An Aspect-Oriented Approach to Relating Security Requirements and Access Control[⋆]

Azadeh Alebrahim[1], Thein Than Tun[2], Yijun Yu[2],
Maritta Heisel[1], and Bashar Nuseibeh[2,3]

[1] paluno – The Ruhr Institute for Software Technology,
University of Duisburg-Essen, Germany
{azadeh.alebrahim,maritta.heisel}@paluno.uni-due.de
[2] Department of Computing, The Open University, Milton Keynes, UK
{t.t.tun,y.yu,b.nuseibeh}@open.ac.uk
[3] Lero – The Irish Software Engineering Research Centre,
University of Limerick, Ireland

Abstract. Affecting multiple parts in software systems, security requirements often tangle with functional requirements. In order to separate crosscutting concerns and increase modularity, we propose to represent security requirements as aspects that can be woven into functional requirements. Using problem frames to model the functional requirements, weaving is achieved by composing the modules representing security aspects with the requirement models. Moreover, we provide guidance on how such security aspects are structured to implement a particular access control solution. As a result, such security aspects become reusable solution patterns to refine the structure of security-related problem.

Key words: security requirement, aspect-oriented requirements engineering, security pattern, access control, problem frames

1 Introduction

Aspect-Oriented Programming (AOP) [4] is a programming paradigm that deals with crosscutting concerns at the implementation level in order to achieve a *separation of crosscutting concerns*. The crosscutting concerns are encapsulated into separate modules, known as *aspects*, that can be woven into a base system without altering its structure. This provides support for modularity and maintainability. The aspect-oriented concept has been adapted for earlier stages of software development, known as Aspect-Oriented Requirements Engineering (AORE) [11] to deal with crosscutting concerns at the requirements level. Quality concerns [1] such as security affect several parts of software systems, and are considered as crosscutting concerns. The first focus of this paper is on providing support for the separation of security requirements from functional requirements by modularising them into aspects.
The second focus of this paper is on providing guidance for refining the security aspects at the requirements level by reusing knowledge located in the solution space to bridge the gap between security problems and their solutions. The

[⋆] Part of this work is supported by the German Research Foundation (DFG) grant HE3322/4-1 and the Science Foundation Ireland grant 10/CE/I1855.

elaborated security aspects can be transformed into a particular solution at the design level. We believe that requirement descriptions cannot be considered in isolation and should be developed with architectural descriptions concurrently, as described by the Twin Peaks model [9]. Patterns describe solutions for recurring problems in software development, thus providing a means to reuse knowledge. Security patterns [12] provide solutions for software problems in the context of security. We aim to leverage security patterns as solution artefacts to refine the security aspects.
The remainder of this paper is structured as follows. Section 2 describes our approach by taking into account access control as a crosscutting security concern and problem frames as a requirements engineering method. Section 3 discusses related work. Conclusions and discussions of future work are given in Section 4.

2 Our Approach using Problem Frames and Access Control

The proposed approach has four steps. As suggested earlier, requirements and architectural descriptions should be considered as intertwining artefacts influencing each other. Therefore our method is illustrated as an instantiation of the Twin Peaks model [9] (see Figure 1). Considering security requirements as crosscutting concerns that can be modularised into separate aspects, we refine them by using security patterns as solution artefacts. The weaving of aspects into the functionality of the software system is achieved by composing the refined aspects with the functional artefacts. To illustrate these concepts, we use access control as the example and problem frames [3] as the requirements engineering method. We use the problem frames approach, because it allows us to navigate between the problem space and the solution space, while exploring problem structures using problem diagrams and solutions structures using patterns. This ability to express and relate the structures of problems and solutions is crucial for the proposed approach.
Problem frames are means to describe and classify software development problems. A problem frame represents a class of software problems. It is described by a frame diagram, which consists of domains, interfaces between them, and a requirement (see Figure 3). The objective of problem solving is to construct a machine (i.e., software) that controls the behaviour of the environment (in which it is integrated) in accordance with the requirements. Requirements analysis with problem frames decomposes the overall problem into subproblems, which can also be represented by problem diagrams. A problem diagram is an instance of a problem frame. Machines in problem diagrams represent solutions for functional requirements. We call them functional machines to distinguish them from machines representing solutions for security requirements that we introduce later in this section.

2.1 Identifying Access Control (AC) as an Aspect

Different modules representing different functional requirements in a system usually share common security concerns. Treating security requirements in isolation

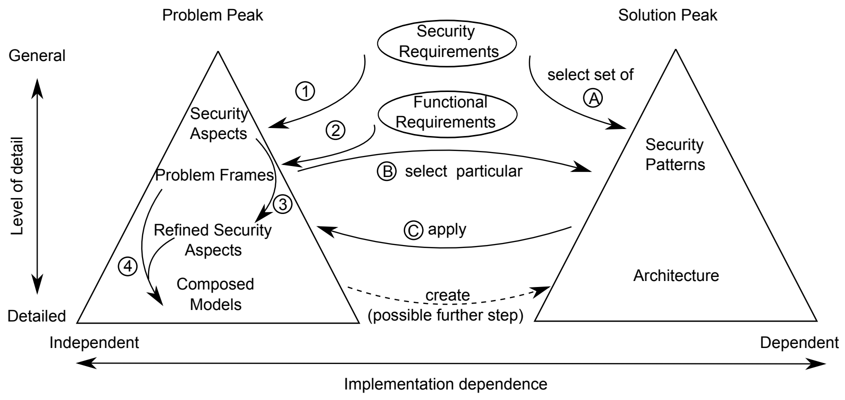

Fig. 1. Overview of our method embedded in the Twin Peaks Model [9]

from the functional requirements enables increased modularity and maintainability of the software. This idea is supported by AORE, which seems to be a promising approach to deal with quality requirements as crosscutting concerns to be separated from the functional requirements [11]. The first step in our method is therefore to identify the crosscutting security concerns that can be captured as a single security requirement, represented as an aspect (step 1 in Figure 1).

Access control verifies whether a subject has the permission to access an object within the system. Therefore each user (*subject*) requesting access to the sensitive parts of a system (*object*) should be checked for a permission. Thus, we could express the security requirement addressing access control over a functional requirement as follows:

– SR: Only *subject* with permission to access the *object* before carrying out a function.

We introduce the *advice frame* to express such an access control requirement. The advice frame is illustrated in Figure 2. There is a *Subject* assumed to be a *biddable* domain, as shown by B in the lower right of the rectangle. The subject issues *commands* requesting access to an *Object*, which can be modelled as either a causal domain or a lexical domain. The *Controller* machine shall authorize the subject, validate the command and change the state of the object according to the command. If a user is not authorized or a command is not valid, the Controller machine shall do nothing. We make the behaviour of the identified security machine more concrete by describing its specification as follows:

```
SUBJECT INPUT: userId,command,object
IF SUBJECT INPUT is valid
THEN (Controller (C) does changeState;
Controller (C) performs do(command,object);)
ENDIF
```

Note that the identified security concern is considered in isolation in this step. Therefore it cannot be fully specified. We refine the specification as we proceed.

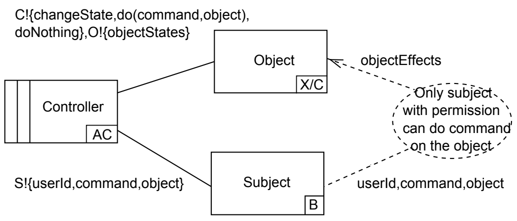
C!{changeState,do(command,object),
doNothing},O!{objectStates}
Object
X/C
objectEffects
Controller
AC
Only subject with permission can do command on the object
S!{userId,command,object}
Subject
B
userId,command,object

Fig. 2. *Advice frame* representing access control

2.2 Capturing Functional Requirements into Problem Diagrams

At the step two, we model functional requirements by using problem frames (see step 2 in Figure 1). Most security relevant systems contain sensitive information that should not be accessible to all users of the system. Sensitive information to be protected in a system is represented as either a causal or a lexical domain. Users are represented as biddable domains. Therefore such problems are generally modeled in problem frames either by the *commanded behaviour frame* containing a causal domain as sensitive information or by the *simple workpieces frame* containing a lexical domain as sensitive information, illustrated in Figure 3.
The commanded behaviour problem frame represents the problem of controlling some parts of the (*Controlled domain*) in the physical world by the machine (*Control machine*) according to commands issued by the *Operator*. The simple workpieces problem frame describes the problem of creating or editing a text or graphic (*Workpieces*) by the machine (*Editing tool*) according to the *User* commands (for details, see [3]).

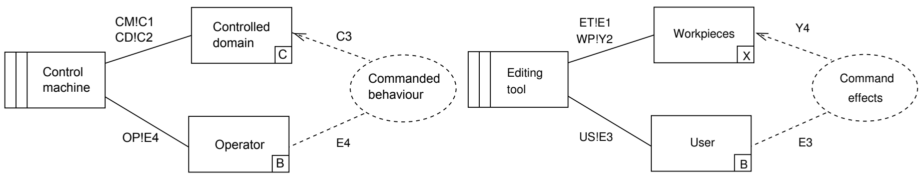

Fig. 3. Commanded Behaviour and Simple Workpieces Frames

2.3 Refining the Access Control Aspect using the RBAC Pattern

Security patterns [12], located in the solution space, provide a widely accepted means to build secure software. Usage of security patterns as solution artefacts aids to address security aspects in the problem space, which is the aim of step 3. Substeps A–C illustrated in Figure 1 deal with selecting and applying the most appropriate security pattern in order to refine the identified security aspect. Exactly how a pattern is selected in this approach is a topic for further research. In this work, we describe a way to structure the security machines, which are considered as black boxes with an unknown structure so far, using using security patterns.

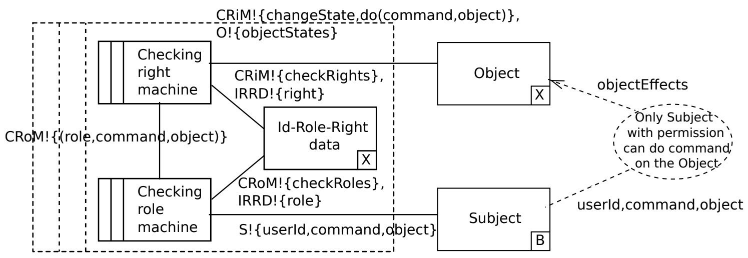

Fig. 4. Refined advice frame by using RBAC pattern

Since verifying permission is a frequently recurring problem in security relevant systems, it has been treated by several access control patterns [12]. Access control patterns define security constraints regarding access to resources. *Role-Based Access Control (RBAC)* provides access to resources based on functions of people in an environment (*roles*) and the kind of permission they have (*rights*). The *User* represents a registered user with certain *id* assigned to a predefined *Role*. Roles are assigned *Rights* in accordance with their functions. *Rights* define and check what resource the user is authorized to access.

Looking at the RBAC pattern in the solution space gives aid to decompose the advice machine. We identify two subproblems, *Checking role* and *Checking right* (see Figure 4). The *Checking role* subproblem represents the problem of checking the role assigned to the *User*, who is represented by the biddable domain *Subject* in Figure 4. The *Checking role machine* verifies whether the subject id is contained in the *Id-Role-Right data*. If there is no id-role relation the machine does nothing. If such a relation exists the machine passes a pair of role and command on to the *Checking right machine*. We specify the *Checking role machine* as stated below:

```
SUBJECT INPUT: userId,command,object
Checking role machine (CRoM) identifies the role of userId
IF there is a role for userId THEN Checking role machine (CRoM) passes
(role,command,object) to Checking right machine (CRiM);
ENDIF
```

The *Checking right machine* checks if a particular role is authorized to perform an operation on the object. If the subject with the particular role holds the right to perform the command, the machine changes the state of the object and performs the operation. Otherwise the machine does nothing:

```
INPUT: role,command,object
Checking right machine (CRiM) checks whether the role is allowed to perform command on the object
IF the role is allowed THEN (Checking right machine (CRiM) does changeState;
Checking right machine (CRiM) performs do(command,object);)
ENDIF
```

2.4 Weaving Aspects into Problem Diagrams

We now introduce a *weaving frame* to compose the refined security aspect with the problem diagrams (step 4 in Figure 1). The weaving frame includes all the

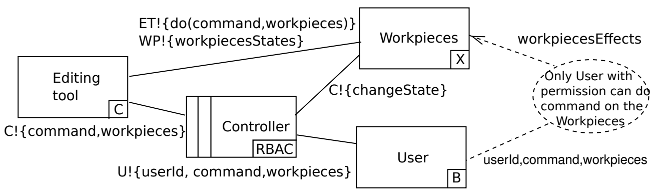
ET!{do(command,workpieces)}
WP!{workpiecesStates}
Workpieces
X
workpiecesEffects
Editing tool
C
Only User with permission can do command on the Workpieces
C!{changeState}
C!{command,workpieces}
Controller
RBAC
User
B
U!{userId, command,workpieces}
userId,command,workpieces

Fig. 5. Abstract *weaving frame*

domains from the basic problem frame (commanded behaviour or simple workpieces), including both the functional machine and the advice machine. We complete the specification of the external behaviour of the security aspect outlined in step 1. The internal behaviour remains unchanged as specified in step 3.
The weaving is achieved by mapping domains in the basic problem frame to domains in the advice frame. The domains *Controlled domain/Workpieces* in Figure 3 are mapped to the *Object* domain in the advice frame, and the domain *User* to the domain *Subject*. We consider the case for the simple workpieces frame as functional frame in the following. The weaving of the functional frame commanded behaviour is carried out analogously. We define *join points*, which represent transformation rules to transform the functional frame into the weaving frame by means of weaving of the advice frame. Changes in addition to mappings are italicised. In order to affect the behaviour of the functional machine by verifying user inputs, we place the advice machine on the interface between the *User* domain and the *Editing tool* (see Figure 5).

Join points	User = Subject, Workpieces = Object, Editing tool = Editing tool, E3 = {userId,command,object}, Y4 = objectEffects, Y2 = {objectState}, E1 = {do(command,workpieces)}, *ADD domain Controller, ADD interface with phenomena C!{command,workpieces}, ADD interface with phenomena C!{changeState}*

The advice machine passes the valid command to the functional machine, which performs directly the operation on the Workpieces domain according to the User command. We specify the advice machine as follows:

```
USER INPUT: userId,command,workpieces
IF USER INPUT is valid
THEN (Controller (C) passes (command,workpieces) on Editing tool;
Controller (C) does changeState;)
ENDIF
```

Figure 6 illustrates how the refined RBAC aspect woven into the simple workpieces frame diagram.
Applying our method to a software development problem, we achieve modularity as the access control requirement is now captured as an RBAC aspect. As a result potential changes to this module do not affect the functional models, increasing the maintainability of the system. We made the internal structure and behaviour of the access control machine more concrete by applying the RBAC pattern as solution and describing its specification in detail. Here the access control machine

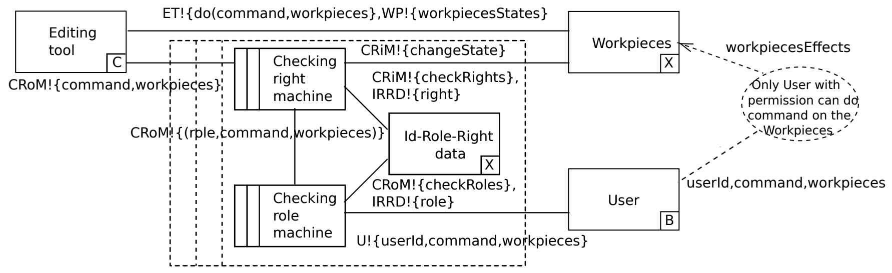
ET!{do(command,workpieces},WP!{workpiecesStates}
Editing tool
C
CRoM!{command,workpieces}
Checking right machine
CRiM!{changeState}
Workpieces
X
workpiecesEffects
CRiM!{checkRights}, IRRD!{right}
Only User with permission can do command on the Workpieces
CRoM!{(role,command,workpieces)}
Id-Role-Right data
X
Checking role machine
CRoM!{checkRoles}, IRRD!{role}
User
B
userId,command,workpieces
U!{userId,command,workpieces}

Fig. 6. Refined weaving frame

is not considered as a black box anymore: we have taken one step further towards implementing RBAC as a particular solution for the access control requirement, thus bridging the gap between the problem and the solution space.

3 Related Work

An AORE model to support the separation of functional and quality concerns is proposed by Rashid et al. [11]. Quality concerns are identified and refined as aspects, which are prioritised in order to resolve conflicts among them. In contrast to our work, Moreira et al. [7] augment the AORE model by a uniform treatment of functional and quality concerns. The method proposed in [8] integrates crosscutting quality attributes into the functional description after identifying and specifying them. However it does not consider solution approaches to refine the crosscutting quality attributes as our approach does. Unlike the goal aspect approach [13], where quality softgoals are refined as aspectual tasks, problem frames-based approach allows navigating between them through physically connected interfaces.

There exists some work that relates aspect concepts to problem frames. Laney et al. [5] propose resolving inconsistencies when composing multiple problem frames. Here we specialise the composition frames to weave security aspects into functional structures. The approach proposed by Lencastre et al. [6] incorporates aspect concepts into problem frames by extending an existing meta-model to express crosscutting relationships between different element types of problem frames. Their work does not focus on treating quality requirements as aspects.

The aforementioned approaches in contrast to our work only discuss methods to incorporate crosscutting concerns into the requirement models. We take one step further towards bridging the gap between the problem and the solution space.

The security Twin Peaks model [2] (an elaboration of the original Twin Peaks model [9]) is a framework for developing security in the requirements and the architectural artefacts in parallel. Taking architectural security patterns into account, the model supports the elaboration of the problem and the solution artefacts. Similar to our work, a method to bridge the gap between security requirements and the design is proposed by Okubo et al. [10]. This method introduces new security patterns at the requirements and the design level, in contrast to our approach that reuses the existing security design patterns at the requirements level.

4 Conclusions and Future Work

We have proposed a method using problem frames to refine the security aspects in the problem space by using the artefacts located in the solution space. We have selected access control as one important security concern to illustrate the refining process. The benefits of our approach are twofold. The first is that we separate security requirements from functional requirements and encapsulate them into separate modules as aspects. Thus we achieve a separation of concerns that increases the modularity of the software. The second benefit is that we give guidance how the security aspects need to be structured to fit a particular solution. To this end we have used security patterns as solutions artefacts to refine the problem structure.

In future work, we will investigate how to find the most suitable security pattern in the set of available security patterns. Finding the most suitable security pattern depends on the context and also on the functional requirements. We will extend the scope of this work by considering different security requirement aspects that need different security patterns to be satisfied.

Acknowledgments. We would like to thank Takao Okubo, Nobukazu Yoshioka, and Haruhiko Kaiya for useful discussions.

References

1. L. Chung, B. Nixon, E. Yu, and J. Mylopoulos. *Non-functional Requirements in Software Engineering.* Kluwer Academic Publishers, 2000.
2. T. Heyman, K. Yskout, R. Scandariato, H. Schmidt, and Y. Yu. The security twin peaks. In *ESSoS'11*, LNCS 6542, pages 167–180. Springer, 2011.
3. M. Jackson. *Problem Frames. Analyzing and structuring software development problems.* Addison-Wesley, 2001.
4. G. Kiczales and E. Hilsdale. Aspect-oriented programming. In *ESEC'01/FSE-9*, pages 313–, USA, 2001. ACM.
5. R. Laney, L. Barroca, M. Jackson, and B. Nuseibeh. Composing requirements using problem frames. In *RE'04*, pages 122–131. IEEE Computer Society, 2004.
6. M. Lencastre, J. Araujo, A. Moreira, and J. Castro. Analyzing crosscutting in the problem frames approach. In *IWAAPF'06*, pages 59–64, USA, 2006. ACM.
7. A. Moreira, J. ao Araújo, and A. Rashid. A concern-oriented requirements engineering model. In *CAiSE'05*, pages 293–308. Springer, 2005.
8. A. Moreira, J. a. Araújo, and I. Brito. Crosscutting quality attributes for requirements engineering. In *SEKE'02*, pages 167–174, USA, 2002. ACM.
9. B. Nuseibeh. Weaving together requirements and architectures. *IEEE Computer*, 34(3):115–117, 2001.
10. T. Okubo, H. Kaiya, and N. Yoshioka. Effective Security Impact Analysis with Patterns for Software Enhancement. In *ARES'11*, pages 527–534, 2011.
11. A. Rashid, A. Moreira, and J. Araújo. Modularisation and composition of aspectual requirements. In *AOSD'03*, pages 11–20, USA, 2003. ACM.
12. M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad. *Security patterns: integrating security and systems engineering.* John Wiley & Sons, 2005.
13. Y. Yu, J. C. S. do Prado Leite, and J. Mylopoulos. From goals to aspects: Discovering aspects from requirements goal models. In *RE*, pages 38–47. IEEE Computer Society, 2004.