerns, respectively. Observe in Table 10 that the values are in general higher in the rows for these concerns than in the remaining concerns. We explain in Section 3.5 how to use these values to establish different concern-oriented metrics. We can also see in Table 11 how there are functional concerns crosscutting other concerns (both functional and non-functional). This situation suggests using aspect-oriented techniques to isolate and refactor the crosscutting concerns. In the activity shown in Section 3.6, we show how these crosscutting concerns may be refactored so that the modularity of the system is highly improved. Sometimes refactoring a certain crosscutting concern removes the crosscutting dependencies between both crosscutting and crosscut concern. However, if two given concerns A and B are crosscutting each other, what concern should be refactored, A or B? Section 3.5 shows how to take such decisions by an empirical analysis driven by the set of concern metrics defined in [20].

3.5. Empirical modularity analysis

To have empirical data to decide which concerns should be refactored, in [20] a set of concern driven metrics were introduced. These metrics are based on the crosscutting pattern presented in Section 2 and, thus, complement the aspect mining process presented with a statistical model. The utilisation of the aspect-oriented metrics may also allow the developer to detect possible errors in the process (false positives). In [20] these metrics were validated by a double process: internal and external validations. By the internal validation, the accuracy of the metrics for assessing crosscutting properties was proven. In other words this validation demonstrated that the metrics measure what is expected. By the external validation, the utility of the metrics regarding other software quality attributes was demonstrated. In particular, this validation shows how crosscutting negatively affects software stability so that the higher the degree of crosscutting for a concern, the more unstable the use cases that implement this concern are. By this validation, the utility of the metrics was empirically proven.

To make this paper self-contained, the definitions of the metrics presented in [20] are summarised here. In particular, three different types of metrics were defined: scattering, tangling and crosscutting metrics. These metrics (summarised in

Table 11
Crosscutting matrix for the CVFS example.

		Insert File	Retrieve File	Commit File	Update Working Files	Remove File	Restore File	Store Message	Retrieve Message	Difference	Tag a Set of Files	Branch a Set of Files	Merge Set of Files	Persistence	Visual Representation	Concurrency	Logging	Security
		Functional concerns												NF concerns				
Functional concerns	Insert File																	
	Retrieve File										1	1					1	1
	Commit File																	
	Update Working Files												1			1	1	1
	Remove Files																	
	Restore File																	
	Store Message								1					1	1			1
	Retrieve Message							1						1	1			1
	Difference																	
	Tag a Set of Files		1									1					1	1
	Branch a Set of Files																	
	Merge Set of Files																	
NFCs	Persistence	1		1				1	1							1	1	1
	Data Representation							1	1	1								1
	Concurrency			1	1									1			1	1
	Logging	1	1	1	1	1					1	1		1		1		1
	Security	1	1	1	1	1		1	1	1	1	1	1	1	1	1	1	

Table 12
Aspect-oriented metrics based on dependency matrix.

Metric	*Definition*	*Relation with matrices*	*Calculation*
NScattering (s_k)	Number of target elements addressing source element s_k	Addition of the values of cells in row k in dependency matrix (dm)	$=\sum_{j=1}^{\lvert T\rvert} dm_{kj}$
Degree of scattering (s_k)	Normalisation of NScattering (s_k) between 0 and 1		$=\begin{cases}\frac{\sum_{j=1}^{\lvert T\rvert} dm_{kj}}{\lvert T\rvert} & if \ \sum_{j=1}^{\lvert T\rvert} dm_{kj} > 1 \\ 0 & if \ \sum_{j=1}^{\lvert T\rvert} dm_{kj} = 1\end{cases}$
NTangling (t_k)	Number of source elements addressed by target element t_k	Addition of the values of cells in column k in dependency matrix (dm)	$=\sum_{i=1}^{\lvert S\rvert} dm_{ik}$
Degree of tangling (t_k)	Normalisation of NTangling (t_k) between 0 and 1		$=\begin{cases}\frac{\sum_{i=1}^{\lvert S\rvert} dm_{ik}}{\lvert S\rvert} & if \ \sum_{i=1}^{\lvert S\rvert} dm_{ik} > 1 \\ 0 & if \ \sum_{i=1}^{\lvert S\rvert} dm_{ik} = 1\end{cases}$
Crosscutpoints (s_k)	Number of target elements where the source element s_k crosscuts other source elements	Diagonal cell of row k in the crosscutting product matrix (ccpm)	$= ccpm_{kk}$
Concerns crosscut (s_k)	Number of source elements crosscut by the source element s_k	Addition of the values of cells in row k in the crosscutting matrix (ccm)	$=\sum_{i=1}^{\lvert S\rvert} ccm_{ki}$
Degree of crosscutting (s_k)	Addition of the two last metrics normalised between 0 and 1		$=\frac{ccpm_{kk}+\sum_{i=1}^{\lvert S\rvert} ccm_{ki}}{\lvert S\rvert+\lvert T\rvert}$

Table 12) are automatically calculated from the dependency matrix and assist the process presented in taking decisions about the analysis of crosscutting. As we can see in the third column of Table 12, all the metrics are obtained from the different matrices of the conceptual framework. Then, we just need to obtain the dependency matrix to perform the whole empirical modularity analysis since the rest of the matrices are automatically derived from the dependency matrix.

Then, using these metrics and the different matrices obtained for our running example, the CFVS, the empirical results are obtained (Fig. 3(8)). In particular, in Table 13 the results obtained for the metrics related to scattering and crosscutting are shown whilst Table 14 shows the results obtained for tangling metrics.

Table 13
Scattering and crosscutting metrics for the CFVS.

	Metrics				
Concerns	Nscattering	Degree of scattering	Crosscut points	Concerns Crosscut	Degree of crosscutting
Insert File	1	0	0	0	0
Retrieve File	3	0.176	3	4	0.205
Commit File	1	0	0	0	0
Update Working Files	2	0.117	2	4	0.176
Remove Files	1	0	0	0	0
Restore File	0	0	0	0	0
Store Message	2	0.117	2	4	0.176
Retrieve Message	2	0.117	2	4	0.176
Difference	1	0	0	0	0
Tag a Set of Files	2	0.117	2	4	0.17
Branch a Set of Files	1	0	0	0	0
Merge Set of Files	1	0	0	0	0
Persistence	3	0.176	3	7	0.294
Visual Representation	2	0.117	2	4	0.176
Concurrency	3	0.176	2	5	0.205
Logging	6	0.352	4	9	0.393
Security	14	0.823	11	15	0.812
Globals / Avg.		0.127			0.164

Table 14
Tangling metrics for the CFVS.

	Metrics	
Use cases	NTangling	Degree of tangling
Insert File	4	0.235
Retrieve File	2	0.117
Change File	1	0
Commit File	5	0.294
Update File	4	0.235
Conflict Management	1	0
Show Message	4	0.235
Show Differences	3	0.176
Remove File	2	0.117
Undo File	1	0
Tag File	3	0.176
Branch File	5	0.294
Merge Files	3	0.176
Assign Permissions	1	0
Log Activities	1	0
Check Access Rights	1	0
Store Message	4	0.235
Globals / Avg.		0.104

As may be observed in Table 13, the concerns with a higher degree of scattering and crosscutting are those defined as NFCs, i.e. security, logging, persistence and concurrency. However, there are other concerns (both functional and

non-functional) which also present a considerable degree of scattering and crosscutting. As an example, the retrieve file functional concern has a degree of scattering and a degree of crosscutting of 0.176 and 0.205 respectively, being even higher than the values obtained for the visual representation NFCs. Regarding the tangling metrics, the use cases where more concerns are tangled are shown in Table 14. For instance, it may be observed how commit file and branch file are the use cases with higher values of degree of tangling metric. All these values obtained for the metrics are used for deciding which concerns should be refactored in the next step (aspect-oriented refactoring). Obviously, those concerns with a higher degree of crosscutting should be firstly considered for being refactored. Moreover, the refactoring of the implementation of these concerns usually means that crosscutting is also removed from the implementation of other crosscutting concerns. The same could be said for considering tangling metrics. It is obvious that those use cases where more concerns are tangled should be considered for performing a refactoring (externalising the implementation of the crosscutting concerns). Even, the metrics allow the consideration of the definition of threshold values as a limit to consider crosscutting concerns (those with values for the metrics higher than the threshold). In this paper we have not considered this option since the main purpose of the empirical analysis is to obtain a classification of the concerns according to the degree of scattering or crosscutting (Fig. 3(9)). However, we do not discard establishing these threshold values in future analyses.

3.6. Aspect-oriented refactoring

As we mentioned previously, once the crosscutting concerns are identified, we may refactor them so that they are encapsulated into separated entities. Actually the refactoring is performed by isolating the use cases that implement crosscutting concerns. In other words, the refactoring is performed for target elements that address crosscutting source elements. To decide which concerns should be refactored, a qualitative assessment process is performed by an expert. This process is fed by the empirical analysis presented in the previous section so that the expert uses the values obtained by the metrics to take such decisions. As an example observe that in the CFVS, the concerns with a higher degree of crosscutting are the NFCs, especially security, logging, persistence and concurrency. These crosscutting concerns are the candidates to be refactored first. Sometimes, the refactoring of a crosscutting concern also removes the presence of crosscutting of a different concern. Then, usually the crosscutting concerns with a lower degree of crosscutting do not need to be refactored. Observe how the crosscutting concerns refactored are those with a higher degree of crosscutting (the aforementioned NFCs). There are some crosscutting concerns with a low degree of crosscutting which are not refactored, e.g. update working files. This crosscutting concern was being crosscut by the NFCs. Then, once these crosscutting concerns are refactored, it also remains isolated.

This section shows how to refactor the crosscutting concerns using aspect-oriented techniques at the requirements level. In particular, we adapt the technique used in [31], where the authors present a method to modularise volatile concerns at the requirements level. They utilise a use case pattern specification [32] and some templates to "mark" the use cases which address volatile concerns. Pattern Specifications (PS) are a way of formalising the reuse of models. The notation for PS is based on the Unified Modelling Language (UML) [27]. A PS describes a pattern of structure or behaviour defined over the roles which participants of the pattern play. Role names are preceded by a vertical bar ("|"). A PS can be instantiated by assigning concrete modelling elements to play these roles. As in [31], we extend the notion of PS from that of [32] by allowing both role elements and concrete modelling elements in a PS. Then, this technique is used to refactor the use cases that implement crosscutting concerns. This is illustrated by showing an example based on the CFVS.

In particular, the concerns which are classified as crosscutting are also classified as roles. Then, we mark the use cases implementing these concerns (using the special symbol "|") and they are modelled using a PS model (Fig. 3(10)). A new ⟪*crosscut*⟫ relation is added to the diagram which is used to relate use cases implementing crosscutting concerns to those which are considered as the base system. The information needed to decide where to add these new relations (⟪*crosscut*⟫) is derived from the results obtained by the metrics (introduced in Section 3.5) and the dependency matrix. On one hand, the results obtained by the metrics indicate the concerns that should be refactored (those with a higher degree of crosscutting). On the other hand, the dependency matrix is used to trace the use cases which address these crosscutting concerns. Fig. 14 depicts a part of the use case diagram of the system with the marked use cases and ⟪crosscut⟫ relations. Fig. 13 shows the same part of the use case diagram in the original system (without refactoring). As we can see in these figures, by refactoring, the direction of the relations is changed (⟪*crosscut*⟫ instead of ⟪*include*⟫) so that the base use cases (those implementing base concerns) are independent of the crosscutting concerns, fulfilling the AOSD obliviousness principle.

Then, once we have isolated the implementation of the crosscutting concerns, we may evolve the system and change the crosscutting concerns using composition rules. By these composition rules we may compose different activity diagrams (in particular, we use activity PS [32]). As an example, in Fig. 15 we show three activity diagrams (Fig. 3(11)) which represent the main flows of three use cases: update working files, manage conflicts and check access rights. These activity diagrams are composed (Fig. 3(12)) using the composition rules shown in Figs. 16 and 17. Then, we can easily change the concurrency feature or the security policy just by composing the base activity diagram (for update working file) with other activity diagrams (using different composition rules).

Of course, there are other techniques (different from aspect-oriented ones) that we may use to get a high level of flexibility or configurability. Sometimes, we may use design patterns [33] to improve flexibility and reusability of particular designs. However, as has been demonstrated in several publications [34,35], the utilisation of aspect-oriented techniques considerably improves the benefits obtained by the utilisation of some design patterns.

Fig. 13. Use case diagram before refactoring.

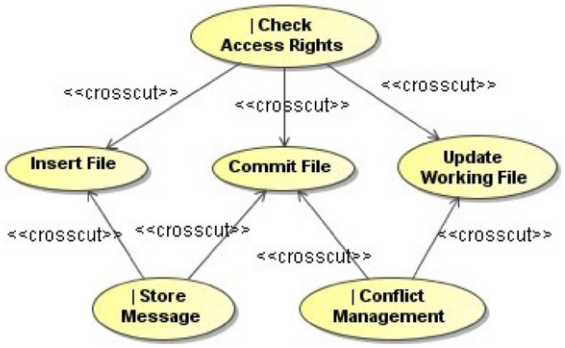
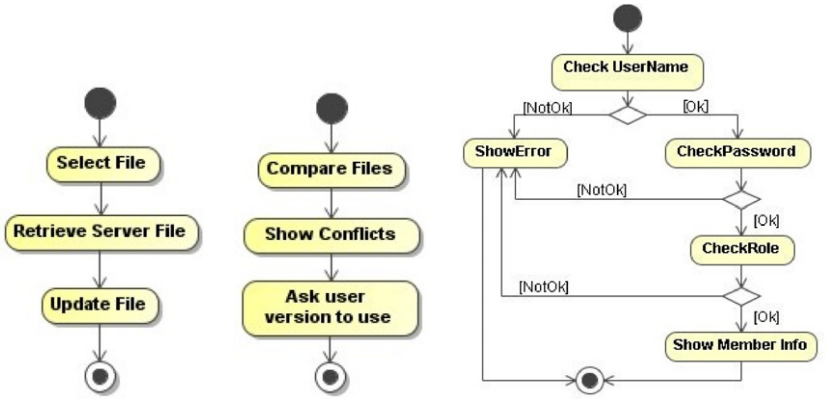


Fig. 14. Use case diagram marked.



(a) Update working file. (b) Manage conflicts. (c) Check access rights.

Fig. 15. Activity diagrams for several use cases of the CFVS system.

Compose Update working file **with** Check access rights

Insert Check user name **before** Select file

Fig. 16. Composition rule to add check access rights functionality to update working file.

Compose Update working file **with** Manage conflict

Insert Compare files **after** Retrieve server files

Fig. 17. Composition rule to add manage conflicts functionality to update working file.

4. Discussion and validation

In this section we provide a double validation of the process. Firstly, we show the application of the process to a real system which has been used to assess modularity and stability in software product lines [36]. We use the set of concern-oriented

Table 15
Concern metrics introduced in [14,38].

Metric	*Definition*
Concern Diffusion over Architectural Components (CDAC)	Counts the number of components addressing a concern.
Lack of Concern-based Cohesion (LOCC)	Counts the number of concerns addressed by the assessed component.
Degree of scattering (DOS)	This is defined as the variance of the concentration of a concern over all programme elements with respect to the worst case. Concentration measures how much a concern is concentrated in a component
Degree of tangling (DOT)	This is defined as the variance of the dedication of a component for all the concern with respect to the worst case. Dedication quantifies how much a component is dedicated to a concern.

Table 16
Different releases of MobileMedia system [36].

Release	Description
r0	MobilePhoto core
r1	Error handling added
r2	Sort photos by frequency and edit label feature added
r3	Set favourites photos added
r4	Added a feature for copy photo to another album
r5	Added a feature for sending photos by SMS
r6	Added the feature for playing music
r7	Added the features for playing videos and capture media

Table 17
Features and releases where they are included.

Feature	Releases		Feature	Releases
Add Album	r0 - r7,		Sorting	r2 - r7
Delete Album	r0 - r7,		Set Favourites	r3 - r7
View Album	r0 - r7,		View Favourites	r3 - r7
Create Photo	r0 - r7,		Copy Photo	r4 - r7
Delete Photo	r0 - r7,		Send Photo	r5 - r7
View Photo	r0 - r7,		Receive Photo	r5 - r7
Label	r0 - r7,		Music Control	r6, r7
Persistence	r0 - r7,		Play Video	r7
Error Handling	r1 - r7		Capture Media	r7

metrics presented in [20] (and used in our aspect mining process) to compare the results obtained with those provided by other authors' similar metrics. Then, we show that our metrics (based on the crosscutting pattern) obtain values consistent with the other metrics, validating our aspect mining process.

Secondly, we compare the framework with similar early aspect approaches, such as EA-Miner or Theme/DOC. Based on this validation we discuss some interesting open issues that may be considered regarding the aspect mining process presented. The complete analysis with all the results presented in this section is also available for the reader in [37].

4.1. Validation in a real application

In this section we show the results obtained by applying the aspect mining process to the requirements of an application called MobileMedia [36]. We have also calculated all the metrics summarised in Section 3.5 so that we have more detailed data for identifying the crosscutting concerns in the system. Finally, the values obtained by our metrics are compared with those obtained by similar metrics introduced by other authors. In particular, we used the metrics introduced in [14,38]: *Concern Diffusion over Architectural Components (CDAC)* and *Lack of Concern-based Cohesion (LOCC)* metrics introduced in [14] and *Degree of Scattering (DOS)* and *Degree of Tangling (DOT)* metrics introduced in [38]. All these metrics are also used to perform modularity analyses and they are defined in Table 15.

4.1.1. The MobileMedia system

The MobileMedia [39] is a product line system built to allow the user of a mobile device to perform different options, such as visualising photos, playing music or videos, and sending photos by SMS (among other concerns). In [36] the authors used a modification of the system to perform different analyses about modularity and stability in software product lines mainly at an architectural and programming level. Our work complements those previous analyses since we focus on modularity at the requirements level.

Table 18
Dependency matrix for the MobileMedia system in release 7.

		Usecases																						
		Add Album	Delete Album	Add Media	Delete Media	View Photo	View Album	Provide Label	Store Data	Remove Data	Retrieve Data	Edit Label	Count Media	View Sorted Media	Set Favourite	View Favourites	Copy Media	Send Media	Receive Media	Music Control	Access Media	Play Video	Capture Media
Concerns	Album	1	1				1																
	Photo					1																	
	Label	1		1			1	1				1					1		1				
	Persistence	1	1	1	1	1	1	1	1	1	1		1		1		1	1	1		1		
	Error Handling	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1		1		1
	Sorting					1	1						1	1							1		
	Favourites						1								1	1							
	Copy					1											1				1		
	SMS					1												1	1		1		
	Music																			1			
	Media		1	1	1		1												1		1		1
	Video																					1	
	Capture																						1

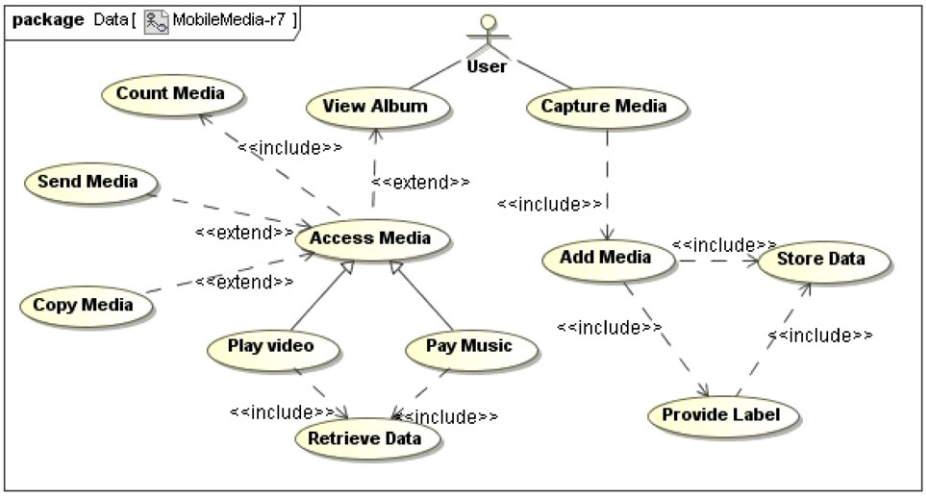


Fig. 18. Part of use case diagram for release 7.

The system has about 3 KLOC and it has been built as a product line in eight different releases. Each of these releases adds some different features to the system. For instance, release 0 implements the original system with just the functionality of viewing photos and organising them by albums. Releases 1 and 2 add error handling and the implementation of some optional features (sort photos by frequency or edit labels) respectively. In Table 16 we show the different releases with the features added in each release (see [36] for more details). We also show in Table 17 the different features taken into account in the product line and the releases where they are involved. To better understand the system, we show in Figs. 18 and 19 some details about the requirements models used for release 7 (which includes all the features). Fig. 18 shows part of the use case diagram used for release 7 whilst Fig. 19 presents a reduced version of the template description for the use case Play Video (involved in release 7).

Then, the aspect mining process presented in this paper was applied to the requirements of each release. To apply this process, we considered the different features of each release as the elements of source and the use cases implementing the system as the target domains respectively. Based on these two domains and the process explained in Section 3, a dependency matrix for each release is built showing the use cases contributing to the different features. The features used for the analysis are shown in Table 17 (the main features of the MobileMedia product line). In the same table, we show the releases in which these features are included. We also considered some NFCs which were identified: "persistence" and "error handling". The dependency matrix built for release 7 is shown in Table 18. In this matrix, the concerns have been shown in the same order that they are added to the system throughout the different releases (this is why they are not classified into functional or non-functional). Based on this dependency matrix, we derive the rest of matrices described in Section 2.2 (scattering,

```
Usecase: Play Video
Actor: Mobile Phone (system) and User
Description: The user can play the video available in device memory
Preconditions:
   1. Application must be launched
   2. The user should have previously selected the view album option.
   3. A video file must be available
Postconditions: The selected video is played in the device.
Trigger: User selects to play a video
Basic flow:
   1. The user selects an album
   2. The device populates the available video files
   3. The user selects a video file to be played
Alternative flow:
   1. There are no video available on the mobile phone
Inherits from: Access Media usecase
Includes: Retrieve Data usecase
```

Fig. 19. Play video use case description.

Table 19
Crosscutting matrix obtained for the MobileMedia system in release seven.

Concerns	Album	Photo	Label	Persistence	Error Handling	Sorting	Favourites	Copy	SMS	Music	Media	Video	Capture
Album			1	1	1	1	1				1		
Photo													
Label	1			1	1	1	1	1	1		1		
Persistence	1	1	1			1	1	1	1		1		1
Error Handling	1	1	1	1	1		1	1	1		1		
Sorting	1	1	1	1	1	1					1		
Favourites	1		1	1	1	1			1		1		
Copy		1	1	1	1	1		1			1		
SMS		1	1	1									
Music					1	1	1	1	1				1
Media	1		1	1									
Video													
Capture						1	1	1	1		1		1

tangling, crosscutting product and crosscutting matrices). Since release 7 represents the system with all the features included, we have focused on this release in this section. However, the reader may access the whole analysis in [37], where we have applied the process to the eight releases of the system.

Based on the dependency matrix built, the crosscutting matrix shown in Table 19 is obtained. In this matrix we can see how there are several crosscutting concerns (both features of the system and NFCs). Moreover, the values obtained in the matrices are used in the next section for assessing the degree of crosscutting of the different features of the product line. In particular, we automatically calculate the metrics introduced in Section 3.5.

4.1.2. Analysing the modularity of the system

In this section we show the results obtained after calculating the metrics introduced in Section 3.5 to all the releases of the MobileMedia product line. The main goal of this analysis is to have a global vision of the results obtained for all the releases and to discuss them. Moreover, based on the dependency matrix built for each release, we have also calculated the metrics introduced in [14] and [38]. Then, we may compare the results obtained by our framework with the results obtained by other authors' metrics.

Firstly we focus on the results obtained by our metrics. As we mentioned in Section 3.4, when there are several concerns crosscutting each other, we may decide which concern to refactor using aspect-oriented techniques. The metrics used by the framework supports such decisions. Note that they provide important information about modularity of source or target elements (e.g. degree of crosscutting of concerns or degree of tangling of use cases). We show in Figs. 20 and 21 the results obtained for our degree of scattering and degree of crosscutting metrics respectively. To decide the candidate crosscutting concerns (since MobileMedia is a product line, we consider features as being the concerns of the system), we take into

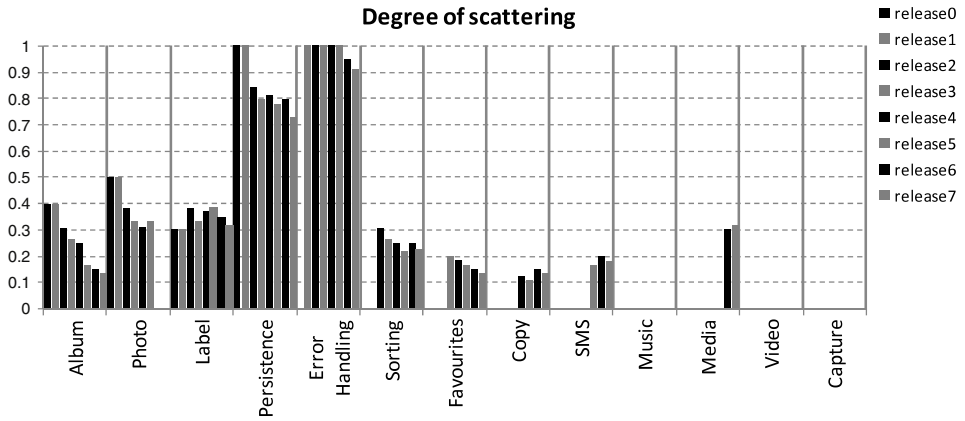


Fig. 20. Degree of scattering in the MobileMedia releases.

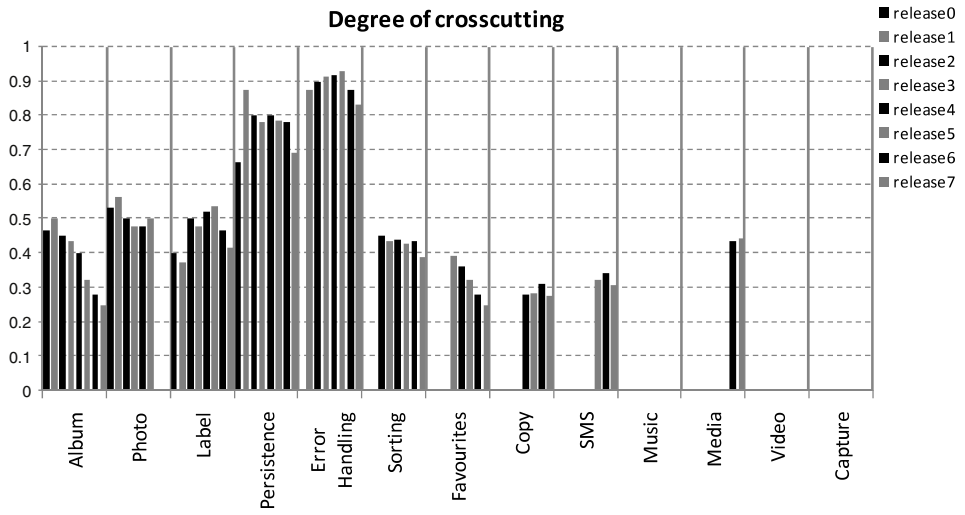


Fig. 21. Degree of crosscutting in the MobileMedia releases.

account the combination of both scattering and crosscutting metrics. Tangling metrics may be also considered for analysing problems related to the cohesion of each use case.

As we can see in the figures, the concerns or features with the highest degree of crosscutting are those related to non-functional requirements (persistence and error handling). They also have the highest degree of scattering. We can also see that the degree of scattering and crosscutting for these concerns remains considerably high in all the releases. We can see a small decrease of the degree of scattering and crosscutting for these concerns in the final releases. This is due to the fact that the changes introduced by these releases either do not involve any operation related to persistence or the operations needed for persisting data are reused for other use cases previously introduced in the system. For instance, when we add the feature receive photo, it reuses the behaviour of another use case called add photo (which includes the behaviour for storing the photo data). For error handling the explanation is the same.

Note that there are other features that should also be considered as a candidate to be aspectised (see Figs. 20 and 21). In particular, photo, label and sorting features have a considerably high degree of crosscutting. Moreover, the values obtained for these features remain constant along the releases. This is due to the fact that sorting and labelling behaviour is shared by all the different kinds of media used in the application (photo, music and video). We can see also how the scattering and crosscutting of photo feature is removed in release 6 (the values of the corresponding metrics are zero). This is due to the addition of the media feature in release 6. This feature is responsible for all the common functionality of the three kinds of media files: photo, music and video. Then, when we add this feature in release 6, a big part of the behaviour previously implemented by the use cases related to photo feature is now implemented by the use cases related to media feature (then we remove crosscutting for photo feature). Note also that media feature presents scattering and crosscutting in these last releases.

Finally, we can see how there are features, such as album with a degree of scattering and crosscutting around 0.4 in the first releases. However, the values obtained for album considerably decrease along all the releases. This feature could be considered as a candidate crosscutting concern in first releases. However, the whole results for all the releases provide a

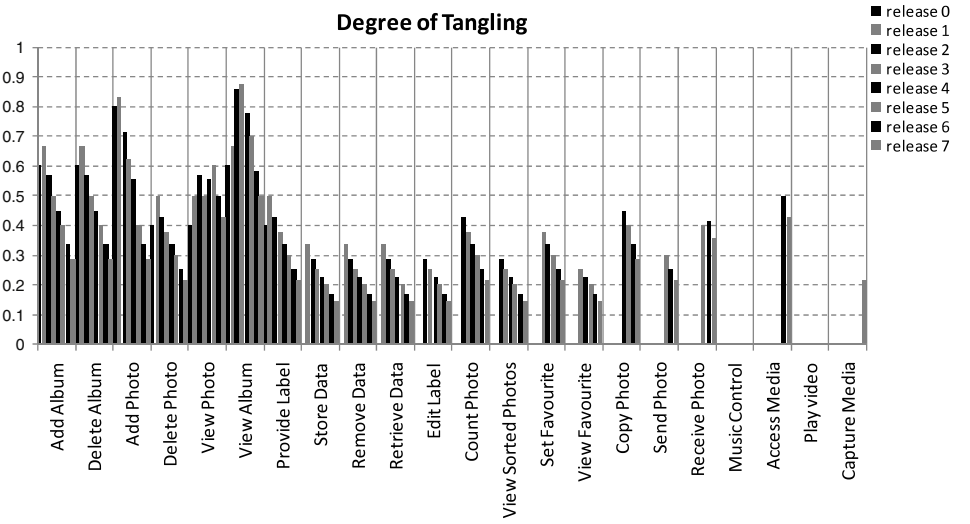


Fig. 22. Degree of tangling for use cases in MobileMedia.

global view of this feature and we can see how, unlike the rest of the crosscutting concerns, the degree of crosscutting for album decreases when we add new features to the product line. Then, we concluded that this feature could be considered as a false positive in the first releases. Note that in the first two releases, the number of use cases of the system is lower than in the last releases. Observe also that degree of scattering and degree of crosscutting metrics are calculated using the total number of source and target elements. Then, in small systems, the number of false positives could be higher than in bigger systems. This is the reason why when we add more features to the system, crosscutting for album quickly decreases. Therefore, we can also conclude that in real systems (with many features or concerns) the values obtained by the metrics are really helpful for assessing the modularity of the system. Moreover, the use cases where album is realised are tangled with other features such as persistence, label or error handling, which have been considered as crosscutting features. Then, refactoring these crosscutting features would also remove tangling from the use cases where album is addressed, thus reducing the degree of crosscutting for album feature. This illustrates how the concern-oriented metrics may be used to decide which concerns to refactor. Note that the metrics may also be used to establish threshold values that allow the identification of these false positives. In particular, the concerns with a degree of crosscutting lower than these threshold values could be discarded.

To better understand why the album feature is considered as a false positive, we also show in Fig. 22 a graphic showing our degree of tangling metric for the use cases of the MobileMedia in the eight releases. We can see in this graphic how the tangling for the use cases is in general higher in first releases than in later ones. This is also due to the lower numbers of use cases in the first releases and the fact that the degree of tangling metric is also calculated considering the total number of target elements. This fact also influences the results obtained for the degree of crosscutting (see Fig. 21) since it is calculated based on a special combination of both scattering and tangling. Note also that the use cases where the album feature is addressed, AddAlbum, DeleteAlbum and ViewAlbum (see Table 18), present a lower degree of tangling in the last releases than in the first ones. These results are consistent with our aforementioned conclusion where we considered album as a false positive.

From these results, we can also extract important information about the adaptability and reutilisation of the MobileMedia product line. As has been shown in some publications [40,41], the utilisation of aspect-oriented techniques may improve the reutilisation of product lines by removing dependencies between features. In this example, label and sorting are optional features and, as we mentioned before, they are crosscutting other features of the system, both mandatory (such as album or media) and optional (such as favourites). Then, reutilisation and adaptability of the product line are endangered by the dependencies created by these crosscutting features (Sort and Label). The problem is even more important for the mandatory features which are crosscut by these two optional features. The utilisation of the process presented in this paper may help to identify these situations and to solve them using the aspect refactoring shown in Section 3.6.

Finally, we have compared the results obtained by our metrics with those obtained by similar metrics defined by other authors. In Tables 20 and 21 we show the average of the values obtained by all the metrics (ours and those presented in [14,38]). The former shows the metrics assessed for the source elements whilst the latter shows those that are assessed for target ones. Due to space reasons we do not show here the metrics for all the releases, however the whole analysis is available in [37]. Note that the metrics introduced in [14,38] are tied to specific deployment artefacts (architectural components and source code lines respectively), thus we needed to adapt the metrics to assess modularity at the requirements level. On one hand, we just took into account use cases instead of architectural components for *concern diffusion over architectural components* and *lack of concern-based cohesion* [14]. On the other hand, since *degree of scattering* and *degree of tangling* metrics [38] are defined using a finer granularity level (lines of code), we calculated these metrics by counting control flows or steps of the use cases. To count the control flows, we used the description or template for each use case (see an example in

Table 20
Metrics assessed for source elements.

	Average of all releases							
Authors	Conejero et al. [20]					Sant'Anna et al [14]		Eaddy et al. [38]
Metrics / Concerns	Nscattering	Degree of scattering	Crosscut points	Concerns vrosscut	Degree of crosscutting	Concern diffusion over usecases	Concern diffusion over flows	Degree of scattering
Album	3.63	0.26	3.63	5.25	0.39	3.63	8	0.77
Photo	4.13	0.3	3.88	4.13	0.38	4.13	7.38	0.62
Label	5.38	0.34	5.38	6	0.46	5.38	7.88	0.82
Persistence	12.8	0.85	12.4	6.38	0.77	12.8	25.1	0.98
Error Handling	15.9	0.98	15.9	7	0.89	15.9	27.6	0.99
Sorting	4.33	0.25	4.33	7.33	0.43	4.33	5.33	0.78
Favourites	3	0.17	3	6	0.32	3	4	0.66
Copy	2.5	0.13	2.5	6.25	0.29	2.5	2.5	0.61
SMS	3.67	0.18	3.67	6.67	0.32	3.67	3.67	0.76
Music	1	0	0	0	0	1	3	0
Media	6.5	0.31	6.5	8.5	0.44	6.5	12.5	0.85
Video	1	0	0	0	0	1	1	0
Capture	1	0	0	0	0	1	1	0
Globals/Avg		0.27			0.34			

Table 21
Metrics assessed for target elements.

	Average of all releases			
Authors	Conejero et al. [20]		Sant'Anna et al.[14]	Eaddy et al. [38]
Metrics / Use cases	Ntangling	Degree of tangling	LOCC	Degree of scattering
Add Album	3.875	0.475	3.875	0.846
Delete Album	3.875	0.475	3.875	0.826
Add Photo	4.5	0.568	4.5	0.886
Delete Photo	2.875	0.350	2.875	0.702
View Photo	4.5	0.506	4.5	0.833
View Album	6	0.694	6	0.900
Provide Label	2.875	0.350	2.875	0.702
Store Data	1.875	0.200	1.875	0.494
Remove Data	1.875	0.200	1.875	0.494
Retrieve Data	1.875	0.200	1.875	0.494
Edit Label	2	0.211	2	0.559
Count Photo	3	0.316	3	0.699
View Sorted Photos	2	0.211	2	0.497
Set Favourite	3	0.294	3	0.693
View Favourite	2	0.196	2	0.554
Copy Photo	4	0.365	4	0.791
Send Photo	3	0.254	3	0.683
Receive Photo	4.666	0.391	4.666	0.789
Music Control	1	0	1	0
Access Media	6	0.464	6	0.867
Play video	1	0	1	0
Capture media	3	0.214	3	0.717

Fig. 19). Then the granularity level used by the metrics at the requirements level is consistent with the original definition of the metrics (we just changed the kind of artefacts used, use cases instead of implementation classes). Based on the idea of counting use cases steps or control flows, we also adapted the *concern diffusion over architectural components* metric to take into account control flows instead of use cases. Then, we considered the *concern diffusion* metric over two types of artefacts: use cases and control flows (coarse and fine granularity level, respectively). In cases where we used flows as the

Table 22
Comparative table with the different case studies and tools.

	Conference Review System (CRS)		Portuguese Highways Toll System (PHTS)		Siemens Toll System (STS)		Course Management System (CMS)		PetStore		
	Original	Crosscutting pattern	Original	Crosscutting pattern	EA-Miner	Crosscutting pattern	Theme/Doc	Crosscutting pattern	Theme/Doc	LSA	Crosscutting pattern
Concerns identified	8	10	7	4	21	16	5	6	4	10	11
Functional Concerns	4	6	–	–	13	6	4	4	–	–	7
Non-Functional Concerns	4	4	7	4	8	10	1	2	–	–	4
Candidate Early Aspects	4	8	7	4	14	13	1	4	15[#]	19[#]	7
False Positives	0	3	0	0	2	1	0	2	12	16	1
False Positive rate[*]	0	3/5 = 0.6	0	0	2/4 = 0.5	1/4 = 0.25	0	2/4 = 0.5	12/5 = 2.4	16/5 = 3.2	1/5 = 0.2
False Negatives	1	0	0	3	5	5	1	0	3	3	0
False Negative rate[*]	1/5 = 0.2	0	0	3/7 = 0.42	5/17 = 0.29	5/17 = 0.29	1/2 = 0.5	0	3/6 = 0.5	3/6 = 0.5	0
Actual Crosscutting Concerns	5		7		17		2		6		

[*] For more information about how to calculate false positive and negative rates see [45].
[#] The approaches based on Theme/Doc identify aspectual requirements. This is the reason why the number of early aspects is higher than the number of concerns.

target element, we changed our dependency matrix to show in each cell the number of flows of a use case addressing the corresponding feature or concern (instead of being a binary matrix).

As we can see in these tables, the values obtained by our metrics are consistent with those obtained by other authors. There are some equivalent metrics which obtain the same values, e.g. *concern diffusion over use cases* and *Nscattering* or *Ntangling* and *lack of concern-based cohesion*. This is due to the fact that they are assessing the same concept. In general, we can see how the higher the values for our *degree of scattering* and *degree of crosscutting* metrics are, the higher *concern diffusion* and *degree of scattering* (defined in [38]) metrics are as well. The same occurs with the metrics assessed for target elements (tangling). Then, we can ensure that the metrics are behaving as expected. The differences between the values obtained by the different metrics are mainly caused by the different granularity level used by them (e.g. our *degree of scattering* metric utilises use cases whilst *degree of scattering* in [38] is adapted to use control flows). As an example, the values for *degree of scattering* in [38] are in general higher than the values obtained by our *degree of scattering* metric. Our conclusion was that these metrics may complement each other in different situations where different granularity levels are needed. For instance, the metrics defined in [38] are suitable to apply them at programming level (lines of code). However, in earlier phases of the development (like requirements), we encourage utilising metrics with a coarser granularity level. Note that, unlike the metrics defined in [14,38], our metrics are not tied to any specific deployment artefact and they are generic enough to be used at any abstraction level.

4.2. Comparison with other approaches

In this section we show the application of our process to other case studies used by similar approaches (which also deal with crosscutting concerns at early stages). The goal of the section is also twofold: on one hand, to validate again the process presented in the paper, showing the feasibility of the approach; on the other hand, the identification of some open issues that should be considered about mining aspects at early stages.

For the validation of the process, we have applied our process to five different examples: a Conference Review System (CRS) (used in our previous work [4] without syntactic and dependencies based analyses), the Portuguese Highways Toll System (PHTS) (used in [42]), a Siemens Toll Gate System (STS) (used by the EA-Miner tool in [43]), a Course Management System (CMS) (used by Theme/Doc [18]) and the well-known PetStore example (used by Theme/Doc and its extension with Latent Semantic Analysis (LSA) both in [44]). Then we compare the results obtained by the original authors with the results obtained by our aspect mining process. Note that we have also compared the process with the results obtained in our previous work [4], where the framework did not use the syntactical and dependency based analyses.

A summary of the results obtained has been shown in Table 22. As we can see in the table, we have represented the number of concerns (both functional and non-functional) and crosscutting concerns identified by the different approaches in the case studies. We have also represented the false positive and negative rates for each example. False positives refer to crosscutting concerns wrongly identified by the tool whilst the false negatives are related to crosscutting concerns that the different tools do not identify. Note that false positives and negatives are decided by manually analysing the results obtained by the different approach. We carefully studied the systems and the results obtained to check whether each crosscutting concern identified was properly identified. Then, if a crosscutting concern identified by one or more approaches is missed by a different approach, we consider this concern as a false negative for the latter (since it has not identified it). Thus, the gold standard used in the analysis is established by the manual reviewing of the results by an expert. In that sense, the actual crosscutting concerns were also calculated by an analysis of the results obtained by the original approach and the application of our aspect mining process. In particular, the actual crosscutting concerns are calculated by the union

of the crosscutting concerns and the false negatives identified by both approaches (the original and ours) minus the false positives (obtained also by both approaches). As an example, in the STS the union of crosscutting concerns and false negatives identified is formed by 19 concerns but the union of false positives is 2. Thus, the actual number of crosscutting concerns is 17. We established that the best technique is the one with the lowest sum of false positives and negatives. In this sense, our framework presents worse values than other manual approaches (e.g. for the PHTS). However, the approach presents better rates than the similar approaches (which identify crosscutting concerns by semi-automatic tools). Of course, in the manual approaches, the engineer may have better knowledge of the domain. However, in real systems, the use of manual approaches may be unfeasible and the utilisation of automatic tools helps the developer in important tasks reducing effort and development time.

4.2.1. Analysing false positives

The crosscutting matrix could lead to the consideration of some false crosscutting concerns (false positives). Sometimes, these false positives may be caused by the decomposition selected for the elements in source or target. In these cases, it is possible to avoid crosscutting by choosing another decomposition of source and target, a possibility determined by the expressive power of the languages in which the source and target are represented.

As we can see in Table 22, we may find false positives in the application of almost all the different approaches analysed. As an example of the explained above, in the CRS presented in [4], our framework identified three functional crosscutting concerns (submission, review and conference management). However, some of these concerns are considered as crosscutting because of the granularity of target elements selected. For instance, the submission process takes place in several use cases: submit paper and change submission (both in the same package). Obviously, these use cases are related to the same functionality. The same situation occurs with review and conference management concerns. We consider these concerns false positives. However, as we have shown in Section 4.1.2, the utilisation of our aspect-oriented metrics suite allows the developer to identify such situations. By the addition of these metrics to the process presented here, the results obtained are more realistic and the developer has more information to take decisions. Moreover, the application of the process to the different releases of the MobileMedia system showed how the problem of false positives may emerge in small systems with just a few features. However, when the system becomes more complex with the addition of new features, the metrics showed a lower degree of scattering and crosscutting for those features (identified as false positives). Then, the situation is detected and avoided. As mentioned previously, the metrics may also be used to set threshold values so that any concern which presents crosscutting but with a degree of crosscutting lower than this value could be considered as a false positive. In this paper we integrated the metrics into the aspect mining process to classify the source elements according to their degree of scattering and crosscutting. This is why threshold values have not been needed. However, we do not discard using them in future analyses.

4.2.2. Analysing false negatives

In the false negatives column of Table 22 we can see how our framework did not identify some crosscutting concerns in some case studies. In most cases, the reason to get these false negatives is that we do not identify as crosscutting concerns those that are not somehow represented in the use case models. For instance, there are some candidate crosscutting concerns like scalability, reliability or compatibility (identified in STS) that are usually mapped to design or hardware decisions later on and they are not represented in diagrams or code. Although we use the NFCs catalogue to identify such concerns, the crosscutting matrix does not identify them as crosscutting concerns since they are not mentioned in the use cases based representation. This is the reason why the framework also did not identify some crosscutting concerns in the PHTS (e.g. availability or correctness). As a different example, in [36] the authors used the Model-View-Controller (MVC) architectural pattern for implementing the MobileMedia system shown in Section 4.1.1. They considered this issue as an important concern for the system (at architectural level). However, since this is a developer's design decision related to adaptability and maintainability of the system, this concern is not explicitly present in the requirements and it was not identified by our process.

In the STS case study, we did not identify a crosscutting concern that EA-Miner did. This concern is what they called charge calculation. However, the authors in [43] claim that they focused on the requirements related to communication. We took the same requirements for our analysis and we did not find any presence of the charge calculation concern in those requirements. We think this concern could be considered as a false positive by their approach.

4.2.3. Accuracy of requirements

Since we are applying our framework at early phases, the first source of information for the process is the set of requirements (usually in text). Sometimes, there are some NFCs which may not be inferred by means of an automatic analysis of the requirements. For instance, in the PHTS presented in [42] the authors identified some NFCs by manually analysing the requirements documents and extracting conclusions about them. As an example, in the text, "*If an unauthorised vehicle passes through it, a yellow light is turned on and a camera takes a photo of the plate*" the photo must be quickly taken, otherwise the plate will not appear in it. Then, response time is present in such a requirement. In the application of our framework to the PHTS we obtained less crosscutting concerns that the authors did in [42]. We did not identify the non-functional crosscutting concerns that were implicitly (but not explicitly in the text) present in the requirements. The problem of

accuracy of requirements is also present in the rest of tools analysed. As an example, in the STS, the EA-Miner identified 21 concerns. However, as the authors say in [43], these concerns are the result of editing and sorting the original set of concerns identified by the tool. Then, the engineer must spend quite some time selecting the concerns of the system after applying the tool to the original requirements.

The problem of the accuracy of requirements is also identified in [46], where the authors claimed that writing quality requirements is also a best practice for automating traceability. In that sense, they suggest that requirements should correct, unambiguous, complete, consistent, prioritised, verifiable, understandable, identifiable and so on. They concluded that requirements that are well written and organised provide better traceability results than those that are randomly created.

A lesson learned from the results obtained in Table 22 is that, in some cases, the results obtained by the manual approaches may be better than those obtained by automatic approaches. However, in real systems with higher sets of requirements, the utilisation of manual approaches is unfeasible and the automatic tools may save much time for the developer. Anyway, like in the other approaches, the requirements engineer may assist our process by changing the dependency matrix as he considers necessary. The results obtained may be improved based on his experience in the current system or systems previously developed. Moreover, the developer could assist the process by the utilisation of techniques for documenting the process, e.g. shadowing or marking the use cases artefacts, as suggested in [36] (the authors use shadowing techniques at source code level).

4.2.4. The NFCs catalogue

The utilisation of a NFCs catalogue or repository improves the identification of such concerns in the systems. We noticed such improvements in the different case studies we have analysed. For instance, in the CRS case study we identified a new non-functional crosscutting concern, data presentation. In the second case study, the PHTS, we also identified some non-functional crosscutting concerns (such as persistence and data presentation) that the authors did not identify in [42].

In the STS example, the two approaches compared use a similar catalogue so that in that sense the results obtained are similar. For the PetStore example, the Theme/Doc also uses a set of keywords. However, this catalogue just contains words related to the application domain, ignoring NFCs common to different domains. As we can see in Table 22, the results obtained by applying the LSA approach to the PetStore are similar to those obtained by our framework.

This fact was also observed by Cleland-Huang et al. in [22]. In this work, the authors presented an automated method to classify non-functional requirements in software requirement specifications. They claimed that a first detection of the relevant keywords related to each non-functional requirement allows an important improvement of the results obtained in next projects (using these keywords identified). Moreover, the authors indicated the lack of standardised non-functional requirements catalogues so that the introduction of the catalogue used in this work aims at bringing this gap.

4.2.5. Granularity of source and target elements

As mentioned previously, concern scoping is one of the major issues in the aspect-oriented area. Sometimes it is not trivial to decide what a concern is in a particular system and the concern decomposition leads to developers' expertise. In that sense, granularity selected to decompose source and target elements may affect the results obtained by any aspect mining process. One has to decide the granularity level selected to analyse the mappings between source and target. Even, alternative decompositions are possible to avoid the problem of crosscutting concerns, sometimes using design decisions, such as the utilisation of design patterns [33]. However, as has been demonstrated in several publications [34,35], sometimes design patterns are not enough to solve these modularity problems. The problem in these cases is related to the limited expressivity power of the languages used and new constructs are needed, provided by aspect-oriented languages.

The decision of selecting the interplay between source and target decompositions is not trivial. Observe that, in [46], the authors presented the selection of the suitable trace granularity as one of the best practices to be considered for obtaining automated traceability methods. As an example, Egyed et al. [47] evaluated the advantages of tracing at lower levels of granularity versus the effort needed to create the links between the elements at this finer granularity level. They concluded that the benefits obtained by improving the granularity of trace links beyond a certain level were really limited. In that sense, by applying our aspect mining process to several case studies and comparing with similar approaches, we checked that the results obtained by the process were consistent with those obtained by the other approaches. This fact validates the process presented even when the source and target decompositions have not been selected by the authors of this process (we used the original systems presented by other authors). Anyway, by using the expertise obtained by applying the framework to different abstraction levels [21,4] and case studies (shown throughout Section 4), we obtained some indications (as an oracle) of the granularity level that should be used at different abstraction levels. We identified the need for using fine granularity levels (e.g. classes or methods) at programming (and detailed design) phases and coarser granularity levels (e.g. use cases or components) at early stages of development (like requirements or architecture). The use of the generic crosscutting pattern allows the utilisation of the wished granularity level depending on the abstraction level and the purpose of the analysis.

5. Related works

There are some works which have used similar analyses to those presented here to identify crosscutting behaviour. In [9] the authors presented an approach which combines the three main techniques of aspect mining: fan-in, identifier

and dynamic analyses. However, all these techniques have been traditionally used just at the programming level. In [48] the authors introduce an approach to discover aspects in domain-specific models. This approach uses a clone detection technique to find out similarities between models. The application of aspect mining techniques at modelling levels rather than source code has important benefits for software development. However, the approach presented in [48] may not be applied to different stages of development or across several refinements levels.

Several works have introduced the need for using aspect-oriented techniques in early stages of development. In [1] the authors introduce *AORE with arcade* that proposes the separation and composition of aspectual and non-aspectual requirements. This work provides a general requirements engineer process which may be instantiated using different concrete techniques [49]. As an example, in [42] a concrete instantiation is presented where viewpoints and XML-based composition mechanisms are used. However, this work is mainly focused on the modelling and composition of aspectual requirements and it only identifies non-functional requirements as aspectual requirements. Moreover, this identification is performed manually. In [50] a method to model aspectual requirements and to compose them with the base system is proposed. In this case, this work uses a goal-oriented requirements approach where aspects are identified based on the relations between functional and non-functional goals. In particular non-functional requirements are represented as soft-goals and those with a high fan-in are marked as candidate aspects. However, again this technique only identifies NFCs as early aspects and this identification is carried out manually. In [51] use case diagrams are also utilised to identify crosscutting relations based on include and extend relations. However, unlike our proposal, they do not deal with NFC identification so that some crosscutting concerns may be missed. A deeper analysis of all these approaches (among others) may be found in [49].

More recently, the work presented in [52] has shown a comparison between different approaches using syntactic and semantic composition mechanisms to weave aspectual and base requirements. The authors demonstrate that semantic based composition mechanisms have important benefits being less fragile and more expressive than syntax based ones [52]. However, this work mainly focuses on the composition of aspectual requirements previously identified (and not in the previous task of identification).

The identification of crosscutting concerns at requirements stages has also been investigated in works such as [17,18]. In [17] the authors use a Natural Language Processing technique to identify base and crosscutting concerns in requirements documents. However, this approach does not provide support for traceability analysis across several refinement levels. As we introduced in [4], our conceptual framework also supports traceability analysis so that the process presented here could be extended to later phases of the development (such as architecture design or detailed design). The approach presented in [18] also uses a textual analysis to identify crosscutting behaviour at requirements documents. As we discussed in the functional concerns identification, the developer must identify a set of key actions to use as input for the tool. A similar approach was also introduced in [22], where the authors use a statistical model to decide whether a non-functional requirement is present in the requirements of a system. In particular, this approach is based on a two step process: in the first step the relevant keywords related to a particular non-functional requirement are manually selected; the second step allows the automated identification of these non-functional requirements and others in different requirement specifications. As the authors explain in [22], the results obtained by the approach are really good and the relation between actual non-functional requirements identified and false positives or negatives is really convincing. However, this approach just deals with the identification of non-functional requirements as candidate aspects and functional requirements are left out of the analysis. Moreover, all these approaches may not be applied to other artefacts different from text such as use cases.

Another interesting work presented in [23] provides a framework for tracing concerns between several refinements level. The authors also use XQuery to perform the queries over XML data. Nevertheless, this work is focused on the traceability of concerns and lacks support for identifying crosscutting concerns.

Regarding the concern-oriented metrics, as shown in Section 4.1, there are similar aspect-oriented metrics suites as introduced in [14,38]. The matrices used in our aspect mining process framework may assist in the visualisation and application of the metrics presented in those publications. Moreover, the crosscutting product and crosscutting matrices provide specific measures for the degree of crosscutting. Then, our metrics complement the metrics suites presented in [14, 38]. There are other authors who have also presented concern-oriented metrics. In [53], Ducasse et al. introduce four concern measures: size, touch, spread and focus. Wong et al. introduce in [54] three concern metrics: disparity, concentration and dedication. These three metrics are used in [38] to define degree of scattering and degree of tangling. Again, all these metrics are tied to the programming level. In [20] we showed a deeper comparison between our metrics and all of these metrics and demonstrated how modularity is correlated to stability.

6. Conclusions

Aspect identification is still one of the main challenges in aspect-orientation. AOSD is meaningless unless crosscutting concerns are properly identified in software systems. Moreover, the identification of these crosscutting concerns at early stages allows the incorporation of the benefits of AOSD in these early phases of development. In that sense, this paper presented a whole process to identify crosscutting concerns at the requirements level. The process is based on the conceptual framework that we introduced in [4] where a crosscutting pattern is defined relating to source and target domains. The framework is independent of any abstraction level so that the aspect mining process is independent of any specific requirement artefacts. The framework has been extended by adding some syntactical and dependencies based analyses. By means of these analyses, the dependency matrix used by the framework may be automatically obtained. XML was used

as a standard language to model the elements of source and target. In particular, we have used XMI to represent the UML diagrams that we use. For the syntactical analysis a NFCs catalogue was used. Both, the syntactical and dependency based analyses, have tool support.

The process was validated showing its application to different systems. In this validation, an aspect-oriented metrics suite was used. These metrics may provide the developer with important data to be used in complex applications and to discern false positives or negatives. The metrics used were also compared with other similar metrics demonstrating that the results obtained were consistent with those obtained by the other metrics. This first empirical study was conducted using the MobileMedia, a product line built in eight different releases, so that the crosscutting features of the different releases were identified. The process was also compared with other similar approaches discussing the results obtained and other important open issues. In that sense, we showed how the manual approaches could get better results in small systems but in complex ones its utilisation is not advisable and the use of automatic approaches may improve the results obtained.

Moreover, the empirical analysis of crosscutting relations at early stages of development allows the comparison of the results obtained with similar studies performed at later stages (e.g. detailed design or programming). Using this comparison we could test different hypotheses, such as, whether similar properties of crosscutting concerns are found to be indicators of software anomalies at later stages or what probabilities of early crosscutting measurements lead to false warnings (i.e. false positives or negatives) at source code level [20]. In that sense, the early aspect-oriented refactoring aims to avoid the crosscutting concerns identified at early aspects causing modularity anomalies at source code level. Note that early discovering of aspects is not only relevant for architectural design but also for design and code level, since the candidate aspects may be also evaluated at these levels, minimising the need to mine and refactor aspects from the code. Although this paper just focuses on requirements, a key contribution of our approach is that it may be applied across several refinement levels allowing traceability analysis [4]. Moreover, the crosscutting pattern has other interesting applications such as the quantification of crosscutting or stability analysis [20]. We also applied it to the identification of volatile concerns at requirements [55].

Acknowledgements

This work has been carried out with the support by MEC under contract TIN2008-02985. We thank Bedir Tekinerdogan for allowing us to use the CFVS example and Alessandro Garcia and Eduardo Figueiredo for allowing us to use the MobileMedia case study and for their helpful comments on the first versions of this paper. Finally, we would also like to thank the anonymous reviewers and the different editors for their useful comments and reviews of this paper.

References

[1] R. Filman, T. Elrad, S. Clarke, M. Aksit, Aspect-Oriented Software Development, Addison-Wesley, Boston, USA, 2004.
[2] G. Kiczales, J. Lamping, A. Mendhekar, C. Meada, C. Lopes, J. Loingtier, J. Irwin, Aspect-oriented programming, in: Proc. 11th European Conference on Object-Oriented Programming, ECOOP, Jyväskylä, Finland, 1997.
[3] E. Dijkstra, A Discipline of Programming, Prentice Hall, Upper Saddle River, NJ, USA, 1976.
[4] K. van den Berg, J. Conejero, J. Hernández, Analysis of cross-cutting in early software development phases based on traceability, in: Transactions on Aspect-Oriented Software Development III, in: LNCS, vol. 4620, Springer, 2007, pp. 73–104.
[5] G. Kiczales, Cross-cutting. AOSD.NET glossary, 2005. http://aosd.net/wiki/index.php?title=Cross-cutting.
[6] A. Kellens, K. Mens, P. Tonella, A survey of automated code-level aspect mining techniques, in: Transactions on Aspect-Oriented Software Development IV, in: LNCS, vol. 4640, 2007, pp. 143–162.
[7] S. Breu, J. Krinke, Aspect mining using event traces, in: Proc. of 19th International Conference on Automated Software Engineering, ASE, ISBN: 0-7695-2131-2, Linz, Austria, 2004, pp. 310–315.
[8] M. Bruntink, A. van Deursen, R.v. Engelen, T. Tourwé, On the use of clone detection for identifying cross-cutting concern code, IEEE Transactions on Software Engineering 31 (10) (2005) 804–818.
[9] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, T. Tourwé, Applying and combining three different aspect mining techniques, Software Quality Journal 14 (3) (2006) 209–231.
[10] D. Shepherd, T. Tourwé, L. Pollock, Using language clues to discover cross-cutting concerns, in: Proc. of the 1st International Workshop on the Modeling and Analysis of Concerns, St. Louis, USA, 2005.
[11] P. Tonella, M. Ceccato, Aspect mining through the formal concept analysis of execution traces, in: Proc. of 11th IEEE Working Conference on Reverse Engineering, WCRE, Delft, The Netherlands, 2004.
[12] T. Tourwé, K. Mens, Mining aspectual views using formal concept analysis, in: Proc. of 4th International Workshop on Source Code Analysis and Manipulation, SCAM, Chicago, USA, 2004.
[13] A. Garcia, C. Lucena, Taming heterogeneous agent architectures, Communications ACM 51 (5) (2008) 75–81.
[14] C. Sant'Anna, E. Figueiredo, A. Garcia, C. Lucena, On the modularity of software architectures: a concern-driven measurement framework, in: Proc. of the 1st European Conference on Software Architecture, ECSA, Madrid, Spain, 2007.
[15] E. Baniassad, P. Clements, J. Araújo, A. Moreira, A. Rashid, B. Tekinerdogan, Discovering early aspects, IEEE Software 23 (1) (2006) 61–70.
[16] Early Aspects, Aspect-oriented requirements engineering and architecture design, 2007. http://www.early-aspects.net/.
[17] A. Sampaio, R. Chitchyan, A. Rashid, P. Rayson, EA-Miner: a tool for automating aspect-oriented requirements identification, in: Proc. of the International Conference on Automated Software Engineering, ASE, CA, USA, 2005.
[18] E. Baniassad, S. Clarke, Theme: an approach for aspect-oriented analysis and design, in: Proc. of the 26th International Conference on Software Engineering, ICSE, Edinburgh, Scotland, 2004, pp. 158–167.
[19] J. Hannemann, G. Kiczales, Overcoming the prevalent decomposition in legacy code, in: Proc. of Workshop on Advanced Separations of Concerns at 23rd ICSE, Toronto, Canada, 2001.
[20] J. Conejero, E. Figueiredo, A. Garcia, J. Hernández, E. Jurado, Early cross-cutting metrics as predictors of software instability, in: Proc. of the 47th International Conference Objects, Models, Components, Patterns, TOOLS Europe, LNBIP 33, Zurich, Switzerland, 2009, pp. 136–156.
[21] K. van den Berg, J. Conejero, J. Hernández, Identification of cross-cutting in software design, in: Proc. Aspect Oriented Modeling Workshop at 5th AOSD, Bonn, Germany, 2006.

[22] J. Cleland-Huang, R. Settimi, X. Zou, P. Solc, Automated classification of non functional requirements, Requirements Engineering Journal 12 (2007) 103–120.
[23] B. Tekinerdoğan, M. Akşit, F. Henninger, Impact of evolution of concerns in the model-driven architecture design approach, Electronic Notes in Theoretical Computer Science 163 (2) (2007) 45–64.
[24] M. Robillard, G. Murphy, FEAT a tool for locating, describing, and analyzing concerns in source code, in: Proc. of the 25th International Conference on Software Engineering, ICSE, Portland, USA, 2003, pp. 822–823.
[25] S. Sutton, I. Rouvellou, Concern modeling for aspect-oriented software development, in: R.E. Filman, T. Elrad, S. Clarke, M. Aksit (Eds.), Aspect-Oriented Software Development, Addison-Wesley, Boston, USA, 2004, pp. 479–505.
[26] N. Wilde, M. Buckellew, H. Page, V. Rajlich, L. Pounds, A comparison of methods for locating features in legacy software, Journal of Systems and Software 65 (2003) 105–114.
[27] Unified modeling language 2.0 superstructure specification, 2004. http://www.omg.org/cgi-bin/doc?ptc/2004-10-02.
[28] XMI Mapping Specification, v2.1, 2005. http://www.omg.org/technology/documents/formal/xmi.htm.
[29] XQuery 1.0 An XML Query Language. W3C recommendation, 2007. http://www.w3.org/TR/xquery/.
[30] R. Filman, D. Friedman, Aspect-oriented programming is quantification and obliviousness, in: Workshop on Advanced Separation of Concerns, OOPSLA, Minneapolis, USA, 2000, pp. 21–35.
[31] A. Moreira, J. Araujo, J. Whittle, Modeling volatile concerns as aspects in: Proc. of the 18th Conference on Advanced Information Systems Engineering, CAISE, in: LNCS, vol. 4001, Louxemburg, 2006, pp. 544–558.
[32] R. France, D. Kim, S. Ghosh, E. Song, A UML-based pattern specification technique, IEEE Transactions on Software Engineering 30 (3) (2004) 193–206.
[33] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design patterns, in: Elements of Reusable Object-oriented Software, Addison-Wesley, Upper Saddle River, NJ, USA, 1995.
[34] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, A. Staa, Modularizing design patterns with aspects: a quantitative study, in: Transactions on Aspect-Oriented Software Development I, in: LNCS, vol. 3880, Springer, 2006.
[35] J. Hannemann, G. Kiczales, Design pattern implementation in Java and AspectJ, in: Proc. of 17th ACM conference on OOPSLA, Seattle, USA, 2002, pp. 161–173.
[36] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Filho, F. Dantas, Evolving software product lines with aspects: An empirical study on design stability, in: Proc. of the 30th International Conference on Software Engineering, ICSE, Leipzig, Germany, 2008, pp. 261–270.
[37] Mining early aspects based on syntactical and dependencies-based analyses, 2009. http://www.unex.es/eweb/earlyaspectmining/.
[38] M. Eaddy, A. Aho, Towards assessing the impact of cross-cutting concerns on modularity, in: Proc. of Workshop on Assessment of Aspect Techniques, ASAT, Vancouver, Canada, 2007.
[39] T. Young, Using AspectJ to build a software product line for mobile devices, M.Sc. Dissertation, Univ. of British Columbia, 2005.
[40] A. Colyer, A. Rashid, G. Blair, On the separation of concerns in programme families, Lancaster University Technical Report Number: COMP-001-2004, 2004.
[41] M. Griss, Implementing product-line features by composing aspects, in: Proc. of First International Software Product Line Conference, SPLC, Denver, USA, 2000, pp. 271–288.
[42] A. Rashid, A. Moreira, J. Araujo, Modularisation and composition of aspectual requirements, in: Proc. of the 2nd International Aspect Oriented Software Development Conference, AOSD, Boston, USA, 2003.
[43] A. Sampaio, A. Rashid, Report on evaluation of aspect identification tool (EA-Miner) in case studies, AOSD-Europe Network of Excellence, AOSD-Europe-ULANC-33, 2007.
[44] L. Kit, C. Man, E. Baniassad, Isolating and relating concerns in requirements using latent semantic analysis, in: Proc. of the OOPSLA Conference, Portland, USA, 2006.
[45] J. Neyman, E. Pearson, On the use and interpretation of certain test criteria for purposes of statistical inference, in: Joint Statistical Papers, Cambridge University Press, 1967.
[46] J. Cleland-Huang, R. Settimi, E. Romanova, B. Berenbach, S. Clark, Best practices for automated traceability, Computer Journal 40 (2007) 27–35.
[47] A. Egyed, S. Biffl, M. Heindl, P. Grünbacher, A value-based approach for understanding cost-benefit trade-offs during automated software traceability, in: Proc. of the 3rd International Workshop on Traceability in Emerging Forms of Software Engineering, Long Beach, USA, 2005.
[48] J. Zhang, J. Gray, Y. Lin, R. Tairas, Aspect mining from a modeling perspective, International Journal of Computer Applications in Technology 31 (2006) 74–82.
[49] R. Chitchyan, A. Rashid, P. Sawyer, A. Garcia, M. Pinto, J. Bakker, B. Tekinerdogan, S. Clarke, A. Jackson, Survey of analysis and design approaches, AOSD-Europe, D11, 2005. http://www.comp.lancs.ac.uk/computing/aop/papers/d11.pdf.
[50] Y. Yu, J. Leite, J. Mylopoulos, From goals to aspects: discovering aspects from requirements goal models, in: Proc. of the 12th IEEE International Requirements Engineering Conference, Kyoto, Japan, 2004, pp. 38–47.
[51] I. Jacobson, P.-W. Ng, Aspect-Oriented Software Development with Use Cases, Addison Wesley Professional, Upper Saddle River, NJ, USA, 2005.
[52] R. Chitchyan, P. Greenwood, A. Sampaio, A. Rashid, A. Garcia, L. Fernandes da Silva, Semantic vs. syntactic compositions in aspect-oriented requirements engineering: an empirical study, in: Proc. of the 8th International Conference on Aspect-Oriented Software Development, Charlottesville, USA, 2009, pp. 149–160.
[53] S. Ducasse, T. Girba, A. Kuhn, Distribution map, in: Proc. of the International Conference on Software Maintenance, ICSM, Philadelphia, USA, 2006.
[54] W. Wong, S. Gokhale, J. Horgan, Quantifying the closeness between program components and features, Journal of Systems and Software 54 (2000) 87–98.
[55] J. Conejero, J. Hernandez, A. Moreira, J. Araujo, Discovering volatile and aspectual requirements using a cross-cutting pattern, in: Proc. of the 15th. IEEE International Requirements Engineering Conference, Posters, India, 2007.