Access Control Obligation Specification and Enforcement Using Behavior Pattern Language

By

Mohammadhassan Sharghigoorabi

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of
Doctor of Philosophy

in

Electrical and Computer Engineering
Faculty of Engineering and Applied Science

University of Ontario Institute of Technology (UOIT)
Oshawa, Canada

January 2018

Abstract

Increasing the use of Internet-based devices offers novel opportunities for users to access and share resources anywhere and anytime so that such a collaborative environment complicates the design of an accountable resource access control system. Relying on only predefined access control policies based on an entity's attributes, as in traditional access control solutions, cannot provide enough flexibility to apply continuous adjustments in order to adapt to any kind of operative run time conditions. The limited scope and precision of the existing policy-based access control solutions have put considerable limitations on adequately satisfying the challenging security aspects of the IT enterprises.

In this research, we focus on the obligatory behavior that can play an important role in access control to protect resources and services of a typical system. Since traditional access control is performed only once before the resource is accessed by the subject, the access control system is unable to control the fulfillment of obligation while the access is in progress. Practically, such a requirement is implemented in hard-coded and proprietary ways. Consequently, the lack of sophisticated means for specification and enforcement of obligation in access control system decreases its flexibility and may also lead to the security breach in sensitive environments.

We provide a descriptive language that is capable of defining a variety of complex behavior patterns based on a sequence of user actions. Such a description can be used to specify different elements of the obligation in order to attach to a policy language, and it is also used to generate queries for behavior matching purposes.

Moreover, we propose a behavior pattern matching framework to approve the fulfillment of the obligation by looking into the audit logs. However, this method is extremely inadequate for ongoing obligations. Therefore, we proposed a compliance

engine by utilizing complex event processing in order to make a decision to revoke or continue the access in a timely manner. We implemented both frameworks that can be used to approve the obligation fulfillment as well as to evaluate the expressive power and complexity of our proposed language.

Acknowledgements

I would like to take this opportunity to thank all individuals who had contributions to make this research possible. A special thanks to Dr. Ramiro Liscano, my supervisor, for all of his support and recommendation in helping me to develop my research skills and focus on a specific problem in order to complete my PhD dissertation. Also, genuine appreciation to Dr. Kamran Sartipi with whom I started this research and I was initiated into the research domain by his guidance. I would also like to express deep gratitude to my committee members, Dr. Mark Green and Dr. Jeremy Bradbury, for agreeing to serve on my thesis committee and for their constructive comments.

Finally, and most importantly, I would like to thank my family where I have received love and support to pursue my interest. In special, I am thankful to my wife for her patience, and my daughter whose laugh and curiosity gave me enough energy to achieve the goal.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Policy-based access control is a common and flexible mechanism to protect resources in information systems. It helps enterprises to enforce regulations that define who should have access to which resources, and under what circumstances [1]. Such a system with the inclusion of a data leakage prevention component are a necessity for handling and sharing of sensitive resources in the comprehensive security plan of large enterprises. In spite of applying different mechanisms to provide security, IT end users deliberately or carelessly exhibit risky behaviors that may put valuable resources at risk.

Data leakage, that exposes the sensitive information, causes serious issues for an organization regardless of the intentional or unintentional mistakes by users. The IBM Cyber Security Intelligence Index reported that 60 percent of attackers are insiders in 2015, and the number and extent of damage of insider breaches continue to rise year by year [2, 3]. An insider is anyone who has been authorized to access physically or remotely to a company's assets.

In a data breach investigation report [4], 90 data breaches that occurred in 2008 were analyzed. The result showed that 20 percent of breaches resulted exclusively from the actions of insiders. Of all insider cases in 2008, about two-thirds were the result of deliberate action and the rest were unintentional. Insider breaches were much more damaging than those caused by other sources. According to the published report by United States Government Accountability Office (GAO) [5], the policy violation had the third rank (17 percent) as the reason of information security incidents in 2014. In recent years, subverting access control is still a notable mechanism for attack through exploiting of weaknesses and limitations of authentication and authorization system [6].

Consequently, effective implementation of access control is necessary to protecting the confidentiality, integrity, and availability of information resources of an organization. Access control policies are a good starting point to explore misuse possibilities because insiders usually abuse the gap between access control requirements and implemented policy rules to circumvent regulations. User behavior monitoring is required as a supplementary module for access control in order to review or update security policies in a timely manner. Since quantitative and statistical analysis are usually used for the extraction of applicable knowledge, traditional behavior modeling using qualitative methods to model the behavior is an inappropriate model for human behavior analysis. Therefore, the lack of systematic and comprehensive methodologies and tools have prevented the advancements towards utilizing knowledge hidden in user behaviors [7].

The primary motivation for this thesis lies in the need for a language to specify precisely the user behavior pattern. We are interested in better understanding how a behavior-based language can be used to determine the access control obligation, as well as how such a language is useful to enforce the obligatory behavior.

1.2 Problem Definition

Security vulnerability of the IT systems against malicious attacks to the organizations data integrity has become one of the most costly, hard-to-track and growing threats worldwide. There have been huge efforts and investments on tracking such threats through investigation of the network communications, which have inherent limitations in terms of the scope of online analysis techniques, and the amount of information carried by each individual transaction. Most of these efforts are focused on preventing outside intruders from breaking into the system through disrupting the IT system's authentication mechanism, in particular, in the large and federated distributed systems.

The majority of existing security management systems concentrate on the communication data and ignore the high level behavior of the trusted users who could freely perform harmful actions. The users access patterns to the system resources (i.e. user-behavior) is the most realistic and reliable indicator of the potential malicious actions against system's security and data integrity.

Different access control models have been proposed to protect resources until the access is granted to a subject in order to perform some specific operations. After granting the access, the majority of proposed access control models cannot control access during the access execution and are unable to consider the historical behavior of the subject for the continuity of access. Meanwhile, traditional static access control policies are too inflexible to meet the stateful requirements for modern applications in highly heterogeneous and dynamic environment such as Internet of Things (IoT) and pervasive computing domain. Currently, there is no proper means to specify access control obligation that can fulfill stateful requirements.

Using obligations to handle the stateful requirements raises the question of how

the enforcement will be managed. Since traditional access control is performed only once before the resource is accessed by the subject, the access control system is not able to control the fulfillment of obligation while the access is in progress. Therefore, the lack of capability to check the obligations for the continuity of access purpose may lead to the security breach in sensitive environments. Moreover, modeling user behavior, obligation and finally enforcing the obligatory behavior are challenges that we will address in this thesis.

1.3 Thesis Statement

The goal of this thesis is to specify the obligatory behavior and provide enforcement mechanisms to fulfill access control obligations. The proposed specification and enforcement methods meet the following objectives:

- Increase the specification scope of the access control obligation

  Using a behavior-based language enables the policy administrator to specify a set of planned actions and their correlations as a complex obligation instead of one action, for access control purposes.

- Verify the enforcement of the obligation

  A behavior matching is required to find the instances of a particular behavior to verify the obligation fulfillment. The proposed language can specify the desirable behavior based on the involved events and their constraints in order to drive a matching mechanism to find instances.

1.4 Research Scope and Assumptions

The scope of this research is restricted to the access control obligation. An obligation determines that a subject is forced to do some actions before, after or during access in order to maintain the state of the system. We assume that all accesses are logged and there are no attacks to bypass the access control elements or subvert the functionality of the access control system.

Moreover, we assume that users do not change gradually their behavior in order to remove their traces. Since this research focuses on the enhancement of the specification of obligations and hence the behavior of a threat is known, it is not necessary to create a threat analysis model.

From a general perspective, this research is relevant to the behavior pattern specification and the verification of the obligation fulfillment. The acquisition of events as primitive elements of the behavior is beyond the scope of this research.

1.5 Contributions

The primary contributions of this thesis are as follows:

- Provisioning a model based on ordered set of events to present the features of the user behavior pattern (presented in Chapter 3).
- Proposing a specification language that offers powerful features to demonstrate the structure and semantics of the user behavior patterns. Such a language allows administrators and analysts to express some explicit conjecture about users behavior as well as it facilitates the description of an obligatory behavior (presented in Chapters 3 and 7).

- Designing a model for different kinds of obligation based on the behavior of the user. Modeling the obligation from user behavior perspective allows us to utilize Behavior Pattern Language (BPL) to specify obligatory behavior that can be attached to access control policy language (presented in Chapter 4).

- Applying session initiation protocol to manage the obligation session for the purpose of continuing or revoking the access decision (presented in Chapter 4).

- Designing and developing a pattern matching algorithm that can be used to match the described behavior against audit logs. Moreover, such a pattern matching engine can find the instances of a post-obligatory behavior by applying the exact matching in order to verify the fulfillment of post-obligation (presented in Chapter 5).

- Implementing a compliance monitoring application based on Complex Event Processing (CEP) technology to verify the fulfillment of the ongoing obligation (presented in Chapter 6).

1.6 Thesis Structure

In this chapter as introduction chapter, we have presented and motivated the primary research problem of traditional access control and insider threat, the thesis statement, the research scope, and finally the contributions of this research. The remaining chapters of the thesis are organized as follows:

- Chapter 2 presents the related works and technologies to this research. It includes the overview of behavior modeling and representing, pattern matching

technique and its application in security, access control policy language and obligation model.

- Chapter 3 discusses different aspects of user behavior in security perspective and then provides a model for user behavior. We design a behavior pattern language to present different kind of behavior patterns.

- Chapter 4 provides an overview about the features of access control policy specification language that can support obligation. Then, we present our model for obligation and how it can be attached to the standard access control policy language. Moreover, we introduce our approach for handling the obligation session.

- Chapter 5 presents our proposed method for static enforcement using the behavior pattern matching that relies on valued constraint satisfaction problem. We design and implement a search algorithm to find the instances of the desirable behavior from an audit log in order to verify the fulfillment of post-obligation.

- Chapter 6 discusses our approach to design an active enforcement for ongoing obligation. We present a behavior pattern evaluation model that will be used to make the compliance framework.

- Chapter 7 presents our experiments to prove the capability of BPL to specify the behavior for both enforcement frameworks.

- Chapter 8 contains the conclusion of this research and proposes some potential future directions to improve this research.

Chapter 2

Related Work

There are several works that are relevant to different parts of this research. There is a close relationship between event processing and behavior modeling as well as behavior representation. Behavior computing is another emerging field that is relevant to our research. Behavior computing studies effective methodologies, techniques and technical tools for modeling, representing, analyzing human behaviors characteristics. Section 2.1 presents an overview of behavior modeling, representation and analysis.

Pattern matching is widely used in security domain to identify pattern of interest. An efficient matching engine is the core of most intrusion detection system. Pattern matching is an effective approach that is utilized to find the instances of a particular behavior pattern. Section 2.2 discusses different aspects of pattern matching and its role in security applications.

Obligations are the vital part of many access control policies and they specify mandatory behavior that should be conducted by a user of the access control system in sensitive domains. Such a behavior includes several correlated actions so that different techniques were used to model and represent obligations by policy specification languages. Section 2.3 presents an overview of different access control policy

specification language and the modeling of obligations.

2.1 Behavior Modeling, Representation and Analysis

Behavior computing is an emerging field and young discipline aiming at exploring formal methods for human behavior representation and analysis. Human and machine behavioral modeling and analysis are becoming interesting areas of study in a variety of research domains such as computer security and access control [8, 9], insider threat detection [10], fraud detection [11], finance [12, 13], social network analysis [14], and weblog analysis [15].

Designing an efficient and scalable infrastructure for monitoring and processing behavior patterns has been a major research interest in recent years to provide a mechanism for the enterprise to monitor the behavior patterns and anomalous behaviors of their customers. Behavior data are becoming a valuable asset to be carefully analyzed in order to reveal its explicit and implicit knowledge that cannot be attained just through recorded transactional data. Moreover, appropriate presentations of behavior analysis results are valuable for end users to enable them to make decisions properly.

Behavior modeling and representation attempt to develop representation languages and tools based on formal methods and techniques. The language helps the analyst to illustrate attributes, primitive events, semantics, constraints, and behavior patterns in a detailed and precise manner. The correlations between events based on attribute values along with constraints constitute the semantic of the behavior pattern.

Behavior modeling has been increasingly recognized as a challenge for associating semantics with the human's actions to be used in different environments and for different purposes. Cao et al. [12, 16] consider behavior as individual activities represented by events as well as activity sequences conducted by the user within certain context. They defined four dimensions, as: actor, action, environment and relationship to represent abstractly the user behavior. The assigned attributes allow for describing features of each dimension and Temporal Logic is used to express the properties and relations between elements of a desired behavior. Their approach is similar to the BDI (Belief-Desire-Intention) model [17] developed in multi-agent systems, and proposed model is appropriate for modeling and analyzing activities in abstract level.

In [18] a conceptual language NKRL (Narrative Knowledge Representation Language) provides an ontological paradigm to deal with the most common types of human behaviors. This approach is basically a knowledge representation language to fill the gap between behavior modeling and its translation into computer-usable tools. They interpreted a narrative including a sequence of logically structured elementary events that describe the behavior. In [14] the authors introduce a semantic model for representing and computing behavior in online communities. An ontology was defined to represent all involved entities and their interactions. Representing the behavior pattern by semantic rules has been also proposed in literature so that in [19] Event-Condition-Action (ECA) rules was considered to represent the frequent behavior patterns.

Representing behavior using the modelling languages has been proposed in [20, 21]. The HBML (Human Behavioral Modeling Language) was introduced by Sandell et al. [20] in order to efficiently capture the relationships and behaviors derived from analysis of data streams. The behavior is represented in three orders (0^{th} order, 1^{st}

order and 2^{nd} order) so that the 0^{th} order describes the list of activities that a user may engage to do, the first order describes the 0^{th} order activities along with some environmental context information, and the second order describes the activities with probabilistic characterization.

In our research, we need to specify the inner structure of the behavior pattern in terms of length, sequence of events, and their association. Such specifications are essential to define obligatory behavior. Moreover, a clear description of a particular behavior is required to initiate a full inquiry into the user behavior to ensure good compliance with predefined obligations. Since the proposed behavior modeling in the literature cannot provide our requirements, we introduce a language called Behavior Pattern Language (BPL) to represent accurately a complex behavior pattern. BPL is a language to represent the user behavior through defining a sequence of events and their correlations. The ability to combine events and keep the ordering of events is a basic and powerful specification for a behavior-based language. Consequently, we deal with two kinds of elements in BPL: ordered sets of events and operations. Operations are descriptions of the constraints for defining the correlations between events. The embedded features and operators to BPL improve the capability of BPL to address any categories of behavior.

2.2 Pattern Matching for Security Applications

The operation of matching is extensively used in many areas of software engineering such as data mining, text processing, data integration and query processing. A pattern matching problem that is also called exact pattern matching is to find all the occurrences of a known pattern in a sequence. Different algorithms such as Karp-Rabin, Knuth-Morris-Partt algorithm, Boyer Moore algorithm, Horspool algo-

rithm, have been developed to solve string pattern matching problem [22, 23]. In recent years, string pattern matching algorithms are widely used as powerful tools in database search and retrieval, text processing and editing, lexical analysis of computer programs, data compression, cryptography, the study of genomics and DNA analysis and other applications.

The first approach for string pattern matching is identifying significant structure in a pattern by pre-processing offline, then using the text as input of algorithm to find the similar structure. For example, this approach is used in Knuth-Morris-Pratt (KMP) and Boyer-Moore (BM) algorithms. The second approach is pre-processing the text into particular data structure such as suffix trees and suffix arrays that improves the search of pattern in the text. Burrows-Wheeler Transform (BWT) is typical example of this approach. Transforming both the text and pattern into numeric or binary strings is another approach that is used in some algorithms. For example, the Karp-Rabin algorithm uses the numeric transformation via a hash function, and Bitap algorithm utilizes bitwise operations on binary strings [24].

There are many applications where approximate matches are desired as opposed to find exact matches. Particularly, when the reference pattern is not known, the analyst would like to spot the instances that are similar enough to the reference pattern specification. For example, approximate pattern matching algorithms are very useful for searching a text database using keywords where both the text and the keywords could contain spelling errors. In general, the goal of approximate string pattern matching is to find pattern in a text where one or both of them have suffered from errors.

Comparing between sequences in terms of finding similarity needs a similarity measure. A natural way to compare two sequences is to determine their edit distance defined as the minimum number of editing operations that will transform one sequence

to the other sequence. Many studies focused on developing efficient approaches for sequence similarities based on Edit Distance (sometimes called the Levenshtein distance) measure to calculate distance between strings. The Edit Distance between two strings S and T will be the minimum number of operations to transform S to T. Three operations insert, delete and replacement are defined in this measure. There are some approaches in literature to define similarity measure between sequences through applying Edit Distance measure by assigning different weights to the operations [25]. The problem of defining a similarity measure between two sequences is converted to recognizing and counting the number of all distinct common subsequences between two sequences. Therefore, the Longest Common Subsequence (LCS), that generates the length of the longest common subsequences of two sequences, has been introduced as a similarity measure in literature [26].

Event pattern matching is a query technique where a sequence of input events is matched against a complex pattern. In [27], sequenced event set pattern matching has been formally defined and an automaton-based evaluation algorithm has been proposed to solve the pattern matching problem. Matching a sequence of input events against a pattern that allows to specify a sequence of sets of events is the main problem in sequenced event set pattern matching.

Pattern matching is widely used to recognize data of interest in security applications such as Network Intrusion Detection Systems (NIDS). NIST describes the intrusion detection as a process of monitoring the events occurring in a computer system or network, and analyzing them for signs of possible violations of security policies [28]. In [29], the following three methodologies are considered for intrusion detection:

- Signature-based Detection (SD): A signature is a pattern that represents a

known attack. Since signatures are defined based on the knowledge acquired from a particular attack, SD is also known as knowledge-based detection or misuse detection. SD is a process for monitoring the occurred events and comparing them against predefined signatures to recognize intrusion.

- Anomaly-based Detection (AD): An anomaly is any deviation of a profile from a known behavior. A profile represents an observed behavior extracted from monitoring regular activities over a period of time. A profile is developed based on a set of attributes such as failed login attempts, resource usage, the count of a specific action done, etc. AD is also called behavior-based detection that is a process to specify normal profiles and compare them against observed events to recognize significant attacks.

- Stateful Protocol Analysis (SPA): A stateless protocol considers each request as an independent transaction that is unrelated to any previous request. Therefore, a communication based on a stateless protocol consists of independent packets of request or response, whereas a stateful-based communication requires the server retains session information and traces the status of each pair of request and response. The SPA process, that is also known as Specification-based Detection, is similar to AD, but SPA depends on vendor-developed generic profiles to specific protocols and standards from international standard organizations, e.g. IETF.

Traditionally, the two major perspectives, anomaly detection and misuse detection, are the basis of the majority of applications that have been implemented for intrusion detection. In such applications, a deep packet inspection is required to match a known malicious pattern against incoming packets. Providing time and space efficiency, particularly in high speed network links, is a challenge for pattern

matching applications. Generally, regular expressions are utilized to represent the known attack signatures.

Regular expression-based matching is simple enough for efficient implementation particularly for network security applications since Finite Automata (FA), either in their deterministic (DFA) or in their non-deterministic (NFA) form, is the most popular method for implementation [30, 31]. In DFA, each possible input symbol leads to at most one new state. However, when NFA receives an input, there are multiple choices for a movement as the new state. In particular, every DFA is also an NFA and each NFA can be converted to an equivalent DFA with the same expressive power by using a subset construction algorithm [32].

Regular expression is a common way to present the target pattern in pattern matching applications such as intrusion detection systems. For example, if we define three actions Read(r), Write(r), Send(s) for a typical access control system, then the enforcement of the access control system produces event sequences such as $wrwss$ and $srwrw$. Assume that we define a policy that a user cannot send an email after reading a resource. Enforcing such a policy can prevent the system from revealing private data through the network. If we would like to find patterns that follow the policy, we can use the regular expression $[(s|w)^*|(r^+(r|w)^*)]$ to present the pattern of the acceptable sequences [33].

Using regular expression is inappropriate to specify behavior patterns. The first reason is that events used to form the pattern in practice often contain features that cannot be described by the regular expression. The second reason is that the meaning of the behavior pattern will be presented by a set of correlation between the elements of the pattern so that such correlations cannot be properly presented by the regular expression.

We exploit our proposed behavior pattern language that enables the administrator

to express monitoring needs in descriptive pattern queries. A fundamental difference between BPL-based pattern queries and regular expression matching is that the set of events that match a particular pattern can be widely dispersed in an input streams. The BPL describes features and constraints of the target behavior pattern. Such a description guides the evaluation model to find rightly the instances of a described behavior pattern.

Our research applies the pattern matching technique into user access behavior in an effective way to find the instances of a particular behavior pattern. The structure and semantics of a conjectural behavior pattern are described by our proposed language. The relations between events as primitive elements of a behavior pattern are defined by constraints. Then, we model the problem as a constraint satisfaction and optimization problem to find instances of reference pattern in the event log before and after accessing resources. However, such a model has not enough capability to handle the behavior pattern while the access is in progress. Therefore, we utilize the complex event processing technique to resolve the weakness of our model to monitor the behavior of users during the usage of resources.

2.3 Access Control Policy Specification Language

An access control policy includes a set of rules to specify the permission of a subject for performing some actions on resources if particular conditions are satisfied. Access control attempts to control the activities of legitimate users who have been successfully authenticated. Typically, access control deals only with authorization decisions on users access to target resources. The objective of access control is to prevent from unauthorized disclosure (confidentiality), avoid improper malicious modifications (integrity), and ensure the access for authorized entities (availability). Separating the

policy from the implementation of a system helps the policy to be modified dynamically in order to cover the changes in the strategy for managing the system and controlling the behavior of users without changing the implementation of the underlying components of the system.

Different access control models have been proposed to protect resources until the access is granted to a subject in order to perform some specific operations. After granting the access, the majority of proposed access control models cannot consider the historical behavior of the subject, do not define how to manage the access and they do not consider any kind of control during the access execution.

However, the access is practically permitted as long as the security policy is satisfied. It means that if during the ongoing access, particular attributes change and the security policies do not satisfy anymore, it is essential that the access control system revokes the granted access to the resource. Dynamic and open environments such as web and cloud have high variability and the continuity of an access decision should be handled in some way by the system when the access is in progress. In such environments, new conditions may affect the access decision already taken over initial conditions. Meanwhile, the security policy should be regularly updated in order to prevent from compromising and breaching the system security.

Due to the ambiguity in the natural language to specify the access control requirements, it is not applicable to specify all conditions in requirements by using rules. Using obligation is an alternative way to eliminate the gap between formal requirements and policy enforcement.

An obligation determines that a subject is obliged to do some actions before, after or during access to a specific resource to satisfy some conditions (e.g. deadline). In other words, obligations are requirements that have to be fulfilled by obligation subjects for allowing or keeping the access right. Different languages have been pro-

posed to specify the policy. However, some of them are able to manage the obligation concept.

Obligations are explicitly expressed as the part of access control policies instead of being hard-coded into the main application. However, the majority of applications consider practically the obligation requirements as hard-coded [34]. Integrating obligations into policy increases the flexibility of the management and easy maintenance of the security system. Such a combination allows a system to react quickly to new circumstances and change easily obligation requirements when flaws in existing policies are discovered.

Different aspects of the obligation concept have been studied in the literature. In the first part, we review works that concentrate on languages to specify the policies along with corresponding obligations. In the second part, we mention works that attempt to propose a full-fledged model for the obligation.

2.3.1 Policy Languages to Support Obligations

In terms of language, several policy languages have been proposed to support the specification of obligations in access control policies, such as Rei [35], Ponder [36, 37] and XACML [38].

Rei relies on deontic logic and uses an application independent ontology to represent the concepts of rights, prohibitions, obligations and dispensations. The three flexible constructs: policy objects, meta policies and speech acts allow the administrator to define different kinds of policies. Policy objects are represented as: PolicyObject (Action, Conditions), where conditions are constraints on the actor, action and environment. Rei uses "has" construct to associate a policy object with an entity. A policy consists of several "has" rules. Meta policy specifies the precedence and

priority of policies to resolve the conflict. Rei uses four speech acts delegate, revoke, cancel and request to modify dynamically the policies.

In Rei, obligation is defined as actions should be performed by an entity and are triggered when certain conditions are true. A monitoring service in the policy engine provides information including actions being performed and contextual data to infer which obligations were fulfilled and which obligations were violated. The Rei framework does not provide an enforcement model and the policy engine is only able to reason over policies in Web Ontology Language (OWL) and domain knowledge in order to answer queries.

Ponder is a declarative and object-oriented language to support the specification of security policies by providing different kinds of policies such as authorization, delegation, refrain, and obligation for distributed systems. Ponder uses a domain service to group the subject and the target objects to which policies apply for providing the granularity of the access management. Ponder can define positive and negative authorization policies so that a positive one defines the actions that subjects are permitted to perform and a negative policy specifies the actions that subjects are forbidden to perform on target objects. The existence of negative authorization policies complicates the enforcement process of the policy in the system. Meanwhile, the specification of negative authorization policies increases the probability of conflicts occurring with the positive authorization policies.

In ponder, the obligation policies specify activities that are triggered by an event and should be fulfilled by the subjects within the system. Ponder provides operators to support the sequence of actions and the concurrency between actions for an obligation policy. Ponder compiler compiles the policy specification into a Java class and then a Java object represents it at runtime. Therefore, runtime changes to policy are not possible.

XACML as a standard provides a policy language and a reference architecture to create a fine-grained access control. An obligation is defined by XACML as follows: "An operation specified in a rule, policy or policy set that should be performed by the Policy Enforcement Point (PEP) in conjunction with the enforcement of an authorization decision." A policy may contain one or more obligations each of which is identified by an ObligationID. The FulfillOn attribute takes a value of either Permit or Deny. For example, FulfillOn "Permit" specifies that if Policy Decision Point (PDP) decides to permit the request, the obligation should be enforced by the PEP. Even though the XACML language provides an element called "Obligations" to specify actions, it does not provide standard definitions and features for such actions.

In our approach, we introduced a behavior-based language to describe precisely the obligatory behavior. The language applies the ordered-set of events. The correlations between events will be described by different attributes and corresponding values. Actually, such correlations present the behavior of the subject who is in charge of performing the obligation. Finally, we convert features of the described behavior in the format that can be integrated into XACML policy. Meanwhile, the described behavior is used as a reference pattern to search in the audit logs for approving the fulfillment of the prior and post-obligations. Furthermore, appropriate compliance checkpoints are extracted from the described behavior in order to develop a compliance engine to approve the fulfillment of ongoing obligations.

2.3.2 Obligation Model

Policy-based systems and languages consider the obligation from different points of view, although all of them consider the obligation as mandatory requirements that should be satisfied by a subject. We categorized such views in two groups each of

which focuses on specific features of the obligation.

The first group considers an obligation as a predicate. From the Usage Control (UCON) perspective, an obligation is a predicate that should be evaluated (true or false) in order to verify whether a certain activity has been fulfilled or not. The evaluation of the obligation fulfillment is applied to the request before, during, and after usage decision. Pre-obligation is a predicate that checks if specific activities have been executed or not before usage exercise based on the recorded activities of system's entities. Ongoing is a predicate that should be satisfied periodically during the usage of granted access. If the evaluation of the predicate is postponed after making decision for the request and used for future usage decision, it is called post-obligation that is not supported by original model of UCON [39]. Basically, post-obligations present actions that occur after the usage of the resources in a specific session. Therefore, such actions are required to be executed and a notification of the fulfillment of them can be sent to the service provider. However, the notification has no effect on the decision of the request.

The second group considers the obligation as a duty. For example, Ponder-based systems consider the obligation as activities that should be fulfilled by the managers when a particular event occurs within the system. Similarly, Rei-based systems define the obligation as a task for an entity when a certain set of conditions are true.

In terms of modeling the obligation, different approaches have been discussed in literature. Using different formalisms is an usual method to model the obligation for various purposes. For example, set theory [40], Lamport's temporal logic [41], and logic rules [34] were utilized to model the obligation.

Using state machines to model obligation enforcement was widely explored in literature so that [42, 41] applied formally state machine to model the obligation. Bertino et al. [43] modeled an obligation as a state machine interacting with PEP

and the outside world through events. They extended the XACML architecture by inserting an obligation module inside of environment to capture the state of the application. The PEP implements the obligation logics and communicates with the obligation module using an event mechanism. They used a timer inside the access control component. They defined six states and used the syntax of XACML rule to define a sequence of rules in order to associate with an obligation. Authors assumed that PEP can change the PDP decision, if PEP cannot discharge the obligation.

Using UML profile for modeling obligations was proposed in [44]. They defined a modeling methodology on top of the profile in order to use the profile for modeling Obligation Class Diagrams (OCDs) and Obligation State Machines (OSMs). Generating various test cases to cover different behaviors of the obligation is the main objective of using UML profile.

Our approach to model the obligation is different from the mentioned methods, so that we attempt to model the behavior of the subject who is responsible to fulfill the obligation as opposed to model directly the obligation states. Meanwhile, we proposed our model based on the access control points of view. The current version of XACML cannot support the pre-obligation and ongoing obligation that are widely discussed in usage control perspective [45]. The semantic of obligation is different in XACML. It is considered as an operation (a set of planned actions) that will be performed by PEP after making authorization decision by PDP. Such a view on the obligation is close to the definition of the obligation in [40] that defines an obligation as an action which a subject must be performed during a time frame in the future. Our definition for an obligation is partially different so that we add the concept of sequence and association into planned actions that should be discharged sequentially by the subject during access time.

Chapter 3

Behavior Modeling and Language

This chapter introduces an abstract behavior model and our proposed language to describe a behavior pattern. Behavior representation language and behavior pattern detection and processing techniques are essential parts of the user behavior analysis. The user behavior is hidden within the transactional data. Querying the instances of the particular behavior relies on an appropriate language in order to represent the specification of the desirable behavior patterns.

From the security perspective, users behavior is divided into two groups: normal and abnormal behavior. Intrusion detection uses a variety of techniques to filter the operational behavior of users based on these two categories. As can be seen in Figure 3.1, we consider suspicious behavior as another category inside of the normal behavior. Suspicious behavior is the guess of the administrator about the activities of a user. Such behavior occurs because some users attempt to circumvent security policies or defined obligations. Consequently, the administrator needs a language to generate a conjectural behavior pattern to initiate further investigation.

The objective of this chapter is to introduce such a language. User behavior concepts as well as our model to represent the behavior are formally defined in Section 3.1.

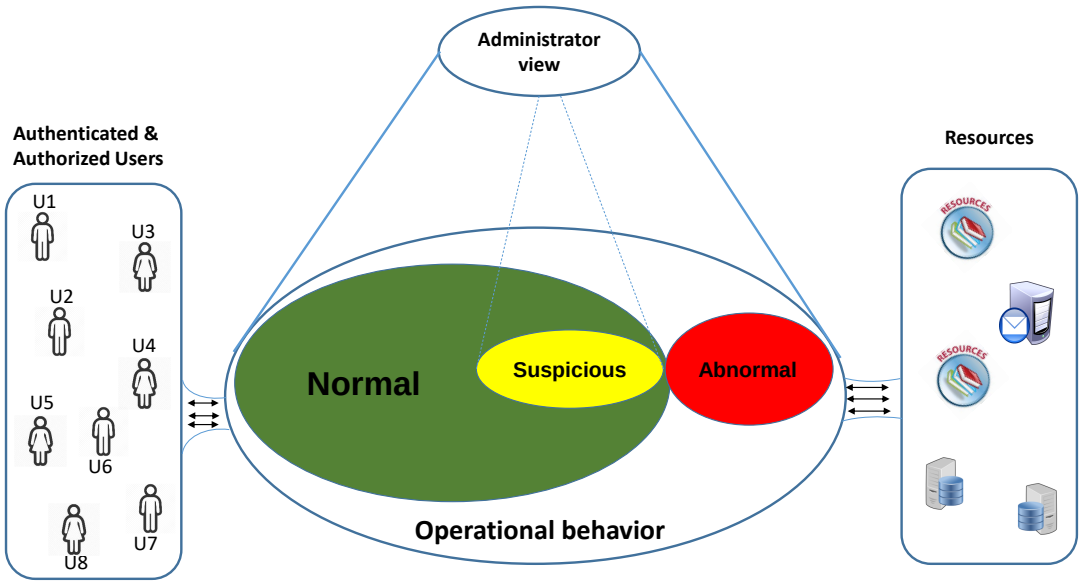

Figure 3.1: The view across the operational behavior from the administrator perspective

The features, syntax, and semantics of our proposed behavior pattern language as a template to compose a behavior will be discussed in Section 3.2. Finally, in Section 3.3, we described different categories of behavior patterns by using our proposed behavior pattern language.

3.1 Behavior Modelling

In this section, we formally define different components that progressively build the proposed user behavior model.

Definition (Attribute): an attribute "a" ($a \in A$ set of attributes) is a contextual information from an actionable and context-aware environment such as hospital, business enterprise, or any other work place. The value of attribute "a" belongs to a particular domain of values D_a. Typical primitive types such as integer, float, text,

date and time are used to define the domain values of an attribute. Table 3.1 specifies the attribute names and their domain values in the context of this thesis.

Table 3.1: A sample of attribute name, and domain value

Attribute Name: a	**Domain of Values:** D_a
User	John, Mike, Emma, Julia, Mary, Robert
Role	Nurse, Physician, Radiologist
Action	Read, Write, Create, Order, View
Resource	Diagnostic Report, Image, Exam
Time	00:00 − 24:00
Date	2000-01-01 , 2099-12-30

Definition (Event): an event "e" is a primitive element of a behavior. An event is represented as a tuple of attributes that describe the characteristics of the interaction between a user and a system resource. The analyst can obtain some basic information from a single event since the semantic information provided by a single event is quite limited. Below, two examples of events with six attributes are shown:
e1= <John, Nurse, Read, Diagnostic Report, 10:00, 2015-06-10>
e2= <John, Nurse, View, Image, 10:30, 2015-06-10>

Definition (Event type T): an event type is determined by the structure of the event tuple in terms of the number of attributes and the domain values of attributes. Two events have the same type if both the number of attributes and their domain values are the same (as e1 and e2 above). Basically, the event type defines a specification for an event set so that all events should have the same structure. Below the event type associated with events e1 and e2 are shown:

T = <User, Role, Action, Resource, Time, Date>

Definition (Set of events): "*E*" is the set of all events that are recorded in the log repository or come from the event stream.

Definition (Behavior B): a behavior is an ordered-set of finite number of events that possess the same event type E_T. For simplicity we also refer to the ordered-set as a sequence. The order of events in the set is determined by an index, which starts with 1.

B= { $e_1, e_2, ..., e_n$ }

Each event in the set is denoted by a single variable e_i. The occurrence order of the events is based on a time stamp. For example, e_{i-1} is before e_i and e_{i+1} is after e_i, however, the time intervals between the events may be different.

Definition (Behavior Pattern P): a behavior pattern consists of two parts "static" and "dynamic". The static part defines the structure of the pattern and includes the number of events and the event type. In fact, it represents the common characteristics (or context) among the events. The dynamic part defines the semantic of the behavior through making correlation between the events in the pattern. Furthermore, a behavior pattern includes a set of constraints that must be satisfied by the user behaviors that match the behavior pattern (i.e., pattern instances). These constraints manage the sequence and association among the events. In other words, the semantics of the behavior pattern is represented by a set of constraints that provide association among the events or specify particular conditions for the pattern to be met by the matching user behaviors.

Definition (Feature): a feature "*f*" ($f \in F$ set of features) of the behavior pattern is a relation between events based on attribute values. It is defined by a tuple containing the following elements:

f = <E1, E2, T, V, o> Where:

$E1, E2 \subseteq B$ are ordered-sets of events

T: Event type

V: Ordered-set of attribute domains (different D_a's), where the order of attribute domains in V is the same as the order of attributes in T.

o: An operation (defined below)

Definition (Operation o): an operation "*o*" (where $o \subseteq E_1 \times E_2$) defines a relation between two sets of events. An operation synthesizes a particular feature by defining constraints between event-sets.

Definition (Constraint c): a constraint C_j is defined as a pair $<t_j, R_j>$where $t_j \subseteq A$ is subset of k attributes (from E1 and E2) and R_j is a k-ary relation on the corresponding subset of domains (D_j). In this case, an evaluation of the attributes (i.e., assigning values to attributes) assigns values from the subset of domains (D_j) to a particular set of attributes, such that the relation R_j is satisfied.

3.2 Behavior Pattern Language

Our definition for behavior which is based on a finite ordered-set of events allows our behavior language to represent the user behavior through a set of operating atomic events. Behavior representation and analysis are close to Complex Event Processing (CEP) as an emerging field that attempts to detect event patterns using continuously incoming events based on an abstraction level. Using general-purpose languages such as C or Java for implementation of event processing features causes extra and complex operations in terms of implementing low level functions for events and query structures. Therefore, designing a specific language for event processing has been emerged as an interesting topic for research. Several languages have been introduced as the result of such research in recent years.

To justify why we need a new event-based specification language, we present an example to illustrate the kind of primitive expressiveness and flexibility we need to handle users' behavior patterns. As an example, we consider a hospital information system comprising of several systems in order to provide facilities for authorized care providers (users) to access resources such as medical information, laboratory information, diagnostic images and associated reports. Events representing users' interactions with the resources. Each event is described by a set of attributes. Now, suppose the administrator must be notified in case of misusing the resources. Depending on the context, the application requirements, and the user preferences, the notion of resource misuse can be defined in different ways. Here we present four cases of misuse behavior and use them to show specifications that should be provided by an event processing language. Misuse can occur when:

i. A nurse first reads and then writes a diagnostic report pertinent to a specific patient more than 3 times within 5 minutes.

ii. A user edits the diagnostic reports of a patient in ward-1 in the morning, and again she accesses those resources when she works in ward-2 in the afternoon.

iii. A physician accesses to resources more than twice the number of her assigned patients within a month.

iv. A user sends requests to access the system resources in less than 3 minutes from two different locations whose distance is more than 100 meters.

Such cases apply a set of constraints to select the relevant events from the event stream or dataset. Two kinds of selection constraints have been used. The first kind chooses events based on attribute values and we call it "*attribute parameterization constraint*". For example, case (ii) considers those events that occur in particular

locations, or case (iii) addresses events whose role attribute value is "physician". The second kind of selection constraints operates on a particular relationship among events, namely "*attribute factorization constraint*". For instance, case (i) affects those events that occur within 5 minutes, actually it defines a timing relationship between events in order to select a sequence of events, or case (iv) filters events according to a location proximity. Furthermore, case (i) combines both parameterization and factorization constraints to define iteration that selects those sequences of events that repeating specific actions. Case (ii) implies negation through limiting the occurrence of events in a given interval. Similarly, case (iii) introduces aggregation by applying arithmetic functions on events relevant to a particular role in order to compare the rate of access to resources and the number of assigned patients.

Finally, when the events have been selected, such cases specify which pattern will be created by those events. Moreover, the inner structure of the pattern in terms of length, sequence of events, and their association will be defined by such cases. A language for behavior pattern not only should be simple and unambiguous for translating constraints, but it also should be able to express all the defined constructs: parameterization, factorization, iteration, aggregation, and sequencing.

We propose Behavior Pattern Language (BPL) that allows the analyst to specify precisely the features of the user behavior patterns. The defined operators represent relationships between the behavior elements. The ability to combine events and keep the ordering of events is a basic and powerful specification for the BPL. Moreover, syntactic simplicity and precise semantics are essential features in order to write patterns succinctly. BPL is a language to represent the user behavior through defining a sequence of events. We deal with two kinds of elements in BPL: ordered-sets of events and operations. Operations are descriptions of the constraints for manipulating the ordered-sets of events. The embedded features and operators to BPL improve the

capability of BPL to address any category of behavior.

3.2.1 Features of BPL Language

BPL takes advantage of the following concepts in high-level programming languages, to define different forms of behavior patterns.

Iterator: Most advanced programming languages provide an object called iterator that enables a programmer to traverse a collection of data. Each data item is considered once during a traversal. The iterator handles the progress of the iteration by providing a reference to the next entry in the collection. We provide this capability for BPL through "each" method and "foreach" statement that can manipulate repeatedly the events in an ordered-set.

Block: The structure of block is extensively used to break up large programs into tractable pieces. It is an important tool for organizing the construction of large programs. Blocks can appear immediately after the invocation of a method. Similar to a method, the block can take parameters. We consider that the statements of the block are located between "DO - END" keywords, and the parameters appear at the beginning of the block and between the vertical bars. In the following example, we assume that B is an event-set containing 5 events of type T that uses an iterator (each method) and a block to assign values to attribute Role for all events.

```
1 Example:
2       B = {e1, e2, e3, e4, e5};
3       B.each DO |event|
4             event.Role = "Nurse";
5       END
```

Module: A module is a programming concept which allows for reusability of code through grouping together methods, classes, and constants. To facilitate the representation of different kinds of patterns, we define a series of methods called operators

to manipulate events. These built-in operators can be organized as modules and then can be utilized to represent different categories of behavior patterns.

Yield: A common use of yield in a programming language is to transfer the control from one point of the code to another point of the code. This capability enables programmers to make a customizable form of an iteration. However, in some situations, it needs to implement or reuse a custom functionality inside a common functionality. We leverage the feature of yield construct in Ruby to implement constraints as custom functionality inside operations as common functionality. In other words, we use yield to separate the parts of the representation that demonstrate the semantics of the pattern from those that define the structure of the pattern.

In Example 1 below, we apply the mentioned features in order to define an operation whose task is to assign values to two attributes of every event in the ordered-set B consisting of 5 events. When operation "OP" is called via "CALL OP (B)", the ordered-set B is passed to it as parameter. In definition of "OP" the keyword "each" plays the role of an iterator and fetches individual events from the ordered-set B repeatedly and passes them to a block as parameters. The block contains a "YIELD" statement which passes each event "e" back to another block that is defined after "CALL OP (B)". This second block then assigns the attributes "User" and "Role" of each event to "John" and "physician". After executing the statements of the second block, the control returns to the "OP" and this procedure repeats for the next event.

```
1  --Example 1;
2  B = {e1, e2, e3, e4, e5};
3  CALL OP (B) DO |event|
4     event.User ="John"
5     event.Role ="physician"
6  END
7
8  DEF OP (set)
9     set.each DO |e|
10    YIELD (e)
11    END
```

```
12 END
```

Below is another example that demonstrates the functionality of yield so that the common functionality is handled by "method2" and custom functionality is handled by the block defined in "method1".

```
1  DEF method1
2     CALL  method2   DO  |b|
3     FOREACH event IN b  DO
4     event.Action = "write";
5     END
6     END  // end of block
7  END  // end of method1
8
9  DEF  method2
10    B = {e1, e2, e3};
11    YIELD (B)
12 END
```

When method1 is invoked, it calls method2. method2 starts executing until yield is reached. At that point control along with the parameter B is transferred to the block that was defined in method1. The block is executed. The value of attribute "Action" changes to "write" for all events. Then, the control returns to method2.

Operation and Operator: What makes a user behavior pattern distinct from another pattern is the properties that exist in the sequence of events representing the user behavior pattern. Correlations between events convey the meaningful features of the pattern. The analyst can show the features of the pattern through defining particular operations on the ordered-set of events. Assigning events to event-sets depends on what events participate in the association and how many event-sets are needed. We consider at most two ordered-sets can be operated in each operation because unary and binary operators will be used in an operation. To represent a feature, the analyst is allowed to adjust the sets in order to define appropriate operations. To specify the relation among events inside an event-set or between two event-sets, we

define several unary and binary operators (Table 3.2). The defined operators can be grouped in three categories.

- The first category containing "NEW" and "INITIALIZE" is intended to define the structure of the pattern. The operator "NEW" creates sufficient placeholders for corresponding events in the pattern. Events in a pattern have the same type, so the placeholders have the same size and structure. The operator "INITIALIZE" assigns the value to a particular attribute for all events in an event-set.

- The second category of operators are used to manipulate the ordered-set of events. The operator "VALUEOF" returns the value of an attribute for a particular event or for all events. Joining two event-sets can be performed by operator "UNION" so that the result of union is a new ordered-set of events. The operator "FOLLOWEDBY" is useful to change the sequence of events in an event-set. The specification of a pattern can be expressed by constraints. Not only do constraints define the correlation between events, but some kind of constraints also affect the whole pattern. The appropriate operations manage the constraints that make correlation between events. However, to handle the second kind of constraints that limit some features of the whole pattern, we use the built-in operators or the combination of operations and operators. In BPL, we also utilize the assertion mechanism (by using keyword "ASSERT") to enforce these kinds of constraints.

- The third category of operators include "DISTANCE", "FREQUENCY" and "EQUALITY" that are designed to apply constraints. Calculating the difference of two attribute values can be used to define particular attribute-based constraints. We define the operator "DISTANCE" in order to calculate the

distance between values of attributes whose types are time or location. It computes the distance of attribute values for each pair of event. It returns an array as the result of difference. For example, we want to enforce three events e1, e2, e3 occur within 2 hours. The DISTANCE (e1, e2, e3, Time) returns the following array that the value of each element should be less than 2.

$[\,(e1.Time - e2.Time), (e1.Time - e3.Time), (e2.Time - e3.Time)] \leq 2$

The operator "FREQUENCY" enforces the number of times a set of events (representing a sequence of actions) should be repeated in the dataset.

3.2.2 Syntax and Grammar of BPL

The BPL code consists of five parts to represent a typical behavior pattern. Figure 3.2 illustrates the structure of a BPL code specification. The declaration is used to describe different entities existing in the pattern. These entities include: attributes, events, sequence of events and some parameters that specify the structure of the pattern. In other words, the skeleton of pattern will be defined in the declaration section, then the pertinent operations and constraints will be called to establish the meaning of the pattern.

The methods in the operation section are used to define the relation between two ordered-sets of events that are chosen by the analyst. These ordered-sets are subsets of the sequence of events that represent the pattern. The BPL allows the analyst to choose at most two sets because the defined operators are unary or binary. The relation can be defined through applying different statements and operators to manipulate attributes and attribute values. If the analyst intends to define particular restrictions for the operation, the relevant assertions will be defined in the constraint section. The methods in the operation section interact with the constraints via the

Table 3.2: The description and syntax of operators

Operators	
Operator	**Description**
NEW	*NEW event-type (number of event)*, returns an event-set that have same type
INITIALIZE	*INITIALIZE (ordered-set of events, attribute, set of value)*, initializes an attribute for all events in the set. The cardinality of event-set and value set is the same.
VALUEOF	*VALUEOF (ordered-set of events or an event, attribute)*, returns the value of attribute for each event in the set or for a particular event.
UNION	*UNION (ordered-set of events, ordered-set of events)*, makes the union of two ordered-sets. The result is an ordered-set as well.
FOLLOWEDBY	*FOLLOWEDBY (event, event)*, the first event is followed by the second event.
DISTANCE	*DISTANCE (ordered-set of events, attribute)*, calculates the distance between attribute values of all generated pairs of events. It can be used for attributes whose type is time or location. The return result is an array containing the difference between attribute values of pairs of events.
FREQUENCY	*FREQUENCY (ordered-set of events)*, returns the frequency of the ordered-set in the dataset.
EQUALITY	*EQUALITY (ordered-set of events, attribute)*, returns true if events have the same value for the attribute.

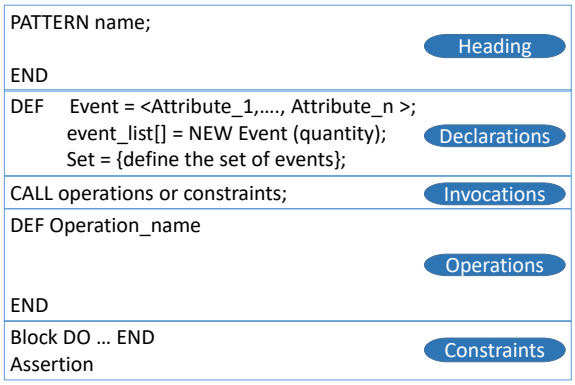

Figure 3.2: The structure of a BPL code to describe a typical behavior pattern

provided language constructs such as block and yield.

Extended Backus-Naur Form (EBNF) is used to describe the syntactic structure of BPL. Appendix A presents the syntax of the BPL.

3.2.3 Semantics

Semantics provides the meaning of the well-formed statements of a language. Semantics should be preserved by language implementation applications such as interpreter or compiler. The syntax is specified by a context-free grammar such as Backus Naur Form (BNF) and its variations. However, there is not a single definition method to explain the semantics. An informal way (i.e. example or text) is usually used to specify the semantics of the language constructs [46].

Operational semantics presents the meaning of a program in terms of how it can be interpreted by a machine. In other words, the operational semantics provides a translation mechanism to convert a program to an equivalent program in another language which is simple for understanding. Plotkin [47] introduced the operational semantics in order to describe language semantics in terms of a state transition system

as a mathematical tool. Such view makes the operational semantics similar to an interpreter.

The operational semantics is defined by state transitions of an abstract machine. An abstract machine consists of four main elements: control stack (c), result stack (r), processor and memory. Instructions are stored in the control stack and the intermediate results are kept in the result stack. A processor performs operations such as comparisons, arithmetic and Boolean. Variables and values are stored in a memory modelled by a function called m. $dom(m)$ denotes the set of locations where m is defined. $m(l)$ denotes the value stored at location l and $m[l \mapsto n]$ maps l to the value n. Formally, an abstract machine is defined by a set of states (configurations) along with transition rules. A state s is defined by a triple $\langle c, r, m\rangle$ of control stack, result stack, and memory. A transition rule is a binary relation between two states.

Definition 1 (semantics of expressions): The value generated by the machine after zero or more transitions for an expression E is considered as the semantics of E [46].

$\langle E.c, r, m\rangle \xrightarrow{*} \langle c, v.r, m'\rangle$

$E.c$: denotes the expression of E on top of the control stack

$v.r$: denotes the value of E on top of the result stack

$\xrightarrow{*}$: denotes zero or more transition steps

Representing the semantics of a language through using abstract machine needs to write a large number of transition rules in detail. Although it is useful for the implementation of the language, it is too detailed for the users of the language. Therefore, we applied structural operational semantics proposed in [47] based on the transition system. In structural operational semantics, the transition rules for a compound statements are inductively defined based on transition rules of substatements.

There are two methods to represent the structural operational semantics [46]: (1)

small-step semantics, based on a reduction relation, and (2) big-step semantics, based on an evaluation relation. In the first method, a transition relation is inductively defined by a set of axioms and rules between states. Axioms are used for reduction whereas rules are applied to specify how to generate new reduction steps based on ones that have been already defined.

The big-step semantics associates each state with its result and abstracts from the details of the evaluation process. The transition systems have the same set of states in big-step semantics and small-step semantics, but the transition rules are different. Transition rules represent individual computation steps in the small-step semantics and full evaluation in the big-step semantics. We use the big-step operational semantics to explain precisely the features of BPL. The transition relation is denoted by $\Downarrow$.

Variable and Constant: We define Rule 3.1 and 3.2 to describe the meaning of constant and variable respectively.

$$\langle c, s\rangle \Downarrow \langle c, s\rangle; if\ c \in \mathrm{D_a}\ \cup\ \{True, False\} \tag{3.1}$$

$\mathrm{D_a}$ denotes domain value for an attribute a. c represents the constant. s shows the state of the transition system.

$$\langle *l, s\rangle \Downarrow \langle v, s\rangle; if\ s(l) = v \tag{3.2}$$

l is a location of memory (variable) and $*l$ denotes the content of the location l and $s(l)$ represents the value of l in the state s.

Operator: The meaning of a statement that uses a binary operator such as arithmetic or comparable is defined by Rule 3.3. The rule says that for an expression (e.g $E1$ *op* $E2$) containing a binary operator, first the value of E1 and then the value of E2 are computed. The final result is generated by applying the operator on both

values.

$$\frac{\langle E1, s\rangle \Downarrow \langle n1, s'\rangle \;\; \langle E2, s'\rangle \Downarrow \langle n2, s''\rangle}{\langle E1 \; op \; E2, s\rangle \Downarrow \langle n, s''\rangle \; if \; n = n1 \; op \; n2} \tag{3.3}$$

Iteration: Iterative constructs cause a statement or sequence of statements (the body of the loop) to be repeated for each item in a collection. BPL uses *foreach* and *each* for iteration whose abstract syntax is: *foreach A do B*; where A is a set whose elements will be fetched one by one and B is the body that should be executed for each element. Rule 3.4 shows the semantics of the iteration. The rule indicates that when an element is successfully returned, the state of program will be changed to s' and the returned element will be saved in location l. The statements in B can use the stored element in the state s' and the execution of statements changes the state to s''. The *skip* means the control is successfully passed to the next statement. Finally, the iteration started in the state s terminates successfully producing s'''.

$$\frac{\langle A, s\rangle \Downarrow \langle true, s'[l \mapsto v]\rangle \;\; \langle B, s'\rangle \Downarrow \langle skip, s''\rangle \;\; \langle foreach \; A \; do \; B, s''\rangle \Downarrow \langle skip, s'''\rangle}{\langle foreach \; A \; do \; B, s\rangle \Downarrow \langle skip, s'''\rangle} \tag{3.4}$$

Operation: BPL calls operations to represent relationships between the behavior elements (events). Each operation has a name, definition and may receive several parameters. We represent abstractly an operation in BPL as an expression of the form $f(x_1, \ldots, x_k) = d$ where $x_1, \ldots, x_k$ denote the parameters and d represents the body of the operation f. Rule 3.5 indicates how an operation is interpreted. First, the arguments of the operation are evaluated and then values are used in the definition of the operation. In other words, the parameters in the body of the operation are replaced with the values of the arguments. The *skip* in the rule shows the control returns to the next statement after calling the operation. We assume that arguments are sent by the reference. Consequently, the results of the operation are directly

assigned to the address (l_i) of arguments.

$$\frac{\langle x_1, s\rangle \Downarrow \langle v_1, s'\rangle \ldots \langle x_k, s\rangle \Downarrow \langle v_k, s'\rangle \quad \langle d[x_i \mapsto v_i], s'\rangle \Downarrow \langle skip, s''[l_i \mapsto v_i']\rangle}{\langle f(x_1, \ldots, x_k), s\rangle \Downarrow \langle skip, s''\rangle} \tag{3.5}$$

Block: Unlike the operation, the block does not possess a name however it can accept parameters. We consider the block as a sequence of statements between keywords *do* and *end*. We define two rules to give a precise meaning to the construct of the block. Rule 3.6 shows the assignment of the actual value (argument) v to the parameter p. The assignment will be done by a reference (pointer) so that the value of *ref* p is a new location whose content is the value of p.

$$\frac{\langle ref\ p, s\rangle \Downarrow \langle l, s'[l \mapsto v]\rangle}{\langle p, s\rangle \Downarrow \langle v, s'\rangle} \tag{3.6}$$

Rule 3.7 shows the meaning of the block definition. According to Rule 3.6, the parameter is referenced to the argument and then the value of argument (content of reference) is assigned to the parameter for each command C. After executing the command, control will be transfered ($skip$) to the next command and the content of the reference will be updated.

$$\frac{\langle p, s\rangle \Downarrow \langle v, s'\rangle \quad \langle C[p \mapsto v], s'\rangle \Downarrow \langle skip, s''[l \mapsto v']\rangle}{\langle DO\ p\ ;\ C\ END, s\rangle \Downarrow \langle skip, s''\rangle} \tag{3.7}$$

Yield: Using *yield* operator was inspired by Ruby programming language so that such feature allows programmers to write code succinctly and efficiently. An operation can be called with parameters as well as a block that itself may possess parameters. We represent abstractly the block as a tuple $\langle p, b\rangle$. p is the parameter and b is the body of the block. Such a block will be invoked by a yield statement defined inside of the operation definition. The values of block's parameters will be provided by yield statement. We assume that the operation call happens in the state s_1 as well as all parameters and statements of the block will be kept at this state. The operation

starts running until it reaches the *yield*. Rule 3.8 shows the assignment of the actual value v to the parameter a. We consider a as the parameter of the *yield*.

$$\frac{\langle ref\ a, s\rangle \Downarrow \langle l, s'[l \mapsto v]\rangle}{\langle a, s\rangle \Downarrow \langle v, s'\rangle} \tag{3.8}$$

Rule 3.9 indicates that the address of the block's parameter will be mapped to the same address containing the value of the parameter a. Then, the control will be jumped to the state s_1 including the body of the block.

$$\frac{\langle a, s\rangle \Downarrow \langle v, s'\rangle \quad \langle ref\ p, s'\rangle \Downarrow \langle l, s_1[l \mapsto v]\rangle}{\langle yield, s\rangle \Downarrow \langle jump, s_1\rangle} \tag{3.9}$$

The parameter of the block will be replaced with the reference to the actual value and Rule 3.10 shows after executing the body of the block, the control will be transfered (jump) to the state s' and the content of the reference will be updated.

$$\frac{\langle p, s_1\rangle \Downarrow \langle v, s_1'\rangle \quad \langle b[p \mapsto v], s_1'\rangle \Downarrow \langle jump, s'[l \mapsto v']\rangle}{\langle yield, s\rangle \Downarrow \langle skip, s''\rangle} \tag{3.10}$$

3.3 Behavior Pattern Examples

Behavioral patterns describe interactions between users and resources in the system. An interaction consists of a set of single events that are somehow related to each other. Each event will be represented by a set of concrete attributes and values. Therefore, such a behavior pattern can present the behavior of a single user or a group of users based on the granularity level of the analysis.

Sequence, association and combination of the events are important relations that can define different categories of behavior patterns in the real world. Sequence is handled by temporal attributes of events. Therefore, BPL always uses an ordered-set of events to represent a behavior pattern and the index of event indicates the order of occurrence of events. Applying constraints on attribute values, usually makes a

relationship between events. For example, a proximity constraint on attributes time and location can represent patterns occur in a particular period of time or within a specific location. BPL has enough capability to model the proximity of events based on a location or range of time. Example 1 and Example 2 demonstrate the capability of BPL to represent the proximity feature of behavior pattern.

Example 1:

We specify the proximity of events based on a time period so that the BPL code 3.1 describes the behavior pattern of a user in any role who first reads and then writes the patients' diagnostic reports more than 10 times from 9am to 10am.

Code 3.1: A proximity of events based on a time period

```
PATTERN Example1;
BEGIN
   DEFINE Event = < User, Role, Action, Resource, Time >;
   E[] = NEW Event (2);
   set1 =  { E[1] };
   set2 = { E[2] };
   attVal = {<Action, ("read","write")>, <
   Resource, ("diagnostic report")>, <Time, (9:00, 10:00)>};
   CALL Op1(set1, set2, attVal);
END

/* define operation and relevant constraint */
DEF Op1 (set1, set2, attVal )
   FOREACH event IN Set1 DO
      Action = attVal[Action].value[0];
      Resource=attVal[Resource].value[0];
      ASSERT Time >attVal[Time].value[0];
      ASSERT Time <attVal[Time].value[1];

   FOREACH event IN Set2 DO
      Action = attVal[Action].value[1];
      Resource=attVal[Resource].value[0];
      ASSERT Time >attVal[Time].value[0];
      ASSERT Time <attVal[Time].value[1];

   ASSERT VALUEOF (set1, User) == VALUEOF (set2, User);
   ASSERT FREQUENCY (UNION (set1, set2)) > 10;
```

```
28 END
```

Example 2:

We specify the proximity of events based on location and time span so that the BPL code 3.2 describes the behavior pattern of a physician who can read the patient's diagnostic report from two different wards in a hospital within 2 hours. We assume that the distance between two wards is less than 100 meters.

Code 3.2: A proximity of events based on location and time span

```
1  PATTERN Example2;
2  BEGIN
3     DEFINE Event = < Role, Action, Resource, Location, Time >
4     E[] = new Event (2);
5     set1 = { E[1], E[2] };
6     attVal = {<Action,("read")>, <Resource, ("diagnostic
         report")>,<Role,("physician")>, <Location, ("L1","L2")
         >};
7
8     CALL Op2 (set1, attVal) DO |event|
9        event.Role = attVal[Role].value[0];
10       event.Action =attVal[Action].value[0];
11       event.Location = attVal[Location].value[0] OR
12       attVal[Location].value[1];
13       event.Resource=attVal[Resource].value[0];
14    END
15    ASSERT DISTANCE (set1, Time) < 2;
16    ASSERT DISTANCE (set1, Location) < 100;
17 END
18
19 /* define operation */
20 DEF Op2 (set1, attVal)
21    set1.EACH DO |e|
22       YIELD (e)
23    END
24 END
```

Example 3:

Some behavior patterns include events that can occur in a particular context such as location, time, etc. Such events share attribute values with some other events. As an

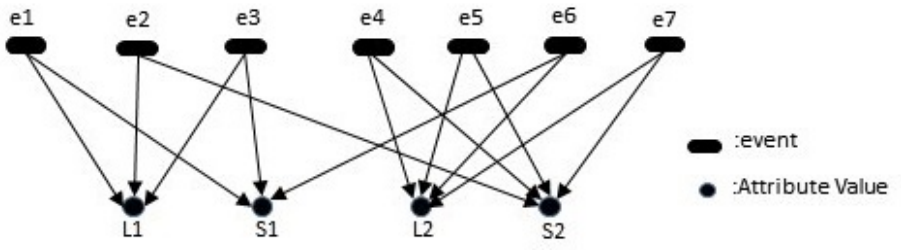

Figure 3.3: Association between events

example, we consider a physician who works in two different clinics (L1 and L2), but he can read the resources from every clinic. A behavior pattern of this user has been shown in Figure 3.3. Event e2 that is occurred in location L1 can read Resource S2 in location L2. When the user as a physician works in L2 also can read resource S1 located in L1. BPL code 3.3 describes the structure of the pattern and corresponding constraints.

Code 3.3: Describes the association between elements of a pattern

```
1  PATTERN Example3;
2  BEGIN
3     DEFINE Event = <User,Role,Action,Resource,Location >
4     E[] = NEW Event (7);
5     set1 = { E[1] ..  E[3] };
6     set2 = { E[4] ..  E[7] };
7     attVal = {<Action,("read")>, <Resource, ("S1", "S2")>, <
         Location, ("L1","L2")>, <Role, "physician">};
8     CALL Op1 ( set1 , set2, attVal) DO |event, Lvalue,
         Rvalue, Avalue|
9        event.Location = Lvalue;
10       event.Role = Rvalue;
11       event.Action = Avalue;
12    END
13    set1 = { E[1], E[3], E[6] };
14    set2 = { E[2], E[4], E[5], E[7] };
15    CALL Op2 ( set1 , set2, attVal) DO |event, value|
16       event.Resource = value;
17    END
18 END
19 /* define operation */
```

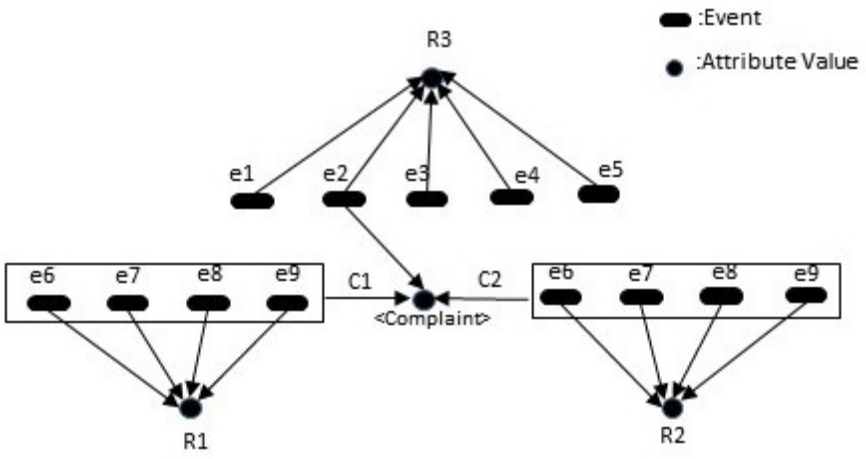

Figure 3.4: Grouping of events based on a particular attribute value

```
20 DEF Op1 ( set1 , set2, attVal)
21    Set1.EACH DO |item1|
22       YIELD (item1, attVal[Location].value[0],
23       attVal[Role].value[0], attVal[Action].value[0] )
24    END
25    Set2.EACH DO |item2|
26       YIELD (item2, attVal[Location].value[1],
27       attVal[Role].value[0], attVal[Action].value[0])
28    END
29 END
30
31 DEF Op2 ( set1 , set2, attVal)
32    Set1.EACH DO |item1|
33       YIELD (item1, attVal[Resource].value[0])
34    END
35    Set2.EACH DO |item2|
36       YIELD (item2, attVal[Resource].value[1])
37    END
38 END
```

Example 4:

Separation is a category of behavior patterns in which events can be divided in some groups. Meanwhile, the groups affect each other based on particular attribute values. To describe a separation behavior pattern, we consider a simple workflow in a clinic to treat a patient. The triage nurse's duties typically consist of measuring

Table 3.3: Events and actions for Example 4

EventID	**Action**
e1	A1:measuring the vital signs
e2	A2:recording the main complaint
e3	A3:documenting in EHR
e4	A4:assigning an examination room
e5	A5:notifying the specialist
e6	A6:taking the patient's history
e7	A7:performing an examination
e8	A8:making a diagnosis
e9	A9:writing a prescription

the pertinent vital signs and identifying the main complaint. She documents the conditions of the patients and assigns an examination room to the patient. She selects and notifies a specialist based on the main complaint. The specialist is responsible for taking the patient's history, performing an examination, making a diagnosis, and writing a prescription. After all ordered services have been completed; the patient is discharged from the clinic.

Table 3.4: Complaint and relevant role for Example 4

Complaint	**Role**
C1	R1: Orthopedist
C2	R2: Urologist
	R3: Nurse

In this example, the actions for treatment of a patient are divided into two groups

of sequential events, which are generated by two different roles: nurse and specialist. The attribute value of role in the second group is defined by user who has specific role in the first group, and it is assigned based on the value of the attribute “complaint”.

Code 3.4: Describes the relation between separated groups of events

```
PATTERN Example4;
BEGIN
   DEFINE Event = <User,Role,Action,Patient,Complaint>
   E[] = new Event (9);
   set1 = { E[1], E[2], E[3], E[4], E[5] };
   set2 = { E[6], E[7], E[8], E[9]};
   attVal = {<Role, ("R1", "R2","R3")> }
   Call Op1 (set1, attVal) DO |event|
      event.Role=attVal [Role].value[0];
  END

   Call Op2 (set1, set2) DO |event, v|
      event.Role=v;
  END
END

/* Define Operation */
DEF Op1 (set, attVal)
   set.EACH DO |e|
      YIELD (e)
   END
END

DEF Op2 (set1, set2)
   Set2.EACH DO |e|
     IF (VALUEOF (set1.event[2], Complaint)) == "C1"
      THEN
        YIELD (e, "R1")
     IF (VALUEOF (set1.event[2], Complaint)) == "C2"
      THEN
        YIELD (e, "R2")
   END
END
```

Chapter 4

Access Control Obligation

Access control mechanism is an acceptable approach in the security domain to allow legitimate users to access different resources in a secure manner. Typically, access control systems deal only with authorization decision on users request to access the target resources. Controlling the activities of authenticated and authorized users is not in charge of the traditional access control systems after granting permission. Furthermore, traditional access control models such as Mandatory Access Control (MAC), Discretionary Access Control (DAC), and Role-Based Access Control (RBAC) do not work properly in pervasive computing environment because such an environment is dynamic and sensitive to contextual information. Moreover, pervasive computing applications usually do not have well-defined security perimeters [48].

Different kinds of policies are required to protect resources in sensitive application domains. Authorization policies represented by a set of rules protect resources through making appropriate access decisions. However, such authorization rules cannot control the activities of the user after granting the access. Integrating obligations into access control policies can dictate the mandatory operation that must be performed by the user before, after, and during the access [49].

Obligations are increasingly being expressed explicitly as part of security policies as opposed to being hard-coded into applications. Supporting obligations in policy increases the flexibility of the management and easy maintenance of the security system. Such a combination allows a system to react to new circumstances quickly and change obligation requirements easily when flaws in existing policies are discovered. Meanwhile, integrating obligations into a security policy is an important part of translating high level security objectives into low level policies. Obligation as an important part of many access control policies is defined by XACML as follows: "An operation specified in a rule, policy or policy set that should be performed by the PEP in conjunction with the enforcement of an authorization decision."

We define the obligation as an obligatory behavior that should be conducted by an entity. Obligation is classified in two perspectives: functional and structural [50]. Functional obligation represents what a subject is obliged to perform with respect to changing or maintaining the state of affairs. In this view, obligation is a commitment that represents a promise to do something or to behave in a particular manner. Structural obligation represents what a subject is obliged to perform in order to fulfill a responsibility such as directing, supervising and monitoring. In this view, obligation is a duty that a subject must perform and it is not specifically part of the access control purpose.

Using obligations to handle the usage of resources raises the question of how the enforcement will be managed. Since traditional access control is performed only once before the resource is accessed by the subject, the access control system is not able to control the fulfillment of obligation while the access is in progress. Therefore, the lack of capability to check the obligations for the continuity of an access purpose may lead to a security breach in sensitive environments. To resolve this issue, we proposed to utilize Session Initiation Protocol (SIP) in order to handle the enforcement of

obligations. SIP is an application-layer control protocol for creating, modifying, and terminating sessions with one or more participants [51].

In this chapter, we propose an approach to integrate obligations into an access control system in order to control the behavior of the user. In Section 4.1, we present a model for different kinds of obligation based on the behavior of the user. We discuss how to integrate the obligation into the access control policy language in Section 4.2, followed by our proposed approach for enforcing the obligation in Section 4.3.

4.1 Obligation Modeling

In access control perspective, the obligation is an operation that should be performed before or after making the authorization decision. It also can be performed during the access time. It means that some planned actions should be fulfilled before or after access decision, or when the access is in progress. The decision can be "permit" or "deny". We consider an obligatory behavior as an obligation that should be conducted by a subject. The following features specify different aspects of the obligation:
Type: obligations are categorized into three types based on when they become effective. (i) pre-obligation specifies actions that should be fulfilled before the access of resources is allowed; (ii) ongoing obligation specifies actions need to be fulfilled during the access session (the usage of resources); and (iii) post-obligation specifies actions required to be performed after accessing resources.
Subject: the subject of an obligation is a person (in the case of user obligation) or a component of the system (in the case of system obligation) that fulfills the obligation.
Object: the obligation object presents to whom the obligation must be applied. The obligation object may not be the same object of the access request. The object is a finite set of resources or state of circumstances.

Operation: the subject must perform a mandatory operation. Practically, the operation is relevant to accessing the resources or its execution affects the state of circumstances. The word "operation" comes from the definition of obligation in XACML standard [38] and we define it as a set of planned actions to achieve a particular purpose. Performing the operation by the subject generates a sequence of events that we call it as the behavior of the subject. Actually, the concept of the obligation dictates the behavior of the subject in a particular situation.

Violation: obligations may include requirements to specify that the fulfillment of the obligation should satisfy particular stateful constraints such as frequency, deadline, location, etc.

According to the aforementioned features, we consider formally the obligation as the behavior of the subject who is in charge of performing the obligatory actions.

Definition (Obligation): An obligation is represented as a tuple of features that describe the characteristics of the obligatory behavior so that Policy Enforcement Point (PEP) is responsible to approve the fulfillment of such a behavior. It is defined by a tuple containing the following elements:

$obl = <w, s, b, o, v>$ Where:

w: represents the obligation fulfillment time

s: subject

b: behavior

o: object

v: condition

The subject s should conduct the behavior b on the object o in order to satisfy the stateful conditions v before, after, or during the access. The defined model for the obligation helps to express the complex obligation specifications in terms of defining a sequence of actions and corresponding conditions.

Specifying precisely the obligatory behavior of the user is a challenge for the policy administrator. The behavior should be described in great detail since PEP defines appropriate compliance checkpoints based on predefined policy and described behavior in order to approve the fulfillment of the obligation. We utilized the Behavior Pattern Language (BPL) to describe the characteristics of such a behavior. BPL allows the access control policy administrator to illustrate behavior patterns and their primitive elements, attributes, and constraints in a detailed and precise manner.

4.2 Integrating Obligation into Policy Language

XACML is a standard developed by the OASIS organization [38] for expressing syntax and semantics of access control policies. It provides a language, a reference architecture and the processing model to describe how to evaluate an access request. Briefly, the XACML promises to be an expressive and flexible policy language and architecture for heterogeneous and dynamic environment.

XACML is totally different from authentication and authorization standards (such as OpenID [52], OAuth [53], etc.) in terms of purpose and application. XACML is not responsible to handle identity management, user approval or delegated access. The main purpose of using XACML is to provide a mechanism to drive the access control logic through policies.

XACML as a de facto standard for the access control separates the policy specification from decision and enforcement mechanism. Such a separation will ease the implementation and validation of the access control as well as decision and enforcement mechanism can be reused across different systems. Separating the different points of the access control in the XACML helps to create a fine-grained access control, however it increases the level of semantic gap between desired behavior (requirement)

and implemented behavior. Misunderstanding of the policy specification, complex interactions between policy and business logic, and programming errors in developing different points of access control are the main reasons that may result in the incorrect implementation of an obligation policy. Any defect in access control software leads to policy violations such as unfulfilled obligation, granted improper permission, and finally security breach through unauthorized access.

A few policy languages support the obligation concept among existing policy languages including proprietary languages. We focus on XACML in this research because of the following reasons:

- Standard: XACML is an open standard access control language that was ratified by the Organization for the Advancement of Structured Information Standards (OASIS). The language and the proposed architecture were revised and approved by the membership of OASIS that creates a large community of experts.

- Distributed: XACML allows policy statements be written by several policy writers and enforced at several enforcement points in a distributed system. The proposed architecture separates the duty of administration point, decision point, and enforcement point. Such a separation facilitates the utilization of XACML in a distributed environment. Meanwhile, XACML is capable of combining the result of evaluating different effective policy rules by using combining algorithm in order to prevent from conflicts.

- Extensible: Using XML in XACML provides enough capability to extend the existing functions and combining algorithms in order to accommodate different requirements. It also supports to develop new functions, data types, attributes, etc. Moreover, several profiles and standard extensions published recently such as REST profile, JSON Profile, Additional Combining Algorithms Profile, Data

Loss Prevention profile for the use of XACML in expressing policies for different domains.

- Generic: XACML is a XML-based language designed to express security policies and access rights to information for Web services, resource management, and enterprise security applications such as firewalls. It does not rely on a proprietary application domain. Meanwhile, using XML enables it to attract the widespread support from the community so that different implementations of XACML has been deployed as open source applications.

The XACML specification document provides a data flow model that shows the interaction between major components (Figure 4.1). XACML does not limit the type of the protocol for communication or the number of repositories. Practically, the communication between entities can be facilitated by using a repository. For example, using a repository simplifies the communications between the context handler and the Policy Information Point (PIP) or the communications between the Policy Decision Point (PDP) and the Policy Administration Point (PAP). The data flow model operates by the following steps [38]:

1. The PAP writes policies and policy sets for a specified target and makes them available to the PDP.

2. The requester sends a request for access.

3. The Policy Enforcement Point (PEP) sends the request along with attributes of the subjects, resource, action and environment to the context handler in its native request format.

4. The context handler creates an XACML request context by adding more attributes and sends it to the PDP

5. The PDP may request any additional information related to subject, resource, action and environment as attributes from the context handler.

6. The context handler requests the attributes from a Policy Information Point (PIP).

7. The PIP collects values for the requested attributes.

8. The PIP returns the requested attributes to the context handler.

9. The context handler includes optionally the resource in the context.

10. The PDP evaluates the policy by using attribute values collected by the context handler.

11. The PDP returns the authorization decision along with obligation to the context handler.

12. The context handler returns the response and obligation to the PEP.

13. The PEP sends the obligation to the obligation service that is responsible to fulfill the obligation.

14. The PEP enforces the authorization decision that can be "permit" or "deny" for access.

The XACML technical committee provides a schema for the XACML policies. Figure 4.2 visualizes the structure of the access policy in XACML. Typical access policies are structured as Policy Sets, Policies, and Rules. A Policy Set contains Policies or optionally other Policy Sets. A Policy consists of a Target, which denotes the target of the policy, a set of rule elements which construct the authorization rule, and a rule combining algorithm. The Target specifies the Subjects, Resources,

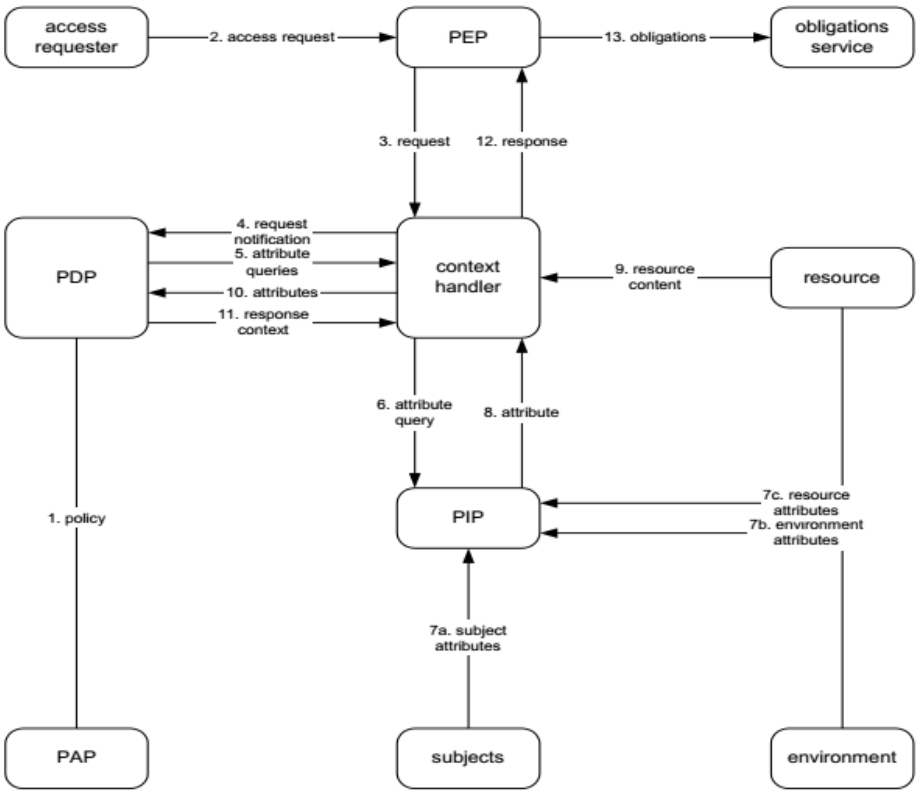

Figure 4.1: XACML data-flow diagram

and Actions on which a policy can be applied. Actually, a Target defines a simple boolean condition that should be satisfied by the attributes value. A Rule includes three main components: Target, Condition, and the Effect of the rule which can be either Permit or Deny. The Target specifies for which kind of access requests the rule can be applied. If no Target matches the request, the decision computed by the PDP is NotApplicable. The Condition elements denote predicates in order to express relations or computations on attribute values. A rule is applicable to an access request if the target of the access request matches the target of the rule and if all the conditions included in the rule are satisfied. Meanwhile, the predefined combining algorithms are used to resolve the conflict among applicable rules during the decision making. Conflict may happen because a policy may contain multiple rules or a policy

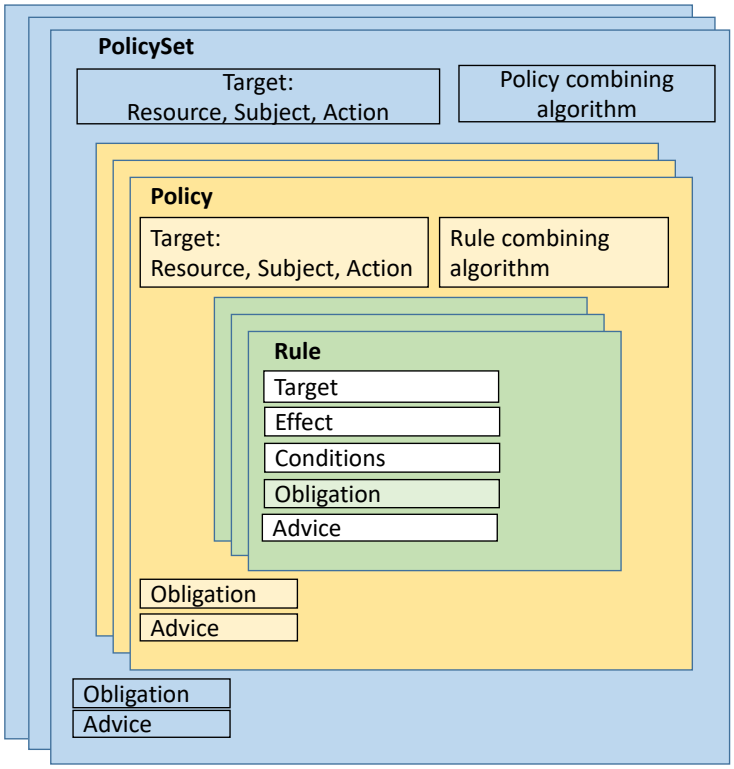

Figure 4.2: The structure of XACML policy language

set may contain multiple policies so that each rule or policy may evaluate to different decisions (Permit, Deny, NotApplicable, or Indeterminate). XACML provides a set of standard combining algorithms to reconcile these individual decisions. Some examples of standard combining algorithms are as follows:

- Permit-unless-deny: if any decision is Deny, the result is Deny, otherwise, the result is Permit.
- Deny-unless-permit: if any decision is Permit, the result is Permit, otherwise, the result is Deny.
- First-applicable: the result is the result of the first decision (either Permit, Deny, or Indeterminate) when evaluated in their listed order.

In addition to the standard set of combining algorithms, XACML includes a standard extension mechanism that can be used to define customized algorithms.

XACML includes the concepts of obligation and advice expressions. An obligation optionally specified in a Rule, Policy, or PolicySet is an operation that must be performed in conjunction with policy evaluation by PEP. An obligation in a rule will be specified by using an ObligationExpressions element to determine action that should be performed after enforcing the authorization decision. Advice is similar to an obligation, except that advice may be ignored by the PEP.

The snippet Code 4.1 of the XACML shows the representation of an obligation that forces the physician to send an email with the content of "Your medical record has been accessed by: physician-ID " to the patient. ObligationId attribute, the authorization decision value, and a set of attribute assignments make the body of <ObligationExpression> element. The FulfillOn attribute defines the authorization decision value (permit or deny) for which the obligation must be fulfilled. There are three AttributeAssignmentExpression elements (lines 6, 15, 21) that provide information for PDP and PEP to assign value to attributes in this example. The first one indicates where the PEP will find the email address in the resource. The second one defines the content of the email. The third one indicates further information for the body of the email in order to provide the subject-ID. We will extend this basic structure by adding more elements in order to express the features of more complex obligations.

Code 4.1: The basic structure of presenting obligation in XACML

```
1<ObligationExpressions>
2  <ObligationExpression
3    ObligationId="obligation:email"
4    FulfillOn="Permit">
5
6    <AttributeAssignmentExpression
```

```
7       AttributeId="attribute:mailto">
8       <AttributeSelector
9         MustBePresent="true"
10        Category="resource"
11        Path="md:record/md:patient/md:patientContact/md:
              email"
12        DataType="XMLSchema#string"/>
13    </AttributeAssignmentExpression>
14
15    <AttributeAssignmentExpression
16       AttributeId="attribute:text">
17       <AttributeValue DataType="XMLSchema#string"
18       >Your medical record has been accessed by:</
             AttributeValue>
19    </AttributeAssignmentExpression>
20
21    <AttributeAssignmentExpression
22      <AttributeDesignator
23        MustBePresent="false"
24        Category="access-subject"
25        AttributeId="subject:subject-id"
26        DataType="XMLSchema#string"/>
27    </AttributeAssignmentExpression>
28  </ObligationExpression>
29</ObligationExpressions>
```

Regardless of the fact that XACML has salient features to express the elements of an attribute-based access control, the lack of a powerful structure makes difficult for a policy administrator to express precisely a complex behavior as an obligation. XACML tries to make the obligation as an integral part of access control policies, however the current version of the XACML lacks the enough capabilities to manage properly the obligations. For example:

1. The current version of XACML lacks well-defined syntax to express a sequence of actions that the subject obliges to do.

2. XACML does not specify clearly all the elements of obligation and it treats obligations as black boxes without specifying how to handle them.

3. XACML considers the obligation as a directive from PDP to PEP on what

action must be carried out after an authorization decision and granting the access.

4. XACML cannot provide sufficient facilities to support the continuity of access decision afterwards an access was granted. It means that the granted permission can be revoked based on the verification of the access decision during the access. Such a requirement is not satisfied by the current version of the XACML.

5. There is currently no generic method to specify the obligations sent from the PDP to the PEP.

XACML allows to express fine-grained access control policies, but it is not expressive enough to define different kinds of obligation particularly ongoing obligation that can be used to control the continuity of a granted access. Specifying obligations using our proposed behavior language is our approach to enhance the expressiveness and capability of the XACML.

Similar to any software artifact, access control authorization rules will be generated after analyzing the requirements. However, authorization rules cannot cover all kinds of requirements. For example, non-repudiation requirements can be hard to implement as access control rules. Therefore, using obligation can be an effective way to meet such requirements. To specify the obligation, we propose to use the BPL. In our approach, we assume that XACML as the standard access control policy specification language has been utilized to specify the authorization policies.

The Figure 4.3 illustrates our method to attach the obligatory behavior to the general elements of an access control policy presented by the XACML. It also shows that BPL will be used to express the obligatory behavior. After describing an obligation, it will be interpreted to elements that we already defined as features of obligation.

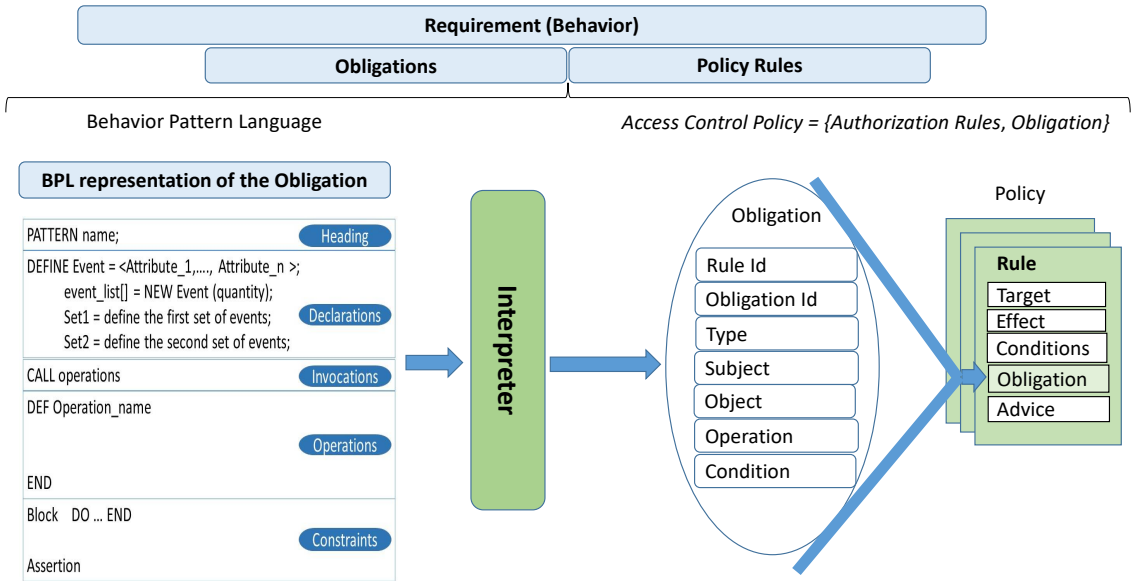

Figure 4.3: Representation of a dictated behavior (obligation) and conversion to XACML policy specification language

Then, these elements will be added as extensible tags into XACML representation of requirements.

4.3 Obligation Enforcement

In general, XACML does not possess specific constructs to address the continuity of policy enforcement. Specifying obligations using a behavior-based language and leveraging SIP protocol are our approach to enhance the expressiveness and capability of the XACML.

The traditional access control is unable to deal with the changes of attribute values of subject, object and environment after the access has been granted. However, values of attributes, that already participated in making an authorization decision, usually changes in dynamic environment. Condition and obligation are two factors that play

significant role in making authorization decision in modern access control systems. Conditions are related to the stateless features of attributes whereas obligations are concerned with commitments of the involved entities. Fulfillment or violation of an obligation often affects the stateful features of attributes.

In terms of approving the fulfillment of an obligation, a predicate represents the obligation in order to verify whether a certain activity has been fulfilled or not. The evaluation of obligation's fulfillment is applied to the access request before, during, and after usage decision. Pre obligation is a predicate that checks if specific activities have been executed or not before usage exercise based on the recorded activities of system's entities. Ongoing is a predicate that should be satisfied periodically during the usage of granted access. If the evaluation of the predicate is postponed after making decision for the current request and used for future usage decision, it is called post-obligation. From the continuity perspective, the fulfillment of ongoing obligation has the main effect on the continuity of the granted access. Therefore, we present a mechanism for checking the adherence to the agreed commitments in functional and ongoing obligations that are expressed by access control policy language.

We need to manage an obligation session that allows the system to monitor the behavior of a user to comply the defined obligation for accessing a particular resource or when the access is in progress. We chose SIP from different session control protocols because of its capabilities for event notification, presence and instant messaging service as well as mature features such as:

- SIP allows two or more participants to establish a session without relying on the content of the media streams. The media streams can be audio, video or any other Internet-based communication mechanism such as distributed games, shared applications, and shared text editors [54].

- The protocol was standardized by the Internet Engineering Task Force (IETF) and was implemented by a number of vendors. SIP allows the endpoints to create, modify, and terminate any kind of media sessions, such as VoIP calls, multimedia conferences, or data communications in distributed systems.

- The SIP protocol supports an intermediate network element called the Back-to-Back User Agent (B2BUA) which operates between two communicating User Agents (UAs). The main advantage of using B2BUA is that it can provide a session management and full control over the session.

- SIP uses the Session Description Protocol (SDP) to help end users to negotiate the characteristics of the session. After session establishment, the protocol behaves as a Peer-to-Peer (P2P) one and allows users send directly messages to each other.

- Utilizing SDP inside of SIP provides a powerful capability to extend new services such as acquiring information about end-points, identifying the originator of a session, and identifying the content-type of a message in a session initiation request.

Increasing the processing load for parsing SIP messages is the main drawback of using standard SIP protocol. Particularly, in constrained environments, processing the large size of text-based SIP messages makes difficult in terms of memory and power consumption. However, recent researches [55, 56] to introduce a lightweight SIP targeted to constrained environments encourage us to utilize the capabilities of it for session management in our approach.

4.3.1 Session Initiation Protocol

Session Initiation Protocol (SIP) is an application layer protocol that is used to set up a media session and help end users to exchange data. SIP provides primitive negotiation capabilities that can be used to implement different services. SIP is based on an HTTP-like request/response transaction model. Each transaction consists of a request that invokes a particular method, or function, on the server and at least one response. Figure 4.4 shows the classic operation flow of a SIP message exchange between two users. To establish a session, the two users invoke several methods back and forth that enable them to agree on a communication session.

To establish a session, the requester (End point A) sends the "INVITE" request that contains a number of header fields. Header fields are named attributes that provide additional information about a message. The body of a SIP message contains a description of the session that is encoded by a protocol called Session Description Protocol (SDP). The "TRYING" response indicates that the "INVITE" has been received and the proxy is working on her behalf to route the "INVITE" to the destination. When the destination receives the "INVITE", it sends the "RINGING" response to requester. "RINGING" response indicates that the destination is deciding to answer the request. If the destination decides to answer the request, it will send "OK" response to requester. After handshaking process, the protocol is considered as a Peer-to-Peer (P2P) one and users send SIP messages directly to each other. Any of the two users can send a "BYE" message directly to the other in order to terminate the session.

SIP protocol provides a logical entity called Back-to-Back User Agent (B2BUA) that operates between two communicating User Agents (UAs) and controls all signaling exchanged between them. In other words, it is a concatenation of a User Agent

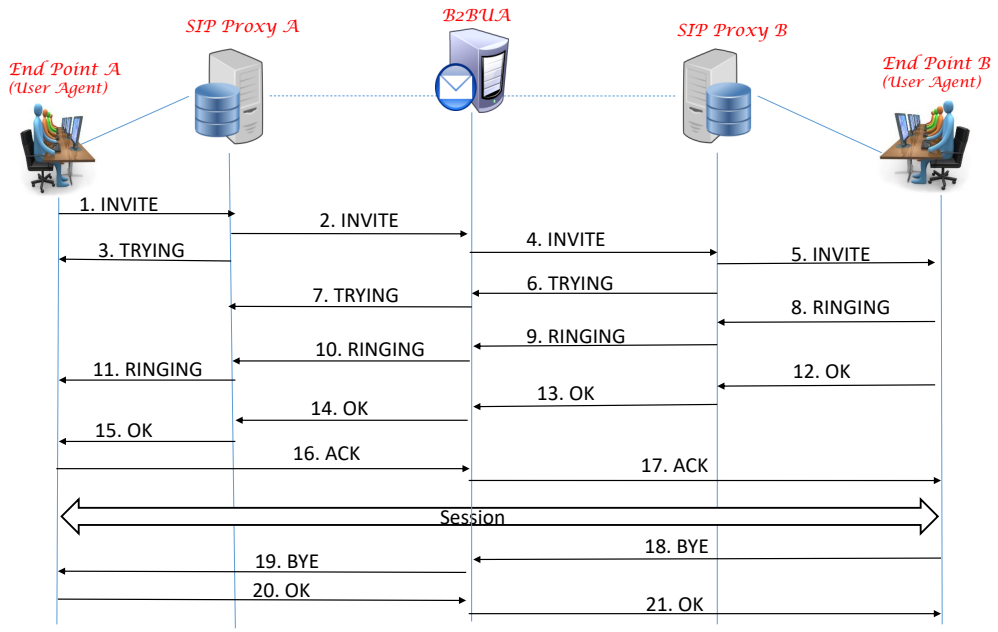

Figure 4.4: SIP session setup example using the back-to-back user agent

Client (UAC) and User Agent Server (UAS). It receives request from the calling SIP user agent (client), and forwards them to the called SIP (server). Therefore, B2BUA as an intermediary prevents the end points from direct exchanging of messages. Applying the B2BUA provides a facility for communication management for the whole duration and full control over the connection.

4.3.2 Wrapping up Obligations in SIP Message

As can be seen in Figure 4.5, a Policy Administration Point (PAP) determines which policies should be defined and deployed to a policy repository. Later on, such predefined authorization rules inside policies can be used by Policy Decision Point (PDP) to make a decision for an incoming access request. A User Agent (UA) generates the request in order to achieve the permission for access to a particular resource. Policy Enforcement Point (PEP) receives the request. PEP transmits the original request to the Context Handler where an XACML request context is constructed. PDP receives

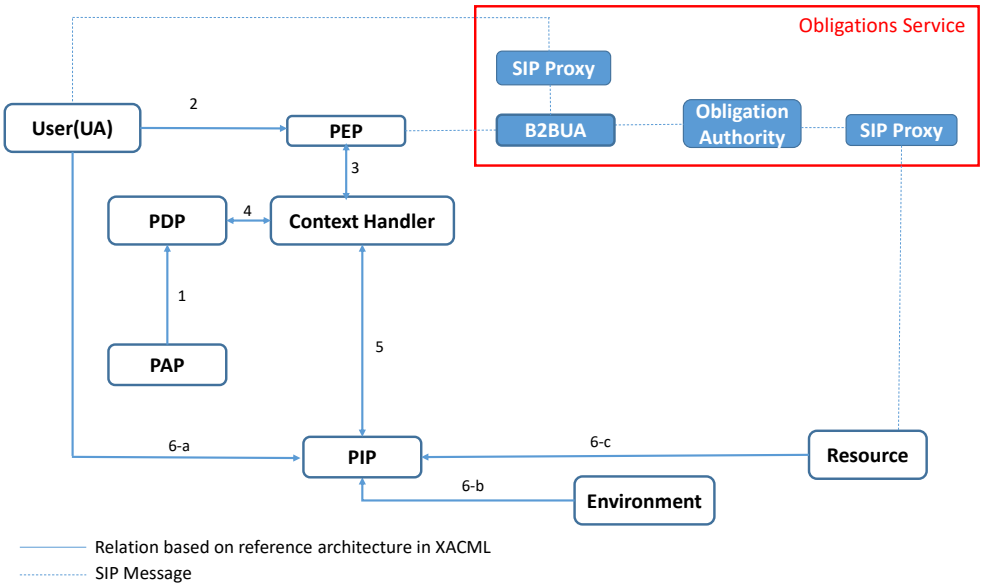

Figure 4.5: Extending the obligation service in XACML reference architecture to handle the prior and ongoing obligations

the XACML request in order to make a decision based on policy rules. For this purpose, PDP requests any additional subject, resource, action, environment and other categories attributes from the context handler. Policy Information Point (PIP) is responsible to provide values for such additional attributes and returns the attribute values to the PDP through the Context Handler. PDP evaluates policies based on provided attribute values to make the decision for granting or denying the request. The authorization decision along with obligations will be sent to the PEP through the Context Handler for enforcement. In summary, the procedure to evaluate an access request based on policy rules is as follows:

1. Flow 1: PAP writes policies and corresponding obligations and make them available to the PDP

2. Flow 2: The access requester sends a request for access to the PEP

3. Flow 3: The PEP sends the request in its native format to the context handler
4. Flow 4: The context handler constructs an XACML request context and sends to PDP
5. Flow 4: The PDP requests attribute values and any additional attributes from the context handler
6. Flow 5: The context handler communicates with PIP to obtain the attribute values
7. Flow 6-a, 6-b, 6-c: The PIP obtains attributes of the subject, resource, action, and environment
8. Flow 5: PIP returns attributes to the PDP through the context handler
9. Flow 4: The PDP evaluates the policy
10. Flow 3: The PDP returns the authorization decision along with applicable obligation to the PEP via context handler

In the XACML workflow, there is no information for obligation enforcement and the role of obligation for the continuity of granted access. We leveraged SIP to provide a communication paradigm between entities in a distributed access control system in order to handle the obligation. For this purpose, we add SIP components such as SIP proxy, B2BUA as well as an Obligation Authority (OA) inside of "Obligations Service" provided by XACML architecture. We consider the OA as a third party (or resource owner) that expects the user to follow a behavior before accessing resources or when the access is still in progress. We describe the interactions between SIP and access control components to handle pre-obligation and ongoing obligation as follows:

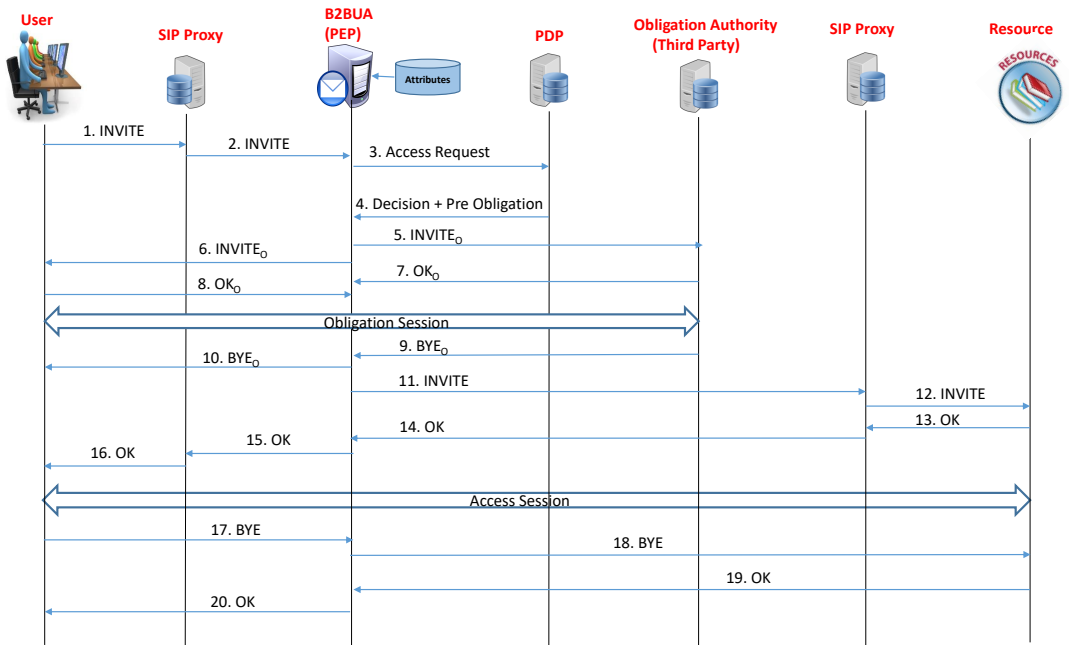

Figure 4.6: Applying a SIP session to handle the pre-obligation

Pre-obligation: In some situations, the fulfillment of an obligation is a prerequisite to allow the user to access a resource. Figure 4.6 shows the sequence diagram in the case of pre-obligation so that the complete execution of an obligation is needed before permitting the user request. We use a B2BUA to achieve a full control over the SIP session. We got the idea from [57] to integrate the B2BUA with the PEP for decreasing the complexity of the implementation. Meanwhile, handling an obligation outside the PEP will greatly complicate the interface between them and decrease the performance. B2BUA is a logical SIP entity that provides a proper opportunity to implement the logics inside of the PEP for managing the obligation life cycle. When a user sends a request to access a particular resource, the request will be wrapped up with SIP message called "INVITE" and will be sent to B2BUA through SIP proxy. PEP extracts the request and sends to PDP to make decision for authorization based on predefined policies and obligations. PDP sends the final decision along with the applicable obligation to the PEP. In this case, we assume that the obligation is a pre-

obligation that should be enforced before granting the permission to the requester. Therefore, the B2BUA suspends the request and sends a SIP "INVITE" message ($INVITE_O$) to both Obligation Authority(OA) and requester (User) to start an obligation session. When the user performs the obligation, the OA approves the fulfillment by sending a "BYE" message (BYE_O) to B2BUA. Then, B2BUA resumes the user's access request and sends it to the resource for establishment an access session.

Ongoing obligation: Ongoing obligations need to be fulfilled in parallel with the requested access. It means that ongoing obligations should be executed when the user performs actions on a granted resource. The user may be permitted to access the resource in the beginning, but if the user fails to fulfill the ongoing obligation, the access will be revoked. Figure 4.7 shows that the user sends the access request wrapped up by SIP "INVITE" message to the B2BUA. Similar to the pre-obligation, PDP evaluates the access request to provide an authorization decision plus an applicable obligation. However, if the obligation is ongoing obligation then the B2BUA establishes an obligation session between the user and the OA which should remain active for the whole period of the access. After obligation session establishment, B2BUA conducts the suspended "INVITE" message to resource through SIP proxy in order to establish an access session. During the access session, if B2BUA receives a BYE_O message from the user to stop the obligation session or B2BUA recognizes the changes of specific attribute values for the purpose of terminating the obligation session, it sends a BYE_O message to the OA to terminate the obligation session. Then, it sends immediately a "BYE" message to the resource and the user to terminate the access session.

Post-obligation: Basically, the post-obligation presents actions that occur after the usage of the resources. Therefore, such actions are required to be performed

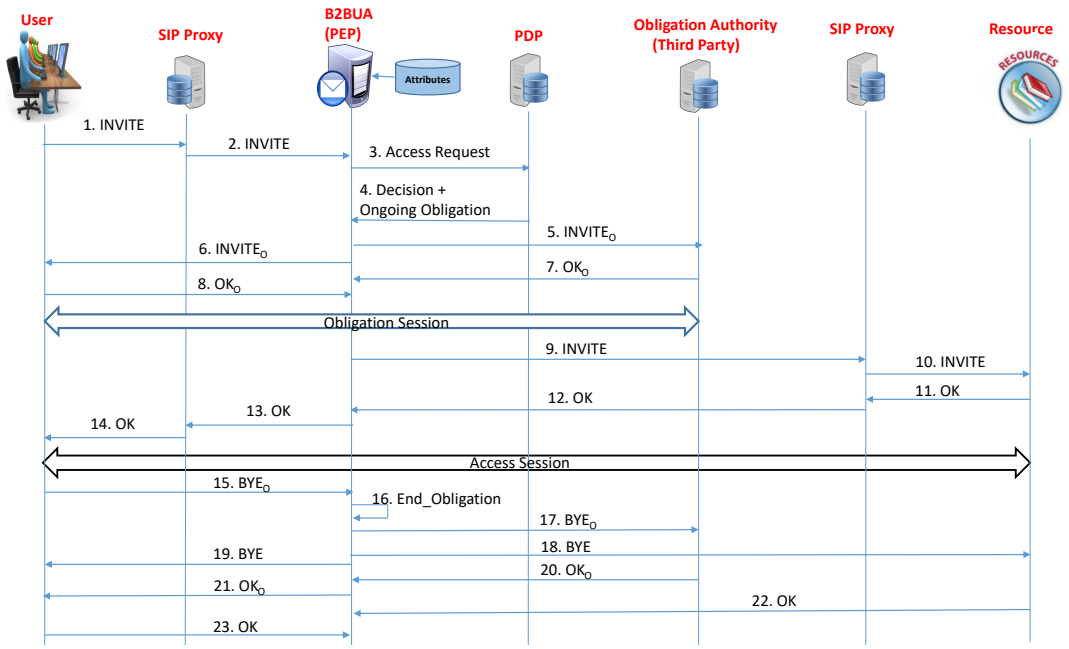

Figure 4.7: Applying a SIP session to handle the ongoing obligation

by the access requester. A notification of the fulfillment of these actions can be sent to the service provider. However, the notification has no effect on the taken decision of the current usage. For example, a user may have to fill in a form at the end of each usage every day or every month. The fulfillment of post-obligation has durability property which guarantees that performed transactions will survive for a long time and they are independent of the active access session. Therefore, defining an obligation session is not an acceptable solution because post-obligation may take a long time for fulfillment. In this case, we rely on the log files in order to investigate the historical behavior of the user through performing a behavior pattern matching to approve the fulfillment of the obligation. We discuss the approach in Chapter 5.

4.3.3 Example Scenario

Since pre-obligation and post-obligation cannot affect on the continuity of the access, we concentrate on specifying the ongoing obligation because the fulfillment of this kind of obligation should be continuously checked after the permission. The continuity of access to a service or resource is a challenge due to the dynamic nature of the environments. In access control, continuity means that a control policy should be enforced not only before an access, but also during the period of the access. For example, the dynamic nature of the context information in a pervasive environment requires the capability of access control model for revocation or continuity of a granted access when certain context conditions fail to hold.

Scenario 1: We focus on combining the smart devices and hospital information system deployed in healthcare centers. Such systems can be accessed remotely or locally by nurses and doctors. The system is able to assign doctors in the capacity of a doctor-on-duty role to a ward during specific time periods. In this role the system allows access to the health records of only those patients who are assigned to the doctor. The system also allows interns to attend in a ward for training purposes. The access permission will be valid until the doctor keeps the role of doctor-on-duty. This role is revoked when the doctor leaves the location or after the end of her duty time. Doctors allow to create different kinds of diagnostic reports about patients so that an intern can access such reports only if the doctor is present in the ward. We consider a situation where an intern begins to access a report and during the access, the doctor leaves the ward. In such a case, the system needs a mechanism to ensure that the ongoing accessing should be revoked due to the absence of the doctor.

We now illustrate how the access control obligation for the Scenario 1 can be modeled and enforced by using our approach. We assume that all doctors, nurses,

interns and patients should register to the HIS system. Meanwhile, entrance doors of important places such as ward, operation room, and conference room have been equipped by RFID readers. Therefore, the system is able to track the individuals, because all personnel have to wear a badge embedded by an RFID tag. The requests of all registered users in different roles to access various resources are evaluated based on authorization rules defined in the policy repository. However, we utilize our approach to resolve some issues existing around fulfilling the obligations such as: how policy administrator can express every detail of the obligation and how a policy engine enforces the obligation during access.

The obligation specifies that an intern can access diagnostic reports in the presence of a doctor. We use the BPL to express the behavior of the intern who is responsible to perform the obligation. In this case, there is a strong association between the behavior of intern and doctor based on the attribute of location. Code 4.2 illustrates the BPL code to represent the dependency of intern's behavior and doctor's behavior. We assume that the behavior of doctor and intern includes one event (line 6 and 7). The structure of the event is defined in line 4 so that it has six attributes. Line 8 defines the domain values for each attribute. We call two methods in line 9 and 10 to assign values to attributes of each event. In line 11, we emphasize that the occurrence of events should be in the same location.

When an intern sends a request, the policy engine evaluates the request by applying policy rules and stateless conditions that are based on the information provided in the access request. The request will be allowed or denied based on the result of the evaluation. However, the authorization system must consider the location of the doctor in the role of doctor-on-duty during the execution of the granted access. In this case, the location of the doctor-on-duty is a mutable attribute that presents stateful condition binding to the resource state. Therefore, continuous obligation enforcement

is required by the authorization system.

According to Figure 4.7, before starting the access session, B2BUA commences an obligation session between the intern and an obligation authority (in this case electronic health record system). After establishment of the obligation session, B2BUA transmits the intern's request to the resource manager in order to create an access session. The values of mutable attributes of involved entities will be periodically collected by Policy Information Point (PIP). Once the location of the doctor changes, an "End_Obligation" message will be generated. Then, B2BUA sends a "BYE" message to both obligation authority and resource manager to terminate the obligation session and access session respectively.

Code 4.2: The BPL description of the obligatory behavior of the intern in Scenario 1

```
1 PATTERN Intern_Behavior;
2 TYPE ongoing-obligation;
3 BEGIN
4   DEFINE Event = <User,Role,Action,Resource,Location,Time>
5   E[] = new Event(2);
6   set1 = {E[1]};
7   set2 = {E[1]};
8   attVal = {<Role,{"intern", "doctor"}>,<Action,{"read","
       write"}>,<Location,{"ward","operating room" }>,<
       Resource,{"diagnostic report"}>};
9   CALL Op1(set1,attVal);
10  CALL Op2(set2,attVal);
11  ASSERT EQUALITY(set1,set2,Location);
12 END
13
14 /* define operation 1-Behavior of intern*/
15    DEF Op1 (set,attVal)
16    Set[E1].Role=attVal[Role].value[0];
17     Set[E1].Action=attVal[Action].value[0];
18     Set[E1].Resource=attVal[Resource].value[0];
19     Set[E1].Resource=attVal[Location].value[0];
20    END
21 /* define operation 2-Behavior of doctor*/
22    DEF Op2 (set,attVal)
23      Set[E1].Role=attVal[Role].value[1];
24     Set[E1].Resource=attVal[Resource].value[0];
25    END
```

Scenario 2: We consider the case of two individuals, say Bob and Alice, who make a video conversation using free softphone applications. When the caller initiates a conversation, the application requires that an advertisement window is displayed on the caller's device while the call is in progress. The caller have to keep the advertisement window active during the conversation. Consequently, if the caller closes this window while the call is still in progress, the application policy term is violated and caller's access should be revoked.

Obligation in Scenario 2 represents the typical example of a usage control that forces the user to open and watch an advertisement during using a multimedia service. Code 4.3 expresses the mandatory behavior of a caller during using the calling service. Execution of the obligation occurs in parallel with using the calling service. The procedure to handle the obligation follows the process shown in Figure 4.7. The caller sends an "INVITE" massage (request) to set up a conversation. The B2BUA receives the decision along with the detail of the obligation from the PDP. Since the obligation is an ongoing obligation, the B2BUA creates an obligation session between the caller and the obligation authority (in this case an advertisement server). The obligation session should keep alive during the conversion between the caller and the callee. After establishing the obligation session, the B2BUA allows to create an access session between them to use the service. If the caller closes the advertisement window for any reason, the B2BUA realizes the termination of watching the advertisement and sends a "BYE" message to the obligation authority to stop obligation session, followed by a "BYE" message to the callee to terminate the access session.

Code 4.3: The BPL description of the obligatory behavior of the caller in Scenario 2

```
1 PATTERN Caller_Behavior;
2 TYPE ongoing-obligation;
3 BEGIN
4   DEFINE Event = <User,Role,Action,Resource,Time>
```

```
5   E[] = new Event (3);
6   set = {E[1],E[2],E[3]};
7   attVal = {<Role,{"caller","calee"}>,<Action,{"call","open"
        ,"watch"}>,<Resource,{"calling_service","advertizement"
        }>};
8   CALL Op1(set,attVal);
9 END
10
11 /* define operation */
12    DEF Op1 (set,attVal)
13      Set[E1].Role=attVal[Role].value[0];
14     Set[E1].Action=attVal[Action].value[0];
15     Set[E1].Resource=attVal[Resource].value[0];
16
17     Set[E2].Role=attVal[Role].value[0];
18     Set[E2].Action=attVal[Action].value[1];
19     Set[E2].Resource=attVal[Resource].value[1];
20
21     Set[E3].Role=attVal[Role].value[0];
22     Set[E3].Action=attVal[Action].value[2];
23     Set[E3].Resource=attVal[Resource].value[1];
24
25     ASSERT EQUALITY DURATION(E1,E3);
26    END
```

Chapter 5

Static Enforcement of Obligation

Searching the obligatory behavior in audit logs is required to approve the fulfillment of a post-obligation. In general, finding the instances of a particular behavior needs a proper pattern matching algorithm. The behavior matching problem is to find the instances of a target behavior pattern so that each instance and the target pattern are very similar in structure and dependency relationship between events. The traditional structure-based matching techniques for relational data is inadequate for behavior matching due to the lack of distinct index of dependency relationships between events. Therefore, we developed a pattern matching engine by using Constraint Satisfaction Problem (CSP) to match a described pattern against events recorded in an audit log.

This chapter is organized as follows: we first provide an overview of CSP problem in Section 5.1. Section 5.2 explains our proposed model for behavior pattern matching. Section 5.3 defines the cost function to optimize the solution of the CSP problem in order to find the best instances of a target pattern. Our approach to reduce the search space is discussed in Section 5.4, followed by pattern matching algorithm in Section 5.5. We also explain how the proposed behavior pattern matching can be used to verify the fulfillment of obligation in Section 5.6.

5.1 Constraint Satisfaction Problem

Constraint programming as a powerful paradigm is to search for a particular state in the search space in which a number of constraints should be satisfied at the same time. Constraints are just relations among variables, or the limitation of values. Modeling a problem as a Constraint Satisfaction Problem (CSP) addresses which relations should hold during the search.

The CSP provides a very expressive and natural formalism to model a variety domains of real-life problems. In general, a standard CSP composed of a finite set of variables, a domain for each variable, and a set of constraints. Each constraint consists of a pair (scope, relation), where scope is a subset of variables that involve in the constraint and relation restricts the values that can simultaneously be assigned to those variables. The aim is to choose a value for each variable so that the assignment satisfies the constraints.

Definition 1: A CSP is a triple P = (X, D, C), where:
$X = (x_1, x_2, ..., x_n)$ is the set of variables called domain variables.
$D = (D_1, D_2, ..., D_n)$ is the set of domains. Each domain is a finite set containing the possible values for the corresponding variable.
$C = (C_1, C_2, ..., C_n)$ is the set of constraints. A constraint C_i is a relation defined on a subset of all variables.

The number of variables in a constraint create the arity of a constraint. For instance, a unary constraint applies to a single variable whereas two variables participate in a binary constraint. If all constraints in a CSP problem are unary or binary, it is called a binary CSP. Any non-binary CSP can be converted into an equivalent binary CSP by utilizing a technique called constraint binarization [58].

Definition 2: An *assignment* is a mapping from the set of values to variables. A

consistent assignment is an assignment that does not violate any constraints, and a *complete assignment* is one where every variable is assigned. A consistent and complete assignment is considered as a solution for a CSP [59, 60].

Definition 3: An assignment A satisfies the constraint C if all the variables in the scope of the constraint are instantiated. A variable is called instantiated when a value from its domain is assigned to the variable (instantiation of variables).

Soft constraints provide an effective way to deal with the real problems that cannot be modeled by classical CSP. Soft constraints provides appropriate method to model a problem with the desired properties considered as preferences whose violation should be avoided as far as possible [61]. There are different ways to present the soft constraints in a model. Using "fuzzy constraints" that is based on fuzzy set theory is an effective way to present the preference [62]. Applying "possibilistic constraints" is another method to model the soft constraints and it is very similar to fuzzy constraints [63]. "Semiring-based CSP" is another approach based on semiring algebraic to model the soft constraints [64].

Associating preferences to the constraints constitutes the CSP as an optimization problem. This kind of CSP in which each constraint of classical CSP will be annotated with a value called weight (cost) denoting the impact of its violation is called a Valued Constraint Satisfaction Problem (VCSP). A VCSP is characterized by a set of hard constraints that must be satisfied, and a set of soft constraints whose satisfaction is desirable. Therefore, solving a VCSP means finding an assignment that optimally satisfying a set of hard and soft constraints [65, 66].

Search is the main algorithmic technique to solve a CSP problem. A complete or incomplete search algorithm can be used to solve a CSP problem. Complete search is called systematic algorithm that finds definitely a solution if one exists. Moreover, such algorithms can be used to show that a CSP does not have a solution. Incomplete,

or non-systematic algorithms do not explore the whole search space. In general, "local search" algorithms behave better for Constraint Satisfaction Optimization Problems (CSOP) [67].

Local search is the famous method that uses the iterative repair approaches. Local search moves from an initial assignment to a solution by making small changes in the current assignment to variables. A set of all solutions that are different from current assignment is called a neighborhood. In each iteration, a solution that has minimum cost value will be selected from neighborhood and considered as the current solution. However, choosing the best solution as a movement increases the risk of local optimum. Therefore, to avoid the local optimum, local search utilizes a random mechanism to select a solution from neighborhood.

5.2 Modeling Behavior Pattern Matching

We use the Valued Constraint Satisfaction Problem (VCSP) to model the behavior pattern matching problem. Formally, a VCSP framework for behavior pattern matching is defined by a four-tuple vcsp = (X, D, C, f), where:

$X = \{x_1, x_2, ..., x_n\}$ is a finite set of variables. We consider every attribute in each event as a variable. For example, x_1 = e1.User ; x_2 = e1.Role; x_3 = e1.Action

$D = \{D_1, D_2, ..., D_n\}$ is a set of domains, one for each variable (i.e., attribute).

C is a set of unary or binary constraints. Unary constraints are defined to restrict the values of one variable. But, a binary constraint defines a restriction between the values of two variables.

f is a cost function.

To define a reference behavior pattern, hard constraints are used to formalize properties that cannot be violated. However, some desired properties can be consid-

ered as preferences, and are modeled by soft constraints whose violations should be avoided as much as possible. A VCSP is characterized by a set of hard constraints that must be satisfied, and a set of soft constraints whose satisfactions are desirable. Therefore, solving a VCSP means finding an assignment (set of events) that satisfies a set of hard and soft constraints on their attributes.

We consider a weight w_i for each constraint so that a very large number (to simulate infinity) will be considered for a hard constraint. The associated weight for each soft constraint is defined based on the number of distinct values for each variable involving in a constraint. We perform statistical analysis on the log file to find the diversity of values for each attribute. For every assignment, a valuation (total cost) is the sum of weights (costs) of violated constraints.

Example: we assume that for a problem with two variables (x1, x2), three constraints have been defined with the costs for violation as follows. The sign ∞ indicates a large number.

$C1 : x1 \in \{1, 2, 3, 4\}; \quad (w1 = \infty, \quad if \quad x1 \notin \{1, 2, 3, 4\})$

$C2 : x1 < x2; \quad (w2 = 3, \quad if \quad x1 \not< X2)$

$C3 : x2 = 4; \quad (w3 = 1, \quad if \quad x2 \neq 4)$

For assignment a1= (x1 = 2, x2 = 4), the total cost will be zero because it satisfies all constraints. The assignment a2 = (x1 = 3, x2 = 2) violates the constraints C2 and C3, so the total cost will be w2+w3 = 4 . For assignment a3 = (x1 = 5, x2 = 4), not only violates the constraint C2, but also violates the first constraint as a hard constraint. The total cost will be w1+w2 = ∞+3= ∞.

5.3 Cost Function

We define a cost function (objective function) to optimize the search algorithm in order to find instances of reference pattern in the dataset. In the VCSP an assignment is the task of assigning a value to a variable. The total cost of an assignment is the aggregation of the costs of constraints that are violated by this assignment.

$$f(C_1, ..., C_n) = \sum_{i=1}^{n} w_i$$

f: cost function

n: the number of violated constraints

w: is a positive integer number assigned to each constraint. This cost shows the effect of a constraint violation.

5.4 Constraint Presentation and Propagation

The feature of the reference behavior pattern is presented by the BPL. Therefore, parsing the BPL code extracts the specification of behavior pattern in terms of length, bound attributes, and constraints. We use the JSON (JavaScript Object Notation) format to present such specifications.

Constraint propagation [59] is a specific type of inference in order to use the hard constraints to reduce the number of legal values for a variable. Constraint propagation decreases the size of search space so that the problem is hopefully easier to solve. Constraint propagation may be combined with search, or it may be performed as a preprocessing step before starting the search. Sometimes this process solves the whole problem, therefore we implement the constraint propagation as a preprocessing step. Constraint propagation can be done based on using the unary constraints or binary

constraints. We propagate the unary hard constraints so that the result will be groups of events satisfying the respective constraint.

5.5 Pattern Matching Algorithm

User behavior pattern matching is the process of searching and bringing together events from a large dataset and comparing them with the reference pattern in order to find out whether they represent the instances of the defined reference behavior pattern. A matching process must be able to analyze and satisfy different kinds of constraints that restrict both the attribute value and the relation among events. Therefore, the specification of reference pattern along with the constraints should be clearly defined. Algorithm 1 shows how our approach extracts the feature of a reference pattern, and Algorithm 2 presents the process of pattern matching to find instances of the reference pattern by satisfying constraints.

When the search space is too large, exhaustive search will fail to give any meaningful results in a reasonable time. Our proposed method is non-systematic algorithm and similar to local search methods [61], it starts with a candidate solution and tries to iteratively improve the initial assignment. Meanwhile, local search can easily be extended to a Constraint Satisfaction and Optimization Problem (CSOP) [59].

The designed algorithm for search (Algorithm 3) considers each event in the reference pattern as a placeholder. To find the instances of the pattern, it starts with an initial assignment that assigns an event selected from an event log to each placeholder. Assigning an event to a placeholder, actually assigns values to several variables (attributes). The search iteratively improves the assignment of a group of variables relevant to the target event in the pattern. The given cost function controls the iterative improvement. The value of cost function is expected to be reduced in each

Algorithm 1 extract-reference-pattern

Input: JSON file

Output: reference behavior pattern B described by the following parameters:

A: set of attributes

Size: number of events in the reference pattern

X: set of bound attributes

C: set of unary or binary constraints

W: set of weights

Local Variable:

key: represents a "key" in JSON file

value: represents the "value" of a key

1: $A = \varnothing, X = \varnothing, C = \varnothing, W = \varnothing, Size = 0$
2: **for** each <key, value> in JSON file **do**
3: **if** key == "eventType" **then**
4: append the value to A
5: **end if**
6: **if** key = = "length" **then**
7: Size = value
8: **end if**
9: **if** key = = "bound-attribute" **then**
10: append the value to X
11: **end if**
12: **if** key = = "constraint" **then**
13: append the value to C
14: **end if**

15: **end for**
16: **for** each constraint in C **do**
17: extract attribute name;
18: w_{i} = number of different values in log file for the extracted attribute;
19: append w_{i} to W;
20: **end for**

iteration by decreasing the number of violated constraints. The search tree is generated as the search progresses. Each node is labeled with a cost value calculated by the cost function.

We denote an assignment of the given CSOP problem by S, and we define a set V_s called conflict set containing all variables that cause violation for constraints under the current assignment S. We call these variables, conflict variables. The variables in V_s are grouped based on respective constraints. A group with the highest weight constraint is selected. If all variables in the selected group are relevant to a particular event-placeholder, that event-placeholder is chosen for replacement. If variables in selected group belong to different event-placeholders, one event-placeholder is randomly selected for replacement. Then, a new event is chosen from the event log such that by assigning its attribute values to the corresponding event in S, the value of cost function decreases. When the V_s is empty, the current S will be a solution for CSOP. Otherwise, a defined threshold is used to stop the iteration. For example, the search keeps modifying the variables until a user specified limit on the number of search steps is exceeded, or the iteration can be stopped if the value of cost function is less than a user defined threshold.

As an example, Figure 5.1 shows the generated search tree to find instances of a behavior pattern consisting of four events each of which contains several attributes.

Algorithm 2 Pattern Matching

Input:

B: reference behavior pattern representation as the tuple (A, X, C, W, Size) extracted from Algorithm 1

E: set of users' events

Output:

P: set of instances of behavior pattern

Local Variable:

g: an event-group that satisfies unary hard constraints relevant to an event-placeholder

G: set of all event-groups

s: an instance of the behavior pattern

1: $P = \varnothing$
2: **for** i in [1 .. Size] **do**
3: **for** each unary constraint c_{i} in C **do**
4: g_{i} = propagate(c_{i}, E)
5: insert g_{i} in G
6: **end for**
7: **end for**
8: **for** step in [1 .. Size of the largest group in G] **do**
9: s = Search (G, C, W)
10: insert s in P
11: **end for**

Algorithm 3 Search Algorithm

Input:

G: set of all event-groups

C: set of constraints

W: list of weights for constraints

Output:

s: an instance of the reference behavior pattern

Local Variable:

V_s: list of conflict variables

f: cost function of global solution

$Current_f$: cost function of the current solution

1: s = [$e_1, e_2, ..., e_n$]; % A complete assignment by choosing one event from each event-group;

2: V_s = [{ $x_1, x_2, ..., x_n\}_{|c_1}, \ldots, \{x_1, x_2, ..., x_n\}_{|c_n}$]

3: $f = \sum_{i=1}^{n} W_{|C_i}$

4: $TryEventPlaceholder = true$;

5: **while** $V_s \neq \varnothing$ and $TryEventPlaceholder == true$ **do**

6: select a constraint with the highest weight from V_s ;

7: **if** all the event placeholders were already selected **then**

8: $TryEventPlaceholder = false$;

9: **else**

10: i= select an event-placeholder relevant to the selected constraint;

11: Count =1;

12: **repeat**

13: choose an event e from an event-group g_i ($g_i \in G$);

```
14:            s[i] = e ;        % The value of i equals event-group index;
15:            Current_f = sum_{j=1}^{n} W_{|C_j} ;
16:            Count = Count + 1;
17:            if Current_f < f then
18:                f = Current_f;
19:            end if
20:        until  f > threshold and Count < IterationNumber;
21:        if f <= threshold  then
22:            break ;
23:        else
24:            V_s = updated V_s
25:        end if
26:    end if
27: end while
28: return s;
```

Each node of the tree represents a complete assignment. The assignment to the root is performed randomly and it is considered the initial solution. The cost function is calculated and the list of conflict variables (V_s) and corresponding event-placeholder will be recorded. Conflict variables will be grouped based on their involvement in constraints. To generate the children, the group participating in a constraint with high cost is selected. Then, the relevant event-placeholder in current solution will be replaced with an event picked from the respective event-group. The selected event should satisfy the hard constraints. The cost function is calculated for new node. If the cost value of the new node is smaller than the cost value of the solution, the current node will be considered as a solution and replaced with the previous solution.

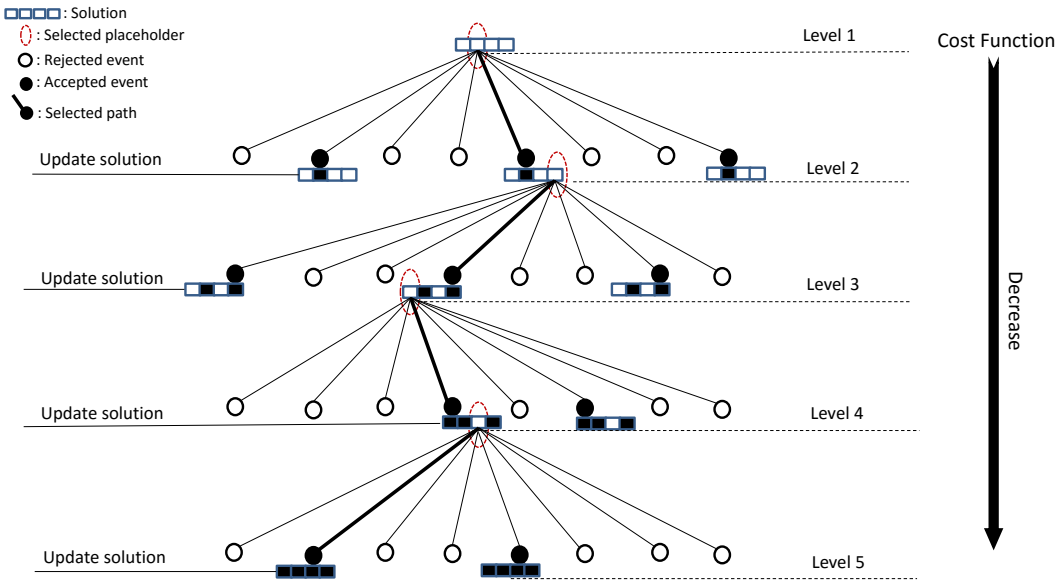

Figure 5.1: A generated search tree for instantiation of a behavior pattern containing four events

This process continues until the number of iteration for improvement exceeds the determined number of steps or the cost value of an improved solution reaches a threshold.

5.6 Behavior Pattern Matching for Post-obligation

Using the concept of hard and soft constraints gives enough flexibility to our proposed behavior pattern matching to match approximately or exactly a desirable pattern against recorded events in a log file. Such flexibility is required to make the method as general as possible in order to search a variety of behavior. In particular, it allows the administrator to find out more about a conjectural behavior through finding approximately the instances of the behavior for further investigation in order to verify the completeness of the access control policies.

Since our proposed behavior pattern matching relies on searching an audit log, it cannot be used to verify the fulfillment all kinds of obligations. Post-obligation is an obligatory behavior that should be performed when the resource access ends and it does not impact the current resource access. Post-obligation has durability property so that we need the historical behavior to verify its fulfillment. Consequently, exact matching instead of approximate matching is required to verify the fulfillment of post-obligation. To initiate exact matching, all constraints should be set as hard constraints to configure the behavior pattern matching. We provide several examples in Chapter 7.

Chapter 6

Active Enforcement of Obligation

In traditional access control, obligations are actions which the user promises to perform after finishing her access. Meanwhile, authorization is assumed to be done before the access is allowed. Therefore, there is no mechanism to validate the actions of a user during the access. The lack of capability for continuous enforcement is a barrier to use the traditional access control in environments where the contextual information changes dynamically. In such environments, continuous enforcement should be applied by evaluating usage requirements and stateful conditions throughout the usage.

Our proposed behavior pattern matching based on CSP is not capable of handling the continuous enforcement because it relies on audit logs. Therefore, we design a matching framework based on Complex Event Processing (CEP) to generate the appropriate response in a timely manner. The framework applies a stream-centric approach to enforce the obligation when the access is in progress. This allows it to connect different event sources to approve the fulfillment of the obligation. Based on the defined time unit, a stream carries zero or more events that have the same format. Any combination of such events in order to satisfy a set of constraints and

features can present the behavior pattern of the event producer. These constraints and features will be expressed by our proposed behavior pattern language.

This chapter is organized as follows: Section 6.1 presents different steps to apply a CEP-based solution for the pattern matching problem. Since the core of a CEP-based algorithm relies on a Finite Automata (FA), we explain the specification of our designed FA in Section 6.2. The structure and data flow of our proposed framework will be discussed in Section 6.3.

6.1 Complex Event Processing

Event Processing is a technique for high speed processing of events on a lower abstraction level in order to recognize significant events or meaningful patterns in an event stream. Technically, Complex Event Processing (CEP) is a subset of event stream processing. The basic concept and architecture of an event processing system has been introduced in [68]. Event Processing Agent (EPA) and Event Processing Network (EPN) are two main components of an event processing system so that EPA applies some logic to generate a set of complex events as output and EPN is a network of event producers, event consumers and a collection of EPAs connected by particular channels. Aggregation-oriented and detection-oriented are two main categories of most CEP solutions. An aggregation-oriented solution is focused on executing on-line algorithms as a response to event data entering the system and detection-oriented is focused on detecting combinations of events called events patterns or situations. A simple example of detecting a situation is to look for a specific sequence of events [69]. In [70] the following steps has been defined as the key idea of complex event detection:

1. Primitive events are extracted from large volume data.

2. Event correlation or event aggregation is detected to create a business event with event operators according to specific rules.

3. Primitive or composite events are processed to extract their time, causal, hierarchical and other semantic relationships.

Complex Event Processing (CEP) has become an applicable technology with a wide range of well-known application in various domains such as financial markets, electronic healthcare systems, and security monitoring systems. Event processing technology can be utilized to solve the problems that are related to processing the large amount of runtime data generated from large information systems, in order to detect undesired behavior (like runtime failures or malicious activities) or other specific behavior patterns of interest. The desirable patterns are defined by a query language in the majority of CEP frameworks. A series of SQL-based languages have been proposed for querying the desired patterns on event streams [71, 72, 73, 74, 75].

Many applications need to recognize patterns in event streams. CEP is an emerging technology that has enough capability for detecting known patterns of events and correlation between them in real-time systems. The work in [76] describes an approach to detect unknown event patterns using a CEP engine. The engine utilizes the event processing agents (EPAs) to analyze the event cloud of an organization. Then, discriminant analysis is applied for recognizing unknown patterns which seem to be a suspicious event combination for the use case of credit card transactions and the fraud problem connected with this kind of payment. Discriminant analysis is a multivariate statistical method for analyzing multidimensional data to discover the relationship between data.

Behavior representation and analysis is closely related to the event processing approaches that have been explored extensively in recent years. CEP is a technique

of processing events on a lower abstraction level in order to recognize significant events or meaningful patterns. Similarly, we represent abstractly the user behavior as an ordered-set of finite number of events so that an event is represented as a tuple of attributes that describe the characteristics of interaction between a user and a system resource. We describe a behavior pattern using our designed language, then the pattern matching engine searches the event logs to find the instances of the described pattern.

6.2 Automaton-based Matching

After expressing a behavior pattern in a descriptive manner, we present a formal evaluation model that employs a non-deterministic finite state automaton enriched with a match buffer. During the execution of the automaton, the buffer collects events that satisfy the conditions to reach a new state. We formulate formally a described behavior pattern as a triple:
$P = (< e_1, e_2, ..., e_n >, F, C)$ where:

1. $<e_1, e_2, ..., e_n >$ is a sequence of events so that each e_i belongs to E_i . Each E_i , $i \geq 1$, is a set of instances of events can substitute for the placeholder i in pattern P so that,
   $1 \leq i \leq n$, $1 \leq j \leq n$,
   $if\ i \neq j\ then\ E_i \cap E_j = \varnothing$.
   We assume $E = E_1 \cup E_2 \cup, .., \cup E_n$ is a subset or the whole of dataset (event stream).

2. $F = \{f_1, f_2, ..., f_n\}$ is a set of features that restrict the attribute values of events. Different kinds of operators make such a feature. We assume θ belongs to set

of operators $\theta \in \{=, <, >, \leq, \geq, \neq\}$. We consider the general form for features $f \in F$ as follows:

$f :\ e.a\ \theta\ c$; limits the value of the attribute a of event e to constant c.

3. C = $\{C_1, C_2, ..., C_n\}$ is a set of constraints that defines the allowed values on a measurable context such as time window, frequency, distance, etc. Such measures are mutable attributes that may be excluded from event structures. They represent the stateful condition of the environment or the correlation between events. We consider each $c_i \in$ C as an aggregation function [77, 78] that accepts events as inputs and the output satisfies the condition on the corresponding context. A constraint can be also defined as $C :\ e.a\ \theta\ \acute{e}.a$ that makes a relation between two events e and $\acute{e}$ based on the attribute a.

Definition (Automaton): Let P be a pattern and V be the set of all event variables (placeholder) in P. A Non-deterministic Finite Automaton (NFA), A, is a six-tuple: $A = (Q, \Sigma, \Delta, q_s, q_f, C)$

- $Q = \{q_1, q_2, ..., q_n\}, q_i \subseteq V$, is a finite set of states
- Σ a dataset includes finite set of events e
- $\Delta = \{\delta_1, \delta_2, ..., \delta_n\}$ is a finite set of transitions of the form $\delta = (q, e, F_\delta)$, F_δ is the transition conditions
- q_s is a start state, $q_s \in Q$
- q_f is the accepting state
- C is a set of constraints

We defined each state of the automaton as a subset of the event placeholders. A transition function $\delta = (q, e, F_\delta)$ directs the movement from a source placeholder q to

a destination placeholder if assigning e to q satisfies the transition conditions F_δ. The F_δ includes a set of conditions that bind the attributes of the substituted event. BPL describes the behavior pattern based on the chronological order of events. Therefore, the last event in the behavior pattern plays the role of the accepting state.

If an input event matches against a placeholder, it moves to a stack. When the automaton reaches to the accepting state, the set of constraints C applies on events inside of the stack. If collected events satisfy all constraints, the content of the stack will determine a matching pattern. We represent graphically an automaton as a graph. Nodes represent states (event placeholder) and edges are labeled with the transition. Each edge is bound by the transition features of the corresponding target event placeholder. In the following example, we describe the translation from BPL description into the equivalent NFA which can faithfully capture the same pattern in the input.

Example 1: Find the behavior pattern of a user in any role who first reads and then writes the patients diagnostic reports from 9am to 10am.

Behavior pattern can be formulated as follows:

$P = (< e_1, e_2 >, F, C)$

P is composed of two events (placeholder), both of which have the same type. Events coming from the source can substitute for e_1 and e_2 if they have features F. The features over events and attributes are given as:

$F = \{f_1 : e_1.action == \ 'read', f_2 : e_1.resource == \ 'diagnosticreport', f_3 : 9 \leq e_1.time \leq 10, f_4 : e_2.action == \ 'write', f_5 : e_2.resource == \ 'diagnosticreport', f_6 : 9 \leq e_2.time \leq 10\}$

These features are grouped in transition functions that have been displayed in Figure 6.1. Transition functions δ_1 and δ_2 guide to find events that can substitute for e_1

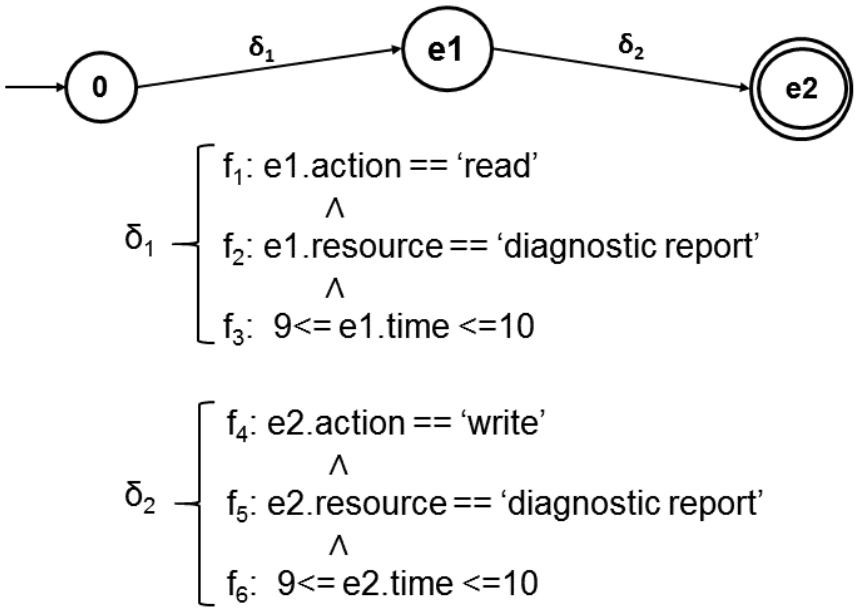

Figure 6.1: NFA for Example 1 along with transition functions

and e_2 respectively. Such events are pushed in a stack. When the NFA reaches to an accepting state, the content of the stack will be checked in order to satisfy constraints. In this example, the constraint over events is defined as:

$C = \{c_1 : equality(< e_1, e_2 >, user)\}$

According to the definition of *equality* in the Table 3.2, the constraint c_1 ensures that the both events have been issued by the same user. However, the user may issue such events in different roles.

The matching process is called execution of a compiled pattern. The execution has two phases: structure checking and acceptance checking. Algorithm 4 shows the procedure for the structure checking (lines 3 to 13) by applying the automaton to capture the instances that own the structure similar to the described pattern. In acceptance checking (lines 14 to 18), we apply the defined constraints on the found instances. If an instance satisfies the constraints, we will consider it as an accepted

instance. Actually, the set of accepted instances will create the response for the query of desirable behavior.

6.3 Compliance Framework

A compliance engine receives a continuous stream of event data that records the interaction between users and resources in order to track compliance with access control obligations. An actual event contains different attributes. Therefore, an event can be represented by some format such as XML, JSON, JAVA object, etc. We represented each event in a stream as a serialized Java object for implementation a prototype of our proposed framework.

In Figure 6.2, Policy Administrator (PA) defines policy rules and uses the Behavior Pattern Language (BPL) to specify expressively the different kinds of obligations. Compliance checkpoints are extracted during modeling and specifying obligations. Based on our discussion in Section 6.2, checkpoints comprise constraints that play the role of semantic filters to approve the instances of a described behavior pattern. We define three categories for compliance checkpoints as follows:

- Proximity: checks the desirable characteristics of the sequence of events within a specific window. Such a window defines the maximum time interval for an event sequence to match the pattern or they can define the distance between locations of events.

- Association: checks the correlation between events based on specific attribute values.

- Stateful condition: The attributes of an entity can be divided into two general groups static and mutable attributes. The value for static attributes is prede-

Algorithm 4 NFA_based behavior pattern matching

Input:

nfa: the parameters of the automaton $(Q, \Sigma, \Delta, q_s, q_f, C)$

B: Buffer

E: event stream

Output:

An instance of the pattern

1: $B \leftarrow \varnothing$
2: $currentState \leftarrow q_s$
3: **for** $i = 1$ to $length(pattern)$ **do**
4: get an event e from E
5: **while** true **do**
6: **if** e satisfies F **then**
7: $B \leftarrow B \cup \{e\}$
8: $currentState \leftarrow \delta(currentState, e, F_\delta)$
9: break
10: **else**
11: get another event e from E
12: **end if**
13: **end while**
14: **if** $(i == length(pattern))$ and $(currentState \cap q_f \neq \varnothing)$ **then**
15: **if** B satisfies C **then**
16: return B
17: **end if**
18: **end if**
19: **end for**

fined. They are assigned by the administrator. For instance, id, role and action are examples of typical static attributes for the entity of the user. The values of mutable attributes are context-aware. They will be determined based on the result of an activity or a business process. Location, time and the level of credit are typical examples of mutable attributes. Meanwhile, there are conditions relating to the activity data that are not presented directly by attributes of a particular entity. Such data are produced by the activities of a business process that changes or keep the state of affairs in the environment. The concept of counter (i.e. The number of access to a resource) or keeping a window open for a while are examples of such states.

When a user sends an access request to the PEP, the request will be forwarded to the PDP for evaluation against the predefined policies. The decision along with obligations will be sent to the PEP and also applicable obligations will be retained in the Obligation Set. In other words, Obligation Set includes all of the obligations that should be fulfilled during the execution of every access request. If the decision is "permit", the PEP creates an access session between the user and resource manager to execute the access decision. Similarly, our proposed framework fulfills the obligation through creating an obligation session. The compliance engine that is actually a Complex Event Processing (CEP) receives all interactive events during a session. The event streams are continuously processed by compliance engine based on the compliance checkpoints of applicable ongoing obligations and updated attribute values in order to monitor the behavior of users. If the user violates the ongoing obligation, the compliance engine sends a signal to the PEP to revoke the session.

The data-flow model of the framework is described as follows:

1. Policy: Policy Administrator (PA) writes policies and make them available to the

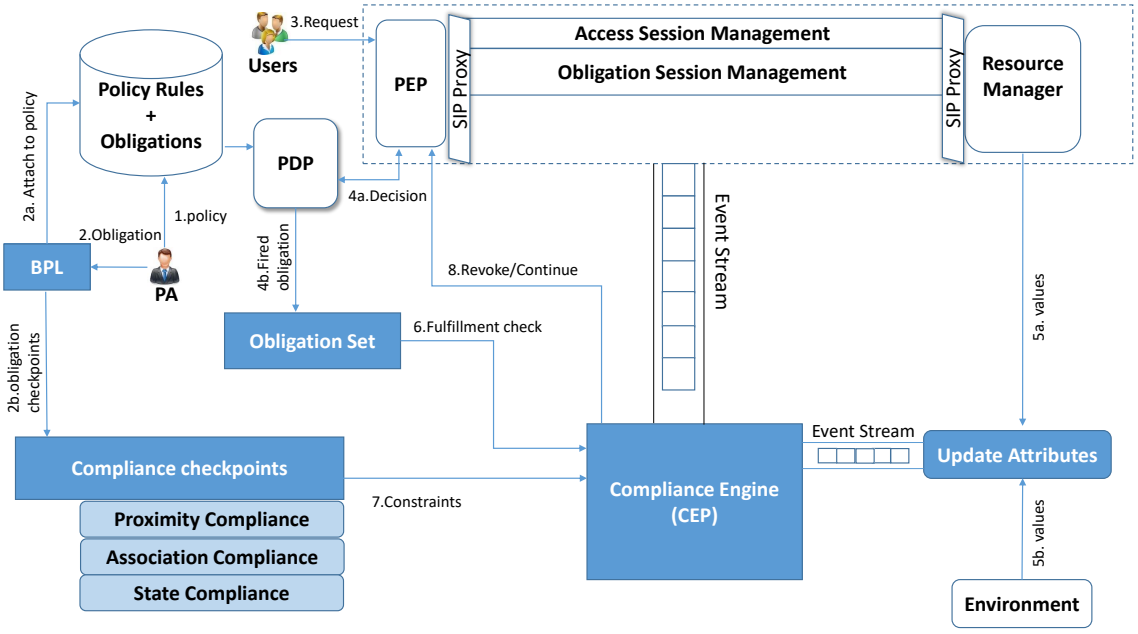

Figure 6.2: An architecture to check the ongoing obligatory behavior

PDP. These policies are sets of rules for specified targets.

2. Obligation: PA can also define any kind of obligations using Behavior Pattern Language. The described obligatory behavior will be interpreted to the format of policy repository and will be attached to corresponding policies or rules (2a). The criteria for the violation of each obligation will be saved in the compliance checkpoint repository (2b).

3. Request: The PEP receives the request of users to access resources.

4. Decision: The PEP sends the request to the PDP where the attributes of the request are compared to predefined policy rules (4a). The decision along with the obligations send to PEP for enforcement. The active obligations are also recorded in the Obligation Set repository (4b).

5. Values: Any changes in attribute values during the enforcement of the access request will be recorded in Update Attribute repository.

6. Obligation ID: The Compliance Engine receives the list of obligations that should be fulfilled when the access is in progress.

7. Fulfillment check: The Compliance Engine compares regularly the user interactions against the obligatory behavior in order to approve the fulfillment or violation of the obligation in defined session.

8. Revoke/Continue: If the compliance engine recognizes the violation of the obligation, it will send a signal to PEP to revoke the access.

Chapter 7

Validation of BPL

Programming languages are very useful to represent an idea in order to transfer it from human to machine. However, programming languages are general purpose languages that are focused on different aspects of the problem to provide a general solution in several domains. In contrast, domain specific languages are created to address the problems in a specific domain. Such languages focus on a subset of the whole problem space. Therefore, the expressiveness of these languages is limited to the most useful operations in the domain. The limited expressiveness prevents the developer from low level details that are not focus on the desirable part of the problem. Finally, domain-specific programming languages can be interpreted to a general-purpose programming language such as Haskell, C, JAVA or translated directly to machine code for execution.

The main purpose of using specification languages is to bridge the gap between the problem domain and the solution domain in order to increase the user productivity. The lack of systematic approaches, comprehensive set of tools and standards make the evaluation of specification languages as a crucial step in the construction of a domain specific language. BPL as a specification language can play the role of a

query language whose semantics formally specifies the desirable operational behavior in a dataset or data stream. To validate the capability of BPL to specify obligatory behavior patterns, we rely on two different frameworks for static and dynamic enforcement. Since the impact of post-obligations and ongoing obligations on the access lifetime is different, we used different approaches for enforcement the described obligation by BPL.

Section 7.1 presents our experiments with executing the BPL-based described behavior on a dataset. This section addresses the specification and enforcement of post-obligations. Section 7.2 introduces the open source tools that we applied to implement the compliance framework to enforce the ongoing obligation. Moreover, the format for representation the event as well as the expressiveness of BPL to specify complex behavior will be discussed in this section.

7.1 Validation for Static Enforcement

The static enforcement is relevant to the post-obligatory behavior where fulfillment or violation does not affect the current access session. Such behavior obliges the user to perform several actions after accessing resources. Moreover, the fulfillment of the post-obligations rely on the historical behavior of users. We present the capability of BPL to express different categories of complex behavior in this section.

7.1.1 Implementation of Behavior Pattern Matching and Validation Framework

As part of this research, the proposed behavior pattern matching has been implemented in a toolkit. The toolkit provides an interactive environment to use BPL to

describe a behavior pattern, extract constraints and drive a search engine to solve the constraint satisfaction problem.

To evaluate the expressiveness of BPL to specify different categories of behavior, we consider three categories of user behavior including sequence, association, and temporal proximity. We define five patterns for each category. Each pattern presents an obligatory behavior that should be performed after accessing resources.

Category 1: Sequence

Sequence means that a series of events occurs in a certain order by focusing on changing the values of only one attribute in each pattern. We consider the behavior of a typical authenticated and authorized user for access different resources in a hospital. We suppose that the user works in different wards of a hospital and her role can change during a month.

Pattern 1: After updating a diagnostic report, she has to notify her supervisor, receive the confirmation, and then send email to the corresponding nurse.

Pattern 2: After downloading the MRI in her local station for more study, she should fill in a form, send it to the supervisor, and then delete the MRI file.

Pattern 3: She has to perform the sequence of actions: read, update, and delete in the role of doctor after downloading a diagnostic report on his local machine.

Pattern 4: After accessing the health record of a patient in the role of nurse, the system should send email to the patient's doctor and then save the information of access in a log file.

Pattern 5: When a patient is removed from her list as a doctor, she has to response the email, and then delete all files that are relevant to the patient.

Category 2: Association

Association shows a kind of behavior that involved events share the values of attributes. We consider the behavior pattern of users who work in different departments

of a hospital. They generate events with shared attribute values.

Pattern 1: A user in the role of nurse, after updating a report, should notify the supervisor and send the report to the corresponding physician.

Pattern 2: A user in the role of triage nurse, after downloading the profile of patients, has to send it to physicians and the supervisor of the emergency ward.

Pattern 3: A physician in the emergency ward, after receiving the patient's profile, should update the profile and update the diagnostic report.

Pattern 4: A physician in the emergency ward, after updating the diagnostic report, should notify the supervisor of the ward, and delete the patient's profile from his local station in the emergency ward.

Pattern 5: A physician in the cardiology ward, after updating the diagnostic report of patient Mike, should delegate her role to an intern and notify the ward supervisor at 10 a.m.

Category 3: Temporal proximity

Temporal proximity describes the behavior of users who perform a sequence of actions during a period. For example, a typical work-flow of a role such as physician or nurse to treat a patient can be described as a sequence of actions within a period.

Pattern 1: A physician in the emergency ward, after receiving the patient's profile, she should notify the triage nurse and update the diagnostic report within one hour.

Pattern 2: A physician, after receiving the patient's profile, should perform a sequence of actions such as read the diagnostic report, update diagnostic report, and delete diagnostic report within 8 hours during a day.

Pattern 3: After downloading an MRI image, the radiologist has to write a diagnostic report and delete the image from her machine within 8 hours.

Pattern 4: When a patient left the ward and a user in the role of nurse is denied to access the patient's health record, she should perform actions in order delete profile,

delete exam, and delete report within 8 hours

Pattern 5: An intern, after receiving a report from her supervisor, she should read, and then delete it within 8 hours.

7.1.2 Querying Behavior Patterns

We assume that an administrator is interested in studying the historical behavior to find instances of a particular behavior pattern. Such a pattern can present the behavior of users who comply policies with post-obligations. On the other hand, the administrator can write a query by changing attribute values of a particular post-obligation. In this case, each query pattern presents a conjectural behavior as a new scenario to study the historical behavior. If the pattern matching engine finds any instances, it proves that the post-obligation has been violated.

For this purpose, the administrator uses BPL to describe the specification of a desirable behavior pattern. Parsing BPL code generates a JSON file including pattern structure along with constraints that express the relation between events. Our proposed pattern matching algorithm uses the specification of the described pattern as a reference in order to find instances of the behavior pattern.

Example 1: The administrator is interested in finding behavior of users who performed a sequence of actions after downloading an MRI file in order filling in a form, sending, and then removing .

Code 7.1: A query for sequence of actions in the post-obligation

```
1 PATTERN Example-1;
2 BEGIN
3    DEFINE Event = < User, Action, Patient, Location,
        Resource, Date, Time >
4    E[] = new Event (3);
5    set1 = {E[1], E[2], E[3]};
6    attVal = {<Action, ("write","send", "delete")>};
7    CALL Op1 (set1, attVal) ;
```

```
END
   /* define operation */
DEF Op1 ( set1, attVal)
   Set1[E1].Action = attVal[Action].value[0];
   Set1[E2].Action = attVal[Action].value[1];
   Set1[E3].Action = attVal[Action].value[2];
   ASSERT EQUALITY (set1, Resource);
END
```

Parsing the BPL code of Example 1 generates a JSON file that describes the structure of the behavior pattern in terms of length, bound attributes, and the specification of constraints.

Code 7.2: JSON file extracted from the described behavior in Example 1

```
{
"name": "Pattern Example-1",
"length": "3",
"bound_variable": [
   {"var1":"action"  },
   {"var2":"resource" }   ],
"constraint": [
   {   "type": "unary",
      "operand1": "event1",
      "attr": "action",
      "value": "A-17",
      "operator": "=="
   },
   {   "type": "unary",
      "operand1": "event2",
      "attr": "action",
      "value": "A-15",
      "operator": "=="
   },
   {   "type": "unary",
      "operand1": "event3",
      "attr": "action",
      "value": "A-19",
      "operator": "=="
   },
   {   "type": "binary",
      "operand1": "event1",
      "operand2": "event2",
      "attr": "resource",
      "operator": "=="
   },
```

```
32     {    "type": "binary",
33          "operand1": "event2",
34          "operand2": "event3",
35          "attr": "resource",
36          "operator": "=="
37     }
38 ]
39 }
```

Example 2: The administrator is interested in finding the behavior of physicians in the emergency ward so that they update the profile and update the diagnostic report after accessing the patients record.

Code 7.3: A query for association pattern in the post-obligation

```
1  PATTERN Example-2;
2  BEGIN
3     DEFINE Event = < User, Action, Patient, Location, Role,
          Resource, Date, Time >
4     E[] = new Event (2);
5     set1 = { E[1], E[2]};
6     attVal = {<Role,("physician")>, <Resource,("profile","
          diagnostic report")>, <Action,("update")>, <Location,
          ("emergency-ward")>};
7     CALL Op1 (set1, attVal) DO |event|
8        event.Role = attVal [Role].value[0];
9        event.Action = attVal [Action].value[0];
10       event.Location = attVal[Location].value[0];
11    END
12 END
13    /* define operation */
14 DEF Op1 (set1, attVal)
15    set1.EACH DO |e|
16       YIELD (e)
17    END
18    Set1[E1].Resource = attVal [Resource].value[0];
19    Set1[E2].Resource = attVal [Resource].value[1];
20    ASSERT EQUALITY (set1, Date);
21 END
```

The BPL parser generates a file in the format of JSON including the specification of the reference pattern as follows.

Code 7.4: JSON file extracted from the described behavior in Example 2

```
{
"name": "Pattern Example-2",
"length": "2",
"bound_variable": [
   {"var1":"role" },
   {"var2":"action" },
   {"var3":"resource" },
   {"var4":"location" },
   {"var5":"date" }  ],

"constraint": [
   {  "type": "unary",
      "operand1": "event1",
      "attr": "role",
      "value": "R-1",
      "operator": "=="
   },
   {  "type": "unary",
      "operand1": "event1",
      "attr": "action",
      "value": "A-10",
      "operator": "=="
   },
   {  "type": "unary",
      "operand1": "event1",
      "attr": "resource",
      "value": "S-1",
      "operator": "=="
   },
   {  "type": "unary",
      "operand1": "event2",
      "attr": "resource",
      "value": "S-7",
      "operator": "=="
   },
   {  "type": "unary",
      "operand1": "event1",
      "attr": "location",
      "value": "L-16",
      "operator": "=="
   },
   {  "type": "binary",
      "operand1": "event1",
      "operand2": "event2",
      "attr": "role",
      "operator": "=="
```

```
47      },
48      {   "type": "binary",
49          "operand1": "event1",
50          "operand2": "event2",
51          "attr": "location",
52          "operator": "=="
53      },
54      {   "type": "binary",
55          "operand1": "event1",
56          "operand2": "event2",
57          "attr": "action",
58          "operator": "=="
59      },
60      {   "type": "binary",
61          "operand1": "event1",
62          "operand2": "event2",
63          "attr": "date",
64          "operator": "=="
65      }
66  ]
67  }
```

Example 3: The administrator wants to find instances of the behavior of all physicians who should perform a sequence of actions such as read the diagnostic report, update diagnostic report, and delete diagnostic report within 8 hours during a day.

Code 7.5: A query for proximity pattern in the post-obligation

```
1  PATTERN Example-3;
2  BEGIN
3      DEFINE Event = < User, Action, Patient, Location,
           Resource, Date, Time >
4      E[] = new Event (3);
5      set1 = { E[1], E[2], E[3] };
6      attVal = { <Action, ("read","update", "delete")>, <Role,
           ("physician")>, <Resource, ("diagnostic report")>};
7      CALL Op1 ( set1 , attVal) DO |event|
8          event.Role = attVal[ Role ].value[0];
9          event.Resource = attVal[Resource].value[0];
10     END
11 END
12 /* define operation */
13 DEF Op1 (set1, attVal)
14     set1. EACH DO |e|
```

```
15        YIELD (e)
16     END
17     Set1[E1].Action = attVal[Action].value[0];
18     Set1[E2].Action = attVal[Action].value[1];
19     Set1[E3].Action = attVal[Action].value[2];
20     ASSERT DISTANCE (set1, Time ) < 8;
21     ASSERT EQUALITY (set1, Date);
22 END
```

The following file will be generated after parsing the above BPL code. The file contains the structure as well as constraints of the behavior pattern. As can be seen the file gives information about the size of pattern, attributes that participate in constraints and the specification of constraints.

Code 7.6: JSON file extracted from the described behavior in Example 3

```
1 {
2 "name": "Pattern Example-3",
3 "length": "3",
4 "bound_variable": [
5    {"var1":"action" },
6    {"var2":"role" },
7    {"var3":"date" },
8    {"var4": "time"} ],
9
10 "constraint": [
11    {  "type": "unary",
12       "operand1": "event1",
13       "attr": "action",
14       "value": "A-3",
15       "operator": "=="
16    },
17    {     "type": "unary",
18       "operand1": "event2",
19       "attr": "action",
20       "value": "A-10",
21       "operator": "=="
22    },
23    {  "type": "unary",
24       "operand1": "event3",
25       "attr": "action",
26       "value": "A-16",
27       "operator": "=="
28    },
29    {     "type": "unary",
```

```
30          "operand1": "event1",
31          "attr": "Role",
32          "value": "R-1",
33          "operator": "=="
34      },
35      {   "type": "binary",
36          "operand1": "event1",
37          "operand2": "event2",
38          "attr": "Role",
39          "operator": "=="
40      },
41      {   "type": "binary",
42          "operand1": "event2",
43          "operand2": "event3",
44          "attr": "Role",
45          "operator": "=="
46      },
47      {   "type": "binary",
48          "operand1": "event1",
49          "operand2": "event2",
50          "attr": "date",
51          "operator": "=="
52      },
53      {   "type": "binary",
54          "operand1": "event2",
55          "operand2": "event3",
56          "attr": "date",
57          "operator": "=="
58      },
59      {   "type": "binary",
60          "operand1": "event1",
61          "operand2": "event2",
62          "attr": "time",
63          "operator": "::",
64          "value": "8"
65      },
66      {   "type": "binary",
67          "operand1": "event2",
68          "operand2": "event3",
69          "attr": "time",
70          "operator": "::",
71          "value": "8"
72      }
73  ]
74  }
```

Table 7.1: The features of dataset and parameters of pattern matching engine

	Example-1 (Sequence)	Example-2 (Association)	Example-3 (Proximity)
Number of patterns	5	5	5
Instances of each pattern	10	10	10
Size of dataset-Events	30000	30000	30000
Number of traces (sequence of events)	3000	3000	3000
Number of variable	2	5	4
Hard constraint	5	14	10
Soft constraint	0	0	0

We assume that Example 1, Example 2, and Example 3 are queries of behavior patterns described by the administrator for Category 1, Category 2, and Category 3, respectively to study the historical behavior of users. The administrator is interested in finding all instances of each example in order to ensure that users comply the defined post-obligations. Therefore, based on the extracted features of each described behavior in BPL, the pattern matching engine attempts to find the instances. Table 7.1 shows the configuration of the pattern matching algorithm for each query.

The configuration of the VCSP solver determines the quality of generated results by the pattern matching engine. For example, any changes to the number of variables, hard constraints, or soft constraints affects the accuracy, memory and time usage. In this experiment, we did not consider any soft constraints because the goal is to find the instances of a post-obligatory behavior. Consequently, we need the exact matching to verify accurately that users comply post-obligations.

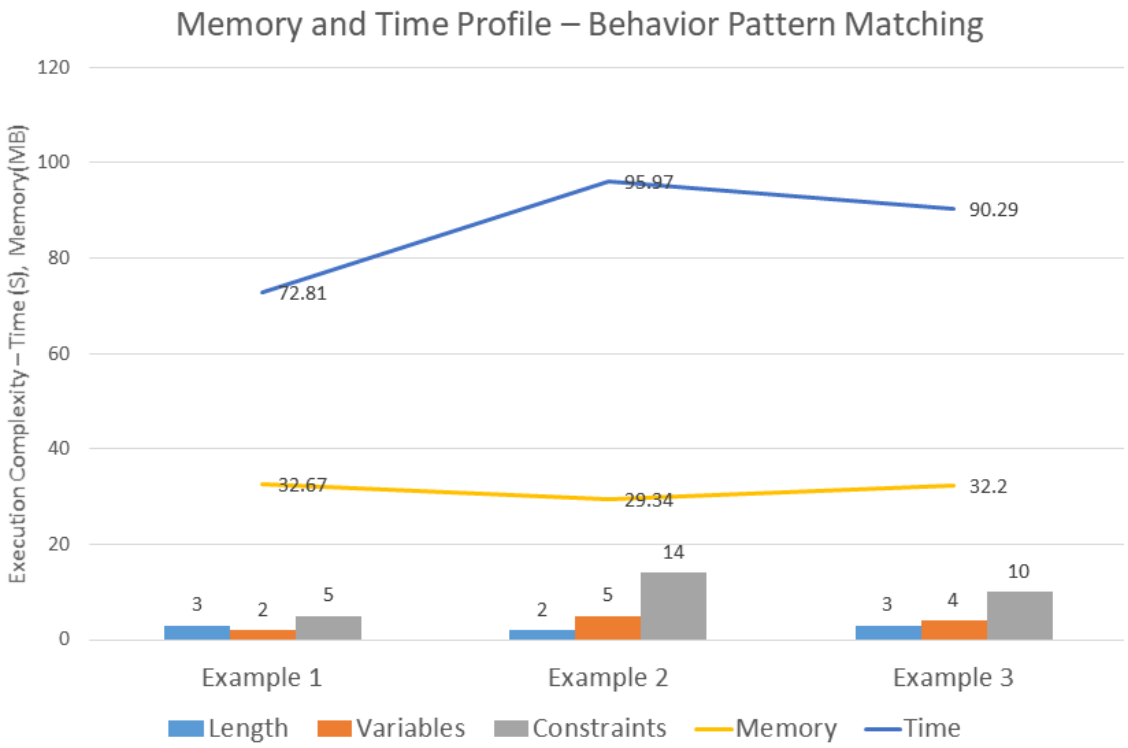

Figure 7.1: Parameters of pattern matching and time and memory usage for three examples

Experiments were conducted on a PC with Intel Core i7 2.0 GHz CPU and 8GB RAM running Windows 10 system. Our dataset was a collection of access events from the distributed medical imaging system. The dataset follows the RFC 3881 standard [79] that defines the format of data to be collected and the set of attributes that need to be captured for security auditing in healthcare domain. We processed the collected logs and converted them into attributed events based on the provided schema in RFC 3881. The dataset was very small for our experiment. Therefore, we extended the dataset by generating synthetic events. We used a dataset generator developed in [80] to generate events. The generator receives parameters of each pattern by a JSON format in order to generate and inject the instances of the pattern into dataset. For

each category of the user behavior, we generated a dataset containing around 30000 events of 100 users during 30 days.

We implemented the preprocessing of event dataset and VCSP solver in Python programming language. In Figure 7.1, x-axis shows the solver parameters and y-axis shows the execution time and memory for each query. Each group of bar graphs shows the value of parameters (i.e. pattern length, number of bound variables, number of constraints) of the solver for each example. We measured the execution time and memory spent on finding the instances of each example. We will discuss the impact of changing parameters on the execution complexity of query patterns in the next section. Furthermore, we will present the capability of BPL to express more complex patterns through changing the parameters of each query.

7.2 Validation for Active Enforcement

Active enforcement is relevant to ongoing obligation where the enforcement is required to be occurred during the access lifetime. Therefore, the result of enforcement impacts the continuity of the access. Our proposed framework leverages the complex event processing technology to verify the enforcement of the obligation.

7.2.1 Validation Framework

Figure 7.2 shows the architecture that utilizes different open source tools to implement the continuous compliance engine.

Event Producer: PEP generates a continuous stream of event data that records the interaction between the user and the resource manager during a session in order to track compliance with access control policies and obligations. Moreover, a stream of sensor data will be collected in order to monitor the particular conditions of the

environment or the status of resources.

Message Broker: The stream of event data will be sent to a Message Broker that supports a publish-subscribe architecture for sharing event data as messages within the event processing modules. We chose Apache Kafka [81] as a real-time streaming data pipeline for our implementation because of:

- The capability of publishing and subscribing to streams of records.
- Building real-time streaming data pipelines that reliably get data between systems or applications
- Building real-time streaming applications that transform or react to the streams of data

Kafka uses a topic as a container where records are retained. Topics are partitioned. Records within a partition are uniquely identified by a sequential id number called the offset. Each record consists of a key, a value, and a timestamp. Several consumers can subscribe to the data written to a topic. Kafka provides four APIs to communicate with other applications.

- Producer API: allows an application to publish a stream of records to one or more Kafka topics.
- Consumer API: allows an application to subscribe to one or more topics and access the stream of records.
- Streams API: allows an application to process an input stream from one or more topics and produce an output stream.
- Connector API: connects Kafka topics to existing applications or data systems through building and running producers or consumers.

Kafka is run as a cluster on one or more servers. Kafka cluster typically consists of multiple brokers to maintain load balance. Each broker can handle hundreds of thousands of reads and writes per second. Kafka brokers are stateless. Consequently, Apache ZooKeeper [82] is used for maintaining their cluster state.

Zookeeper: ZooKeeper was designed to store coordination data such as status information, configuration, location information, etc. ZooKeeper performs its tasks through a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. ZooKeeper is used for managing and coordinating Kafka brokers.

Flink CEP: Flink is an open-source framework for distributed stream processing on data that is continuously produced. Flink processes continuous data at large scale and can provide accurate results in the case of out-of-order or late-arriving data. The main features of Flink are state management, handling of out-of-order data, and flexible windowing that are essential for computing accurate results on unbounded datasets. Flink provides stateful computations. It means that Flink can maintain an aggregation or summary of data that has been processed over time, in contrast to stateless computation, that the application looks at each individual event and creates some output based on the last event.

Figure 7.2 also shows a bottom-up view of the Flink stack. Flink can run in the cloud or on premise. In our experiment, we deployed it on a laptop with a local JVM (Java Virtual Machine). Flink's core (runtime) relies on a distributed streaming dataflow engine so that data is processed as an event-at-a-time rather than as a series of batches. Applications can use the capability of Flink through APIs such as DataStream API, DataSet API, Table API, and Streaming SQL. We utilized DataStream API to implement filtering, defining windows, and aggregating on event streams. Flink also provides several libraries so that we used the Complex Event

Processing (CEP) library to create a BPL query execution engine.

FlinkCEP is the complex event processing library that allows an application to easily detect complex event patterns in an event stream. An administrator describes a behavior using BPL based on significant features and constraints of the behavior. FlinkCEP provides a PatternAPI that we use to convert the described pattern to the format that can be used by Flink pattern matching engine. According to our proposed model for behavior pattern evaluation, a described behavior is formally formulated as a triple: $P = (< e_1, e_2, ..., e_n >, F, C)$.
Each pattern consists of multiple states representing by $< e_1, e_2, ..., e_n >$. Each e_i as a placeholder plays the role of a state. The simple conditions in F direct the movement from one state to the next. Such conditions can be the contiguity of events or a filter condition based on an attribute value. The PatternAPI provides a *where* method that can specify conditions.

Adding multiple conditions for a state is possible by combining conditions using the logical AND or logical OR operators. The strict contiguity of two succeeding events can be created via the *next* method. Strict contiguity means that no other events can occur in between two matching events, whereas Non-strict contiguity means that other events are allowed to occur in-between two matching events. The method *followedBy* can specify non-strict contiguity.

The implemented pattern matching engine based on FlinkCEP matches the specified pattern against input stream and captures a sequence of events that satisfy the simple conditions. We have to apply constraints to select those sequences that satisfy exactly the characteristics of desired pattern. The PatternAPI provides a method called *within* to specify a temporal constraint for the pattern to be valid. Other type of constraints can be implemented as a *PatternSelectFunction* that is required for *select* method.

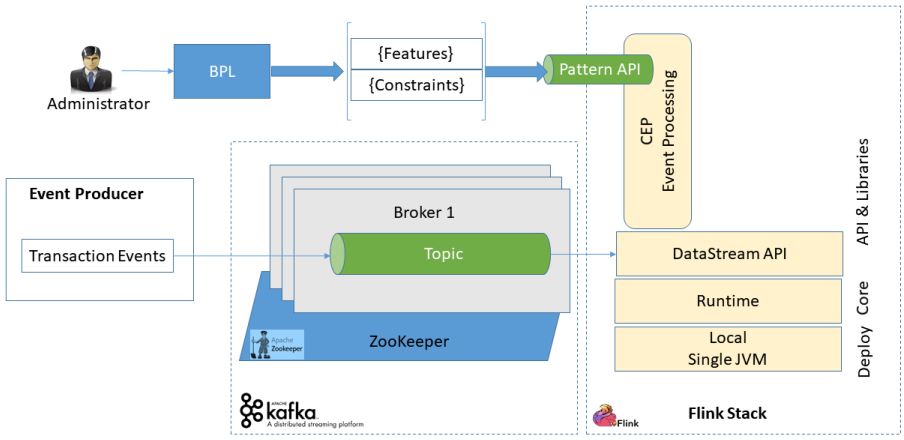

Figure 7.2: The architecture to implement the compliance framework

7.2.2 Event Representation

Flink supports different categories of data types such as Tuples, special POJOs (Plain Old Java Object), Primitive Types, Regular Classes, and Hadoop Writables. Flink applies the following restrictions to support POJO data type:

- The class must be public.
- It must have a public constructor without arguments.
- All fields are either public or must be accessible through getter and setter functions.

It seems that the fulfillment of such requirements makes a POJO class similar to a JavaBeen class. However, as shown in Appendix B, we use POJO data type to present the event.

7.2.3 Pattern Query

The data stream model considers data that are not stored into a traditional relational database because of inherent nature of such data which rely on contiguity, rapidity,

and real-time concept. It is inefficient to simply load the arriving events into a traditional relational database and then write queries for desirable patterns in typical application domains such as processing data generated by sensor networks, IP network traffic analysis, and mining text message streams. Such query languages depend totally on the format of data representation. For example, SQL works truly on the table through providing relational operators. Since querying for behavior patterns can be applied on different sources with various format, SQL-based query is not efficient. We need a language with proper flexibility and minimum dependency to data representation. Consequently, we propose BPL language with low dependency to the data representation in order to query a variety of behavior patterns. However, the challenge is how we can measure the expressive power and evaluation complexity of the BPL.

In this section, we clarify how we measure the expressiveness and evaluation complexity of the behavior pattern language as a pattern-based query language. The expressiveness of a query language is the set of queries it can express [83]. In other words, the expressive power shows what can and what cannot be expressed in the query language. The supported operators and structures provide enough flexibility for the language to compose different kind of queries. Therefore, the expressiveness of the BPL is determined by its supported operators and structures to determine different specifications such as sequence, frequency, conjunction, disjunction, and a variety of constraints for a particular behavior pattern. Therefore, the group of common features that can be used to measure the expressiveness of BPL consists of:

- Event type: event representation in terms of number of attributes and domain values

- Event abstraction: combining of events to make complex event

- Pattern representation: the structure for representing the pattern
- Event pattern constraints: relationships between events
- Aggregation concept: counting of events, counting the sequence of events, average on attribute values, etc.

We measure the expressiveness by the number of example queries selected from pattern space. The set of all users traces in a typical access control system creates the behavior pattern space. Since there is an infinite number of pattern queries, we choose patterns in order to address the mentioned features or the combination of them. Each pattern is described in BPL and then is tested whether the proposed framework can evaluate the described pattern.

It is obvious that there is a trade-off between the expressiveness of a query language and the computational resources needed to evaluate a query stated in the language. The complexity of evaluation presents whether the language is usable practically to specify a query. We look at the complexity in terms of processing time and the memory usage for executing the query.

We put query examples in several categories so that they address the expected features in behavior patterns, although they may have overlap in some features. Each query will be defined to detect the instances of a particular pattern. These patterns are different in terms of pattern length (number of events), association degree (shared attribute values/total attributes) and the number of constraints.

Sequence:

Sequence presents the behavior pattern of those users who generate a series of events only by changing the value of a specific attribute. Therefore, the queries in this

category detect a sequence of events where a particular attribute value changes sequentially. We consider the following query that inquires into the behavior of users who work in different wards of a hospital and their role can change during a day. However, we want to find the behavior of users whose actions change sequentially.
Query 1: Find behavior of users who perform first "read" action and next "write" action.

In terms of expressiveness, the sequence is a combination of events to address the ordered-based relationship between events. Since BPL utilizes the ordered-set to present the pattern, it has enough expressiveness to present such relationships. To measure the complexity, we apply the query on a synthetic event stream using our proposed framework.

Code 7.7 shows the description of Query 1 in BPL. The code presents the structure, features, and constraints that will be used to configure the pattern matching component implemented in FlinkCEP library.

Code 7.7: The BPL code for describing the Query 1

```
1 PATTERN Seq_Query1;
2 BEGIN
3    DEFINE Event = <User, Action, Patient, Location, Resource
        , Date, Time>
4    E[] = NEW Event(3);
5    Set1 = {E[1], E[2]};
6    attVal = {<Action, ("read", "write")>};
7    CALL Op1 (set1, attVal);
8
9 END
10 /* define operation */
11 DEF Op1 (Set1, attVal)
12    FOREACH event IN Set1 DO
13       Set1[E1].Action = attVal[Action].value[0];
14       Set1[E2].Action = attVal[Action].value[1];
15    ASSERT EQUALITY (set1, User);
16 END
```

FlinkCEP uses the BPL description to form a target pattern in order to match

against incoming event stream. For this purpose, as shown in Code 7.8, the target pattern is increasingly created by defining states that delegate the corresponding events and their features. Measurement is a POJO class (Appendix B) whose properties will be inherited by EventMeasurement to make the type of each state. Line 2 assigns a name to the first state (called "First") and then defines the condition by using *where* clause. The condition plays the role of a transition function to move to the next state. The Line 5 specifies the name of the second state (called "Second") and its condition. The code is written based on Java 8 lambada expressions.

Code 7.8: The syntax is used by FlinkCEP for the Query 1

```
1 Pattern<Measurement, ?> targetPattern = Pattern
2   .<Measurement>begin("First")
3   .subtype(EventMeasurement.class)
4   .where(evt -> evt.getAction().equals("read") )
5   .next("Second")
6   .subtype(EventMeasurement.class)
7   .where(evt -> evt.getAction().equals("write") )
```

Each discovered pattern is an instance of the target pattern so that it is fully compliant with the structure of the target pattern. However, we also need to apply constraints to find the exact matched patterns. In this case, the same user should issue the both events that create the pattern. We implement the constraints as a pattern selection function that utilizes the *select* method provided by Flink.

To discover the complexity trend of Query 1, we increase the length of target pattern (query length) and keep constant the value of the association degree.

The average processing time for different version of query 1 has been illustrated in Figure 7.3. Furthermore, Figure 7.4 shows the memory usage for query processing. As shown in these figures, as the size of query for a sequence pattern increases, the complexity (time and space) of query processing will be grown. Memory usage increases nearly linear by augmenting the target sequence pattern. In contrast, the

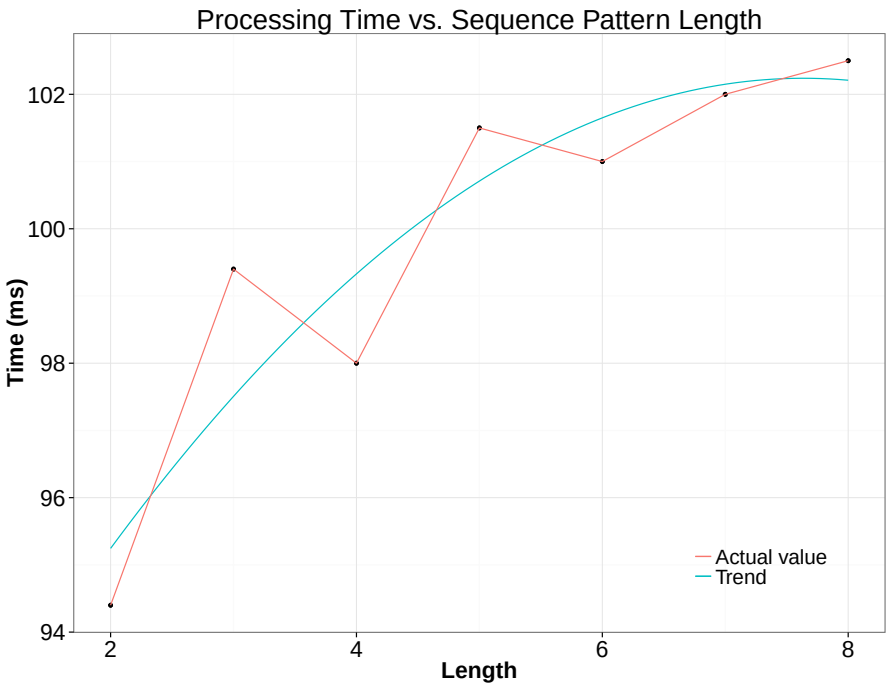

Figure 7.3: Processing time: the time required to process sequence pattern queries

processing time tends to be constant after a remarkable increase up to length 6.

Association

The association query pattern searches the instances of a pattern with the association between events. Association can intuitively be viewed as a multiple-way correlations between attributes. To quantify the level of association, we look at attributes as a free or bound attribute. It means that if an attribute participates in a constraint that shows the correlation between events, it will be a bound attribute. The number of bound attributes over the total attributes defines the association degree of a behavior pattern.

Query 2: Find the behavior of all users who send access request from emergency ward, urology ward, and cardiology ward.

We consider the Query 2 as a basic query for a pattern with low association degree.

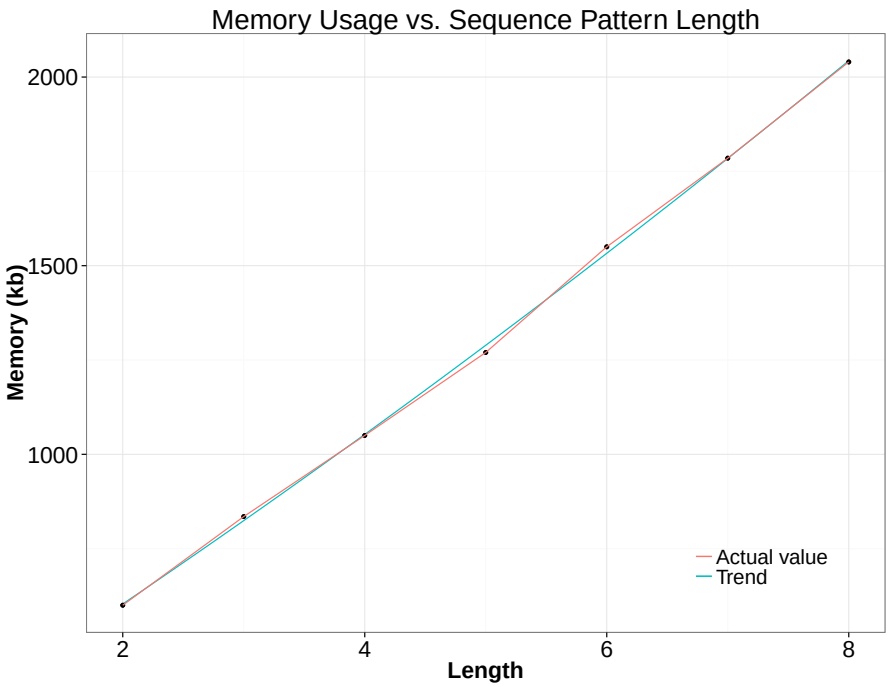

Figure 7.4: Memory usage: the amount of memory needed to process sequence pattern queries

During our experiment, we investigate how increasing the association degree affects the complexity. BPL has enough express power to describe such queries because of its operators that provide mechanism to access attributes of each event and make correlation between them.

The Query 2 can be described as a pattern with three events issued by a user from three different locations. In our experiment, we consider eight attributes for each event. The attribute value of *UserId* is shared between all three events in case of Query 2. Therefore, the association degree is 1 over 8. To calculate the complexity, we keep constant the number of events, but increase incrementally the association degree.

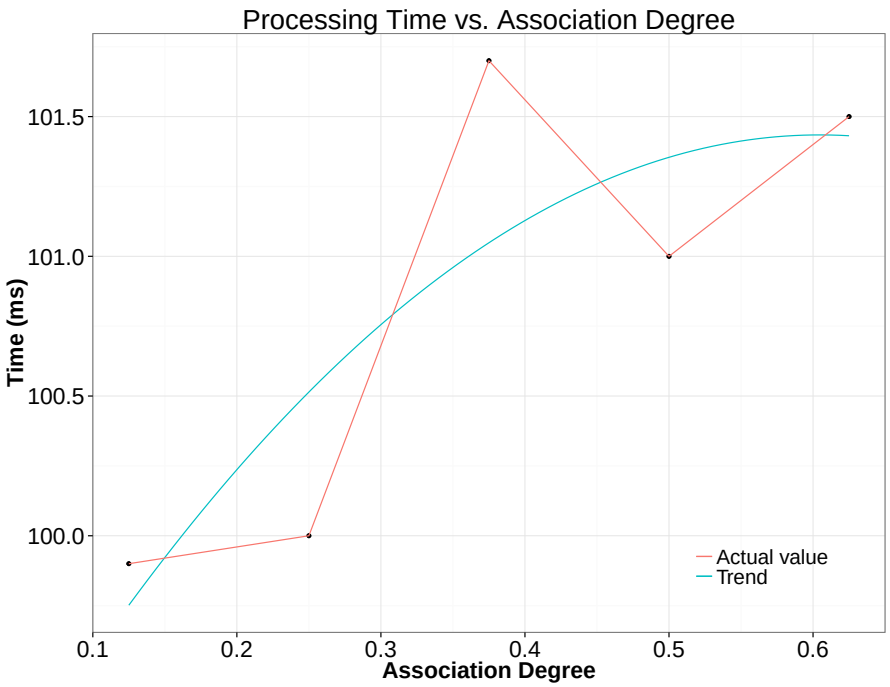

Figure 7.5: Processing time: the time required to process association pattern queries

Proximity

Proximity describes the behavior of users who perform a sequence of actions that are needed to keep a state during of a process in the particular time. For example, stateful processing is an important part to monitor the behavior of a user who must watch an ad after using a free service for three times within five minutes in order to keep using the service.

Defining a window is the common mechanism to take a snapshot of the stream in order to verify a continuous state across the stream. BPL provides enough express power to address different kind of windows that can be defined based on a specific attribute. Time-based windows such as tumbling or sliding are more common in event stream processing. Tumbling window is useful for dividing stream in to multiple discrete batches each of which tumbles once the window is evaluated. Sliding window

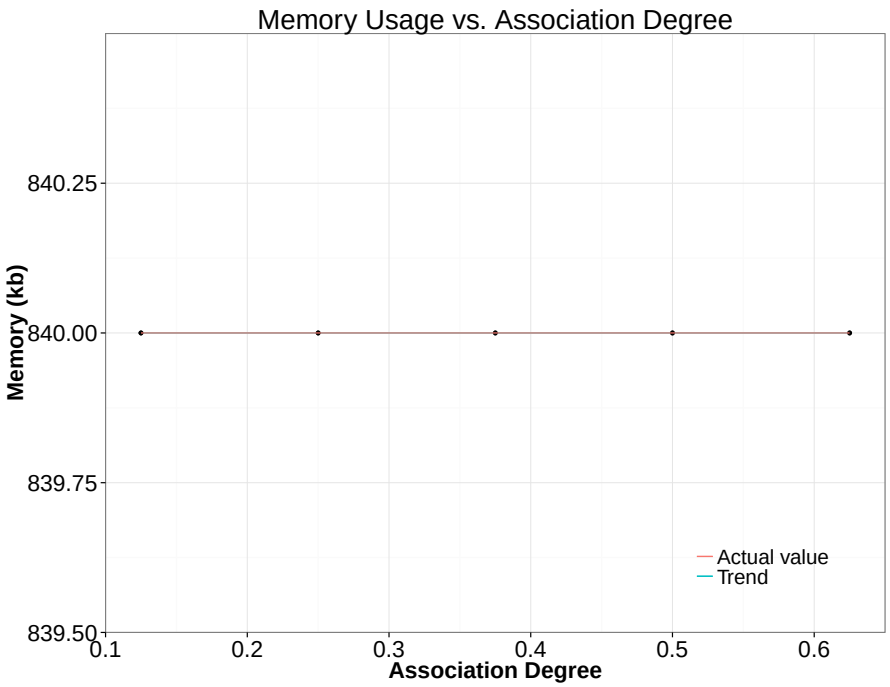

Figure 7.6: Memory usage: the amount of memory needed to process association pattern queries

creates overlapping windows compared to discrete windows in tumbling window. We define Query 3 that inquires the behavior including three sequential actions in a time window.

Query 3: Find behavior of users who perform actions in order "A-1", "A-2" and finally "A-3" action within 10 seconds.

To measure the complexity of proximity query, we change the size of time window during experiment and then measure the CPU time and memory usage for the query processing. Figures 7.7 and 7.8 illustrate the results for time and memory usage respectively. Both will rise by increasing the size of window.

Negation

The expressiveness of BPL is not enough to address the negation concept in behavior

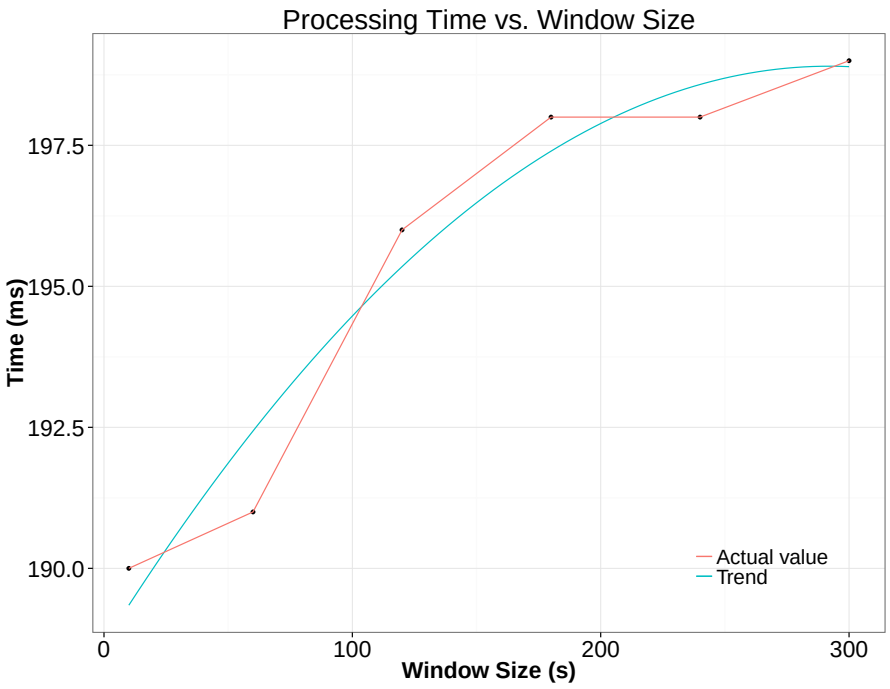

Figure 7.7: Processing time: the time required to process proximity pattern queries

pattern. Conceptually, order-set can cover the negation, but we did not consider a negation operator for BPL. In access control perspective, we model the obligatory behavior in positive view. It means that we expect that the obligatory sequence of events occur certainly in the system. However, negation concept deals with a situation that non-occurrence of events is desirable. Meanwhile, it is not straightforward to efficiently model the non-occurrence of events by our proposed NFA when there are correlations between the negated and non-negated events. If we look at the negation as a gap in the sequence of events then it can be specified by BPL through ignoring negated events. In this way, we actually do not model directly the non-occurrence events.

Repetition

Repetition in a pattern is called Kleene closure (e^*) that means an event e can oc-

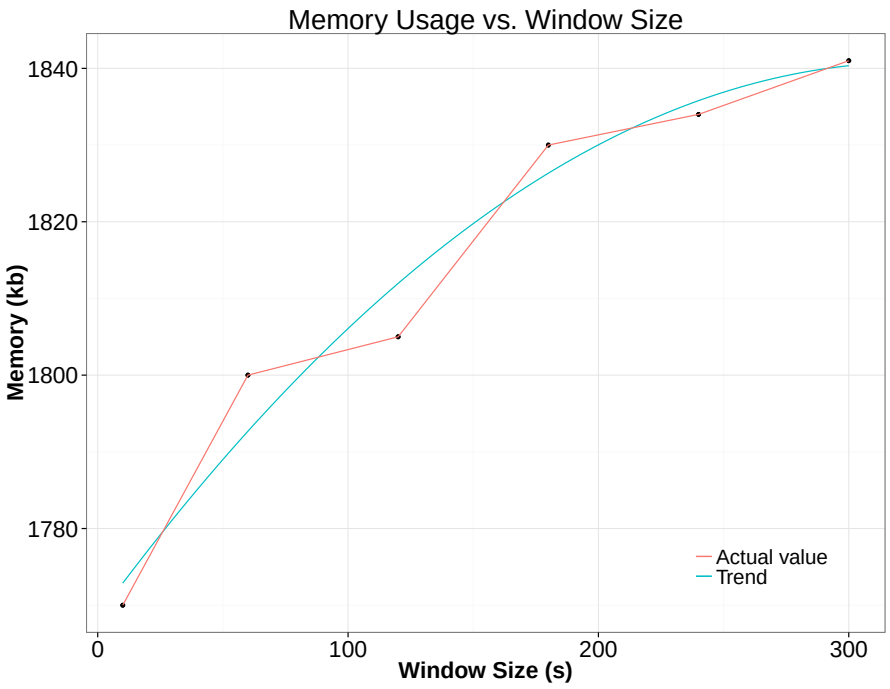

Figure 7.8: Memory usage: the amount of memory needed to process proximity pattern queries

cur zero or more times. Kleene closure has been well studied for regular expression matching. Using Kleene closure on event steam imposes new requirements that fundamentally distinguish it from conventional regular expression matching. For example, how to select relevant events to the Kleene closure from a stream mixing relevant and irrelevant events. Moreover, the termination criteria in the complex event processing problem also differ from regular expression matching where the input is a finite string. However, the input in CEP is an infinite stream and queries can proceed as far as possible by skipping irrelevant events.

If a pattern presents behavior that includes repetitive events, BPL cannot express such a behavior pattern if the number of occurrence has not predefined. Consequently, it is not straightforward to specify precisely the pattern for the proposed framework.

Chapter 8

Conclusion and Future Work

8.1 Summary

Chapter 1 introduced the motivation of this thesis by explaining the challenges of existing policy-based access control systems. The goal of this thesis is to design a language to describe precisely mandatory behavior as obligations and integrate them into policy rules in order to mitigate the misuse of access control policies. Moreover, such a language enables the administrator to write queries in meticulous detail to verify the enforcement of the obligation. In Chapter 2, we reviewed the related works to this thesis including an overview of behavior modeling, pattern matching techniques, and obligation models in different access control policy languages.

Our proposed model for behavior was discussed in Chapter 3. We modeled the behavior of a user based on the order-set of events. We also presented the syntax and semantics of the behavior pattern language to specify a user behavior. Chapter 4 presented our model for obligation and our method to leverage the Session Initiation Protocol to handle the obligation session. Moreover, we proposed an approach to integrate the described obligatory behavior into XACML as the standard access

control policy language.

We developed a CSP-based pattern matching engine that is able to perform exact and approximate matching in order to match the described behavior against the recorded users' events in Chapter 5. Setting the engine to do exact matching enables it to approve the enforcement of the post-obligation where the obligatory behavior should be performed after accessing a resource. Such pattern matching is not applicable for ongoing obligations. Therefore, in Chapter 6, we designed a CEP-based framework to develop a continuous enforcement mechanism for obligation while the access is in progress. We utilized our proposed behavior pattern language BPL, as the pattern query language in this case. We presented our experiments to use BPL for static and active enforcement in Chapter 7. Moreover, the expressiveness of BPL to generate different kinds of queries was evaluated in that chapter.

8.2 Limitations

The proposed approach in this thesis to specify the obligatory behavior has the following limitations:

- We used BPL to specify a limited range of behavior patterns such as association, sequencing, and proximity. In practice, there are a variety of behavior patterns by combining events. It is required to extend the operators of BPL in order to specify such patterns.

- Considering a user behavior as an ordered-set of events has limited our proposed pattern matching algorithm to work on ordered events. Therefore, the developed framework is unable to find patterns on a stream including the out-of-order events.

- BPL limits the description of a pattern based on the number of events required to fulfill the behavior requirements.

8.3 Conclusion

Policy violation is still one of the major reason of information security incidents in recent years. Access control as an important component of security system relies on predefined policies. The majority of access control models consider the obligation as a black box and do not provide clear specifications and enforcement models. In this research, we looked at different aspects of obligations and proposed a generic model based on the behavior of the subject who is responsible to fulfill the obligation. Such a model not only helps us to provide semantics for obligatory behavior by combining primitive events, but also enables us to perform empirical practices as opposed to present just a logical model.

The standard policy specification language, XACML, does not have enough expressiveness to address complex behavior as an obligation. Therefore, we proposed Behavior Pattern Language (BPL) that allows us to specify precisely the features of the user behavior patterns. We used the BPL to describe the obligatory behavior for two purposes. First, the described features of the behavior were used to specify the obligation elements of the policy. Second, we used BPL to describe a behavior pattern as a target pattern, called query, in order to find the instances of it. This usecase was applied to enforce the obligation.

We considered two different approaches for enforcement with respect to the type of obligations. Static enforcement was proposed for the post-obligation. We modeled the static enforcement as a constraint satisfaction problem. Solving such a problem relies on searching in audit logs to verify the fulfillment of the post-obligation. Active

enforcement was proposed for ongoing-obligation. This kind of obligation impacts the current access session. Therefore, we leveraged the capabilities of SIP protocol to create the obligation session in order to manage the current access session. We used the complex event processing technique to create a framework.

We developed both our proposed enforcement mechanisms. We ran several experiments for each mechanism in order to prove that our proposed language is able to specify different categories of obligatory behavior. The experiments prove that our proposed methods for obligation specification and enforcement are generic to be applied into various application domains unlike the hard-coded and proprietary approaches that are used to solve this problem.

8.4 Future Work

Pervasive computing environments are growing fast due to the developing reliable applications based on IoT and cloud computing. Since the conditions of access to resources cannot be predefined well, existing access control paradigms are not efficient in such environments. Considering the behavior of involved entities to design the access control is a big step towards having a comprehensive and reliable access control system in dynamic environments. We have identified the following areas for extension of the work presented in this thesis:

- Adaptive access control systems are essential for dynamic environments because they rely on context and trust that help the system to adapt for changes in the behavior of users. Trust-based mechanism to provide access is getting to be the dominant method for access control in cloud environments. However, finding a trust threshold is often a challenge for such access control systems. Examining the behavior of users is required to balance the level of trust against risk. The

proposed approaches in this thesis are capable of specifying and matching the desirable behavior against a set of events in order to run such examinations. However, we need to extend our method to measure the amount of deviation between a behavior pattern and a common behavior. The deviation can be used as a parameter to assign a level of risk to the user.

- The accurate implementation and enforcement of different components of access control policies depend on the syntactic and semantic correctness of policy specification. The access control system includes different points to handle user's requests, make the decision to grant or deny the access and finally to enforce the obligation. The implementation of such points is an error-prone task because the specification language often lacks flexibility to express the complex logic inside of security and obligation requirements. Moreover, developers may seed malicious codes or implement the logic of authorization enforcement inappropriately. Therefore, testing is really essential in order to ensure that the implemented policies syntactically and semantically satisfy the intentions of the policy authors.

  Presenting a fault model and using mutation testing are the most common methods to test XACML policies. A fault model represents the types of faults that typically occur in the policy. Syntactic faults are a result of simple typos and XSD (XML Schema Definition) can detect them in XACML-based policies. However, semantic faults are associated with the logical constructs of the policy language such as policy combining algorithm, rule combining algorithm, rule evaluation order, and using improperly logical AND (AllOf) instead of logical OR (AnyOf) or vice versa.

  A fault model is used to create policy mutants in order to measure the effective-

ness of the fault detection in an automatic test generation. A mutation operator is used to implement the fault model. In other words, the mutation operator interprets the fault model in order to modify the appropriate policy elements to generate the faults. The modified policy is called a mutated policy. Actually, each mutant is a variation of the original policy where a particular type of fault is injected. Mutation testing involves a given policy, a set of mutation operators, a number of mutant policies, and a set of access control requests. Each request will be evaluated against both the original policy and mutant policy. If the responses are different, the mutant policy will be killed. Otherwise, a potential faulty situation has been found. Using model checking is another approach to find the differences between a logic representation of security requirements and policy rules. Further researches and experiments are required to choose the effective methods for testing the access control obligations.

- The proposed static enforcement relies on a search algorithm because we modeled the behavior pattern matching as a constraint satisfaction problem. Using a constraint propagation technique allows our algorithm to satisfy all hard constraint in order to perform the exact matching. However, the proposed search algorithm is non-deterministic. Such an algorithm does not promise to find all instances of a particular behavior pattern for the approximate pattern matching purposes.

  Moreover, scalability is a challenge for the search algorithm particularly in the case of approximate matching. Further researches are required to find the appropriate heuristic to decrease the search space. Applying machine learning techniques such as clustering or classification may be useful to define such a heuristic.

- Since this research presents our methods to develop the capabilities of BPL to specify and enforce different kinds of obligations, we focus on the exact matching in both static and active enforcement. BPL is also useful to study about the suspicious behavior. Such behavior is conducted by trusted users who misuse resources and hence compromise the system security aspects. In this context, a thorough analysis of the users behavior will provide the opportunity for the system administrators to identify the users whose behaviors are suspicious and may jeopardize the data privacy and integrity of the IT enterprises. Suspicious behavior is very similar to the normal behavior so that finding manually the difference by the administrator is a challenging task.

  The exact description of a suspicious behavior is a starting point for further investigations. BPL can be useful for this purpose and we call a described behavior as a conjectural behavior. Once the behavior is defined, a pattern matching engine will search the systems event log repository to discover the instances of the defined behavior that satisfy different hard and soft constraints. We need to extend our proposed framework to examine the instances in order to verify that the described behavior is a malicious behavior.

Bibliography

[1] G. Antoniou, M. Baldoni, P. Bonatti, W. Nejdl, and D. Olmedilla. Rule-based policy specification. In *Secure Data Management in Decentralized Systems*, pages 169–216. Springer Science+Business Media, LLC, 2007.

[2] IBM security, reviewing a year of serious data breaches, major attacks and new vulnerabilities. In *IBM X-Force Research, 2016*. https://www.ibm.com/security/data-breach/threat-intelligence, Accessed 20 Sep, 2017.

[3] Vormetric Data Security. 2015 vormetric insider threat report, trends and future directions in data security. In *Research Report*. https://dtr.thalesesecurity.com/insiderthreat/2015/, Accessed 20 Sep, 2017.

[4] Data breach investigations report. In *A study conducted by the Verizon Business RISK team, 2009*. http://www.verizonbusiness.com/resources/security/reports, Accessed 20 Sep, 2017.

[5] Information security, cyber threats and data breaches illustrate need for stronger controls across federal agencies. In *United States Government Accountability Office (GAO), 2015*. http://www.gao.gov, Accessed 20 Sep, 2017.

[6] IBM security, the year of the mega breach. In *IBM X-Force Threat Intelligence Index, 2016*. https://www.ibm.com/security/xforce, Accessed 20 Sep, 2017.

[7] L. Cao. Behavior informatics: A new perspective. *IEEE Intelligent Systems*, pages 62–80, 2014.

[8] M. Yarmand, K. Sartipi, and D. Down. Behavior-based access control for distributed healthcare systems. *Journal of Computer Security*, pages 1–39, 2013.

[9] M. Zerkouk, A. Mhamed, and B. Messabih. User behavior and capability based access control model and architecture. In *Springer Science and Business Media*, pages 291–299. Springer Science and Business Media, 2013.

[10] S. Stolfo, S. Bellovin, S. Hershkop, A. Keromytis, S. Sinclair, and S. Smith. Insider attack and cybersecurity: Beyond the hacker. Springer, New York, 2008.

[11] R. Rieke, M. Zhdanova, J. Repp, R. Giot, and C. Gaber. Fraud detection in mobile payments utilizing process behavior analysis. In *IEEE International Conference on Availability, Reliability and Security*, pages 946 – 953. IEEE, 2013.

[12] L. Cao. In-depth behavior understanding and use: the behavior informatics approach. *Journal of Information Sciences, vol. 180, no. 17*, pages 3067- 3085, 2010.

[13] A. Kirou, B. Ruszczycki, M. Walser, and N. Johnson. Computational modeling of collective human behavior: The example of financial markets. pages 33–41. Springer Verlag Berlin Heidelberg, 2008.

[14] S. Angeletou, M. Rowe, and H. Alani. Modelling and analysis of user behaviour in online communities. pages 35–50. Springer Verlag Berlin Heidelberg, 2011.

[15] R. Vishnu Priya and A. Vadivel. User behaviour pattern mining from weblog. *International Journal of Data Warehousing and Mining, vol. 8, no. 2*, pages 1–22, 2012.

[16] C. Wang and L. Cao. Modeling and analysis of social activity process. pages 21–35. New York, Springer, 2012.

[17] C. Wang and L. Cao. Reasoning about rational agents. Cambridge: MIT Press, 2000.

[18] G. P. Zarri. Behaviour representation and management making use of the narrative knowledge representation language. In *Behavior Computing*, pages 37–56. London, Springer-Verlag, 2012.

[19] H.L Bui. Survey and comparison of event query languages using practical examples. Ludwig Maximilian University of Munich, 2009.

[20] N. F. Sandell, R. Savell, D. Twardowski, and G. Cybenko. Hbml: A language for quantitative behavioral modeling in the human terrain. In *Social Computing and Behavioral Modeling*, pages 180–189. Springer, 2009.

[21] W. Grieskamp and N. Kicillof. A schema language for coordinating construction and composition of partial behavior descriptions. In *SCESM06, Shanghai*, pages 59–66, 2006.

[22] H. Gharaee, S. Seifi, and N. Monsefan. A survey of pattern matching algorithm in intrusion detection system. In *7th International Symposium on Telecommunications (IST'2014)*, pages 946–953. IEEE, 2014.

[23] G. Navarro. Pattern matching. *Journal of Applied Statistics, vol. 31, no. 8*, pages 925–949, 2004.

[24] D. Adjeroh, T. Bell, and A. Mukherjee. Exact and approximate pattern matching. In *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching*, pages 187–263. Springer Science+Business Media, LLC, 2008.

[25] G. Cormode and S. Muthukrishnan. The string edit distance matching problem with moves. In *ACM Transactions on Algorithms (TALG), Volume 3 Issue 1*, pages 1–19, 2007.

[26] H. Wang. All common subsequences. In *IJCAI-07*, pages 635–640, 2007.

[27] B. Cadonna, J. Gamper, and M. H. Bhlen. Sequenced event set pattern matching. In *14th International Conference on Extending Database Technology*, pages 33–44. ACM, 2011.

[28] K. Scarfone and P. Mell. Guide to intrusion detection and prevention systems. In *NIST Special Publication 800-94*. National Institute of Standards and Technology (NIST), 2007.

[29] H. Liao, C. R. Lin, Y. Lin, and K. Tung. Intrusion detection system: A comprehensive review. *Journal of Network and Computer Applications, vol. 36*, pages 16–24, 2013.

[30] E. Norige and A. Liu. A de-compositional approach to regular expression matching for network security applications. In *36th International Conference on Distributed Computing Systems*, pages 680–689. IEEE, 2016.

[31] X. Yu, W. Feng, and D. Yao. O3fa: A scalable finite automata-based pattern-matching engine for out-of-order deep packet inspection. In *2016 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 1–11. IEEE, 2016.

[32] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation, Third Edition*. Addison-Wesley, 2007.

[33] F. Yan. Efficient IRM enforcement of history-based access control policies. Master's thesis, University of Regina, 2008.

[34] Y. Elrakaiby, F. Cuppens, and N. Cuppens-Boulahia. Formal enforcement and management of obligation policies. *Data & Knowledge Engineering, vol. 71, no. 1*, pages 127–147, 2012.

[35] L. Kagal, T. Finin, and A. Joshi. A policy language for a pervasive computing environment. In *4th International Workshop on Policies for Distributed Systems and Networks (POLICY'03)*. IEEE, 2003.

[36] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. In *Workshop on Policies for Distributed Systems and Networks, Bristol, UK*, pages 18–39. Springer-Verlag LNCS, 2001.

[37] Ponder2: The ponder2 policy environment. www.ponder2.net, Accessed 20 Sep, 2017.

[38] eXtensible Access Control Markup Language (XACML), version 3.0, oasis standard, january 2013. https://www.oasis-open.org/, Accessed 20 Sep, 2017.

[39] J. Park and R. Sandhu. The UCON_{ABC} usage control model. *ACM Transactions on Information and System Security, vol. 7, no. 1*, pages 128–174, 2004.

[40] K. Irwin, T. Yu, and W. H. Winsborough. On the modeling and analysis of obligations. In *The 13th ACM conference on Computer and communications security*, pages 134–143. ACM, 2006.

[41] X. Zhang, F. Parisi-Presicce, R. Sandhu, and J. Park. Formal model and policy specification of usage control. *ACM Transactions on Information and System Security (TISSEC), vol. 8, no. 4*, pages 351–387, 2005.

[42] B. Katt, X. Zhang, R. Breu, M. Hafner, and J. Seifert. A general obligation model and continuity-enhanced policy enforcement engine for usage control. In *The 13th ACM symposium on Access control models and technologies (SACMAT'08)*, pages 123–132. ACM, 2008.

[43] N. Li, H. Chen, and E. Bertino. On practical specification and enforcement of obligations. In *The second ACM conference on Data and Application Security and Privacy (CODASPY12)*, pages 71–82. ACM, 2012.

[44] I. Rubab, S. Ali, L. Briand, and Y. LeTraon. Model-based testing of obligations. In *14th International Conference on Quality Software*, pages 1–10. IEEE, 2014.

[45] M. Colombo, A. Lazouski, F. Martinelli, and P. Mori. A proposal on enhancing xacml with continuous usage control features. In *Grids, P2P and Services Computing*, pages 133–146. Springer US, 2010.

[46] M. Fernndez. *Programming Languages and Operational Semantics, A Concise Overview*. Springer-Verlag London, 2014.

[47] G.D. Plotkin. *A structural approach to operational semantics*. Technical Report, Aarhus University, Denmark, 1981.

[48] M. Toahchoodee. *Access Control Models for Pervasive Computing Environments*. PhD thesis, Colorado State University, Fort Collins, Colorado, 2010.

[49] L. Gomez and S. Trabelsi. Obligation based access control. In *OTM Confederated International Conferences on the Move to Meaningful Internet Systems*, pages 108–116. Springer-Verlag Berlin Heidelberg, 2014.

[50] C. Feltus, M. Petit, and M. Sloman. Enhancement of business IT alignment by including responsibility components in rbac. In *CAiSE 2010 Workshop Busital 10, Hammamet, Tunisia*, pages 61–75, 2010.

[51] J.Rosenberg, H.Schulzrinne, G.Camarillo, A.Johnston, J.Peterson, R.Sparks, and et al. SIP: Session initiation protocol. RFC 3261. 2002. https://www.ietf.org/rfc/rfc3261.txt, Accessed 20 Sep, 2017.

[52] Openid home page. http://www.openid.net/, Accessed 20 Sep, 2017.

[53] The oauth 2.0 authorization framework. https://oauth.net/, Accessed 20 Sep, 2017.

[54] H. Schulzrinne and E. Wedlund. Application-layer mobility using sip. In *Mobile Computing and Communications Review, VOL. 4, NO. 3*, pages 47–57. ACM, 2000.

[55] S. Cirani, M. Picone, and L. Veltri. A session initiation protocol for the internet of things. In *Scientific International Journal for Parallel and Distributed Computing-Scalable Computing: Practice and Experience, VOL. 14, NO. 4*, pages 249–263. SCPE, 2015.

[56] V. Miskovic and D. Babic. An architecture for pervasive healthcare system based on the ip multimedia subsystem and body sensor network. In *FACTA UNIVERSITATIS Series: Electronics and Energetics, VOL. 28, NO. 3*, pages 439 – 456, 2015.

[57] G. Karopoulos, P. Mori, and F. Martinelli. Usage control in SIP-based multimedia delivery. In *Computers & Security, NO.39*, pages 406–418. Elsevier, 2013.

[58] F. Bacchus and P. van Beek. On the conversion between non-binary and binary constraint satisfaction problems. In *In Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI-98)*, pages 311–318. AAAI Press, 1998.

[59] S. J. Russell and P. Norv. Artificial intelligence, a modern approach. Pearson Education, Inc., Third Edition 2010.

[60] D. L. Poole and A. K. Mackworth. Artificial intelligence: Foundations of computational agents. Cambridge University Press, 2010.

[61] F. Rossi, P. V. Beek, and T. Walsh. Handbook of constraint programming: Foundations of artificial intelligence. In *Handbook of Constraint Programming: Foundations of Artificial Intelligence*. Elsevier Science and Technology Books, Inc., 2006.

[62] D. Dubois, H. Fargier, and H. Prade. Fuzzy constraints in job-shop scheduling. In *Journal of Intelligent Manufacturing, Vol 6, No 4*, pages 215–234, 1995.

[63] T. Schiex. Possibilistic constraint satisfaction problems or how to handle soft constraints? In *8th Int. Conf. on Uncertainty in Artificial Intelligence*, pages 268–275, 1992.

[64] S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint satisfaction and optimization. *Journal of the ACM, Vol. 44, No. 2*, pages 201–236, 1997.

[65] T. Schiex, H. Fargier, and G. Verfaillie. valued constraint satisfaction problems: Hard and easy problems. In *In Proceedings of the 14th international joint conference on Artificial intelligence*, pages 631–637. ACM, 1995.

[66] N. Verbiest, C. Cornelis, and Y. Saeys. Valued constraint satisfaction problems applied to functional harmony. In *In Proceedings of the IFSA-EUSFLAT*, pages 925–930, 2009.

[67] O. Kamarainen and H. El Sakkout. Local probing applied to scheduling. In *8th International Conference, CP*, pages 155–171. Springer Verlag LNCS, 2002.

[68] O. Etzion and P. Niblett. Event processing in action. Manning Publications Co., 2010.

[69] Complex event processing. http://en.wikipedia.org/wiki/Complex event processing, Accessed 20 Sep, 2017.

[70] Y.H. Wang, K. Cao, and X.M. Zhang. Complex event processing over distributed probabilistic event streams. *Journal of Computers and Mathematics with Applications , Vol 66, No.10*, pages 1808–1821, 2013.

[71] S. Rudolph D. Anicic, P. Fodor and N. Stojanovic. Ep-sparql: A unified language for event processing and stream reasoning. In *in Conference on World Wide Web , Hyderabad*, pages 635–644, 2011.

[72] F. Bry and M. Eckert. A high-level query language for events. In *IEEE Services Computing Workshops(SCW'06)*, pages 31–38. IEEE, 2006.

[73] S. Rozsnyai, J. Schiefer, and H. Roth. Sari-sql: Event query language for event analysis. In *IEEE Conference on Commerce and Enterprise Computing*, pages 24–32. IEEE, 2009.

[74] R. S. Barga, J. Goldstein, M. Ali, and M. Hong. Consistent streaming through time: A vision for event stream processing. In *3rd Biennial Conference on Innovative Data Systems Research (CIDR), California*, pages 363–373, 2007.

[75] A. Aztiria, J. C. Augusto, R. Basagoiti, A. Izaguirre, and D. J. Cook. Learning frequent behaviors of the users in intelligent environments. *IEEE Transactions on Systems, Man and Cybernetics, vol. 43, no. 6*, pages 1265–1278, 2013.

[76] A. Widder, R. V. Ammon, P. Schaeffer, and C. Wolff. Identification of suspicious, unknown event patterns in an event cloud. In *Proceedings of the 2007 inaugural international conference on Distributed event-based systems*, pages 164–170. ACM, 2007.

[77] M. Grabisch, J. Marichal, R. Mesiar, and E. Pap. *Aggregation Functions: Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 2009.

[78] Y. Qi, L. Cao, M. Ray, and E. A. Rundensteiner. Complex event analytics: Online aggregation of stream sequence patterns. In *Proceedings of the ACM SIGMOD international conference on management of data*, pages 229–240. ACM, 2014.

[79] G. Marshall. RFC 3881: Security audit and access accountability message XML data definitions for healthcare applications. 2004. https://tools.ietf.org/html/rfc3881, Accessed 20 Sep, 2017.

[80] W. Ma. *User behavior pattern based security provisioning for distributed systems*. PhD thesis, Electrical and Computer Engineering, University of Ontario Institute of Technology, Oshawa, Canada, 2016.

[81] Apache kafka: A distributed streaming platform. https://kafka.apache.org/, Accessed 20 Sep, 2017.

[82] Apache zookeeper: A centralized service for maintaining configuration information. https://zookeeper.apache.org/, Accessed 20 Sep, 2017.

[83] N. Schweikardt, T. Schwentick, and L. Segoufin. Database theory: Query languages. In *Algorithms and theory of computation handbook*, pages 1–48. Chapman & Hall/CRC, 2010.

Appendix A

The syntax of BPL

```
1  Syntax of BPL
2  <BPL> ::= BEGIN <pattern_specification> END
3  <pattern_specification>::=<Declaration><Operation>|
4  <Constraint>
5  <Declaration>::= <event declaration>|<pattern
6  declaration>|<set declaration>|<attVal declaration>
7  <event declaration>::= DEFINE Event    =
8     <    {<atttName>}+    >
9  <pattern declaration>::=<leftside>    =    <event type>
10 <event type>::= NEW Event( <Nnumber> )
11 <set declaration>::= <leftside>    =       {    {<
      eventIdentifier>}+    }
12 <attVal declaration>::= <leftside>    =       {    {<
      valuelist>}+    }
13 <valuelist>::= <attName> {<value>}+
14 <leftside>::= <identifier> | <identifier>[]
15 <Operation>::= <operation header> <operation body>
16 <operation header>::= DEF <identifier> {<parametrs>}
17 <operation body>::= <statements> END
18 <statements>::= <foreach statement>|<ifstatement>|
19 <yield statement>|<block statement>|<call statement>
20 <yield statement>::= yield { <parameters>}
21 <block statement> ::= DO { <parameters>} <statements> END
22 <call statement>::= CALL <method name> { <parameters>}
23 [<block statement>]
24 <foreach statement>::= FOREACH <identifier> IN
25 <identifier> DO
26 <ifstatement>::= IF < expression> THEN <statements>
27 <Constraint>::= <ConstraintExpr>|<call statement>
28 <ConstraintExpr>::= <expression>
```

```
29 < comparable operator><expression>
30 <comparable operator> ::=    <   |   >   |   =   |    !=
      |   <=   |   >=
31 <parametrs>::= set | event | <value>
32 < atttName>::= <identifier>
33 <identifier>::= a string of character
34 <eventIdentifier>::= event id
35 <Nnumber>::= a positive integer value
36 <value>::= integer | float | string
```

Appendix B

POJO Class

```
1  public  class Measurement {
2
3  @SerializedName("id") //global id of the event
4  private int ID;
5
6  @SerializedName("seqid") //sequence id
7  private int SeqID;
8
9  @SerializedName("eventid") //event id in a sequence
10 private int EventID;
11
12 @SerializedName("attnum") //number of attribute inside of
       an event
13 private int Attnum;
14
15 @SerializedName("date")
16 private String d;
17
18 @SerializedName("time")
19 private String t;
20
21 @SerializedName("role")
22 private String Role;
23
24 @SerializedName("location")
25 private String Location;
26
27 @SerializedName("action")
28 private String Action;
29
```

```
30  @SerializedName("res")
31  private String Res;
32
33  @SerializedName("patientid")
34  private String PatientID;
35
36
37  @SerializedName("userid")
38  private String UserID;
39
40  @SerializedName("type")
41  private String Type;
42
43  //    @SerializedName("value")
44  //    private float Value;
45
46  public int getID() {
47  return this.ID;
48  }
49  public void setID(int ID) {
50  this.ID = ID;
51  }
52
53  public int getSeqID() {
54  return this.SeqID;
55  }
56  public void setSeqID(int SeqID) {
57  this.SeqID = SeqID;
58  }
59
60  public int getEventID() {
61  return this.EventID;
62  }
63  public void setEventID(int EventID) {
64  this.EventID = EventID;
65  }
66
67  public int getAttnum() {
68  return this.Attnum;
69  }
70  public void setAttnum(int Attnum) {
71  this.Attnum = Attnum;
72  }
73
74  public String getd() {
75  return this.d;
76  }
77  public void setd(String d) {
```

```
78  this.d = d;
79  }
80
81  public String gett() {
82  return this.t;
83  }
84  public void sett(String t) {
85  this.t = t;
86  }
87
88  public String getRole() {
89  return this.Role;
90  }
91  public void setRole(String Role) {
92  this.Role = Role;
93  }
94
95  public String getLocation() {
96  return this.Location;
97  }
98  public void setLocation(String Location) {
99  this.Location = Location;
100 }
101
102 public String getAction() {
103 return this.Action;
104 }
105 public void setAction(String Action) {
106 this.Action = Action;
107 }
108
109 public String getRes() {
110 return this.Res;
111 }
112 public void setRes(String Res) {
113 this.Res = Res;
114 }
115 public String getPatientID() {
116 return this.PatientID;
117 }
118 public void setPatientID(String PatientID) {
119 this.PatientID = PatientID;
120 }
121
122
123 public String getUserID() {
124 return this.UserID;
125 }
```

```
126
127 //     public float getValue() {
128 //         return this.Value;
129 //     }
130
131 public String getType() {
132 return this.Type;
133 }
134
135 public void setUserID(String UserID) {
136 this.UserID = UserID;
137 }
138
139 //     public void setValue(float Value) {
140 //         this.Value = Value;
141 //     }
142
143 public int getRisk() {
144
145 return 0;
146 }
147 public String toString(){
148 return new String("");
149 }
150 public long getCreationTime() {
151 // TODO Auto-generated method stub
152 Timestamp timestamp = new Timestamp(System.
       currentTimeMillis());
153 return timestamp.getTime();
154
155 }
156
157 }
```