Enforcing authorization constraints in workflows by using a security monitor

Darfoufi, R.

Award date:
2016

Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Security Group

Enforcing Authorization Constraints in Workflows by Using a Security Monitor

Master Thesis

Redouane Darfoufi

Supervisors:
dr. Jerry den Hartog
dr. Fatih Turkmen
dr. Massimiliano de Leoni

Eindhoven, June 2016

Abstract

A workflow, is a collection of related tasks which realize a business goal of an organization. To reach that goal, users process elementary information such as financial data by executing tasks. However, unauthorized users having access to all this sensitive information form a threat to the organization. Users can intentionally leak information, or misuse the process for personal benefits which is marked as fraud. In this regard, authorization constraints are applied on tasks to prevent users from executing a task that will result in a violation. Authorization constraints are evaluated and enforced by software components called security monitors. A security monitor is a unit of code that checks whether a task execution violates a constraint at runtime.

In this thesis, we address the problem of developing a security monitor that enforce authorization constraints at runtime in security sensitive workflows that are representing a business process. This problem emerges from the fact that existing security monitor approaches can handle only a limited number of authorization constraints such as Dynamic Separation of Duty. Furthermore, they are applied only on certain type of workflows. These monitors are crucial to protect the workflows since they are needed to mediate access to the tasks of a workflow by enforcing authorization constraints. Thus, our thesis is dedicated to the introduction of a feasible security monitor that is capable of enforcing authorization constraints that can capture any security property required. More precisely, it describes the development process of monitors. There are two important issues related to security monitors: the formal specification of constraints and their enforcement at runtime.

First, we examined the existing work of security monitors to identify their issues. Based on this examination, we developed a simple notation syntax to specify constraints and a modular security monitor. The presented syntax allows security designers to specify any constraint they desire, while our security monitor verifies these constraints at runtime when a violation is detected. However, just detecting a violation of the constraints is not sufficient to ensure the security of the workflow. Therefore, our security monitor also enforces the constraints when a violation is detected by skipping tasks execution.

This thesis shows an approach that combines a simple formal syntax notation and a modular security monitor based on Aspect-Oriented Programming for specifying and enforcing constraints at runtime. The introduced notation enables the ability to specify constraints loosely coupled from the workflow design time. In addition, the notation does not inherit any complex mathematical formalization which helps security designers to specify constraints with satisfaction. Furthermore, our approach supports automatically transforming specified constraints into enforcement code to prevent human errors. Then, these constraints are embedded in an AspectJ aspect that represents the security monitor.

We validate the security monitor by performing a case study in which we use a workflow that represents a mortgage request of a bank, showing the feasibility of the security monitor. For this purpose, we apply real world constraints on the workflow tasks such as Dynamic Separation of Duty which are covered by most business processes.

Acknowledgments

First I offer my sincerest gratitude to my tutor dr. Fatih Turkmen and supervisor dr. Jerry den Hartog.

I would have never finished this thesis without the help and support of Fatih Turkmen. He always provides me with advices and guides me to clear up my goal ideas. I am especially grateful to Fatih Turkmen for being patience with me during our meetings and for his support to continue working with me to achieve a better contribution. The discussions and ideas that we had during our meetings with Jerry den Hartog were of great importance for this thesis. The comments of Jerry den Hartog were helpful to the contributions of this thesis result. The meetings were very enlightening experience for me, that had taught me a lot.

My sincere appreciation goes out to my parents for their continuous encouragement and support during my time as a student at TU/e.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

A business process is a process to accomplish a specific goal of an organization. Examples of such processes include a mortgage request of a bank, processing orders and shipping products. The computerised facilitation of business processes is referred to as a *workflow*. A workflow consists of a number of activities which are called tasks. A task is a unit of work to process information to reach a desired goal. The tasks of a workflow require human interaction (e.g. employees of an organization) which are usually referred to users. The users of an organization execute the tasks in a specific order. The key objective of the workflow is that it uses technology to improve speed and flexibility of processes. To reach that goal, various types of information is needed such as financial data, e-mails, users information. However, unauthorized users having access to all this sensitive information forms a threat to the organization. Users can intentionally leak information, or misuse the process for personal benefits which is marked as fraud. Or users can accidentally make an error which can lead to inconsistent information. This may cause troubles to the functionality of an organization.

In this regard, access control is an important measure which restricts task executions only to authorized users. *Constraints* are well established extensions of access control, aiming to prevent threats. They are applied on tasks to prevent users from executing a task that will result in a violation. For example, a constraint may impose that a teller of a bank may process the required information of a mortgage request whereas only a manager may decide to grant or reject the mortgage to a customer. This type of constraint is known as *Separation of Duty* (SoD). The idea behind SoD is to prevent fraud by ensuring that no single user can execute two tasks (e.g., teller and manager). Therefore, a task of a workflow is only executed by a user who is authorized to do so.

We distinguish two classes of constraints. The first class corresponds to authorization constraints which have a dynamic nature, and are also called history-dependent constraints. They restrict task executions based on previous task executions. The execution of a constrained task depends on the execution history of the workflow. They are complicated to evaluate since the state of a workflow changes continuously at every task execution. Because of the dynamic environment of the workflow, these classes of constraints are evaluated and enforced by a security monitor. The intention of the monitor is to verify whether the current task execution does not violate any constraint. To achieve this, the security monitor keeps track of who has executed previous tasks. Dynamic Separation of Duty (DSoD) constraint is an example of this. It requires that related security sensitive tasks must be executed by different users. The monitor evaluates the constraint by checking who is executing the current task, and who has executed the previous task. After evaluation the security monitor enforces the decision.

The second class has a static nature and can be encoded to authorization policy. For example, a

policy can specify that a task is only allowed to be executed by a user who works for a specific department. The evaluation of policies is not influenced by tasks execution history.

In this thesis, our main focus is on evaluating and enforcing authorization constraint by the security monitor in workflows.

1.1 Problem statement

Several security monitors have been proposed in literature, but they differ in their enforcement capability of constraints. In more detail, those security monitors can enforce only a limited number of constraints. Thus, they are restricted in the constraints they can handle. The problem lies in expressivity of the specification language in the existing monitors. Either they are too simplistic, in which case it is not possible for security designers to specify any constraint they want, or the specification language inherits an abstract mathematical formalization, which is hard for security designers to understand. Moreover, the implementation of the monitor depends on a specific type of workflow such as hierarchical workflows. As a result, these types of security monitors are not applicable to workflows which require more advanced constraints. For instance, a security designer may require that a task can only be executed by a user who works for a particular department and who has not executed any task yet.

Based on the above discussion we claim that the existing security monitors are too weak to specify and enforce advanced constraints in the dynamic environment of workflows. A more advanced security monitor should be introduced by taking into account real world workflows constraints. Therefore, the research question that must be solved in this thesis is formulated as follows:

Is it possible to develop a formal notation to specify advanced authorization constraints, that can be automatically enforced by a security monitor independent of the workflow implementation?

1.2 Research goal

Security monitoring is composed of two main stages. The first stage is the formal specification of the constraints. In this stage the constraints are specified in an abstract specification language. If this language is very limited in its expressiveness then it is not possible for a security designer to specify advanced constraints to conform the organization's security requirements. This first stage of the security monitor is defined in the first research goal of this thesis:

First research goal. Propose a formal specification notation in which it is possible to specify advanced authorization constraints. The syntax of the specification notation must not be complicated for the security designers, while the notation must still be expressive enough to specify advanced constraints.

The second stage consists of implementing the constraints in the security monitor. The idea is to automatically generate the constraints from the formal specifications into enforcement code. Then, this code is implemented in a modular way in the execution environment (e.g., workflow engine) of the workflow. As a result, having an automatic generation and modular implementation prevent human error and requires much less effort in the integration to existing workflow execution environment. This second stage of security monitor is defined in the second research goal of this thesis:

Second research goal. Design a security monitor that will evaluate and enforce the specified constraints, which are generated from the formal specified constraints in First research goal.

In addition, the security monitor must be implementable with minimal effort as possible for the security designers.

However, the capability of specifying constraints on early stages and their automated enforcement is quite a challenge. In this thesis, a security monitor is presented for specifying and enforcing advanced constraints in a workflow.

1.3 Contribution of this thesis

The main contribution of this thesis is the development of a framework for the enforcement of policies and constraints in a workflow at runtime. Our elaborated framework offers a practical and simple notation to specify authorization constraints which can be used for real world workflows. Our security monitor is based on Aspect-Oriented Programming (AOP) to allow modular integration of the monitor. Thus, our approach allows modularity for cross-cutting concerns which offers the ability to develop the monitor separately from the business process. This approach is a synthesized work of formal notation and aspect oriented programming for specifying and enforcing constraints at runtime. More specifically, the contributions of our thesis are:

- An expressive notation to specify authorization constraints which can be used for real world workflows.
- Authorization policies are described in the standard policy language XACML, which is widely known by most security developers.
- An automated process is proposed to generate enforcement code from the specified constraints.
- An attribute based model tailored to workflows that can capture our dynamic workflow components.
- Separation of concerns offers the capability of loose coupling of components: security developers have the capability to concentrate on the security itself, while business process experts can concentrate on the design of the workflow.

1.4 Structure of the thesis

This thesis is organized as follows.

- Chapter 2 presents the background and literature overview.
- Chapter 3 is dedicated to the first stage of the security monitor. In here, we introduce the formal notation to specify the authorization constraints. First, we introduce an attribute based model tailored to workflows. With this model we can describe the workflow components such as users and tasks. Second, we explain how to specify the constraints using our developed notation. To better illustrate our notation, we give an illustrative example workflow which represents a "mortgage provision request". We apply real world constraints on this workflow to show the expressivity of our notation.
- Chapter 4 presents our elaborated framework for enforcing policies and constraints. The main concept of this framework is the security monitor. We present an access control mechanism that will evaluate and enforce the specified policies. We present an AOP methodology to weave the security monitor in the framework in a modular way. We also give a broad

analysis to support our weaving methodology, and show that it indeed enforces the specified constraints.

- Chapter 5 presents the validation process of our framework. We describe two use cases of the mortgage workflow example. In each use case we test the security monitor to our conditions and show the results of the validation.
- Chapter 6 presents the contributions of this thesis and summarizes its results.
- Chapter 7 presents some ideas for future work.

Chapter 2

Background and Related work

This Chapter is dedicated to the related work and background knowledge in order to present the terminology and technique used in this thesis. The related work is about specifying and enforcing authorization constraints in workflows.

2.1 Authorized Workflows

Workflows have the purpose to achieve a business objective such as a mortgage provisioning request of a bank. To fulfill this business objective, the workflow models related tasks which are performed in some order by humans which are represented as users. A task is a unit of work that is performed by a user (e.g., employee). In these tasks, elementary information is being processed to achieve a desired end goal. Workflows are used to coordinate activities between users and tasks to reduce errors and time. Since a workflow is an automated process, it executes tasks more efficient and tasks can be better monitored. However, internal unauthorized users try to comprise sensitive information. As a consequence, in a workflow where users are required to execute multiple tasks, fraud or error can easily occur. To prevent these kind of threats in workflows, rules are introduced in the form of constraints such as Separation of Duty and Binding of Duty. In authorized workflows constraints are imposed on tasks or users who are executing the tasks. The constraints express which executions may occur and which must not occur. That is, a user shall only be authorized to execute a specific task. In this thesis, we consider that tasks are executed by human intervention.

2.1.1 Designing a workflow

Workflows are designed in a graphical pattern notation such as Business Process and Model Notation (BPMN) [17] and YAWL [21]. In this thesis, we consider only the BPMN language which provides a standard comprehensive set of symbols to design a workflow, see Figure 2.1 [17]. The activities are tasks which contain a unit of work to be processed by users. The circles, also called events are triggers for the activities. A task can only be activated if an event contains a token. The flow of the workflow is given by the arrows. Gateways are used to model decisions or design patterns. A BPMN designed workflow is easy to understand by business experts as well as by non-business experts (e.g., security designer). In this way, workflows can be evaluated against multiple possible executing paths. The paths represent execution orders of the tasks. In that way, business experts can capture the flow of the workflow to detect any problems such as deadlocks. The workflow can be analyzed and improved to achieve better performance. Since it is not the purpose of this thesis to describe workflows modeling techniques, we advice the interested reader

to visit the website [20] for a detailed description about the workflow's languages. The website is a collaboration between the Eindhoven University of Technology and Queensland University of Technology.

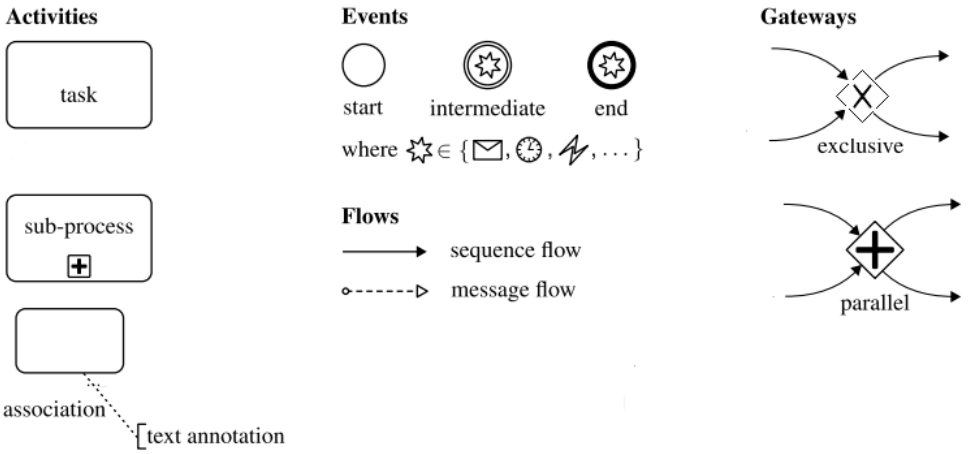

Figure 2.1: Basic BPMN modeling elements

2.1.2 Executing the workflow

The designed workflow is executed in a workflow engine. A workflow engine is software that manages the workflow. Meaning that the designed workflow is implemented in the workflow engine that takes care of the executions of tasks.

2.2 Attribute Based Access Control (ABAC)

Attribute Based Access Control (ABAC) [22] is being used for many years by means to control access of security sensitive objects (e.g., files, applications). In ABAC, objects are the entities that need to be protected from unauthorized access. The subject also referred to as the requester, is the entity that is requesting access to perform an action on an object. These entities are defined as a set of attributes which describe their properties such as name, age, department, etc. Attributes have typically an attribute id and a value pair shape, for example "name=Tom", "age=24" and "department=CSE". Based on attributes values, ABAC takes a decision whether a subject is allowed to perform the requested action on the object or not. The decision is a result of specified policies which are evaluated against the attribute values of the subject, object and environment.

So when a requester makes a request to access an object, the ABAC engine can take a decision based on the current attribute values of the requester, object, environment and the policies. Basically, this describes the core capability of ABAC.

The specified policies are enforced by an Access Control Mechanism (ACM). In order to make a decision, the ACM needs to gather information about the requester, the object that is being protected, the policy and environment information. To evaluate the request, the ACM uses a policy decision point (PDP), a policy enforcement point (PEP) to enforce the decision and a context handler to manage attribute values retrieving to make a decision. Figure 2.2 [8] gives an overview of ACM scenario.

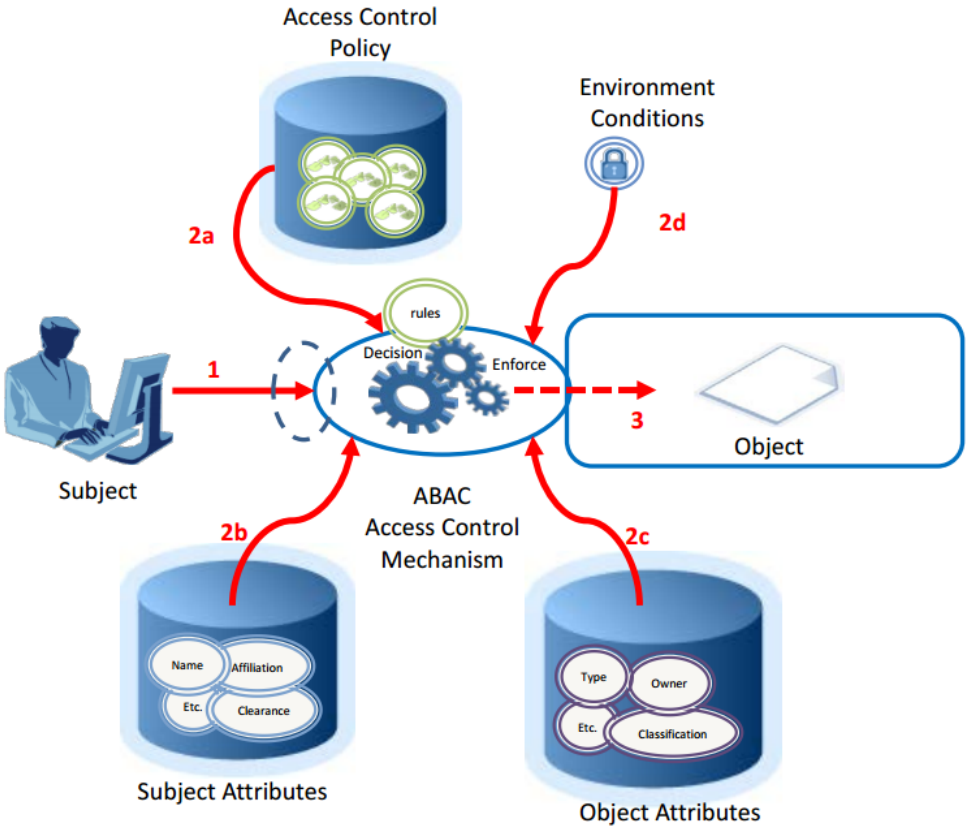

Figure 2.2: ABAC Access Control Mechanism basic scenario

1. A subject makes a request to access a protected object.
2. The received request is evaluated by ACM against:
   (a) the specified policies
   (b) the requester attributes
   (c) the protected object attributes
   (d) the environment condition (e.g., time, location)
3. the requester is granted access if ACM has authorized that

ABAC is a general model of logical access control. In the next Section, we introduce a model which is usually used to implement ABAC ACM.

2.3 eXtensible Access Control Markup Language (XACML)

Extensible Access Control Markup Language (XACML) is a standard attribute based access control that exists since 2003 [18]. XACML consists of a policy language and an infrastructure which takes care of the evaluation and enforcement of the policies. The policy language, implemented in XML, specifies who is permitted to perform an action on a secured object. The policy contains specified rule(s) to control access of the objects. The effect of these rules is a permit or deny result. An example of a XACML policy is given in listing 2.1 [9].

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <Policy PolicyId="urn:org:herasaf:xacml:example:policy:policy1"
3  <div id="_mcePaste">RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-
       combining-algorithm:permit-overrides"></div>
4  <Target>
5  <Subjects>
6  <Subject>
7  <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
8  <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">user1</
       AttributeValue>
9  <SubjectAttributeDesignator
10 DataType="http://www.w3.org/2001/XMLSchema#string" AttributeId="name" />
11 </SubjectMatch>
12 </Subject>
13 </Subjects>
14 <Resources>
15 <Resource>
16 <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
17 <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">task1</
       AttributeValue>
18 <ResourceAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string"
        AttributeId="resource-id" />
19 </ResourceMatch>
20 </Resource>
21 </Resources>
22 <Actions>
23 <Action>
24 <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
25 <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">execute</
       AttributeValue>
26 <ActionAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string"
       AttributeId="action-id" />
27 </ActionMatch>
28 </Action>
29 </Action>
30 <Environments />
31 </Target>
32 <Rule RuleId="urn:org:herasaf:xacml:example:rule:rule1"
33 Effect="Permit">
34 </Rule>
35 </Policy>
```

Listing 2.1: Example XACML Policy

In general the XACML infrastructure consists of four main (colored) components depicted in Figure 2.3 [18]. These four components work together to provide an authorization decision to a requester:

- Policy Administration Point (PAP) is responsible for deploying policies in the policy repository
- Policy Decision Point (PDP) is responsible for the evaluation of the request against the attribute values
- Policy Information Point (PIP) is responsible for the retrieving of attributes to the PDP
- Policy Enforcement Point (PEP) is the front end application that receives the requests and enforces the PDP decision

The following steps shows how XACMl processes a request as depicted in Figure 2.3:

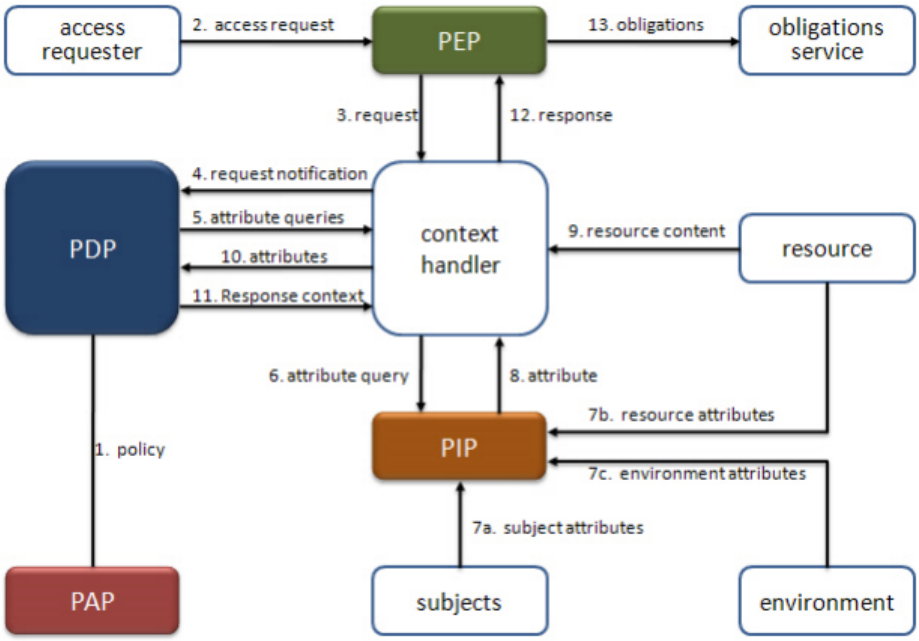

Figure 2.3: XACML Architecture

1. The security designer first defines a policy and he/she deploys it in the policy repository by using the PAP
2. The requester (subject) makes a request to the PEP to access a secured object
3. The PEP forwards the request to the context handler
4. The context handler transforms the request to a XACML request and forwards it to the PDP
5. The PDP queries attributes from the context handler in order to evaluate the request
6. The context handler in his turn queries the attributes from the PIP
7. The PIP returns the attributes of the subject, object and the environment
8. The context handler obtains the attributes
9. The context handler includes the resource in the context, but this is an option
10. The context handler returns the attributes to the PDP
11. The PDP evaluates the request and sends the decision to the context handler
12. The context handler sends the PDP decision to the PEP
13. The PEP enforces the decision by executing any relevant obligation

In the above process description, the context handler is responsible for the communication between the components. The obligation service is responsible for executing the obligations. These obligations depend on the implementation. The XACML process described above is not maintained by every XACML engine. Every XACML engine has its own process for evaluating requests. It can be that the PEP and the PDP are integrated as one component, or in some cases the context handler is removed from the framework. However, the general idea of XACML is inherited by all the existing XACML engines.

2.4 Aspect Oriented programming (AOP)

AOP is a way of programming which lets programmers concentrate on the cross-cutting modules, also called concerns. To achieve that purpose it uses cross-cutting concern as a mechanism, which is also known as separation of concern. The basic idea of cross-cutting in AOP is adding a functionality in the main program without changing the main program code. An example of such functionality is security, logging, etc. Adding these functionalities as a separate concern prevents tangling of code with the main program. For this purpose, AOP uses the unit module *aspect* to implement a concern that crosscut multiple modules of the main program. The *aspect* is woven in the main program by using an AOP language that reads the aspect code and generates the final main program.

There exists many AOP languages, such as AspectJ, Spring AOP, JBoss AOP, etc. Which one to use depends on the application development requirement. However, AspectJ is one of the most advanced AOP language. For our security monitor implementation we will be using AspectJ. In the next Section, we will explain the AOP terminology by using an AspectJ example.

2.4.1 AspectJ features

AspectJ is an AOP extension created for the Java programming language. A Java class is called in AspectJ an aspect which can have the same extension as the Java class. When an aspect is compiled in a AspectJ compiler a class file is produced which complies with Java byte code of the JVM (Java Virtual Machine). So a valid Java class is also a valid AspectJ aspect. This has the advantage that programming in Java, AspectJ benefits the Java tooling. Therefore, AspectJ makes it easy for Java programmers to use it for AOP goals. More important, AspectJ is compatible with Java extension such that:

- AspectJ program code runs on standard JVM (Java Virtual Machine).
- AspectJ can be used in integrated development environments such as eclipse. In addition, it can also be used with existing tools such as maven and spring framework AOP.
- Programming AspectJ programs is just like programming in Java.

AspectJ aspect syntax can be written in different ways. For our security monitor, the annotation method is used to write the syntax aspect code. Although, the different methods to write the syntax are almost similar, the annotation methods is preferred for the this thesis. Since AspectJ is an extensive and advanced AOP implementation, it is out of the scope for this thesis to explain AspectJ in detail such as the difference between the syntax methods. For the interested reader, the book "AspectJ in Action" [13] gives an excellent detail view about AspectJ and AOP.

For simplicity, first we introduce the AOP concepts and then we use an AspectJ example to explain them:

Target : This is an object that is being *advised* by an aspect. A target can be a Java class or a method of a class.

Join Point : A join point is a point of a program. For example, a join point can be the execution of a method in Java language, parameter of a method, etc.

Pointcut : This is a query which selects a set of join points, where non-functional code must be executed.

Advice : This is the actual non-functional code executed by the aspect. The advice is executed at particular joint point in the main program. In AOP the advice is referenced to a pointcut that defines at which join point in the program the advice code must be executed. The advice is similar to the methods of Java and is only executed when triggered by the pointcut.

Listing 2.2 presents an example of a Java class *Account* which contains sensitive information stored in the variable *salary*. The information stored in the variable *salary* can be retrieved by calling the method getSalary(). The security designer has decided that when ever this method is called, a notification should be made.

```
class Account{
    double salary; //security sensitive information
    ....
    @Check
  public int getSalary(){
    return this.salary
  }
}
```

Listing 2.2: Example of an AOP target

Listing 2.3 presents an example of an AspectJ aspect. The aspect notifies every call made to the Java object *Account*.

```
@Aspect
public class Notify{
    @Around("@annotation(Check)")
  public Object notification(ProceedingJoinPoint point){  //start of the advice
  System.out.println("The method "+point.getSignature().getName()+" is being called
      ");
  point.proceed();
  System.out.println("The method "+point.getSignature().getName()+" has been called
      ");
  } //end of the advice
}
```

Listing 2.3: Example of an Aspect

The aspect `Notify` (see listing 2.3) looks as a regular Java class. With the annotation @Aspect the aspect is created as an AspectJ aspect. In this aspect, the pointcut @annotation(Check) specifies where exactly the Notify code will be executed. In listing 2.2 this is marked as the join point with annotation @Check. The pointcut will intercept the method getSalary() whenever called. This pointcut identifies the method `public int getSalary()` as the executing method which must be intercepted.

The advice identified by the method `notification(ProceedingJoinPoint point)` is the aspect code executed at a specific point in the program specified by the pointcut and the join point. The advice, as showed in listing 2.3, is associated with the pointcut @annotation(Check). The advice uses the argument type `ProceedingJoinPoint` to support the *Around* advice. ProceedingJoinPoint supports many methods such as `proceed()`. For more information see [13]. The advice will run before the execution of the method `getSalary()` if the join point is matched by the pattern defined by the pointcut. The advice prints out two notification messages. The first one specifies the method name intercepted by the pointcut. Afterwards, the advice lets the method `getSalary()` continue its execution by calling the method `proceed()`. The second message prints out that the method has been executed.

The weaving process is a mechanism for integrating the aspects into the main program [4], just as the aspect `Notify` with the Java class `Account`. A weaver integrates the aspect code of the advice at the right place defined by the pointcut and the join point.

2.5 Related work

In this Section, we provide an overview of the related work in the area of monitoring and enforcing authorization constraints applied on security sensitive workflows.

An algebra (SoDA) is provided in [15] to specify *Separation of Duty* (SoD) constraints. SoD aims to prevent fraud and error by requiring that associated security sensitive tasks should be executed by multiple users. The algebra uses the concept of terms to express SoD constraints, and it requires that the set of users who are performing the task must satisfy the specified term. However, it handles only SoD constraints. Constraints such as Binding of Duty (BoD) are not supported. An advantage of the proposed algebra it is not bounded to a specific workflow model. Unfortunately, it is not shown how to map the specification to a workflow configuration.

Basin et al in [3] describe how they have adapted SoDA to their purposes. They managed to transform a SoDA specification into a process algebra CSP model. It represents a so called SoD-secure workflow which is a combination of formal models of workflows, RBAC and SoD constraints. The last ones are specified in process algebra CSP for two reasons: 1) CSP supports tracking traces which fits in their goal to describe task executions of a workflow. 2) CSP supports parallel synchronization which allows Basin et al to decompose their implementation environment into loosely coupled components. This requires minimal interaction between the business process expert and security designer. The produced CSP model serves as the concept of their implementation based on *Software as a Service* [2]. It is a new approach to enforce SoD constraints on workflows, introduced as *Sod as a Service*. It enables loose coupling between the components of which their implementation exists of. The proposed implementation supports quick adaptation of constraints and the dynamic business environment such as technological changes.

In [6] Clara Bertolissi et al. proposed an automated synthesized runtime monitor to enforce constraints in workflows. The runtime monitor is capable of enforcing constraints while the workflow still can terminate successfully. They have managed to solve the *Workflow Satisfiability Problem* (WSP) which consists of checking whether there exists a user to task assignment such that: 1) the workflow can successfully terminate, 2) while all the constraints can be satisfied. However, their runtime monitor is capable only to enforce two constraints: Separation of Duty and Binding of Duty. Their approach consist of an offline phase and an online phase. In the offline phase, the execution of the tasks is specified by using a transition system symbols with the corresponded constraints of the tasks. Then this specification is supplied to a model checker. This will generate all possible execution traces of the tasks. In that way, all possible terminating traces which satisfy the constraints are explored. In the online phase, the terminating traces with the related constraints are transformed into a Datalog program. This program represents the monitor. The monitor is implemented in a Datalog engine. The engine is capable of answering queries whether a user is able to execute a task without violating the WSP. Besides the fact that the monitor is only able to enforce SoD/BoD constraints, calculating the terminating traces is only limited to a number of tasks. The model checker can handle only five tasks at a time, if the model checker gets more than five tasks supplied it will consume a lot of time to produce a terminating trace.

Crampton et al. [10] go much deeper into solving WSP in workflows. They propose to use a propositional LTL to model a workflow, and in that way deciding whether there exists a workflow instance that adheres to a workflow specification without violating the imposed constraints. Their methodology is not bounded to a particular type of workflow. Their methodology can compute a set of solutions with minimal users as possible to execute the tasks of a workflow. Unfortunately, they do not have an enforcement mechanism of how to enforce the constraints. Their work is more on abstract level.

Slim Kallel [11] propose a framework to enforce authorization constraints that is most related to our work. They follow more the traditional method of the runtime monitor concept to enforce constraints. They propose a three phase approach to specify and enforce constraint in workflows. The

first phase is the high-level specification of constraints for which they have developed a complex specification language based on LTL and Z notation called TemporalZ. The language integrates LTL in the Z framework. However, their specification language can specify more constraints than just SoD or BoD. As an illustration, they have specified RBAC access control and SoD/BoD in their framework. Once these constraints have been specified they are supplied to the theorem prover Z-EVES. These theorem prover will detect whether there exists any contradiction in the specified constraints, and this process is performed in the second phase. In the last phase, the specified constraints are transformed into enforcement aspects of the AOP language ALPHA. The transformation is performed automatically which prevents human errors. The process start with specifying RBAC and the constraints by using TemporalZ. This is performed by the security designer for which he/she has to learn the TemporalZ language. The lack of usability of this language makes it hard for the security designers to easily catch the language. Then, the programmer develops the main application using Java which will execute the workflow process. The implementation of this application must conform to the constraints specification.

Ayoub et al. [1] propose an approach to enforce constraints at task execution. In contrast to other works, they let the business process designer to specify the access control policies (ACP) instead of the security designer. They integrated the design of a workflow and constraints specification in one approach. At design time, the business process expert can impose ACP on tasks. For this purpose, they use a so called business rule activity that is available in BPMN specification. This is an activity (e.g., task) in which the business rules are specified that can either evaluate to permit or deny. When a secured task is about to be executed, first the imposed activity is executed which will evaluate the request for task execution. If the rule defined in the activity is evaluated to permit then the task is allowed to be executed, otherwise it is denied. The policies are enforced by a business rule engine implemented in XACML. However, we believe that security properties should be threated as a separate concern. This has the advantage that a business expert can concentrate on the actual workflow design. And the security designer can specify constraints independent of the workflow pattern.

We believe that the design of a workflow is the responsibility of a business process designer, and he/she should not get involved in the security constraints development process. In this context, a workflow is first designed in a design modeling tool by a business process designer. Then it is up to the security designer to decide on which tasks the constraint should be imposed. The designed workflow is then executed by a workflow execution engine. Unlike to [5] and [1], we believe that constraints should be enforced independent of the workflow design-time, the executing engine or a workflow application. Business designer should focus on the design of a workflow and security designers on the enforcement of constraints, which requires minimal communication between them. Unlike to [6], the generating process of the monitor must be independent of the workflow type or how many number of tasks the workflow must have. In addition, the security designer must have the capability to specify constraints which can be used for real world workflows. [11] conforms most to our work as described above. However, their specification process is difficult to use for the security designer. This is because their specification language inherits abstract mathematical formalization which are hard to understand. Furthermore, it requires some effort to learn that language to specify constraints. Our findings are also confirmed in their conclusion section, and in their future work they will provide a framework that is more usable for security designers. In addition, the workflow execution application must be developed conform the specified constraints. Hence, business process designer must keep in mind which constraint are imposed during design time.

For this purpose, the security monitor in thesis is written in separate concern independent of the business logic (e.g. workflow engine) and integrated in a modular way by means to enforce the constraints. This separation of concerns has many advantages. It enables the loose coupling between the executing engine that executes the workflow, specification of constrains and the enforcement mechanism. Furthermore, our specification notation does not inherit any difficult mathematical formalization which make it usable for security designers. Moreover, it is not limited

to specify only SoD constraints.

Chapter 3

Specifying Authorization Constraints for Workflows

In this Chapter an approach is presented for specifying authorization constraints which are enforced by the security monitor in a workflow. This approach introduces an expressive syntax to capture real world security properties of workflows.

The first stage of the security monitor is the formal specification of the constraints. In this stage the security designer should formally specify constraints based on workflow security requirements. The literature overview presented in Chapter 2 showed different solutions to build security monitors that enforce authorization constraints. These works considered only a limited number of constraints imposed on the workflow tasks. In addition, some solutions were only applicable to a specific type of workflows. They cannot be applied to other workflows. There is a lack of expressivity for specifying authorization constraints which captures real world security properties. This is because some of them use a limited formal language to specify constraints. Therefore, it is important to have an expressive syntax to specify authorization constraints for real world security properties.

In Section 3.1 an mortgage provisioning example is presented as a workflow scenario that is used to explain the specification of the constraints in the introduced specification approach. Then, some example constraints are presented which are widely accepted as powerful constraints. We will use these constraints to explain our formal syntax notation.

In Sections 3.2 and 3.3, the concept of formal specification of constraints is introduced. First, we describe an attribute-based model adapted to our purposes. The components in this model are used to express the workflow entities. Then, we explain how to use this model by mapping the mortgage provisioning workflow to its components. Finally, the formal approach is introduced how to specify authorization constraints by using our simple and yet expressive notation.

3.1 Scenario: Mortgage provisioning process

In this Section, a workflow scenario is presented that is used to demonstrate the approach to specify authorization constraints. The scenario is concerned with how mortgage requests of clients in a bank named X are handled and is depicted in Figure 3.1. The process of the request is modeled in the BPMN modeling tool. This scenario is used throughout this thesis. Bank X has some strict rules when granting mortgage to customers. The costumers can apply for a mortgage request via online graphical user interface. The customers have to log in by providing their

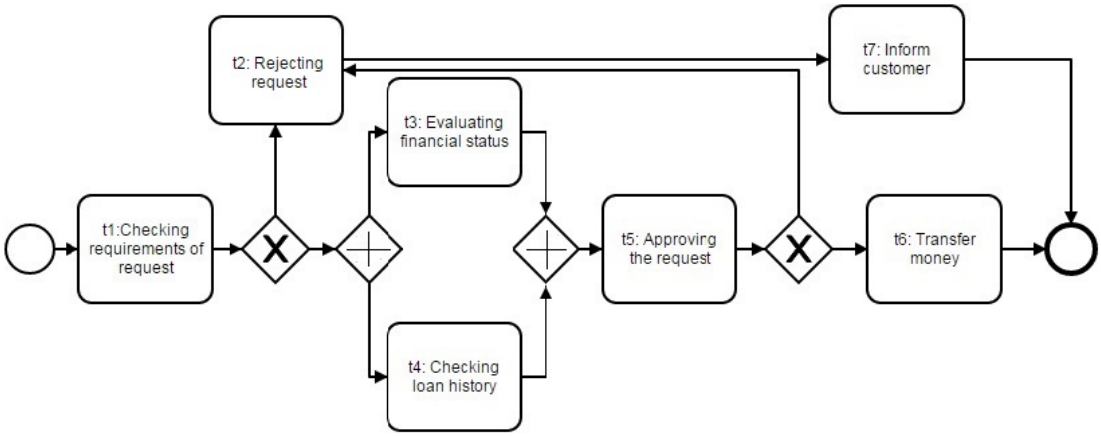

Figure 3.1: Mortgage provisioning request workflow modeled in BPMN

credentials. Once the customers are logged in they must fill in an online form to start the request for a mortgage. Bank X requires that the form must be completely filled in and customers must attach some required personal information (salary, work contract, living address, etc). Once the form is completely filled in and the customers clicks send, the workflow will be started to handle the request.

In task *t1* an employee checks whether the customer has fulfilled the request requirements. After task *t1* is finished, the workflow can go two directions. It either continues with task *t2* or it follows the path to the other tasks. The workflow forces the token to follow only one path of the two when task *t1* is finished. The choice to follow only one path is enforced by the cross sign showed in Figure 3.1. Which path to take is based on the decision made in task *t1*. If the employee in task *t1* decides that the customer does not fulfill all the requirements, then the request will be rejected. Rejecting requests is processed in task *t2*. Furthermore, if the request is rejected in *t1*, the bank's policy stipulates that task *t2* must be executed by an employee who has a senior level compared to the employee who has executed task *t1*. Afterwards, the customer must be informed why his request is rejected. this is performed in task *t7*. Informing the customer can be done by letter, e-mail, telephone or any other communication method used by the bank X. This bank has the restriction that task *t7* must be performed by the same employee who has processed the task prior task *t7*. So the same employee who executed *t2* must execute task *t7* and he/she must explain why the request is rejected.

However, if the employee in task *t1* decides that all the requirements have been fulfilled, then the request is further processed in tasks *t3* and *t4*. These tasks are executed in parallel, which is visualized by the plus sign shown in the workflow. Bank X has strict rules concerning the financial status of customers. For example, a customer must have a working contract for at least one year, the salary must adhere to some basic income, the customer must have an official living address, etc. All this information is checked by an employee in task *t3*. Furthermore, it is important for the bank that the customer does not have an unacceptable loan and mortgage history by other banks. An employee can gather and check this information in task *t4*. In addition, the employee can also check whether the customer still has some current loans/mortgage by other banks. Since task *t4* processes sensitive information, it has the restrictions that the task can only be executed by an employee who has more than two years of work experience. In task *t5* an employee can decide whether to accept or reject the request. A decision can only be made by an employee if all the required information is gathered in tasks *t3* and *t4*. Therefore, task *t5* can only be executed if the tasks *t3* and *t4* are successfully finished. This requirement is enforced by the plus sign which is followed directly by task *t5*. If for some reason the request is rejected, the workflow continues with task *t2*. Finally, if the request is granted, the money can be transferred to the customer's

account number. Transforming money is processed in task *t6* by an employee of the department transaction management. In addition, task *t6* has the restriction that it must not be executed by the same employee who has executed task *t5*.

3.1.1 Example authorization constraints

We introduce a small number of authorization constraints which are enforced by the security monitor. These constraints are representing our dynamic constraints which can only be evaluated during the run of the workflow. With these constraints it is shown how effortless it is to specify and enforce constraints in the proposed syntax notation. The introduced examples are selectively chosen which are covered in most research works. We will consider below three basic types of history-dependent authorization constraints which are adopted by most business processes [10]:

Dynamic Separation of Duty (DSoD)
The primary goal of DSoD is preventing fraud in workflows. This goal is achieved by dissipating the execution of tasks to multiple authorized users for a workflow instance. DSoD ensures that within a workflow instance, a user who has executed a former task cannot execute a later task related to the former task. DSoD has proven itself to be very useful in workflows. In this thesis Dynamic Separation of Duty is considered as the history-dependent constraint.

Example. Consider the mortgage workflow depicted in Figure 3.1. Bank X has the restriction that the tasks *t5* and *t6* must be executed by two different users. This is a DSoD authorization constraint, since task *t6* can only be executed after checking the identity of the user who has executed task *t5*. If the user who wants to execute task *t6* is the same user who has executed task *t5*, then the user is prohibited from executing task *t6*.

Dynamic Binding of Duty (DBoD)
DBoD has the same goal as DSoD, but the approach is different. DBoD does the opposite of DSoD. It makes sure that some tasks must be executed by the same user.

Example. Bank X has also the restriction that the tasks *t2* and *t7* must be executed by the same user. This is a DBoD authorization constraint, since task *t7* can only be executed after checking checking the identity of the user who has executed task *t2*. If the user who wants to execute task *t7* is the same user who has executed task *t2*, then that user is allowed to execute task *t7*.

Seniority constraint (SC)
This constraint restricts users with a specific organization level from executing a task. If a user has a specific level then he may or may not execute some tasks. This constraint specifies a task must be executed by a user who has a senior level compared to the user who executed the previous task.

Example. Bank X of the mortgage example uses cautionary measures when requests are rejected after task *t1*. Therefore, it enforces the restriction that task *t2* must be executed by a user who has a senior level than the user who has executed task *t1*. The constraint is imposed on task *t2*.

The specified authorization constraints are applied on tasks. However, if the constraint involves more than one task, on which task should the constraint be applied? To illustrate, consider the mortgage example where the user who executes task *t6* must not be the same user who has executed task *t5*. This constraint is applied on task *t6*, since that task is executed later in time. Therefore, it is always assumed that the constraints are applied on the tasks which are being executed later in time. Referencing back to the mortgage scenario, the restrictions applied on the tasks are mapped to the above described authorizations constraints. Bank X has the constraint

that task *t2* can only be executed by a user who has a senior level than the user who has executed task *t1*; this is actually the seniority constraint described above. Finally, the DBoD constraint is imposed on task *t7*.

3.2 Attribute-based model for workflows

We present a model based on attributes tailored to workflows. This model describes the components of a workflow: tasks, users and environment. In this model tasks are executed by users.

The model consists of three entities: subjects, objects and environment. Informally, subjects are associated with the users who are performing the tasks of a workflow. The objects are associated with the tasks contained in a workflow. All these entities are expressed in a set of attributes. Basically, these set of attributes describe the users and tasks properties such as *name*, *age*, etc. Therefore, attributes are the core components of the model.

Base on the attribute values, a decision can be made to either allow or deny a task execution. The decision is based on a boolean statement by comparing the attribute values. For example, user.age = 23 where user is an entity and age is the attribute of the user. With this flexibility of attributes we can describe any property of the entities. More importantly, we can specify constraints for real world workflow scenarios.

The value of an attribute can be atomic or set-valued. For example, a user can work for different departments of an organization, and a department attribute of a user can hold multiple departments. Furthermore, attributes are defined to be functions which map an entity to an atomic or a set of values. These values which are returned by the function attribute, are necessary for the security monitor to make a decision. Some of these attributes are dynamic and others are static. A dynamic attribute value changes during the run of a workflow. As for a static attribute its value does not change during the run of a workflow.

Based on the above description, the components of the model are presented in Table 3.1. U is the finite set of existing users and UA is the set of attributes which describe the properties of users of U. Every attribute in UA maps a given user to a specific value in the domain of that attribute. T is the finite set of tasks contained in a workflow and TA is the set of attributes which describe the properties of tasks in T. Every attribute in TA maps a given task to a specific value in the domain of that attribute.

As shown in Table 3.1 the function *dom* is a range function which takes an attribute as input, and it returns the values from an attribute domain. For simplicity, it is assumed that all returned values of the attributes are sets even if it is an atomic value. An attribute function can also have an empty set $\emptyset$ as a value. This can be the case especially for dynamic attributes. So when the function *dom* is applied on an attribute with an empty set, it will return the value $\emptyset$ of empty set.

In the following Section, the mortgage workflow example of bank X is expressed in the model components.

3.2.1 Mapping mortgage workflow to the model

The mortgage provisioning request workflow depicted in Figure 3.1, is used as an illustration to show how effortless it is to use the model components. The workflow contains seven tasks all executed in a particular order. The set of tasks is T = {t1, t2, t3, t4, t5, t6 ,t7}. For simplicity, there are also seven users who will execute these seven tasks. So the set of users is U ={u1, u2, u3, u4, u5, u6, u7}. Next, the attribute sets UA and TA are extended with attributes which describe the properties of users and tasks. Which attributes to use depends on the workflow process. First,

Component	Specification	Description
	SETS	
Users	U	Set of users
Tasks	T	Set of tasks of which the workflow consists of
Workflow	E	Set of workflow components
Subject Properties	*UA*	set of the User attributes
Object Properties	*TA*	set of the Task attributes
Workflow Properties	*EA*	set of the Workflow attributes
	ATTRIBUTE FUNCTIONS	
Range function	dom : att $\in$ UA $\cup$ TA $\cup$ EA $\rightarrow$ Domain value of att	function that returns an attribute value
User att funct	$\forall$ ua $\in$ UA: U $\rightarrow 2^{dom(ua)}$	Returns attribute value of a user
Task att funct	$\forall$ ta $\in$ TA: T $\rightarrow 2^{dom(ut)}$	Returns attribute value of a task
Workflow att funct	$\forall$ ea $\in$ EA: E $\rightarrow 2^{dom(ea)}$	Returns attribute value of a workflow

Table 3.1: Basic components of the model

every user needs to be uniquely identified, so the attribute *name* has that purpose. Since the function attribute *name* is a property of the user, then $name \in UA$. So the mapping of this attribute is:

$$\text{name} : \text{U} \rightarrow 2^{dom(name)}$$

Although the returned value of the function attribute *name* is a set, it still contains only one value. A user can have only one unique name. Furthermore, bank X requires that employees must have some number of years of work experience before they are allowed to execute a particular task. More precisely, task *t4* of the mortgage workflow can only be executed by a user who has more than two years of work experience. So the attribute $workexperience \in UA$ stores the users number of years they have worked for bank X. The formal mapping of this attribute is:

$$\text{workexperience} : \text{U} \rightarrow 2^{dom(workexperience)}$$

Furthermore, bank X also specifies levels on the users. A user can have for example CEO level or just junior level. In the workflow process only users with a specific level can execute some tasks. The levels are expressed in numbers. For example, a junior level has the number 1 and a CEO has number 5. These levels are stored in the attribute $level \in UA$. So the formal mapping of this attribute is:

$$\text{level} : \text{U} \rightarrow 2^{dom(level)}$$

The attribute set of users is now $UA = \{name, workexperience, level\}$. The properties of tasks are different from those of users. The set of attributes $TA = \{name, requester, executed\}$ contains three attributes with the following formal mapping:

$$\text{name} : \text{T} \rightarrow 2^{dom(name)}$$

$$\text{requester} : \text{T} \rightarrow 2^{U}$$

executed : T $\rightarrow 2^U$

The attribute *name* uniquely identifies the task, and has the same rules applied to it as the users attribute *name*. The attribute *requester* contains the user who is assigned to execute the task. The attribute *executed* keeps track of who has executed the task. This is a dynamic attribute since the value can only be filled in during the run of workflow.

Both attributes *requester* and *executed* contain users as value which are entities. As described above, entities of U and T consists of set of attributes UA and TA.

The requester's name of task t is stored in *t.requester.name*. Hence, the access of attributes is comparable with a tree structure as depicted in Figure 3.2. It visualizes the attribute set TA of

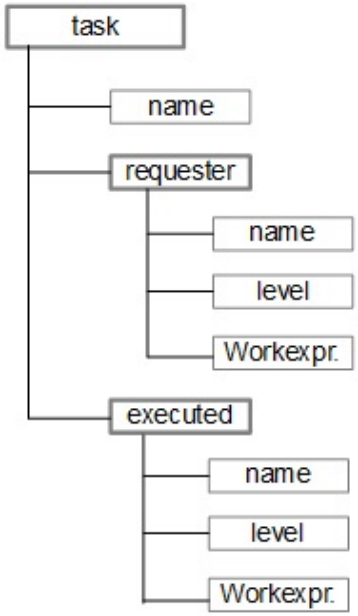

Figure 3.2: Task expressed as a set of attributes

the task entity. Note that the attributes *requester* and *executed* contain users entity. Therefore, the attributes {name, level, workexperience} $\in UA$ are branched under the attributes *requester* and *executed*.

Until now the basic core components of the model are described. The most important component was the attributes function. They take an entity (user, task, etc) as input, and return an atomic value or a set value of the attribute domain. The mortgage workflow of bank X was used as an example to express the workflow in the model components. In there it was shown how effortless it was to use the components to express the mortgage workflow. In the next Section, the formalization of constraints is described and applied on the mortgage example.

3.3 Specification of authorization constraints

In this Section, the formal specification of the authorization constraints is introduced. In our presented literature overview in Chapter 2, we showed that existing languages are to weak to specify real word constraints. Some of these languages have a big lack of usability. We solve these problems by introducing a simple notation, yet still very expressive to specify constraints which can be used in real world workflow scenarios.

The syntax notation is based on basic predicate logic and mathematical operators. Our syntax exploits the components of our model described in previous Section to specify constraints. The constraints specifies a condition whether a user is allowed or not to execute a task. Base on the attribute values, the constraint can be evaluated either to true or false, i.e. allow or deny.

3.3.1 Syntax operators to specify authorization constraints

Mathematical operators (e.g. $=, <, >$) have a defined semantics that are well known. These operators return either false or true after validation. For example, $9 > 8$ returns true while $8 > 9$ returns false. And because they are simple to use, they are used to evaluate conditions of the constraints. A condition can be user.age > 18, where age is an attribute of the entity user. So they are used to compare attribute values and they can only be applied on attribute values. They can not be applied on entities. This raises a problem because constraints are imposed on tasks which are entities. In order to use mathematical operators to specify the constraints, a *syntax operator* is introduced as in the way mathematical operators are used when evaluating conditions. Unlike to the mathematical operators, the syntax operator takes only tasks as input. Definition 3.1 represents the formal definition of the syntax operators to define authorization constraints.

$$OP_{n,att_m} = (t_1 \times .. \times t_n \times 2^{dom(att_1)} \times .. \times 2^{dom(att_m)}) \rightarrow \{true, false\} \tag{3.1}$$

More importantly, the syntax operators are representing the type of the authorization constraints as shown in Table 3.2.

Authorization constraint	Operator	Description
DSoD	t $\neq_{executed.name,requester.name}$ t'	user who is performing the task t' must be different from the user who executed task t
DBoD	t $=_{executed.name,requester.name}$ t'	user who is performing the task t' must be the same user who executed task t
SC	t $\succ_{requester.level,executed.level}$ t'	user who executes task t must have a senior level than the user who has executed previous task t'

Table 3.2: Syntax operators to specify the authorization constraints

The syntax operator OP can be represented by any mathematical operator (e.g. $>, <, =$) depending on the purpose of the authorization constraint. However, the semantics of the mathematical operators do not hold for the syntax operators. The semantics of the syntax operators are actually defined by the security designer and are discussed in Section 3.3.2. The syntax operator of definition 3.1 contains two parameters n and att_m which clearly distinguish the syntax operator from the mathematical operator.

The parameter n $\in N$ is the number of tasks an operator OP can have as input. These input tasks are the tasks on which the authorization constraint will be imposed. The parameter att_m is a set of attributes and $att_m \in TA \cup UA \cup EA$. The syntax operator uses the values of these set of attributes $AV_m = \{2^{dom(att_1)}, .., 2^{dom(att_m)}\}$ for evaluation. According to definition 3.1 a constraint will always be imposed on tasks. Thus a syntax operator has two forms: 1) OP_{n,att_m}, in this case the constraint is applied on the relation between the tasks $\{t_1, .., t_n\}$. For example, the constraint can be a Dynamic Separation of Duty that involves a number of tasks. 2) $\text{OP}_{n,dom(att)}$ where att $\in att_m$, in this case the constraint is applied on a specific task which needs to be evaluated on a specific attribute value. Finally, the syntax operator evaluates either to true or

false. The syntax operator represents the syntax of the authorization constraints. That is, when a security designer wants to specify constraints on certain tasks he/she uses the syntax operator in the way described here above in accordance with definition 3.1.

Example. We present an example on how to define the constraint DSoD by defining a syntax operator according to definition 3.1. The concept of DSoD specifies that two tasks must be executed by two different users and not by the same user. The syntax operator $\neq_{executed.name,requester.name}$ which represents DSoD authorization constraint as described in Table 3.2, takes two tasks as input t and t'. The syntax operator uses two attributes for evaluation. The first attribute function *requester* $\in$ *TA* of the task t' contains the user who is actually executing the current task. The second attribute function *executed* $\in$ *TA* of the task t contains the user who has executed the task. Both attributes returns a subject entity, which is in this case the user(s). These returned users must be different in order to evaluate the constraint to true. By comparing the values of the users attribute name, which contains their unique identifier, we can see whether these users are the same or not. Hence, the formal definition of the syntax operator DSoD is $\neq_{executed.name,requester.name} \in OP_{2,executed,requester}$ according to definition 3.1. That is, the syntax operator is applied on two tasks and it evaluates the constraint on the attributes $\{executed.name, requester.name\}$. This constraint evaluates to true if and only if the attributes *name* are not equal. To compare the values of the attribute *name*, we use the mathematical operators. This process is described in the next Section.

Referencing back to the mortgage example, the authorization constraints described in Section 3.1.1 are depicted in Figure 3.3 and are represented with the dotted lines. The authorization constraints are specified by using the syntax operators encapsulated in Table 3.2.

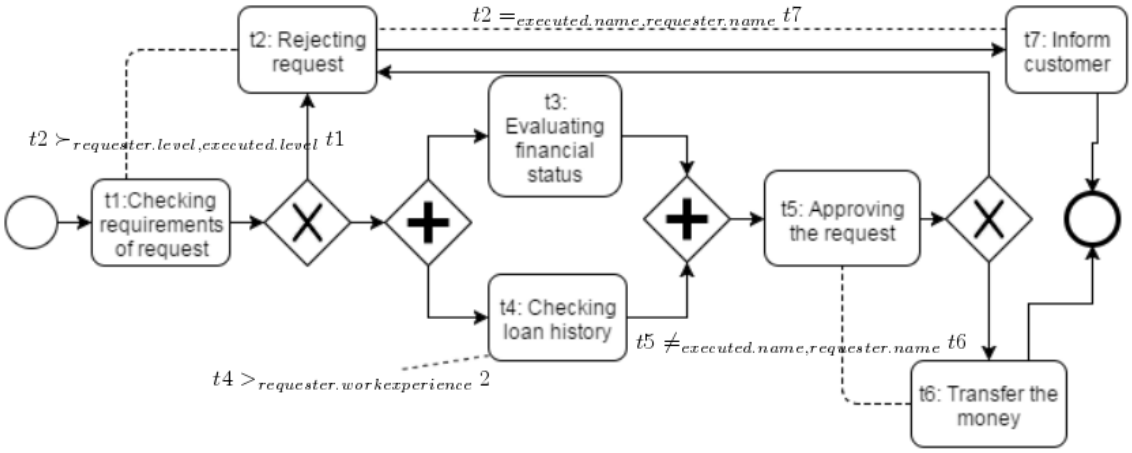

Figure 3.3: Authorization constraints of the mortgage workflow

3.3.2 Semantics to define syntax operators

The formal definition of an authorization constraint represents the semantics of a syntax operator. They are expressed in first order logic to reason about a specific constraint condition. This condition can evaluate to true or false, depending on the condition. For instance, a condition can be that certain tasks or users must adhere to a specific attribute value by using mathematical operators. So this definition process is a combination of first order logic, function attributes and mathematical operators.

The constraint DSoD in Table 3.2 which is applied on the task *t'* since that task is executed later in time then the task *t*. The user who is assigned to execute task *t'* is stored in the function

attribute *requester*. In order to retrieve this user, the task *t'* is passed as a parameter to the function attribute: *requester(t')*. And the user who has executed task *t*, is stored in the function attribute *executed* which contains an atomic value. If they contain different users then the DSoD will evaluate to a true statement. The semantics for this inequality is formally defined as follows:

$$t \neq_{executed.name,requester.name} t' \iff \forall u \in executed(t), u' \in requester(t') : u \neq_{name} u'$$

The right hand side of the bi-implication compares whether the returned users *u* and *u'* are equal or not. If they are not equal then the right hand side evaluates to true. The syntax operator will only evaluate to true if and only if the right hand side evaluates to true.

To check whether the users *u* and *u'* are not the same the operator $\neq_{name}$ is used. This operator is not a syntax operator even though its syntax is the same as the syntax operator, since it takes users and not tasks as input. It is also not a mathematical operator, since it takes users as input which are subjects and not attributes. In this thesis, these operators are called sub syntax operators. The name syntax operator comes from the fact that they also take entities such as subjects as input for evaluation, while syntax operator takes only the entity task as input. The name sub comes from the fact they are never used in specifying the constraints, only syntax operators are used for specifying the constraints. Thus, sub syntax operators are only used for the semantics and not for the syntactic use. The sub syntax operator uses the function attribute $name \in UA$ for evaluation. The formal semantics of the sub syntax operator $\neq_{name}$ is given as follows:

- $u \neq_{name} u \iff (\forall n \in name(u), n' \in name(u') : n \neq n')$

The names *n* and *n'* are the unique names of the users. To compare them, the mathematical operator $\neq$ is used. The condition $n \neq n'$ can only evaluate to true if *n* and *n'* are not equal. Hence, the DSoD operator is semantically defined as follows:

$$t \neq_{executed.name,requester.name} t'$$
$$\iff \forall u \in executed(t), u' \in requester(t') : u \neq_{name} u'$$
$$u \neq_{name} u'$$
$$\iff (\forall n \in name(u), n' \in name(u') : n \neq n')$$

The DSoD constraint is applied on the task *t6* of the mortgage workflow depicted in Figure 3.3. Task *t6* is substituted for *t'*, and *t5* for *t*. The semantics of DSoD operator are defined as a tree structure. The top of the tree contains the syntax operator, the node contains the sub syntax operator and leaf contains the actual mathematical operator which evaluates the condition. The main advantage of our notation is that we only use the syntax operator to specify the constraints as we did for the mortgage workflow. The semantics description of the other authorization constraints depicted in Figure 3.3 are presented in appendix A.

Until now, only the authorization constraints encapsulated in Table 3.2 were discussed in this Chapter. These powerful authorization constraints were used to present the expressivity and usability of the presented syntax operator. But with the presented approach, it must be obvious that with the syntax operator it is possible to express other constraints as well. For instance, it is also possible to express that a user can execute a task if he/she has minimal years of workexperience. This constraint is imposed on task *t4* of the mortgage workflow as depicted in Figure 3.3. The semantic of this constraint can be found in appendix A.

The semantics used to define the syntax operator is simple and practical to use. In the semantics, the universal quantifier is used to predicate about the members of an attribute. This raises a

problem when we want to predicate about only one member. For example, suppose now that in the mortgage workflow we want to specify the constraint that a task can only be executed by a user who works for a specific department. However, users can work for multiply departments. The idea now is to check whether in the set of departments of the user, there exists the department that is required by the task restriction. Following the described approach on how to specify the constraints, first the attribute set *UA* is extended with the attribute *department*. Then the syntax operator to specify the above described constraint is defined as follows:

$$OP_{n,att_m} = (t_1 \times 2^{dom(department)}) \rightarrow \{true, false\} \tag{3.2}$$

Where the syntax operator $\doteq_{requester.department} \in OP_{1,dom(department)}$ is specified according to definition 3.2.

The semantics of the syntax operator is given as follows:

$$t \doteq_{requester.department} D$$

$$\iff \begin{cases} true & \text{if } \exists u \in requester(t) : \exists d' \in department(u), d \in D : d' = d \\ false & otherwise \end{cases}$$

So the constraint states that users who want to execute the task t must be working for the department contained in the atomic-set D. However, this is a rare case and therefore for most cases we will be using syntax operators according to definition 3.1. But just to show that it is also possible to express this type of constraints, the above constraint example about departments is given where existential quantification is used.

Chapter 4

Implementation

This Chapter describes how the introduced framework in chapter 3 is implemented. This framework is a unity of XACML and AOP. For this purpose, XACML is well suited for the introduced framework because it can represent the introduced model described in Section 3.2. In literature, XACML is the most referenced implementation for attribute based access control models. XACML provides a standard mechanism to specify policies based on attributes. This policy is evaluated by a XACML evaluation engine. There are a number of XACML engines such as HerasAF XACML engine [9] and Sun XACML [19]. In this thesis, HerasAF is used to implement the XACML engine that evaluates the (XACML) authorization policies.

We address the security monitor that enforces our authorization constraints described in previous chapter. More precisely, we introduce an approach of runtime evaluation of constraints on top of the authorization policies. They ensure that only authorized users can execute certain tasks of the workflow at runtime. This is described as the second stage of the security monitor. In the runtime enforcement stage, we build the monitor in the aspect-oriented language AspectJ to encode the enforcement of constraints.

4.1 Architecture of the framework

To prevent unauthorized tasks executions, policies and constraints are specified and enforced by our presented framework depicted in Figure 4.1. The framework consists of a workflow engine, XACML access control mechanism, a front-end application and a security monitor. The blue colored components are our developed parts, and the gray colored components are black-boxes which will not be addressed in this thesis. When a user wants to execute a task, he/she makes a request to the front-end application. The security monitor will intercept this request and decide whether the task execution is allowed or not based on the specified constraints. Then, the request is forwarded to XACML engine which contains the decision mechanism to allow or deny the user executing the task based on the specified policies. Finally, the PEP will return either a allow or deny decision.

4.1.1 Workflow engine

The workflow engine is the component that executes the workflow. There are many engines that can be used to execute a workflow. Our framework is not attached to a specific engine, the developers can use any engine they want. Developers can attach their engine to our framework

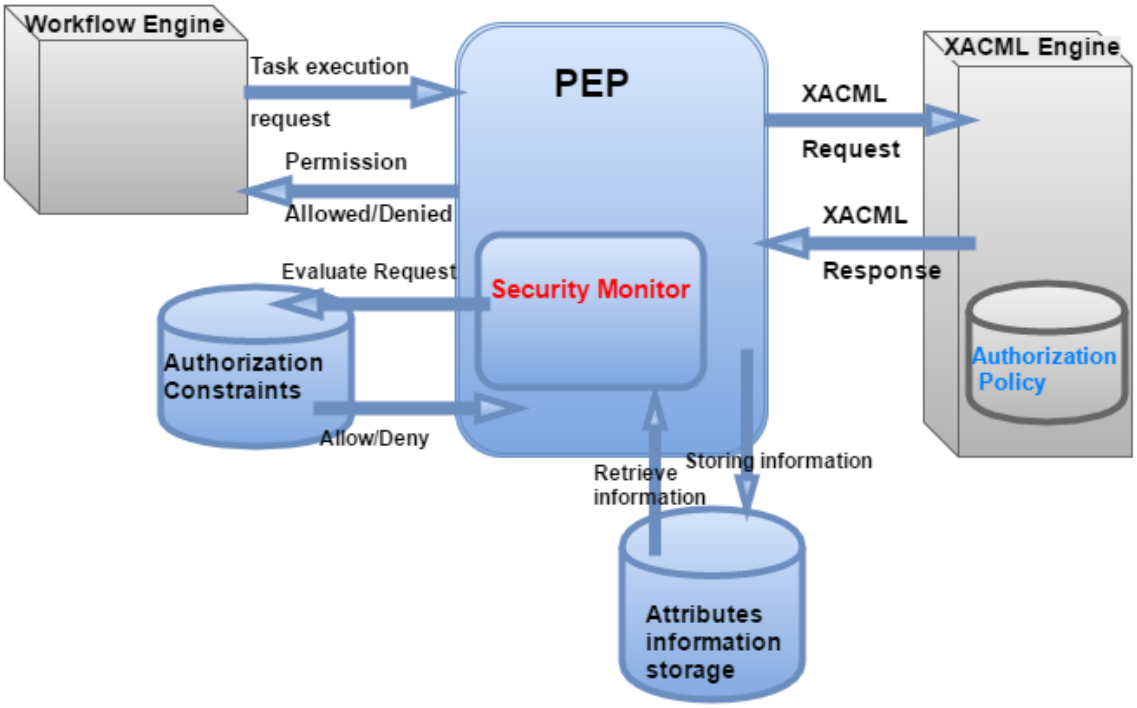

Figure 4.1: Architecture of the framework

by simply making an instantiation of the PEP. Developers can leverage any workflow engine with minimal effort.

4.1.2 HerasAF XACML engine

HerasAF XACML is a Java open source XACML v2.0 engine implementation. The main advantage of HerasAF is that it contains a factory to create a simple PDP which takes care of the initialization of all the required components. The XACML Policy Decision Point in HerasAF is represented by the *simplePDP* component. This component is responsible for the evaluation of requests against the authorization policies. For this purpose, policies are specified in XACML policy and then depolyed in the Herasaf Policy Repository. After this deployment, the engine is ready to receive and evaluate requests. This thesis will not go into detail about the XACML engine and its mechanism, it is only used to evaluate requests against the authorization policies sent by the policy enforcement point.

4.1.3 Specifying authorization policies

Authorization policy is classified as static constraint. The authorization policies are transformed to a XACML policy which later on are deployed in the policy repository for evaluation. So when a task execution request comes in, the PDP will evaluate this request against the stored authorization policy. XACML provides a standard mechanism to specify policies [18]. For this reason, in this thesis we do not address this specification.

4.2 Authorization Policy Enforcement

As depicted in Figure 4.1, *Policy enforcement point* (PEP) is the *front-end application* developed in this thesis for the framework. PEP is the responsible component for the enforcement of the authorizations. The process of this enforcement is described below:

1. The process starts with specifying the policies in an XACML policy which is deployed in the policy repository of the XACML engine. After the security designer has specified the authorization constraints they are uploaded in our framework.
2. The PEP receives a request from the workflow when a task is going to be executed.
3. This request is converted to a XACML request by the PEP and passed to the XACML Policy Decision Point (PDP).
4. The PDP evaluates the request against specified policy; the response is composed of XACML release decisions.
5. Based on the PDP response and security monitor, the PEP enforces the decision.

The specified constraints, which we describe as the dynamic constraints are evaluated by the security monitor. This process is conducted between the second and the third step. Thus, before the request is sent to the PDP. The evaluation process of the authorization constraints is described in Section 4.3.

4.2.1 Request evaluation and decision

Once the XACML request is received, the PDP identifies this request by matching it against the specified policies. Then, the PDP evaluates the request against the matched policies to achieve a decision. The decision of the PDP is either Allow, Deny or Not Applicable. The last decision is returned if no matched policies are identified. Our framework will permit the task execution if the decision is either Allow or Not applicable.

The framework depicted in Figure 4.1 shows that the security monitor is integrated in the PEP. The monitor will intercept every request made to the PEP. However, only the tasks on which constraints are imposed will be evaluated, the others will be skipped. The decision of the monitor is either Allow or Deny. If the monitor allows the task execution, then the request is forwarded to the PDP for further evaluation. Otherwise, the final decision will be Deny.

The PEP will only permit the task execution if both the monitor and the PDP allow the task execution. If one of them denies the task execution then the task execution will be denied. And this decision is returned by the PEP.

The security monitor is described in details in Section 4.3

4.3 Authorization Constraints Enforcement Mechanism

This Section is dedicated to the security monitor, which is our enforcement mechanism based on runtime monitor [7, 14] concept. The monitor is responsible for enforcing authorization constraints on security-sensitive tasks at runtime. First, we discuss the basic runtime monitor concept in general and then we introduce the expected concept of our security monitor wrt "runtime monitor". Finally, we introduce a program instrumentor tool which can implement our security monitor concept in an efficient way.

In general, a runtime monitor goal is to continuously monitor an application behavior. The purpose of this monitoring is to detect any violation of the specified properties to which the application should adhere. The approach consists of two main stages. First, the properties are expressed in a formal language as it is described in details in Chapter 3. Second, from the formal specification the monitors are automatically synthesized and integrated in an application to enforce the desired specified properties of the application. The generating process is the mapping from the formal specification (e.g., authorization constraints) to a specific machine code. For example, datalog, Aspect Oriented Programming (AOP), etc. An important property of runtime monitor is the automated enforcement of the desired properties. That is, if a violation is detected, the runtime monitor will enforce certain decisions to satisfy the desired goal of the specified properties. Runtime monitoring has proven to be a reliable and a widely accepted approach to enforce security properties. The monitor in this thesis is developed with the purpose to monitor and enforce authorization constraints at task executions of a workflow. Therefore, the term security monitor is used instead of the general term runtime monitor. Nevertheless, the concept is still the same. In this Chapter we will focus more on the second stage which is the enforcement of the authorization constraints adapted to our security monitor.

The monitor bases its decision on the historical execution of tasks. In literature [7, 14], this is known as a trace which keeps track of the information related to the execution of tasks, for instance, keeping track of who has executed which task. *Locations* [7, 14] are the relevant places where the code is inserted in order to capture the task executions and to evaluate them against the constraints. These places can be determined to be either manual or automatic, depending on the business logic. Manual means the code is inserted manually in relevant places of the business logic. We avoid this tangling of code, since we want to have a loose coupling of components. For our monitor, we choose that these places are detected automatically. Furthermore, the security monitor must automatically enforce the constraints at runtime as it is the traditional method of "runtime monitors". The automatic enforcement consists of checking the current state of the task execution and the specified constraint conditions.

The enforcement mechanism is implemented by using Aspect Oriented Programming (AOP) [12]. AOP is a programming paradigm which supports other paradigms such as object oriented programming. It supports the modularization of concerns which cross cuts the business logic of an application. The key point of AOP is the modularity unit called *aspect*. The aspect is a separate concern that contains a non-functional code such as security properties that cross cuts all modules of a business logic. After specification the aspect is woven in the business logic. This separation of concerns makes our loosely coupling of components achievable. This has the advantage that the monitor is developed separately and can be later integrated in our framework. The pointcuts defined in the aspect which matches a set of join points, are identified as the *locations*. A join point is a point in a program where some code is being executed (e.g., task execution). At this point which is identified by the pointcut, an advice will be executed. An advice is a piece of AOP code that can influence the execution of a program.

In this context, the security monitor is implemented in the AOP AspectJ tool. AspectJ is an aspect oriented extension to Java for practical usage. The key objective of this tool is the simplicity of implementing aspects in a modular way. Aspects are compiled into Java byte code. This simplifies the integration of the security monitor into the framework which is a Java environment. AspectJ is a powerful AOP tool and it is well documented. For this reason, the monitor is implemented in this tool.

4.3.1 Security monitor requirements

The security monitor monitors a security sensitive workflow which represents a business process. This workflow can have a large number of tasks that needs to be executed by different users. In addition, all these tasks executions must adhere to specified authorization constraints. This

means that the security monitor must be able to monitor a large number of tasks. Furthermore, the monitor must be able to evaluate every task request with minimal overhead. Another factor that is taken in consideration is the development environment. That means, the chosen approach must lead to a security monitor that is implementable in the framework with minimum configuration for the security developer. Therefore, the security monitor has the following requirements:

- *Complete mediation*: The security monitor must be able to monitor every task execution of the workflow.
- *Economy of Mechanism*: The security monitor must evaluate only the tasks on which authorization constraints are applied on. Any other tasks must not be evaluated.
- *Feasibility*: The security monitor must be implementable with less configuration required for the security developer.

In Section 4.4.3 it is shown that our security monitor adheres to the above mentioned requirements.

4.4 Implementation of security monitor

In this Section, we describe how the security monitor should automatically enforce constraints at runtime when a request is sent to the PEP. This process constitutes of intercepting the request of an executing task, then evaluating it against the specified constraint. After evaluation, the monitor decides whether the task is allowed to be executed or not. For this purpose, AspectJ aspect is proposed as a solution to evaluate and enforce constraints at runtime.

First, an explanation is given about how the monitor should intercept tasks executions and what the goals are that we wanted to achieve for this monitor. Then, we will show that due to implementation limitation, we had to adapt our goal. In other words, we adapted the implementation approach of the monitor. The concept of the monitor is explained with an AspectJ monitoring idea throughout this Section.

4.4.1 Security monitor goal

Figure 4.2 is used as an example to explain the idea of the security monitor goal for the presented framework. The security designer specifies for each constrained task an authorization constraint. Then this specification is transformed in an AspectJ aspect that will function as the security monitor for that task. For every constraint a separate aspect is created. Each aspect contains a defined pointcut. A pointcut matches one or more join points. A join point marks an executing point in the workflow, in our case this is the task execution. Furthermore, the aspect layer advice contains the code that is being executed at a join point when it is matched by the pointcut. In our case, this code will be the specified constraint. Thus, the specified constraints are embedded in the advice layer.

Figure 4.2 depicts three aspects. Each of them contains an authorization constraint specified for a constrained task. *Aspect1* is an aspect which contains an authorization constraint specified for task *t2* of the mortgage workflow example. The other two aspects contain authorization constraints defined for other tasks. Every time a task makes a request to the framework only the aspect which holds the constraint for the requesting task must evaluate it. Thus, the tasks are marked as the *join points*. And the pointcut in the aspect is defined such that it must match this join point. In Figure 4.2 user *u2* is requesting whether he/she is allowed to execute task *t2*. Since aspect *Aspect1* is a constraint defined for *t2*, the security monitor invokes *Aspect1* after intercepting the request. The vertical beam indicates that *Aspect1* is invoked by the security monitor to evaluate

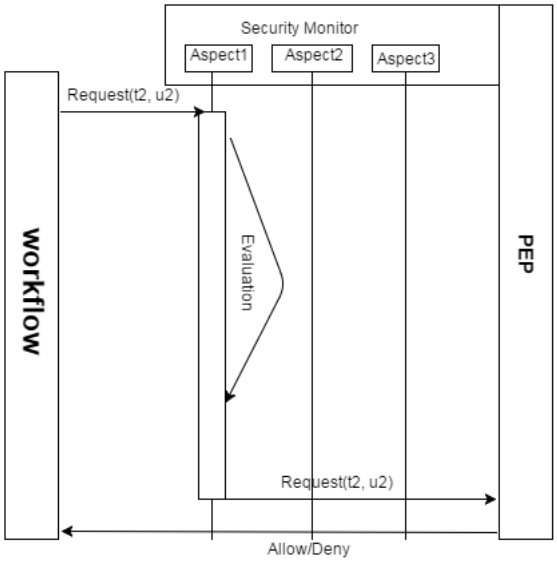

Figure 4.2: Security monitor sequence diagram

the request for task *t2*. During this process the request is not allowed to continue its flow to the PEP until the security monitor decides to. At the end of the evaluation the request continue its flow to the PEP. When the request was intercepted by the security monitor, *Aspect2* and *Aspect3* were not invoked during the evaluation process. This approach makes the security monitor really flexible since only tasks with authorization constraints are intercepted by the associated aspects. This approach makes the constraints manageable. The intercepting mechanism introduced in this approach makes the goals of the security monitor achievable. That is, all constrained tasks are intercepted by the security monitor and evaluated by the associated aspects. This is how we wanted to implement our security monitor in thesis. For this purpose, the interception mechanism plays a major role. However, this mechanism is limited to the borders of the presented architecture, which are the blue sectors (see Figure 4.1). That means we cannot mark the tasks as join points. As a consequence, this has limitation on the implementation of the security monitor for which we had to adapt our intended approach to implement the monitor. In the next Section we explain how we have solved the interception problem of tasks at runtime and how we have achieved a feasible monitor that is implemented in our framework.

4.4.2 Intercepting tasks at runtime

We present an analysis work of how security-sensitive tasks should be intercepted, which is an important process of the monitor. The key objective of this Section is about the implementation of the interception mechanism. The analysis work is conducted in the range of AspectJ cross-cutting feature and the introduced framework presented in Section 4.1. This work reveals where, when and how to intercept the security-sensitive tasks at runtime. The result of this work is a feasible monitoring approach for the security monitor that is described in Section 4.4.3. Basically, we wanted to implement the monitor as described in the previous Section. For that, we present three different implementation approaches. But only one approach will be used that supports our monitor implementation described in Section 4.4.3.

In the first approach we wanted to create for every specified constraint a separate aspect. The aspect will automatically intercept the task execution for which it was created for. After the interception, it will perform the evaluation of the constraint. But this approach requires that the workflow engine must be involved in the constructing mechanism. Unfortunately, the work-

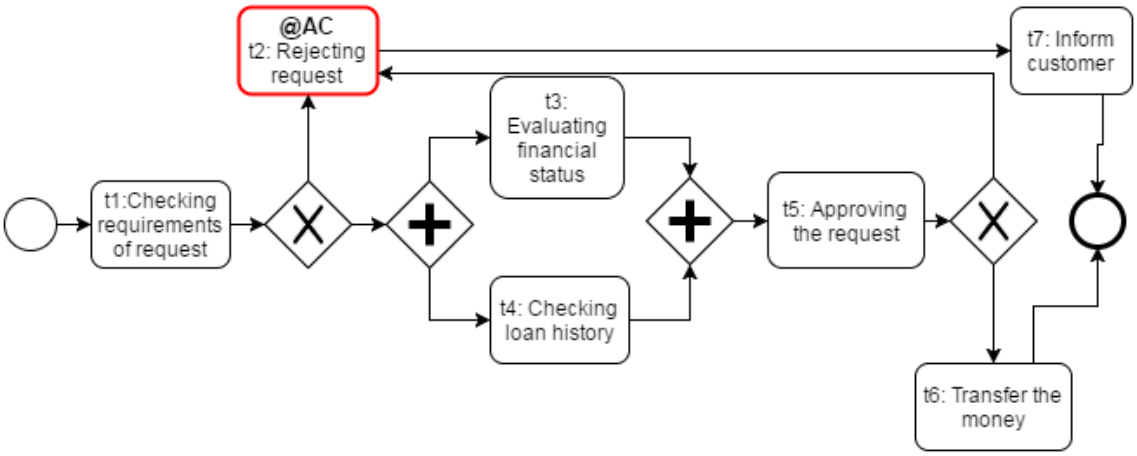

Figure 4.3: Using annotated marker to define a join point

flow engine is out of the blue border of the framework. The second approach does not require involvement of the workflow engine while it has the same achievement and functionality as the first approach. The difference is that the aspects will intercept every incoming request sent to the framework, even if the aspect is not applicable for the this request. Thus, the problem with this approach is that every created aspect will try to evaluate every task execution. Therefore, this approach can cause excessive overhead which can make the framework become very slow. Finally, the third approach will intercept every request, however it evaluates only the requests for which constraints are imposed. The last approach seems to be the best method to construct the security monitor.

First approach. The implementation of the monitor starts with the introduced idea in Section 4.4.1 and depicted in Figure 4.2. The aspect layer that is responsible for interception is the pointcut layer. This pointcut intercepts join points when a match is found. AspectJ annotation marker is used to annotate these join points. In order to invoke only the aspect linked to a specific task, that task needs to function as the join point. To illustrate, consider the example depicted in Figure 4.3 where an authorization constraint is applied on task *t2*. Aspect1 is the created aspect that contains the specified constraints for task *t2* as shown in Listing 4.1. To monitor task *t2* it is configured as a join point by annotating it with the marked annotation @AC. In order to let the monitor invoke only the aspect created for task *t2* the pointcut must only match the annotation @AC as shown in Listing 4.1.

```
@Aspect
public class aspect1{
....
Around("@annotation(AC)")
public Object Constraint(ProceedingJoinPoint pjp){
//defined constraint for task t2
}
}
```

Listing 4.1: Defining a pointcut

Although the described approach above is achievable with AspectJ, it cannot be implemented as security monitor for the framework. The reason for that are the join points, which are the tasks of the workflow. The workflow is a business process that is being executed by a workflow engine. Unfortunately, the workflow engine in the framework is a blackbox that cannot be monitored by the security monitor. So it is impossible to annotate the tasks as join points. Hence, the proposed approach to build the security monitor is not applicable for the presented framework. Therefore, we adapt this approach and try to overcome the mention problems.

Second approach. In this approach we adapted the first approach to overcome the mentioned problem in there. The only component of the framework that our monitor can monitor is the PEP. An important functionality of the PEP is receiving and handling of the tasks requests. The PEP contains the method `Request()` to process all these requests. Since the security monitor needs to intercept the requests it is obvious that the *Request* method should be marked as the join point as shown in Listing 4.2. This also means that the PEP is configured to be the *target* of the security monitor.

```
@Intercept
public void Request(){
//code for handling the request
}
```

Listing 4.2: Defining a join point

The first approach had many different join points which were the tasks. Each pointcut of the aspects had its own matching pattern to match a specific annotation defined for the task that needs to be evaluated only by that aspect. Unlike the first approach, this second approach has only one join point: the method *Request* shown in Listing 4.2. The pointcut of every created aspect must match the marked annotation @INTERCEPT. As a consequence, all the aspects are having the same pointcut definition. If a request comes in, then it must go threw all the created aspects. That also includes tasks requests on which no authorization constraints are applied. The problem with this approach is how to invoke the aspect only associated with the incoming request. Ideally this would be if the interception can be based on the parameters values. The pointcut in the aspect could be defined such that it is must match a certain value of the parameter *task* in Listing 4.2. For example consider Figure 4.2, where *Aspect1* is created for task *t2*. The pointcut in *Aspect1* can be defined such that it must match the value *t2* of the parameter *task*. In that way the request sent from task *t2* is intercepted and evaluated only by *Aspect1*. In AspectJ it is possible to annotate the parameters of a method as a join point. Unfortunately, AspectJ pointcut definition can only match the data type of the parameters. So the interception will be based on the parameters data type and not on the parameters value. Thus, it is not possible to solve the problem mentioned above.

A proposed solution would be to implement a check mechanism in the aspects. When an aspect is invoked, it needs to check whether the incoming request is associated with it. To illustrate, referencing back to the example depicted in Figure 4.2. First, *Aspect1* intercepts the incoming request. Then *Aspect1* checks whether it needs to evaluate the request, which is the case since the aspect is created for task *t2*. The only problem is that the aspects *Aspect2* and *Aspect3* will also intercept the request after it has been evaluated and released by *Aspect1*. In addition, the two other aspects will also check whether they need to evaluate the request or not. However, this solution is not very efficient. To illustrate, suppose the workflow consists of a hundred tasks and for every task there is an aspect created. Then the security monitor will perform 10.000(100 tasks x 100 aspects) of checks when the workflow finishes. Thus, This second approach is not suitable for the framework. In the next Section, a more suitable approach is presented to implement the security monitor.

4.4.3 Feasible approach to intercept tasks at runtime

In the previous Section we have described two approaches to implement our monitor. However, due to implementation limitation they were not applicable for our framework. Based on these approaches we present a feasible approach on how to implement the monitor.

To solve the problem of the approaches described in the previous Section, there will be only one aspect created for all the authorization constraints which represents the *security monitor*. The method *Request* is marked as the join point, and the pointcut in the aspect is defined to match this

join point. The advice layer that enforces our constraints must now be able to enforce multiple constraints.

However, if there are abundance of tasks then the security monitor needs to monitor all these tasks. But the security monitor is only concerned about the tasks on which the authorization constraints are imposed. In the worst case, the monitor can intercept requests for which no constraints are applied as shown in the previous Section. This is not an appropriate way of intercepting tasks executions and in fact it can lead to an unnecessary overhead to the framework.

To solve this problem, we foresee the aspect of a checking mechanism. In this mechanism the aspect checks whether it needs to evaluate an intercepted request or not. If not, the PEP continues processing the request. If the aspect does need to evaluate the request then it will evaluate the request against the defined authorization constraints.

In addition, with this approach all the requirements mentioned in Section 4.3 of the security monitor are achieved. Complete mediation: our monitor is capable of intercepting every request for evaluation. Economy of mechanism: our monitor evaluates only tasks on which constraints are imposed on. Feasibility: since the join point and the pointcut are already defined, the monitor is automatically integrated in the PEP. No human intervention is required. This last approach is used to implement the security monitor of the framework.

The aspect used in this approach is transformed to the actual security monitor, which is the developed enforcement aspect that incorporates the generated formal specified authorization constraints. For each request, our monitor is evaluating and enforcing distinguishable applicable constraints related to the request.

4.5 Evaluation and enforcement of constraints

This process starts with specifying constraints imposed on tasks of the workflow that need to be secured according to organizational policy. The tasks which do not need to be secured are left out. After the specification, the constraints are automatically generated and stored in the authorization constraints repository as depicted in Figure 4.1. It is an automated process but only for the four constraints that we have specified and used in the mortgage workflow.

The pointcut defined in the aspect of our security monitor intercepts task requests at execution and forwards them to the *around advice*. This advice is the aspect layer that is responsible for checking and enforcing the authorization constraints at runtime. This advice first checks whether a constraint is imposed on the task. If this is the case, then the advice evaluates and enforces the related constraint. Otherwise, the evaluation process will be skipped. This *checking mechanism* will prevent overhead as described in Section 4.4.3. The related constraints codes are instantiated from the authorization repository for evaluation and enforcement. If the constraints evaluate to **true**, then the task is allowed to be executed by the requesting user. Otherwise, it is prohibited from execution. Hence, our monitor is executing different constraints at each request interception.

The evaluation process consists of comparing the current state of the attributes values of the request against those specified in the constraints. After the evaluation process, the request will continue its flow to the PEP so that it can be forwarded to the PDP. The interception of requests is performed automatically.

The monitor is enforcing history dependent constraints, which means that the evaluation process of these constraints is task history dependent. In more detail, the monitor needs to know who has executed previous tasks in order to evaluate the current task execution. Keeping track of this information is stored in the *Attributes information storage* depicted in Figure 4.1.

4.5.1 Deploying specified constraints in repository

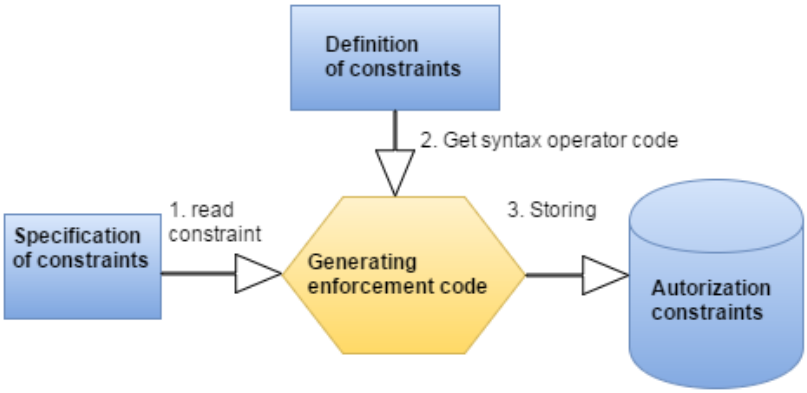

Figure 4.4: Generating enforcement code

In Figure 4.4 the definition of constraints are the semantics of the constraints that we have defined in Chapter 3. These are coded manually. Once the specified constraints are uploaded in our framework, our generator will produce enforcement code based on the defined semantics. So the mapping is done automatically for all the specified constraints. In our mapping we do not have to consider the pointcut or the joint point. These are already specified (see Section 4.5). This eases the mapping of the constraints. All this code is placed in the method **evaluate**. This is a method which contains the actual enforcement constraint code. The method is a standard method of the interface *Constraint* that we have defined. This interface is written in Java. For every constraint, an instance of the *Constraint* interface will be created. Finally, once the mapping is completed and put in the method evaluate of the interface *Constraint* it can be stored in the repository. The method *put* of the repository expects the task name as a key and the value is the creating of a new interface *Constraint* with the overwritten method evaluate. The evaluate method is executed by the *around advice* embedded in our security monitor. Which code to execute, depends on the intercepted request.

After the specification, actual enforcement code is generated and deployed in the authorization repository. See Figure 4.4 of this process. However, this process is limited to the only four constraints that we have expressed in Chapter 3.

4.5.2 Constraints repository

One of the important functionality of the security monitor is executing the appropriate constraint. Since a workflow can contain an abundance of tasks, it is of great importance that the executing process must be very efficient. Therefore, the constraints are stored in a *HashMap* data structure with the associated task as depicted in Figure 4.5.

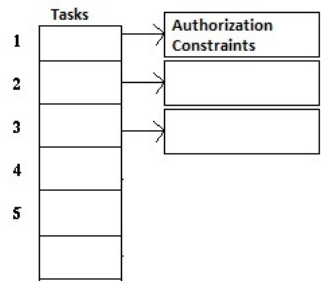

Figure 4.5: Repository containing the constraint with associated tasks

HashMap is a simple and practical data structure to store objects in an efficient way. This HashMap is the collection or repository of all the existing constraints imposed on the tasks. The tasks are used as keys, and the constraints are the related values of those keys. The advantage of a HashMap is that the aspect advice can check whether a constraint is imposed on a task or not very efficiently. First, it receives the parameters task and the executing user. The task is used as the key to look for the related constraints. Second, the constraint code will be executed for evaluation. If there is no constraint imposed on the task, then no code will be executed at all. Therefore, only the tasks on which the authorization constraints are applied are evaluated. All the other tasks are skipped. HashMaps are usually used to store a large number of records, which is suitable for a workflow with a large number of tasks.

Chapter 5

Validation

In this Chapter we validate our security monitor. Validation is the process to evaluate the security monitor by checking whether it fulfills the condition. This condition is enforcing constraints at runtime which are imposed on tasks of a workflow. For this purpose, we present two different use cases of the mortgage workflow introduced in Section 3.1 and modeled in Section 3.2. In each use case, we show the enforcement of constraints by the monitor, and demonstrate the evaluation of the constraints in accordance with expected results.

5.1 Use case: No violation of constraints

In this use case, we consider the following trace {t1,t3,t4,t5,t6} of the mortgage workflow. The considered constraints imposed on these tasks are depicted in Figure 3.3. For this use case, we show that there is no violation of the constraints. The tasks of the workflow will be successfully executed. For simplicity, we have seven users that are capable of executing seven tasks. Table

User/Task	t1	t2	t3	t4	t5	t6	t7
u1	X	-	-	-	-	-	-
u2	-	X	-	-	-	-	X
u3	-	-	X	-	-	-	-
u4	-	-	-	X	-	-	-
u5	-	-	-	-	X	-	-
u6	-	-	-	-	-	X	-
u7	-	-	-	-	-	-	-

Table 5.1: User task assignment of the mortgage workflow

5.1 shows which user is requesting a task execution. The entry X means that the user in that row is requesting permission to execute that task of that column. The users of the workflow are employees of bank X which are executing the tasks. Furthermore, there are no polices applied on the workflow. So the PDP evaluates every request as Not applicable. Which means all the requests are allowed by the PDP.

Via the security monitor GUI menu, the constraints are uploaded to the framework as depicted in Figure 5.1. Specified policies are also uploaded via the GUI menu.

The mortgage workflow starts when a mortgage request is submitted by a customer. Once the mortgage is received, employee *u1* will check whether the customer has fulfilled all the requirements

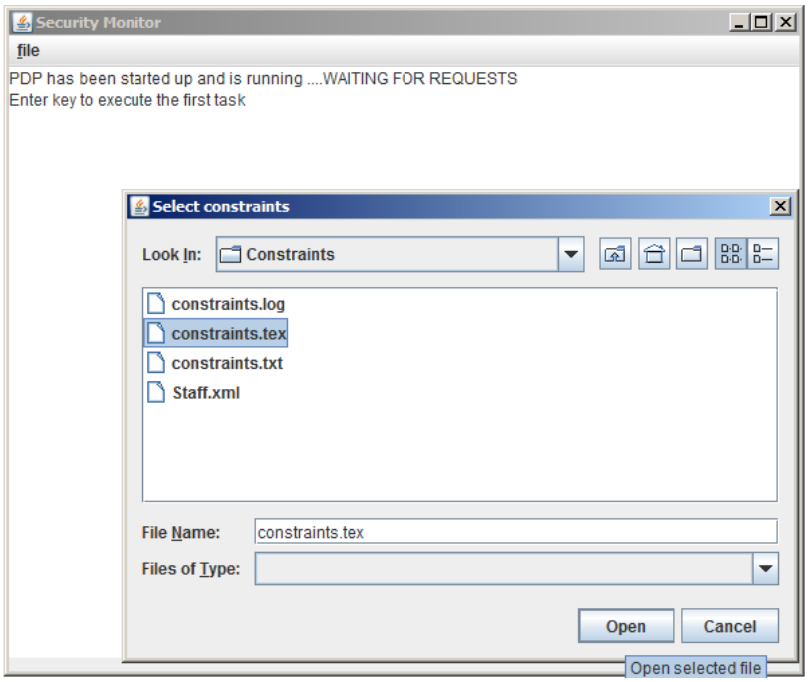

Figure 5.1: GUI of the security monitor

User	Work experience (years)
u1	1
u2	4
u3	6
u4	5
u5	9
u6	3
u7	3

Table 5.2: Users years work experience of the mortgage workflow

concerning mortgage. Before *u1* can execute task *t1*, a *request* is sent to the framework. The framework checks whether *u1* is allowed to execute task *t1*. The monitor intercepts this request and checks whether any constraint is imposed on the task. Since this is not the case, the request is forwarded by the PEP to the PDP for a policy evaluation. The PDP returns allowed as a decision. The attribute *executed* of task *t1* is updated with the value *u1* which means that user *u1* has executed task *t1*.

We assume that the employee has detected no issues and the workflow continues its flow to task *t3*. The workflow engine makes a request whether *u3* is allowed to execute the task. This request will only be evaluated by the PDP, since there is no constraint imposed on it. As expected the PDP allows *u3* to execute task *t3* and the attribute *executed* is also updated. Employee *u3* has checked the customers history and he/she has found that the required administration of the customer is sufficient for the mortgage. However, employee *u4* still needs to check whether the customer has any ongoing mortgage/loans at other banks. This checking is performed in task *t4*. As depicted in Figure 3.3, a constraint is imposed on this task. The constraint states that this task can only be executed by an employee who has more than two years of work experience. The security monitor intercepts the request and evaluates the constraint imposed on this task. For that, the monitor checks the value of attribute *workexperience*. Fortunately, the attribute *workexperience*

of employee *u4* contains the value *5* as given in Table 5.2. Thus, this constraint is evaluated to *true* as expected. Employee *u4* is allowed to execute task *t4*. The attribute *executed* of task *t4* is also updated. Now that both tasks are successfully executed the workflow continues with task *t5*.

Based on the gathered information in tasks *t3* and *t4*, employee *u5* decides whether the mortgage is granted or rejected in task *t5*. There are no constraints specified for task *t5*. Which means that *u5* is allowed to execute task *t5* and also here the attribute *executed* of the task is updated. Thus, employee *u5* decides that the mortgage is granted to the customer.

Transferring the money to the customer is performed in task *t6* by employee *u6*. However, DSoD constraint is imposed on this task. The monitor evaluates the constraint imposed on this task. For this purpose, the monitor checks whether the value of *executed* of task *t5* is not equal to the *requester* which is *u6*. Since, task *t5* is executed by *u5*, the constraints evaluates to true as expected. After updating *executed* of task *t6* the workflow ends.

As expected, the monitor intercepts every *request* and checks whether there is a constraint imposed on the task. The evaluation of the request is skipped if there is no constraint imposed on the task (e.g., *t3*). Otherwise, the request will be evaluated against the specified constraint, as we have seen for the tasks *t4* and *t6*. The constraints are evaluated by the monitor by monitoring the attributes values of the executing task and requesting employee. Base on these attribute values, a decision is made in accordance with the specified constraint as we have seen here above. After the evaluation, the requests are forwarded by the PEP to the PDP for policy evaluation. Hence, our security monitor works inline with the access control mechanism. Furthermore, the dynamic attributes are updated after the request evaluation. The result is showed in Figure 5.2.

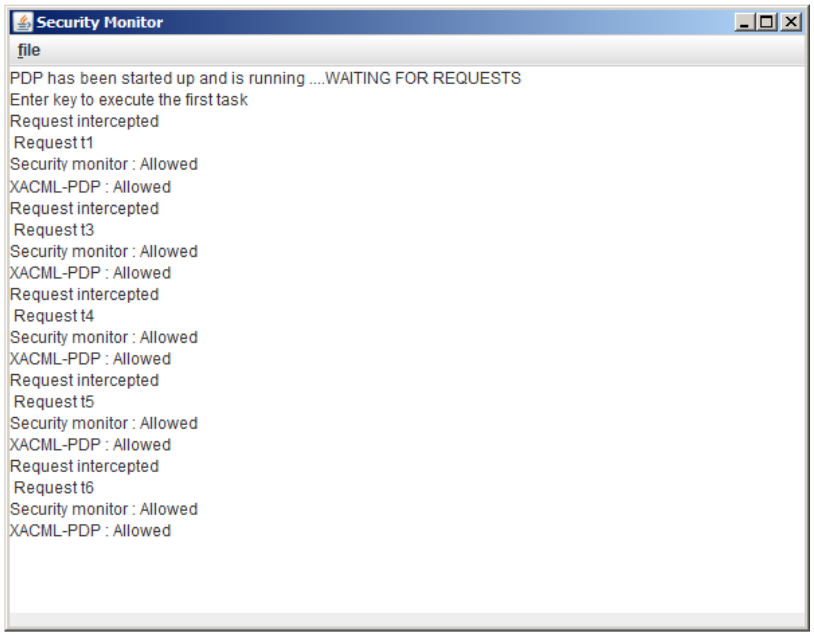

Figure 5.2: Use case: No constraints violation

5.1.1 Use case: Violation of the Dynamic Separtion of Duty constraint

In this Section, we want to show what happens if the constraint DSoD is violated which is imposed on task *t6*. We follow the same execution trace as described in the previous Section and with the same configuration. Except, employee *u6* is now also executing task *t5*. What should happen, is that the monitor will detect a violation and task *t6* is not allowed to be executed. That is, the

monitor will make the decision *deny* and request is also not forwarded to the PDP. Furthermore, the attribute *executed* of task *t6* will be updated with value {} which is the empty set.

Following the same scenario as described in the previous Section until task *t5* which is now performed by employee *u6*. This task is allowed to be executed by the monitor and attribute *executed* is updated with the value *u6*. Now, employee *u6* is requesting to execute task *t6*. This request is intercepted by the monitor, and since the constraint DSoD is imposed on this task the monitor is evaluating the constraint. For that, the monitor checks the value of the attribute *executed* of task *t5*, and compares it with the *requester* of task *t6*. The values are the same and so the constraints evaluates to *false*. Hence, task *t6* is denied to be executed by employee *u6*. The monitor does not let the request being forwarded to the PDP, since it has no use for further evaluation as showed in Figure 5.3. The *executed* of task *t6* is updated with the value {}.

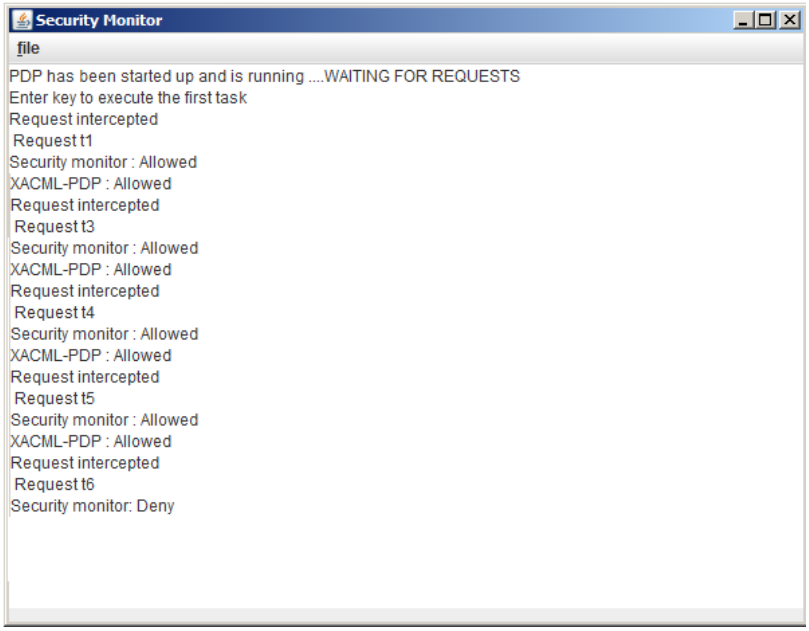

Figure 5.3: Use case: Dynamic Separation of Duty constraint violation

As expected, the monitor enforces the specified constraints also when there is a violation detected. The monitor takes the right decisions and the dynamic attributes are updated with correct values. Furthermore, when a constraint is violated the request is not forwarded to the PDP. We consider that our monitor is working to our expectations.

5.2 Discussion

In this Chapter, the functionality of the security monitor has been demonstrated. We have executed a trace of tasks of the mortgage workflow on which constraints are imposed on. Every time an employee wants to execute a task he/she has to make a request to our framework for evaluation. The monitor intercepts these requests for enforcing the constraints. To prevent overhead, the monitor skips request for which no constraints are imposed on the requesting tasks. But tasks on which constraints are imposed, are always evaluated by the monitor. The monitor evaluates the constraints by monitoring the attributes related to the requesting user and the executing task. The constraints are evaluated either to be true or false, as specified in Chapter 3. The tasks are only allowed to be executed if the constraints are evaluated to be true. If the constraints are evaluated to be false, then the tasks will not be executed.

We believe that if a constraint is evaluated to be false by the monitor it should not be forwarded to the PDP, since the monitor decision is decisive. Even in the case that the PDP allows the execution but the monitor does not allow it, then still the task will not be executed.

After evaluation, the dynamic attribute *executed* is updated with the correct values. Initially, the dynamic attributes contain the value empty {}, meaning that it is an empty set. In our model, we assume that attributes contain atomic-value or set-values. The security monitor can read these values, and it can clearly make a distinction between the types of values. We store all the attribute values as a (Java) String data type because it can be easily converted to other data types.

Chapter 6

Conclusions

In this thesis, we presented a security monitor that enforces authorization constraints at runtime. These constraints are imposed on security sensitive tasks of a workflow. The idea is to prevent unauthorized users from executing tasks at runtime. Although several research works have been conducted to develop security monitors that are capable of enforcing constraints at runtime, they were not applicable for real world constraints. We detailed the limitations of these existing works to show the usefulness of our work. These limitations are due to the expressiveness of the formal languages to specify constraints, and the dependency between the workflow environment and its authorization constraints. The goals of this thesis were to provide a specification notation that is usable for security designers and develop a security monitor to enforce the specified constraints at runtime. Our approach to achieve these goals was applied in two parts presented in this thesis.

The first part of this thesis was dedicated to the expressive formal notation to specify authorization constraints which can capture real world constraints. The security designer must have an easily and expressive notation to specify constraints which can be applied on real world workflows. This is solved in this part. Although, several notations have been proposed to specify more advanced constraints like in [11], the proposed language is too complex to use for security designers. In addition, security designers have to learn the proposed language before they can use it. In our proposed notation, security designers can specify advanced constraints while it is easy to use, because it does not inherit any difficulty of complex mathematical formalization. It consists of basic predicate logic, basic mathematical operators and attributes which describe the properties of users and tasks. Security designers use simply operators to impose constraints on tasks. The usability of predicate logic and mathematical operators allows the security designer to specify constraints with satisfaction. We applied our notation by using real world constraints such as Dynamic Separation of Duty that are widely used in workflows, and are a key concept for business processes. We used an example workflow that represents a mortgage provision process, and imposed a set of constraints that are used in real world workflows. The specification of these constraints demonstrate the expressiveness and usability of the notation. In addition, all the specified constraints can be enforced without considering the workflow implementation.

The second part of this thesis consists of enforcing the specified constraints. We presented a security monitor based on Aspect Oriented Programming (AOP) that enforces authorization constraints at runtime. The specified constraints are mapped inconsiderably to enforcement code thanks to our simple notation. Then the security monitor evaluates and enforces the generated constraints. We used an automated AOP methodology to weave the monitor in our framework at a selective point in a modular way. The security designer only has to specify the constraints intended for the tasks, and the monitor automatically fulfills the evaluation and enforcement of the constraints at runtime. We have already applied this approach to specify and enforce authorization constraints to verify the usability and feasibility of our approach. For this purpose, we used an example

workflow that represents a mortgage provisioning request of a bank.

Finally, we integrated the security monitor in a state of the art access control mechanism (ACM) which is responsible for enforcing authorization policies. Security designers can use the standard specification language that is provided by the ACM to specify policies. The security monitor is integrated automatically and requires no intervention from security designers. Moreover, the tasks are intercepted at execution and evaluated against specified constraints. If a violation is detected, the task is prohibited from execution. Otherwise, the task is allowed to be executed. Furthermore, we conducted a validation process which consists of various use cases. During this validation process, the security monitor has been validated. Results showed that the expected conditions were indeed found. Hence, our research produced a security monitor that is capable of enforcing authorization constraints at runtime.

Taking all this into account, we can conclude that we have met our thesis goals and have provided a framework that is capable of enforcing authorization constraints and policies.

6.1 Discussion

Although the results showed that the security monitor can evaluate and enforce authorization constraints at runtime, there are still some unresolved limitations. The monitor must enforce history dependent constraints. For this reason, a constraint is imposed on the task that is executed later in time than the related task. However, if these related tasks are executed in *parallel* then it is difficult to decide on which task to impose the constraint. This is because we do not know which task will be executed first or later in time when they are structured as a parallel pattern. Therefore, constraints that require history information for evaluation cannot be imposed on tasks that are executed in parallel.

An important aspect of workflows is that they must be able to terminate successfully. Applying constraints on tasks can cause deadlocks which will prohibit the workflow to terminate. In literature this problem is described as *Workflow Satisfiability Problem* (WSP) [6, 10]. It consists of checking whether a workflow can terminate while it still can satisfy the authorizations constraints. The WSP has not been considered in the current study because it requires more sophisticated tooling.

Chapter 7

Future Work

In this section, we provide some future work for our framework. We focus on the extensibility of our security monitor which needs a detailed research work.

Our security monitor has convincing advantages, but there are also limitations to the functionality of the framework. Despite the fact that our framework can generate only the four constraints that we have described, the use of a general generator gives the possibility to generate every specified constraints. The idea is to construct a generator which will read the formal specified constraints and feed it to a model checker. The model checker can verify the consistency of the specified constraints and from that it will automatically generate enforcement code. Several research has been conducted to specify constraints in high level specification language and from that synthesizing security monitors. Our notation is based on predicate logic and can be easily adapted to a model checker syntax such as AspectLTL [16]. This has the advantage that the security designer can confirm that his specification is consistence. In addition, this automated process of synthesizing monitors prohibits errors. Developers can make errors when they manually map specification to code. Thus, automatically generating code helps to avoid errors at implementation level.

Our framework is not complicated to use, but it lacks usability for user preferences. An advanced GUI can be developed to help the security designer using our framework in efficient way. For example, the accessibility of the functionalities (e.g., creating new constraints) can be ensured by using menu items. However, this is more engineering work then scientific.

Bibliography

[1] Mahmoud F Ayoub, Riham Hassan, and Hicham G Elmongui. Esac-bpm: Early security access control in business process management. 13

[2] David Basin, Samuel J Burri, and Günter Karjoth. Separation of duties as a service. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 423–429. ACM, 2011. 12

[3] David Basin, Samuel J Burri, and Günter Karjoth. Dynamic enforcement of abstract separation of duty constraints. *ACM Transactions on Information and System Security (TISSEC)*, 15(3):13, 2012. 12

[4] Kai Bollert. On weaving aspects. *Citeseer*, 1999. 11

[5] Achim D Brucker, Isabelle Hang, Gero Lückemeyer, and Raj Ruparel. Securebpmn: Modeling and enforcing access control requirements in business processes. In *Proceedings of the 17th ACM symposium on Access Control Models and Technologies*, pages 123–126. ACM, 2012. 13

[6] Silvio Ranise Clara Bertolissi, Daniel Ricardo dos Santos. Automated synthesis of run-time monitors to enforce authorization policies in business processes. *ACM-digital library*, 2015. 12, 13, 42

[7] Ylies Falcone. You should better enforce than verify. In *Runtime Verification*, pages 89–105. Springer, 2010. 27, 28

[8] Vincent C Hu, D Richard Kuhn, and David F Ferraiolo. Attribute-based access control. *Computer*, (2):85–88, 2015. 6

[9] Florian Huonder. Heras-af xacml engine, 2015. 7, 25

[10] J.-P.Kuo J. Crampton, M. Huth. Authorized workflow schemas: deciding realizability through ltl(f)model checking. *Springer*, pages 31–48, 2014. 12, 17, 42

[11] Slim Kallel. *Specifying and Monitoring Non-functional Properties*. PhD thesis, Software Technology Group, TU Darmstadt, Germany, 2011. 12, 13, 41

[12] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. *Aspect-oriented programming*. Springer, 1997. 28

[13] RAMNIVAS LADDAD. *AspectJ in Action, Second Edition*. Manning Publications Co., Connecticut, USA, 2010. 10, 11

[14] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009. 27, 28

[15] Ninghui Li and Qihua Wang. Beyond separation of duty: An algebra for specifying high-level security policies. *Journal of the ACM (JACM)*, 55(3):12, 2008. 12

[16] Shahar Maoz and Yaniv Sa'ar. Aspectltl: an aspect language for ltl specifications. In *Proceedings of the tenth international conference on Aspect-oriented software development*, pages 19–30. ACM, 2011. 43

[17] Business Process Modeling and Notation. Bpmn tutorial. `http://www.bpmn-tool.com/en/tutorial/`, 2015 (accessed December 7, 2015). 5

[18] Tim Moses. extensible access control markup language (xacml), 2005. 7, 8, 26

[19] Sun Micro systems. Sun's xacml implementation. `http://sunxacml.sourceforge.net`, 2004-2006. 25

[20] Professor Wil van der Aalst. Workflow pattern home page. `http://www.workflowpatterns.com/`, 2010-2011. 6

[21] Wil van der Aalst. Yawl: Yet another workflow language. `http://www.yawlfoundation.org/`, 2004-2014. 5

[22] Ravi Sandhu Xin Jin, Ram Krishan. A unified attribute-based access control model covering dac, mac and rbac. *DBSec*, 2012. 6

A Authorization constraints

Dynamic Separation of Duty:

$t \neq_{executed.name,requester.name} t'$

$\iff \forall u \in executed(t), u' \in requester(t') : u \neq_{name} u'$

$u \neq_{name} u$

$\iff \forall n \in name(u), n' \in name(u') : n \neq n'$

Dynamic Binding of Duty:

$t =_{executed.name,requester.name} t'$

$\iff \forall u \in executed(t), u' \in requester(t') : u =_{name} u'$

$u =_{name} u$

$\iff \forall n \in name(u), n' \in name(u') : n = n'$

Seniority constraint:

t $\succ_{requester.level,executed.level}$ t'

$\iff \forall u \in executed(t), u' \in requester(t') : u \succ_{level} u'$

$u \succ_{level} u'$

$\iff \forall l \in level(u), l' \in level(u') : l > l'$

Working experience constraint:

$t >_{requester.workexperience} N$

$\iff \forall u \in requester(t), n \in N : u >_{workexperience} n$

u $>_{workexperience} n$

$\iff \forall m \in workexperience(u) : m > n$