RHEINISCH-
WESTFÄLISCHE
TECHNISCHE
HOCHSCHULE
AACHEN

LEHRSTUHL
FÜR INFORMATIK 4
PROF. DR. OTTO SPANIOL
KOMMUNIKATION UND VERTEILTE SYSTEME

Master Thesis

An Obligation Framework and Language for Data Handling in Service Composition

Muhammad Ali
Matriculation number: 284152

June 10, 2009

Reviewers:
Prof. Dr. Otto Spaniol
Prof. Dr. Ulrike Meyer

University Supervisor: Dr. Dirk Thissen
Industry Supervisor: Dr. Laurent Bussard(Microsoft)

Dedicated to my father (May his soul rest in peace) and my mother. May Lord be merciful to them for they have brought me up in my childhood and always supported and prayed for me.

Declaration

I hereby declare that this thesis is my own unaided work except for the official assistance provided by my supervisors as mentioned. Furthermore, this work has not been submitted in any form for any other degree at any university or educational institution. Information derived from the published or unpublished work of others have been acknowledged in the text and list of referenced literature has been provided in the bibliography section.

Aachen, June 10, 2009
Muhammad Ali

Acknowledgements

I am deeply thankful to all those who helped me directly or indirectly during the time I had been working on this thesis.

I would like to express my sincere gratitude to Prof. Dr. Otto Spaniol and Prof. Dr. Ulrike Meyer for approving this thesis and giving me the opportunity to perform this thesis under their supervision.

Special thanks to my immediate supervisors including Dr. Laurent Bussard and Dr. Ulrich Pinsdorf from European Microsoft Innovation Center Aachen and Dr. Dirk Thissen from i4 Chair for their continuous guidance and supervision during the course of this work. This would not have been possible without their support and valuable feedback on my work.

I would also like to thank all my team members and colleagues at European Microsoft Innovation Center Aachen who provided their valuable opinion on the work and direction.

This work is being done under EU FP7 PrimeLife project and I am thankful to all the project partners for their valuable feedback during the thesis.

Finally, thanks to all my friends in Aachen for their encouragement and support.

Abstract

Today's access control languages and other privacy policy languages recognize the importance of usage control/obligations by foreseeing placeholders in the syntax for obligations. However, most of these existing policy languages do not provide any concrete language constructs to actually express obligations. This is a real problem since obligations play an important role in daily business. User privacy is one of the major concerns in today's world with most of the businesses have adhoc and inflexible processes to keep track on customer data. We address this problem and develop a general language for expressing obligations. It is independent from the enveloping policy languages and thus can be used e.g. in XACML. We also provide an enforcement framework design that keeps track of and enforce obligations committed by service providers with its customers.

Contents

List of Figures

1. Introduction

New technological advancements, increasing capacity of hardware and decreasing costs allowed the expansion of the Internet and inspired many businesses to go digital. This not only brought change for existing businesses but also instigated new business scenarios to be realized through the modern computer systems. Service oriented architectures provide a new way to automate existing and new business processes and opened the possibilities of new business dynamics. End users can now interact with the service providers electronically and the interaction, which could be in multiple iterations, is completed within seconds which would otherwise be a long tedious process. Further complex forms of scenarios have been discussed in the literature with multiple businesses interacting or chains of services called to provide the required service.

As the new scenarios emerged, they also brought a new set of problems and areas of research which should be addressed to achieve the full potential of the new technology without any negative repercussions. Security concerns from the perspective of service providers like access to services only to authenticated and authorized users, non repudiation, impersonation by malign requestors etc came to light. On the other side the end user's privacy concerns about the usage and handling of her data are also some of the problems emerged.

Before the advent of the modern policy based management systems, adhoc mechanisms were used to provide such capabilities like tagging user data, incorporating constraints in the functionality etc. These were not extendible, unstructured and informal mechanisms and the need to have policy based systems was first realized among the research community. The result is that now we witness the existence of many of such policy languages targeting different classes of problems like access control, data handling etc.

Traditional access control mechanisms provides the answer to the question *whether access to specific resource by a specific requestor is granted or not?*. It is designed to accommodate the concerns of service providers who control the access to resources[1] under their ownership. On the other hand, the concept of data handling, usage control and obligations defines *how the resource is handled and used* once the access was granted. It is also important to note that the roles *resource owner* and *service provider* are not mutually exclusive and it is also usual that entities may have been behaving both as a resource owner at one instant and resource consumer at another.

We consider that obligations are themselves complex enough to be considered as a separate problem rather than considering them as a sub component of a data handling policy. In reality, they are not just a sub part of a data handling policies but are used widely in other forms of policies. Many of today's access control languages and

[1] The word *resource* is generic and it could be any physical or logical resource under the direct ownership of a service provider including its customer's data

other privacy policy languages recognize the importance of obligations by providing placeholders in the syntax of their language structure for obligations. However, most of the languages we examined, including XACML [24, 29] and PRIME-DHP [4], have this placeholder but do not provide concrete language constructs to actually express obligations. Usually obligations are defined and enforced within a single trust domain and it is up to the service provider to define and enforce application specific obligations in an arbitrary way.

We also consider that obligation rules and data handling rules are somewhat overlapping and one form of statement can be translated into other e.g. *X can share U's data with its business partners* is a data handling statement. However, a corresponding obligation could be *X commits not to share U's data anyone other than its business partners*. The obligation statements depict a binding or commitment on the declarer while data handling rules express rights of the declarer. However, the relationship between the two forms is fuzzy and an open area of research for translating one form of statements to the other which is out of scope of our current work. This thesis work focuses on the problem of expressing and enforcing obligations. The major contributions of this work are mentioned below.

- We formalize real world obligation statements and derive requirements for a general, well formed obligation language (Sect. 1.2).
- We formally describe a syntax fulfilling those abstract requirements (Sect. 3.3). The language is intended for general purposes but it is rich enough to express complex real-world obligations. Moreover it can be extended with domain specific triggers and actions which simplify the process of policy writing in a specific application context.
- The language is generic enough to be independent from the enveloping policy languages, e.g. XACML. This allows combining our work with other languages, which leads to interoperability and enables general post-processing of obligations in the service back end.
- We describe a software framework that keeps track of and enacts obligations (Sect. 4). Although we implemented the obligation engine to verify our assumptions, we deliberately describe the architecture in a generic way so that our findings can be reused for other implementations of obligation engines.
- We give an overview of related approaches in Section 2 and position our work within the state of the art.

As it is evident from the discussion until now that different forms of policy languages like access control policy, data handling policies and obligations address different problems and are neither entirely mutually exclusive nor completely overlapping. It is also not clear whether any parent child relationship or containment relationship exists between these different forms of policy languages. We can even assume that there could be a recursive relationship between these forms of privacy policies which means that an

access control policy can embed data handling rules and obligation rules which in turn can embed access control rules and so on. The enforcement platforms are independent of, but could be integrated with, each other.

In the next section, we present one of the problem scenarios where we would like to apply our solution. However, it is important to note that our results are not restricted to this problem only.

1.1. Problem Scenario

The current work primarily focuses on the application of our obligation policies and their enforcement in a service oriented paradigm. Figure 1.1 shows the traditional and modified user-service provider interplay. In the classical scenario, in Figure 1.1(a), the data owner (or the end user or client) did not have any involvement in the handling of her data. However to involve the data owner, the interplay is modified as shown in Figure 1.1 (b). On request for a new service, the service provider sends back the customizable policy to the client. The client accepts the policy and customizes it and finally sends it back to the provider along with her data.

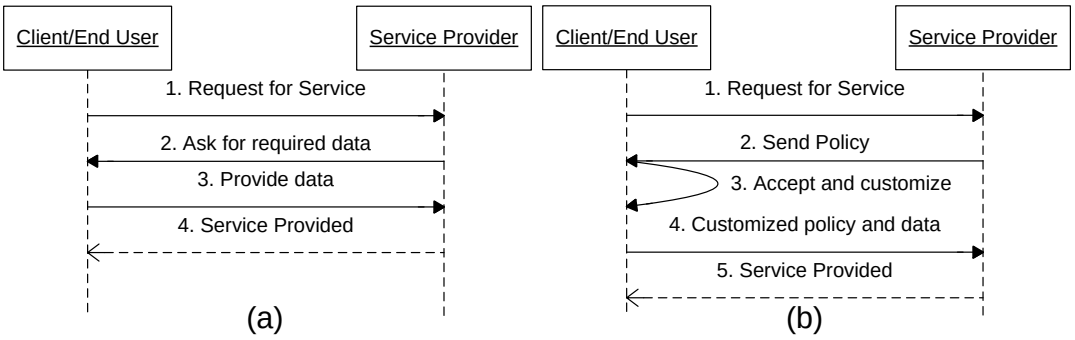

Figure 1.1.: Service and User Interplay

It is important to note the following

- The *policy* as sent by the service provider, in Figure 1.1(b) step (2), could be a data handling policy, obligation policy message or the combination of the two. We want to keep the obligation policy language design independent of the protocol used.
- This is a service provider dominated scenario where the policy is being provided by the provider and the end user only customize it within defined limits and guidelines. This ensures that even after customization[2] of the policy the provider is able to handle and enforce it. The other way could be that the end users send their policy

[2] In general the customization of the policy could also involve a real multi-round negotiation protocol between user and service provider.

to the provider but this creates the problem of ensuring whether the provider is capable to enforce such policies or not.

- The last step shows that the end user send the data along with the policy back to the provider. This is a *sticky policy paradigm* [10, 9], where policy and data travel together through the system. We make no assumption *how* the policy sticks to the data. For reasoning throughout this report, we assume that obligations are part of a policy which stays with the data throughout the data's lifetime.

Figure 1.2 shows the complete illustration of the scenario.

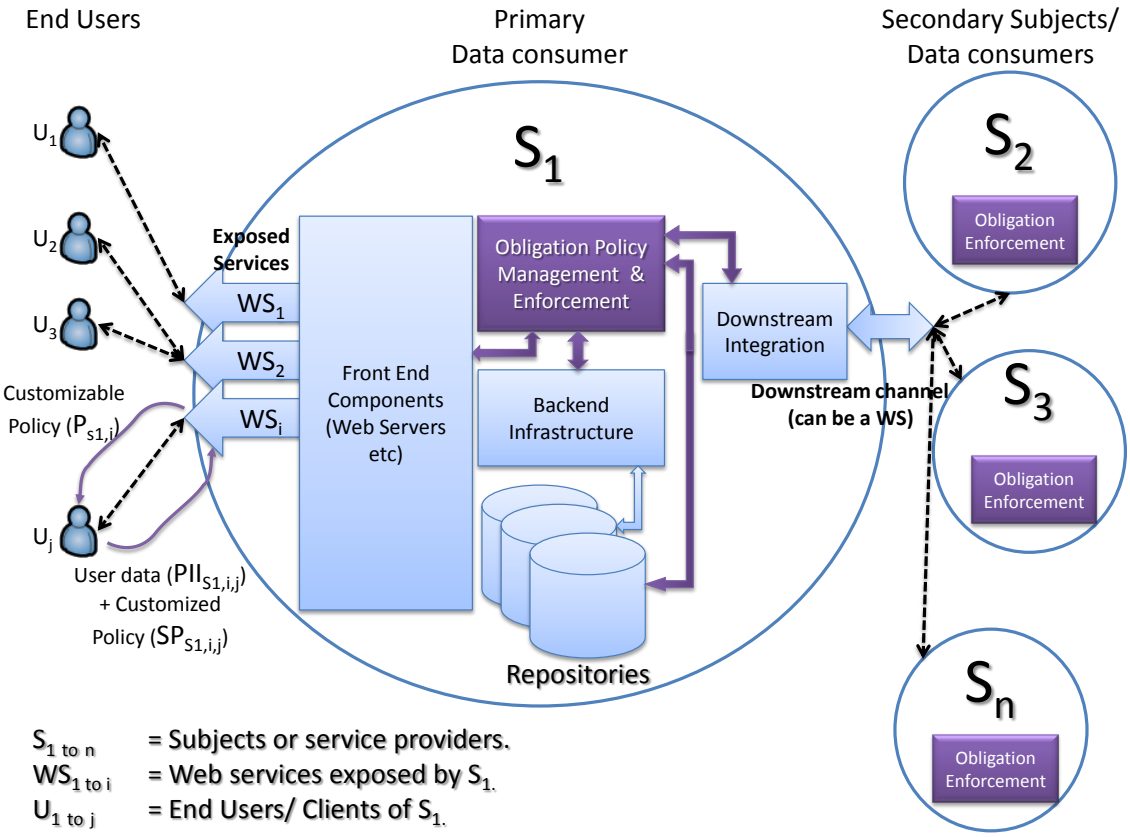

Figure 1.2.: Complete Scenario Illustration

In this thesis we focus on the expression of policies travelling between the users and service providers and the design of the policy management and enforcement framework. These components are illustrated in violet color (dark gray for black and white prints) in Figure 1.2. For clarity we have also illustrated the secondary service providers ($S_{2\ to\ n}$) who use the data from the primary provider (S_1). Each service provider can have multiple

services ($WS_{1\ to\ i}$) exposed for its clients/End users ($U_{1\ to\ j}$) who utilize different services. Here S_1 is behaving as a primary data consumer and $U_{1\ to\ j}$ are data owners.

The customizable policy $P_{S_1,i}$ represents policy sent by the provider S_1 via WS_i to any of its users. This corresponds to Figure 1.1 Part (B) Step [2]. The expression of obligation policy language to express this customizable policy is one of the goals of this thesis.

The user data $PII_{S_1,i,j}$ represents *Personally Idenfiable Information* by U_j for WS_i at S_1. Customized policy $SP_{S_1,i,j}$ represents the policy as returned by the U_j to WS_i at S_1. This returned policy is named as *sticky policy* thus we used *SP*. This corresponds to Figure 1.1 Part (B) Step [4].

Personally Idenfiable Information(PII) is any piece of information which can be used to identify an individual uniquely like Email address, social security number, passport number, national ID card etc.

When interacting with secondary data consumers on the downstream channel, S_1 behaves as a data owner. The decision to share any data with secondary consumers depend on the policies applied on the specific piece of requested information. Thus, the user privacy is preserved and the usage and handling of information is restricted by the promises made in the policy and accepted by the actual data owners ($U_{1\ to\ j}$). The dynamics of downstream data sharing and usage has not been investigated in detail as it was dependent on the expression of obligation policies and enforcement platform. However, this is one of the important parts of future work to enhance the work done in this thesis and extend it to chains of service providers.

Next Section 1.2 presents major requirements of the language and framework. We covered these requirements in our design.

1.2. Major Requirements of the Language and Framework

There has been quite a lot of work done in the area of obligations with different researchers' targeted different problems in expressing and enforcing of obligations. We considered major existing classifications and forms of obligations discussed in literature and developed a set of obligation attributes and types which could cover wide range of real world obligations. Based on our classification, we formulized obligations and developed a specification policy language which could express all such obligations. The final major contribution is the implementation of the obligation enforcement framework which manages the obligation policies expressed in our proposed language.

Except HP's work [7], we have not been able to cite any real implementations of obligation enforcement platforms. Most of the work done in literature targeted towards specification and theoretical approaches to express obligations. Pretschner et. al [28] proposed obligation specification language (OSL) but provided translation mechanisms to translate their language in Rights expression languages (RELs) like *Open Digital Rights Language(ODRL)* [16] to utilize existing REL enforcement platforms. Thus, we fill this missing part and believe that existing obligation expression approaches could be translated into our language and enforced with our platform.

We now present the set of requirements which we have currently addressed in our implementation and then we present additional requirements which are currently not taken into account but are important enough to be mentioned in the text. We compiled this list by looking at scenarios and requirements in [7, 4, 17, 24, 29, 22] and some other published literature.

- *Independence from policy language.* Obligations can be enforced independently from the embedding policy languages offering the placeholder for the obligation. Consecutively, the policy can stand by its own without being embedded inside any other policy. This totally depends on the application and scenario. Thus the obligation framework needs to be technically decoupled from the policy engine.

- *Independence from data repositories.* The obligation handling must be independent from the concrete data repositories. The obligation travels with the data and should be stored along with the data so that the reference does not get lost. However, the important difference is that either we manage both policy and data within our framework or we only allow the obligation framework to manage policies with data stored externally. The policies keep the reference of the data. This design allows easy integration with the existing systems without the need to have huge migration effort. The obligation may refer to personal data stored in a database or to documents stored in a file system.

- *Independence from communication protocols.* The framework must be independent from the communication protocol. For instance, Web Services, REST, or plain HTTP could be used to exchange data and obligations.

- *Support for common obligations.* The obligation handling language should be extendible but not empty. Usual actions such as, for instance, *delete*, *anonymize*, *notify user*, *get approval from user*, *log* should be available with different implementations. Triggers for a time-based and an event-based execution need to be defined.

- *Support for domain specific obligations.* The framework must be open to define additional domain specific obligations. This requires mechanisms to define new types of actions and triggers. In any case the semantic of these new elements has to be understood by all involved communication partners.

- *Support for abstraction of actions.* The obligation language must offer abstract actions which are configurable for the specific purpose. For instance, a *notify user* action might be implemented as sending an e-mail, sending an SMS, sending a voice message, or calling (and authenticating to) a web service.

- *Support for abstraction of triggers.* The obligation language must offer abstract and configurable triggers. For instance, a trigger "access PII" may react both to a query on a database and to a read operation on a file server.

- *Support for distributed deployment.* Depending on the scenario of the service provider, different deployments of the obligation handling framework can be envisioned. A corporate-wide obligation framework could handle obligations associated with data stored in multiple data bases. A local obligation framework hosted on user's machine could address obligations associated with local files. Finally, the obligation framework could also be offered as a "cloud service". We assume that only one obligation framework can be in charge of enforcing obligations regarding a specific piece of data.

- *Support for preventive obligations.* The obligation language shall be able to express preventive statements or negative obligations that forbid the execution of an action. For instance, the obligation to store logs six months will forbid deletion of log files.

Before we go into the details of the obligation language and the design architecture, it is important to briefly identify the stakeholders of such a system. This is important as the decision to procure, deploy and operate any business application depends on the vested interests of stakeholders. The next section discusses briefly some of these actors.

1.3. Stakeholders of the System

Figure 1.3 shows the potential stakeholders of the system who may have direct or indirect interest within the solution provided.

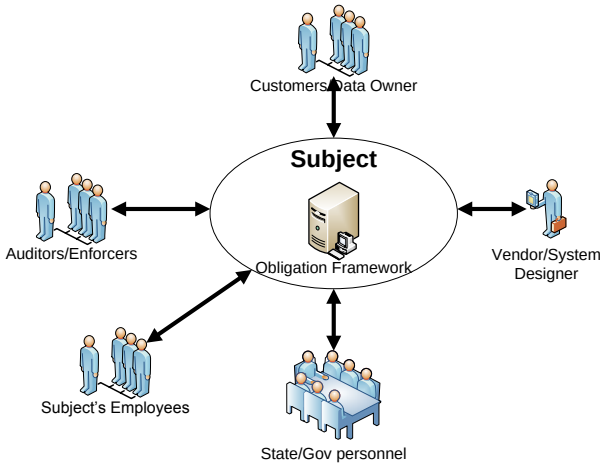

Figure 1.3.: System Stakeholders

Data Owner: These are entities, usually outside the subject's trust domain and are the customers of the organization, who have interest in the correct usage of their sensitive data. They are also concerned about the fulfillment of commitments made by the data consumer.

Subject[3]*/Data Consumer:* Entity which deploys the obligation framework within its trust domain and is liable to fulfill obligations committed against the collection of sensitive data from its customers. In our scenario, subject is also the data consumer. If subject/data consumers do not handle data as promised then subject may lose its reputation, credibility and as a result may lose customers. The major incentive for the subject to invest in such a system could be mandatory legislation, value of service for its customers and social responsibility. It is hard to relate the implementation of such a system which addresses user privacy concerns with the revenue generation of the subject but it could be considered as an intangible asset which may indirectly increase in income, safety against potential lawsuits and repute among customers.

Solution Vendor: Entity which designs the solution (system and accessories) and sells it to its customers. The customer of solution vendor is the Subject/Data consumer as mentioned above. If the designed system does not behave as per standards and existing legislation then the vendor and its respective product both lose its reputation and potential business.

Auditors: Organization or individuals who are incharge of auditing the subject to find out the potential fraudulent behavior. These are responsible to detect the violation of promises made by the subject. The auditing entity can be internal or external. However, they are usually not responsible for the enforcement of obligations but behave as a source of information to the enforcers. They can also audit the system to detect non-compliant implementation of the system and prosecute system vendor for compensations.

Government or State Actors: Concerned about the compliance behavior of the subject as per the legislation. These are one of the potential enforcers and utilize the information provided by auditors and other monitoring agents to enforce the intended behavior. They could also penalize the subject or solution vendor in the event of fraudulent behavior.

Subject's Employees: They are the group of personnel who operate the system for the subject. They create social issues as they perceive an implementation of a new system as against their vested interests like losing a job, hiring or new men trained on the new system, system outsourcing etc. They deem that their lack of ability to operate new system would cost them their job or they will be replaced by a new trained man power. In such cases, they deliberately try to prove the incapability of the new system and efficiency of older methods on which they are trained.

The rest of the report is organized as follows. Chapter 2 compares our work with state of the art, Chapter 3 defines the obligations formally, Chapter 4 presents the enforcement architecture and its working, Chapter 5 presents evaluation results and Chapter 6 concludes with a summary, future work and open problems.

[3]The subject in this definition is the data processor and not the owner of the data who is sometimes referred to as *data subject*.

2. State of the Art

Most of the available policy languages, like XACML (eXtensible Access Control Markup Language) [24, 29, 23], EPAL (The Enterprise Privacy Authorization Language) [5], Ponder [20], Rei [19] and PRIME-DHP (PRIME[1] Data Handling Policy) [4], provide either only a placeholder or very limited obligation capability. Moreover these languages do not provide any concrete model for obligation specification. XACML and EPAL support system obligations only, as no other subject can be expressed in their proposed language. Ponder and Rei on the other hand do allow user obligations, however they do not provide a placeholder explicitly for the specification of temporal constraints and they do not support pre-obligations, conditional obligations, and repeating obligations.

XACML specifically targets access control requirements and this only provided limited obligation capability. PRIME-DHP proposed a new type of policy language which expresses policies as a collection of data handling rules which are defined through a tuple of recipient, action, purpose and conditions. Each rule specifies who can use data, for what purposes and which action can be performed on the data. The idea is inspiring and contributed a new direction to view the problem. The language structure is flat which limits its expressiveness. PRIME-DHP itself also does not provide any concrete obligation model.

Besides the policy languages, we observed publications on expression, enforcement and formalization of obligations. In the next paragraphs, we collected prior art which is directly related to our approach and point out the key differences to our work.

Mont Casassa et al. [7, 8, 27] proposed the idea of having parametric obligation policies with actions and events having variable parameters. This work was done in conjunction with the PRIME-DHP to support obligations. It is by far the closest work to ours thus we compared our work with this work in detail in Chapter 5 of *Evaluation*. The detailed version of HP's work is described in [8]. In [27], authors presented the results of usability tests of the obligation management system. The key question they answer is *Whether an ordinary user can interact with an obligation management system or not?*. They identified four key problem areas for the future obligation management system developers namely trust, enterprise perspective, ambiguous phrases in the policies and obligation setting relative to data or data collection purpose. We currently have not conducted any usability tests with human end users. However, the results from [27] should be taken into account and can be considered as a starting point.

Irwin et al. [17] proposed a formal model for obligations and defines secure states of a system in the presence of obligations. Furthermore, they focused on evaluating the complexity of checking whether a state is secure. However, the proposed obligation

[1] PRIME (Privacy and Identity Management for Europe) was the name of EU FP6 Project. Details can be found http://www.prime-project.eu

model is very restricted and does not support pre-obligations or provisions, repeating and conditional obligations which are required in different domains and scenarios. They addressed the problem of verification of obligation enforcement while we focus on the expression of a wide range of scenarios, supporting all of the above types of obligations. They also briefly mentioned the notion of negative obligations but have not provided handling of negative ones as they perceive that negative obligations can be translated into access control requirements. We perceive negative obligations to be an important aspect and address this too in our model and architecture. So the two research efforts are targeting different problems.

Lupu et. al. [21] also discussed the concept of negative obligations and perceived them different from negative authorizations. The paper focused on the conflicts in the distributed policy management systems where conflicts arise because of incompatible requirements. Lupu el. al. also proposed ponder policy specification language [20] but it lack explicit placeholder supporting the specification of temporal constraints and they do not support pre-obligations, conditional obligations and repeating obligations either.

Pretschner et al. [22, 15, 28] worked in the area of distributed usage control. In [22], they used distributed temporal logics to define a formal model for data protection policies. They differentiated provisional and obligation formulas using temporal operators. Provisions are expressed as formulas which do not contain any future time temporal operators and obligation are formulas having no past time temporal operators. They also addressed the problem of observability of obligations which implies the existence of evidence/proof that the reference monitor is informed about the fulfillment of obligations. Possible ways of transforming non-observable obligations into observable counterparts have also been discussed. We also consider temporal constraints as an important part of obligation statement. However, we deem observability depends on the visibility and scope of the reference monitor and not completely on the obligation rule. The scope could be within the system, within the same trust domain but outside the system, or even sitting outside the trust domain, to observe fulfillment and violations. We currently have not addressed this problem of observability. In [15] they have proposed an obligation specification language (OSL) for usage control and presented the translation schemes between OSL and rights expressions languages (RELs). Thus, the OSL expression could be enforced using existing rights management enforcement mechanisms. We fill this gap by implementing the enforcement platform for enforcing obligation policies without translation. In [28], the authors have addressed the scenario of policy evolution when the user data crosses multiple trust domains and the sticky policy evolves. Currently, we are not focusing on evolution of obligation policy, but it could likely be one of the future extensions of our work where we plan to address the interaction of obligation frameworks at multiple services which is complementary to what is discussed in [28].

Katt et al. [18] proposed an extended usage control (UCON) model with obligations and gave prototype architecture. They have classified obligations in two dimension a) system or subject performed and b) controllable or non-controllable where the objects in the obligation would be either controllable or not. Controllable objects are those that are within a target system's domain, while non-controllable objects are outside the system's domain. The enforcement check would not be applied for system-controllable

obligations where they assume that since system is a trusted entity so there is no need to check for the fulfillment. The model again misses the conditional obligations.

Cholvy et al. [11] studied the relationship between collective and individual obligations. As opposed to individual obligations which are rather simple as the whole responsibility lies on the subject, collective obligations are targeted toward a group of entities and each member may or may not be responsible to fulfill those obligations. They investigated the problem of translating collective obligations into individual obligations. We also consider that the subject of any obligation rule is a complex entity in itself like individual or group, self directed or third party. Our current implementation does not support this but could be extended to include such scenarios.

Ni et al. [25] proposed a concrete obligation model which is an extension of P-RBAC (Privacy aware role based access control) [26]. They investigated a different problem of the undesirable interactions between permissions and obligations. The subject is required to perform an obligation but does not have the permissions to do so, or permission conditions are inconsistent with the obligation conditions. They have also proposed two algorithms, one for minimizing invalid permissions and another for comparing the dominance of two obligations. Dominance relation is the relationship between two obligations which implies that fulfillment of one obligation would cover the fulfillment of other which is analogous to set containment. We believe that the results from this work could also be applied on our proposed framework for optimization purposes, but we see that this has strong implications on the consistency check of policies.

Gama et al. [14] presented an obligation policy platform named *Heimdall* which supports the definition and enforcement as a middleware platform residing below the runtime system layer (JVM, .NET CLR) and enforcing obligations independent of application. Opposed to that, we present an obligation framework as an application layer platform in a distributed service-oriented environment which could be used as a standalone business application to cater for user privacy needs. We believe that it is not necessary to have the obligation engine, which is an important infrastructure component to ensure compliant business processes, as part of the middleware. Moreover our service-oriented approach supports interoperability in a heterogeneous system environment.

Dougherty et al. [12] also presented an abstract model perceiving obligations as a means for expressing constraints on the future behavior of a system. They consider obligations to have state and can fail to be fulfilled. Rather expressing obligations, they focused on the problem of analyzing the obligations interaction with system in execution. The model cannot express pre-obligations or conditional obligations. We have focused on the expression and enforcement of obligation. However, we also deem that obligations could be stateful entities which fulfillment is achieved in steps.

Mazzoleini et. al. [23] provides a much elaborated overview of XACML policy integration algorithms and techniques for policy matching and equivalence in a distributed scenario. They also presented results with respect to scaling such techniques.

Anne et. al. [3] provides a very detailed comparison of IBM's EPAL and OASIS XACML. The work concludes that XACML is a superset of EPAL and covers a wider range of possibilities as compared to EPAL. It lists down precisely the missing features of XACML which are missing in EPAL.

[30] provides the comparison of IBM's EPAL and W3C P3P[2] policies and concluded the paper with the key differences of the two.

Janice et. al [32] discussed the issue of privacy protection of government data specifically which is made available for use in mashups on the internet. They also proposed the idea of having personal privacy policies specified by data owners whose data is being collected by the government which is complementary to what we propose.

The work present in this paper incorporates some of the enlightening prior art and extends it towards more expressiveness, extendibility, and interoperability. However, we think that some authors addressed different problems, and it would be worthwhile to further combine their results with our approach.

[2]P3P stands for *Platform for Privacy Preferences* details can be found at http://www.w3.org/P3P/

3. Obligations

This chapter focuses on the aspects of obligation rules, semantics and our formalization of the problem of expressing obligations. We start with the formal definition of obligations followed by the classification, formal model and last in the chapter we present consistency and safety issues.

3.1. Definition

We define an *obligation* as:

> Promise made by an entity, the *subject*[1], to another entity, the user. The subject is expected to fulfill the promise by executing and/or preventing a specific *action* at a *particular time* or due to a certain *event* and optionally under certain *conditions*.

From the definition, it could be inferred that the obligation statements bind the declarer to fulfill its commitments. The words in italics in the above definition are components of an obligation and form the building block of an obligation rule. A collection of obligation rules formulate an *obligation policy*[2].

3.2. Aspects of Obligations

It is close to impossible to develop an exhaustive list of obligation statements existing in real world. We could encounter many complex forms of commitments made between individuals, organizations and logical entities. Additionally, the semantics of these commitment and promises may be different in different domains like healthcare, financial services etc.

Our first goal is to identify and classify such promise and obligation statements along with their different attributes and aspects. Many of these aspects have also been discussed in existing literature. We have considered all of them from existing work as well as from our own research by investigating different obligation statements from different domains.

[1] The subject in this definition is the data processor and not the owner of the data who is sometimes referred to as *data subject*.

[2] A policy language is a set of rules and vocabulary which specify how to write a well formed policy using this language. This is analogous to grammar of any natural language and the sentences expressed in that language.

This research effort helped us to identify and derive common building blocks of such obligation statements. The same building blocks are then taken as the policy language constructs to formalize obligations.

3.2.1. Enforcement Mechanism

The first and the prime important aspect of obligations is the tone of statement. We could have *positive obligations* where the commitment is stated in a positive tone. Example 1 below depicts two such cases where the subject, *Hospital X*, commits to its patients in a positive tone.

Example 1 Positive obligation statement

1: Hospital X commits to delete patient's history in 1 month.
2: Hospital X commits to notify patient whenever his information is shared.

We could also encounter obligations which are expressed in a negative tone. Example 2 shows examples of such statements. [17, 21, 28] also discussed briefly about negative obligations.

Example 2 Negative obligation statement

1: Hospital X commits not to share patient's history to anyone.
2: Hospital X commits not to use patient's history for any statistical purposes.

We propose to have different enforcement mechanisms for enforcing positive and negative obligation statements. For positive statements it is evident that the subject is required to take actions, within defined timelines and conditions, for successful enforcement.

For negative obligations, the subject is supposed to stop and inhibit certain actions till the deadline for enforcement. We classify these two forms of enforcement respectively as *proactive* and *preventive* enforcement. Now let's identify proactive and preventive actions from the above examples.

Example 3 Obligation statement with enforcement mechanism

1: Hospital X commits to delete patient's history in 1 month.
   ▷ *Delete* action to be performed proactively.
2: Hospital X commits to notify patient whenever his information is shared.
   ▷ *Notify* action to be performed proactively.
3: Hospital X commits not to share patient's history to anyone.
   ▷ *Share* action must be inhibited.
4: Hospital X commits not to use patient's history for any statistical purposes.
   ▷ Some *usage* must be inhibited.

During the course of investigation, we found that there are obligation statements which may not be negative in tone but semantically they are opposite of some proactive

action and their enforcement can only be done through preventive enforcement mechanisms. Example 4 depicts an example of a statement which is not negative in tone but enforcement is still being done by prevention. Action *store* is semantically opposite to

Example 4 Semantically negative obligation statement

Hospital X says X will always keep patient's data stored

action *delete*. The statement is equivalent to *NOT deleting patient's data*. We provided both enforcement mechanisms in our proposed architecture which is discussed in detail in Section 4.

3.2.2. Conditionality

It is very usual to have some form of constraints defined with the obligation statements. This was one of the very important aspects of obligations that the subject relates the fulfillment of committed promises on some conditions. If the conditions are not fulfilled then the subject is not held liable for not fulfilling its promises. Let's take the following example which we have extended from the previous section and added conditions to both positive and negative form of obligations.

Example 5 Obligation statements with conditions appended

1: Hospital X commits to notify patient whenever his information is shared **If** patient is registered at the hospital.
▷ Positive obligation with a condition specified.
2: Hospital X commits not to share patient's history to anyone **for** 10 years.
▷ Negative obligation with a condition specified. *for* is used to express a *timeframe* of the rule validity.

We covered this aspect by providing *application condition* construct in our obligation rule. Thus, a set of conditions could be attached to an obligation rule to make it applicable. Because of the inclusion of conditional expressions with each obligation rule, the expressiveness of the policy increases. The conditions on the obligation rule can be composed of temporal or non-temporal constraints.

Though we cannot vary the level of strictness of an atomic rule with these conditions except that either the rule would be active or inactive but we can vary the level of strictness of the whole policy by activating more rules or deactivating them with respect to time, environment and/or server state. Any obligation rule without an accompanying condition is considered *active* all the time.

To avoid the problem of time synchronization in temporal constraints, as the obligation policies will be shared across different systems, we restrict the literal values to be *absolute* and not relative like in the following Example 6

In the above example, *1 month* is ambiguous in a distributed scenario and must be replaced with absolute values when translated from static templates to actual policy in execution within the system.

Example 6 Obligation statement having temporal constraint with relative values

1: Hospital X commits to delete patient's history in 1 month.

3.2.3. Iteration

The third aspect is the iterative nature of obligation statements. The simplest forms of obligations, including most of the examples given until now, are fulfilled once (except negative obligations). However, we could encounter statements which are required to be fulfilled multiple times iteratively. When the fulfillment of an obligation is required multiple times during its life then we consider them as iterative or repeating obligations.

Example 7 Iterative or repeating obligation

1: Bank X commits to send account statement to customer C every 3 months.

The above example shows that the obligation will be triggered every 3 months. This aspect is being covered in our model by allowing multiple triggers, including periodic triggers, to be attached with a single obligation rule.

3.2.4. Stateful Obligations

Another important aspect of obligation statement is their stateful nature. In existing literature, we mostly found that obligations are either fulfilled or violated which are two atomic states. However, we consider that the state of the obligation could be somewhat fuzzy with many sub-states in between and the fulfillment or violation is achieved in steps. Such obligation statements are termed as *stateful* as their state need to be stored. Let's consider the following examples.

Example 8 Stateful Obligations

1: Hospital Y commits to share patient data only **3 times**.
2: Bank X commits to reimburse account holder A $100 in **10 installments** in first year after account opening.

Rule 1 in Example 8 puts a constraint which limits the execution of obligation only first three times. Rule 2 above can only be considered fulfilled once the whole amount is reimbursed and will be fulfilled in steps.

3.2.5. Time Boundedness

We could have obligations which are supposed to be fulfilled in a finite time or alternatively in an infinite time. We define infinite obligation which fulfill the following two conditions.

- Does not have any temporal constraints/conditions defined. This makes the rule active infinitely.

- Contain only non-deterministic triggers (see Section 3.3.1). Alternatively, if there is any deterministic trigger then the rule itself is repetitive and needs to be fulfilled infinitely.

Example 9 Time boundedness aspect of obligation

1: Bank X commits to give customer C a gift for referring a new account within 1 month of account opening.
▷ Time bounded obligation to be fulfilled within 1 month after account opening.
2: Bank X commits to give interest of 4% per annum.
▷ Unbounded obligation with no end time defined and will continue to be fulfilled infinitely.

It is recommended/required to have a condition attached to such obligations to stop their execution at some point in the future. In Rule 2, the condition could be *"account closure"* condition. We can also have un-bounded negative obligations where the subject is supposed to prevent some action to fulfill the obligation.

Example 10 Negative unbounded obligation

1: Bank X Says X Will Never Disclose Your Data To Anyone
▷ It's an unbounded negative obligation with no end period defined.

The above mentioned aspects were the most important which we addressed in our solution. There are certain aspects of the obligation statements which are beyond the scope of the work accomplished until now and will be addressed as part of future work. However, their overview would be worth mentioning as they provide the room of further research in these dimensions.

3.2.6. Observability

Pretschner et al [22] discussed the problem of observability of obligations and suggested methods to translate observable obligations into non-observable ones. However, we perceive that observability is not completely dependent on the rule itself but highly related to the authority and scope of the evaluator and monitor. Figure 3.1 shows different scopes of the evaluators. The external evaluator can only monitor the communication channels external to the subject trust domain. Internal evaluator can also monitor the internal communication between obligation enforcement platform and other infrastructure of the subject. Lastly the monitor inside the framework is the one which could log audit trail and monitor the actual processing of the enforcement framework.

Some obligation rules can be observed externally like Rule 1 in Example 11 while Rule 2 is inherently unobservable externally. Thus *observability* of obligation rule is dependent on the spectrum of visibility of monitor/evaluator and *type of rule*. Further work in this direction is beyond the scope of this thesis.

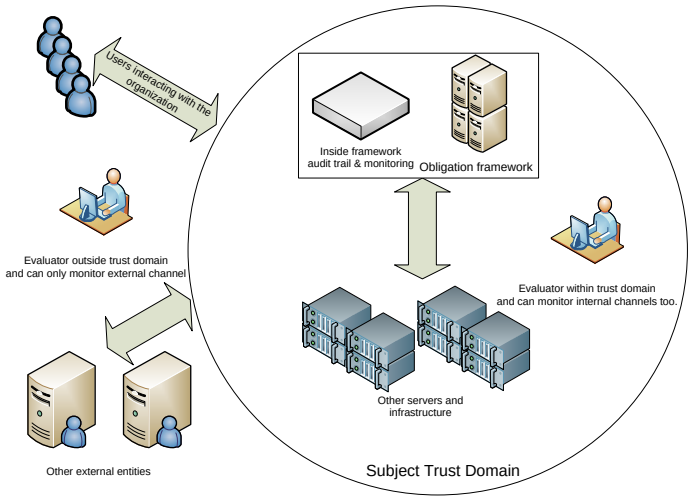

Figure 3.1.: Observability of obligations with respect to the scope of evaluators

Example 11 Observability of obligations

1: Service X commits to notify user @email when user's data is shared.
2: Service X commits to delete user's data in 1 week

In Example 11, the notification email is something which can be monitored externally. In Rule 2 however, the obligation is non-observable as it can't be monitored externally whether the subject has fulfilled it or not.

3.2.7. Delegation

Delegation of obligations could be another interesting area for extension. The basic idea is to allow constructs in the language which enable the policy issues to commit promises/obligations on behalf of other subject. Let's consider the following examples from the previous section.

Example 12 Examples of delegation of obligation

1: Hospital X declares that X's partners commits to pay compensation for using your data.
2: Team leader Y commits that his team members A,B will participate in the conference.
3: Service X declares that partners of Y commits to delete your data.
▷ Delegation of obligation by X to group of entities *partners of Y*. This is two level delegation and in theory these obligations can also exist.

In Rule 1, X is declaring on behalf of his partners which requires underlying reason like mutual agreement etc. Rule 2 depicts a scenario where Y commits on behalf of her subordinates and the underlying reason to commit something like that is authority and

position.

Obligation delegation scenarios like in Example 12 create ambiguity whether the commitment can be made or not on behalf of others. The simplest case is that policy declarer should only commit promises about itself. However, we can find cases very common in the real world where commitments are being made on behalf of others and behind those declared commitments there are some underline reasons like authority, influence, mutual agreement etc.

Responsibility Structure

Complementary to the problem of delegation of obligations is the proportion of responsibility in the cases of collective subjects. Cholvy el. al [11] mentioned this aspect of obligations. Whether we consider an obligation to be fulfilled collectively like by everyone, by only one among the directed subject group etc. Obligations where the subject of the rule is an individual entity has the full responsibility of fulfilling it.

Example 13 Responsibility structure of obligation

1: Hospital X commits to pay compensation for using your data.
2: Hospital X declares that X's partners will pay compensation for using your data.
3: Team Leader X commits that his team will participate in the conference.

In collective obligations, the subject of the rule can be a named group, set of attributes which define a group of individuals or collection. In Example 13 Rule[2] the subject is X's partners which is a collection. The responsibility is not defined whether one of the partners will pay, all of them contribute equally or each will pay some pre-defined amount etc. In Rule 3, the ambiguity is there whether the whole team will participate, some members of the team will participate etc. Such statement where an obligation is on the whole collection of atomic entities is somewhat complex and need further in depth reasoning. For our scenario, *subject* is a service provider.

This is one of the possible future directions of work to investigate such scenarios where obligations are delegated, promises committed on behalf of a group of entities and their corresponding responsibility structures and finally to extend the current language based on the results. We now present a formal obligation model in the next section.

3.3. Formal Obligation Model

This section defines the semantics of the obligation rule, ruleset and policy structures in detail. We also present how we covered the different aspects discussed in last section through the proposed model.

3.3.1. Obligation Rule

Formally, we represent an obligation rule[3] o as a tuple $\langle s, a, \xi, c, e \rangle$ where s is a subject which is obliged to fulfill the obligation, a is an action which is executed/prevented to fulfill the obligation, ξ is a set of triggers, c is a boolean equation specifying conditions under which the obligation rule would be active and e is the set of events which are sent outward in case of a change in state of obligation e.g. violation or fulfillment. We use O as the set of all possible obligations. s, a and ξ are mandatory components of an obligation rule while c and e are optional.

Subject

Subject s is an identifier to identify the data processing entity that commits to fulfill an obligation; $s \in S$ where S denotes the set of all existing subject identifiers. The subject can be a unique entity which can be the policy issuer itself or there is also a possibility that the policy issuer may declare an obligation on behalf of a third party as a subject. It can even be a group of entities falling under the same category defined through name or a set of attributes.

We could have a number of ways to express a subject within an obligation rule. Let's discuss a very simple obligation and then its variations with respect to the subject entity and keeping everything else as constant.

Example 14 Different examples of *subject*

1: Service X commits to Delete U's Data.
   ▷ Simplest form of obligation where the subject and policy issuer are same entities.
2: Service X declares that Service Y commits to delete U's Data.
3: Service X declares that Partners of X commits to delete U's Data.
4: Service X declares that Partners of Y commits to delete your Data.
   ▷ Delegation examples as presented in Example 12.

In Example 14, the last rule depicts a possibility of having two levels of delegation. In real world, such examples do exist however this is part of possible future extension to the language. It is important to note that in all the rules depicted in Example 14, the policy provider entity is the same and only the subject is changing.

Action

Action is by far the most important part of the obligation clause. Action a is the activity executed to fulfill an obligation and is represented as a tuple $\langle i, p, at \rangle$ with $i \in I$, where I represents the set of all possible action identifiers. Each element of I can be uniquely mapped to available actions within the system using a bijection $map : I \rightarrow A$. The action parameters p is a set of name/value pairs. We classify actions by their action type $at \in \{proactive, preventive\}$

[3] We use the term *obligation*, *rule* and *obligation rule* interchangeably in the report. However, they represent the same thing

- *Proactive* actions require the execution of actions proactively. For example Delete, SendEmail etc.
- *Preventive* actions can only be prevented from executing to fulfill the corresponding obligation promise. This class of actions add lot of expressiveness into our language and does allow negative obligation statements like *Subject X commits never Sharing U's data with anyone* where the action *never share* itself is never executed but the fulfillment is done by preventing the action *share*. For accomplishing this behavior, we need to integrate our framework with external infrastructure of the organization.

This classification is taken to incorporate both enforcement mechanisms of obligations as discussed in Section 3.2.1. We do not allow actions which can be used as both proactive and preventive to avoid consistency problems. The enforcement framework knows which actions are conflicting with other actions within the system. Thus, this meta information aid the consistancy check tools for detecting inconsistant policies.

An obligation rule contains a single action, however we envision that this action itself can be composed of many basic actions arranged in a complex manner. This restriction has been put to avoid ambiguity. Indeed, if we would have two actions, e.g. *Delete* and *SendNotification*, in the same rule and the first one is executed successfully while the second fails the overall status of the rule is undecidable (fulfilled or violated). We consider deciding the status of rules having multiple actions as a separate problem which we may address in future.

Composition of atomic actions into composite actions would increase the flexibility of the language and help users to structure domain specific dialects of the obligation language in a more efficient and error-preventing way. However, the downside is that composite actions introduce a level of complexity that makes it harder to detect contradictions inside an obligation rule. We would need to consider the sub actions to establish the semantic contradiction relation (see Consistency issues in Section 3.5.1).

Perhaps a set of design rules defining how action may be composed could help to deal with this complexity. For instance, there must be a plugin within the system to handle each sub-action of a composite action. Additionally, the set of parameters for a composite action must *satisfy* the set of parameters of each sub action. Here, the composition may itself be important. The output from one sub-action could be an input of the next sub-action within a composite action. Let $a :< i, p, at >$ is composed of n sub actions represented as $a_n :< i_n, p_n, at_n >$ then...

- $\nexists a_x | 1 < x \leq n \wedge map(i_x) \notin \{A\}$
- p satisfy $\bigcup_{1}^{n} p_x$

From very simple to extremely complex possibilities to express actions does exist. We consider actions to be parametric and need a set of parameters for their execution.

Action parameter can be anything. It could be any resource (PII, DB, File, and URI etc). The resource itself may be residing within the system (of the policy provider),

Example 15 Examples of *actions*

1: Service X commits to Notify(Email=XYZ) if U's data is used.
   ▷ Notify is the action, Email is parameter.
2: Service X commits to Delete(U's data).
   ▷ Delete is the action. U's data is the parameter.

remote resource (provided the policy provider has rights on it) or even something which is non-existent at the time of interplay but generated in the future.

For our data handling scenario, we assume that the obligation policy would be sticking with the user PII. This user PII would be an implicit parameter which all the rules within a policy can utilize. We start with the simple examples by changing objects and keeping the rest of the components of the clause as constant.

Example 16 Examples of *action objects/parameters*

1: Service X commits to Delete U's data.
   ▷ U's data is the object and action is applied directly on it.
2: Service X commits to Notify(Email=XYZ)
   ▷ Action is not applied on the policy object (U's PII).

Triggers

Triggers define the types of inward events which result in the execution of the obligation's action. Multiple triggers can be defined for a single obligation. Triggers can be *deterministic* where we know the precise time instant when the trigger will be fired and we classify them as *AbsoluteTimeTriggers*. Such triggers can be defined by a tuple $\langle \tau, d \rangle$ where τ is any absolute point in future time and d is the timeout duration. The rule must be fulfilled before $\tau + d$. Deterministic triggers, in conjunction with temporal conditions, provides the capability to express time bounded obligations as the rule activation time frame and time to trigger execution are known in advance.

Trigger can also be *non-deterministic* and fired in reaction to locally or externally generated events. For performance reasons, we suggest that these events should have the user data unique ID that is used to select the corresponding policy and in turn related obligation. Beside this mandatory information these external triggers may accompany additional parameters depending on their type. Non-deterministic trigger is defined by a tuple $\langle ty, p, d \rangle$ where $ty \in T$, where T denotes the set of all existing trigger types in the system, p is a set of trigger parameters expressed as name/value pairs and d is the deadline duration as defined before. Parameters for trigger are not specified within the policy but the triggers when fired will accompany them. For detailed trigger classification see Appendix B.

Application Condition

Application condition expressions are boolean equations defining whether a rule is applicable. When a trigger is received (e.g. delete resource r), the system takes into account any obligation $\langle s, a, \xi, c, e \rangle$ having such trigger ξ. If the condition c of an obligation is evaluated to true, the obligation's action is executed.

Depending on the result of the action, the obligation will be considered as fulfilled or violated. If the obligation is non-repeating, it will disappear from the system after fulfillment or violation.

The condition expression is expressed as *Sum of Products*. We have shown the grammar of condition expression (cexpr) in EBNF form below.

$$\begin{aligned} cexpr &= \{pterm\}; \\ pterm &= \{cond|cexpr\}; \\ cond &= name, \{parameter\}; \\ parameter &= function|variable|literal; \\ function &= returntype, name, \{parameter\}; \\ variable &= name, type \\ literal &= \{a...z|A...Z|0...9\} \end{aligned}$$

For example, in order to have temporal constraints on the obligation rule we can define a time frame function which can then be used in policies. An example condition expression has been shown below

$$\begin{aligned} cexpr = \quad & (Timeframe(t_s, t_e) \wedge UsageLimit(i)) \vee \\ & (System.State == green) \end{aligned}$$

Timeframe and *UsageLimit* are the function/condition names. Conditions are subset of functions which always have the *boolean* return type and thus can be used in the product terms (pterm). The second product term specifies that if the environment variable *System.State* is *green* then condition is applicable. The two product terms are OR-ed together. The environment variables are specified in the *variable repository*, which is being discussed in Section 4, and values are set/defined by the administrator.

Events

Optionally obligation rule can have outward events which are used to inform other rules within the same policy and external entities about change of status like fulfillment or violation etc. Event of one obligation rule can be a trigger for another rule and through this design we implement the notion of cascaded obligations. Formally, $e \in E$ where E denotes the set of existing events. For internal processing of the events we may attach additional parameters with the outward event which are directed toward the event receiver or handler.

Any two obligation policies, each attached to a specific piece of data, are not assumed to interact with each other. Thus we restrict that the events from one policy could

only be consumed as a trigger within the same policy. However, they can serve as a notification to integrated external systems within the same trust domain but outside the obligation framework.

3.3.2. Rule Set

For actual implementation and clarity purposes, we segregate the term *subject* from the obligation rule. Instead of defining subject repetitively with each obligation in a policy, we define all the subjects first and then rest of the parts of obligation is defined in sequence as *partial rules*. This composition with list of subjects and list of partial rules is defined as *RuleSet*. Ruleset can be defined as a function as follows.

$$
\begin{aligned}
&Ruleset : \Im \times \Re \overset{map}{\rightarrow} \Phi \\
&Where \\
&\quad \Im \subseteq \quad S \\
&\quad \Re = \quad finite\ set\ of\ partial\ rules \\
&\quad \Phi \subseteq \quad O
\end{aligned}
$$

Ruleset must have at least one well defined partial rule and subject $\Rightarrow |\Re| > 0 \wedge |\Im| > 0$. The simple Cartesian product of these two disjoint sets will give us the set of obligations.

3.3.3. Policy

Obligation policy is a set of one or more well defined obligation rules which are consistent as a whole. Formally

$$\rho = \{o : o \in O\} \wedge |\rho| > 0 \tag{3.1}$$

The overall structure of the proposed obligation language has been designed to allow multiple subjects in the same policy. If the subjects are mutually exclusive then the policies can be merged. If they are not then we would need to ensure consistency. The overall structure of the policy is represented below in EBNF form. Mutual exclusion of subject implies that there is no containment relationship between two subjects within the same policy.

$$
\begin{aligned}
ObPolicy &= protocol, rules \\
rules &= \{ruleset\} \\
ruleset &= \{subject\}, \{rule\} \\
rule &= action, \{trigger\}, [cexpr], [\{event\}] \\
protocol &= \text{protocol specific data not yet defined}
\end{aligned}
$$

The obligation policy (Obpolicy) also contains protocol related fields which are used during the interplay between the user and the service provider. These fields are not

used in the internal processing of the policy but are used for mutual authentication, shielding from tampering, confidentiality of the policy on the wire and optimization purposes. Figure 3.2 shows the policy structure graphically while detailed XML schema and example policies are presented in Appendix A .

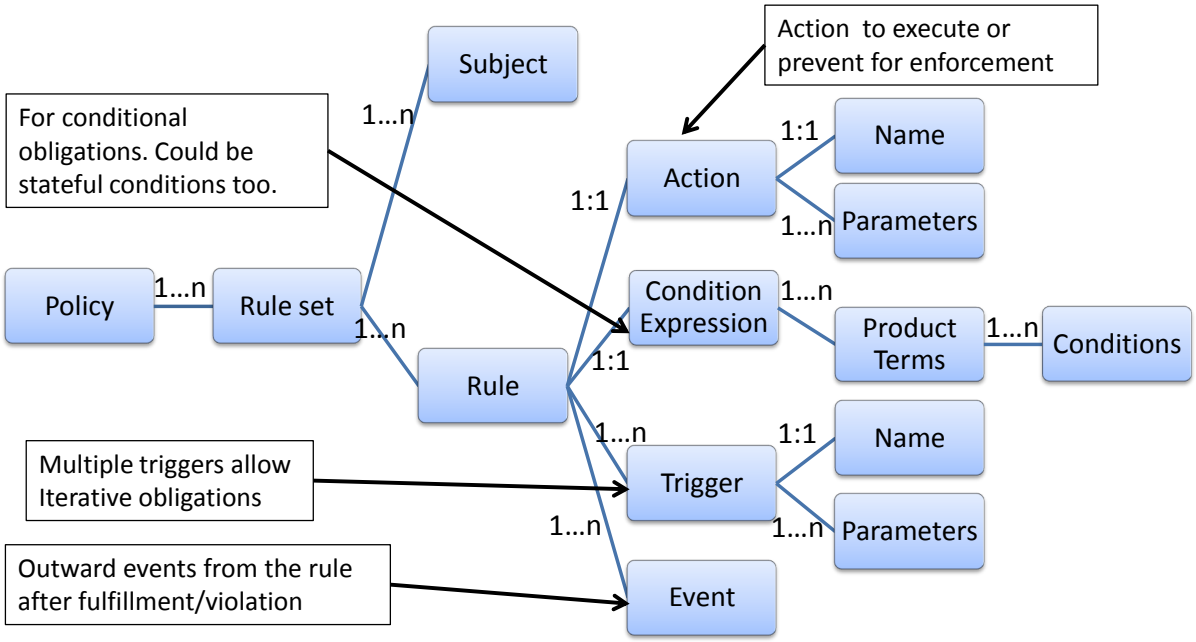

Figure 3.2.: Policy Schema Illustration

3.4. Integration with Existing Policy Languages

The obligation language presented above aim at enforcing obligations that may be defined with existing policy languages. It was also one of our requirements as was mentioned in Section 1.2. This work can be used to enforce obligations specified in access control policies (e.g. XACML [24, 29]) or privacy policies (e.g. PRIME-DHP [4]).

For example XACML provides constructs to express access control requirements but with minimal support for obligation. In this case, obligations are defined and enforced at service-side. When a request is evaluated, the Policy Enforcement Point (PEP) gets a decision and a list of obligations to enforce. The enforcement of those obligations can be handed to our proposed obligation framework. It is important to note that XACML only supports proactive obligations triggered by an access control decision while our framework supports a wide range of obligations.

In other scenarios, the obligations may be pushed by the user to the service hosting the obligation framework. For instance, Enterprise Right Management [1] may link

a document to a license which could in principle specify some obligations. Or, using PRIME-DHP the obligation may be customized by the user before being attached to personal data as a sticky policy. In these cases, obligations associated with data are dynamically added to the obligation framework.

We now present the possible consistency and safety problems that could result is ambiguous or unenforceable policies.

3.5. Properties of Policy Language

Barth et al. [6] discussed properties of policy languages namely consistency, safety, local reasoning and closure under combination. It is impossible to have a language having all of the four properties without compromising the expressiveness [6] however one can target one or two of these which may be of importance for us. We perform reasoning on our proposed language to understand the relation between obligation rules within a single policy against each other.

We focus mainly on consistency issues and up to certain extent cover the safety problems too but major work should be done in this area to identify new problems.

3.5.1. Consistency

The problem of inconsistency arises when a policy contradicts itself. It could be in the simplest form because of the presence of opposite rules and negation operators. Even in the absence of negation operators, two obligation rules can be inconsistent semantically. In order to avoid such problems, the obligation engine must know the relation between different obligation rules. More complex forms of inconsistencies can also arise which may be difficult to detect. To make it clearer, let's consider the following two example rules which are part of the same obligation policy.

Example 17 Inconsistent rules

Service X says X will Delete your data in 1 week
Service X says X will Store your data for 3 months

In Example 17, we are not using any negation operators but the rules are still conflicting each other and thus make the policy inconsistent. The policy consistency checker must know the relation between *delete* and *store* actions to ensure that such problems does not exist.

Let $o_1 = \langle s_1, a_1, \xi_1, c_1, e_1 \rangle$ and $o_2 = \langle s_2, a_2, \xi_2, c_2, e_2 \rangle$ with $o_1, o_2 \in \rho$ be two obligations, where ρ is an obligation policy as defined before. To illustrate inconsistency issues, we first define the *semantic contradiction* relation. We use the operator $\bowtie$ to represent semantic contradiction between two entities. This is not an equality relation and holds the following properties.

- Symmetric: if $o_1 \bowtie o_2$ then $o_2 \bowtie o_1$

- Non Reflexive: $o_1 \not\bowtie o_1$
- Non transitive: if $o_1 \bowtie o_2 \wedge o_1 \bowtie o_3 \not\Rightarrow o_2 \bowtie o_3$

An obligation policy ρ is inconsistent if one of the following conditions is true:

- ρ has an obligation rule which is inconsistent, i.e. $\exists o \in \rho : o \, is \, inconsistent$.
- ρ contains two semantically contradicting obligation rules, i.e. $\exists o_i, o_j \in \rho : o_i \bowtie o_j$.

The first case arises when any rule within ρ is not well written, contains actions or conditions whose processing plug-ins are not present within the system.

The second case occurs when two rules, having the same subject and overlapping conditions, are contradicting each other because of contradicting actions. If the condition is not overlapping then it is not necessarily a consistency error. We define function $IsConditionOverlap(c_1, c_2) \in \{true, false, undefined\}$ which returns whether two conditions do overlap. *Condition overlap* establish whether the two rules could become active within the same time frame in future. If they do then they may be triggered both at the same time and because of the contradiction it would be impossible to execute both. During plug-in design, we define explicitly which actions are contradicting each other within the system. This meta information aid policy writers to write consistent policies.

$$\begin{aligned} &\text{If } a_1 \bowtie a_2 \wedge s_1 = s_2 \wedge \\ &\quad IsConditionOverlap(c_1, c_2) = true \\ &\quad \Rightarrow o_1 \bowtie o_2 \end{aligned}$$

There could also be cases when it would be undecidable to establish the condition overlap relation and in such cases we can only raise a warning to the policy writer. We can take *undefined* as a consistency error but that will reduce the expressiveness of the language.

$$\begin{aligned} &\text{If } a_1 \bowtie a_2 \wedge s_1 = s_2 \wedge \\ &\quad IsConditionOverlap(c_1, c_2) = undefined \\ &\quad \Rightarrow o_1 \bowtie o_2 \text{ is undefined} \end{aligned}$$

Otherwise, *IsConditionOverlap* returns *false* which ensures that the two rules, having contradicting actions, would be active in a separate time frames and it is safe to have them within the policy. A further more complex form or action contradiction exists when actions are a composition of many granular actions. In such cases, we would also need to consider the sub actions to establish the semantic contradictiong relationship. *SubActions(a) = Set of all the sub actions within action 'a'*

$$\begin{aligned} &\text{If } \exists\, a_3 \in SubActions(a_2) \\ &\quad |\, a_1 \bowtie a_3 \Rightarrow a_1 \bowtie a_2 \\ &\text{If } \exists\, a_3 \in SubActions(a_1) \wedge \exists\, a_4 \in SubActions(a_2) \\ &\quad |\, a_3 \bowtie a_4 \Rightarrow a_1 \bowtie a_2 \end{aligned}$$

It is also required that composite actions must be executable themselves. There could be cases when two children of a composite action are incompatible but we still combined them to form a new action which could be used in policies. To form composite actions, it is strictly required to follow action composition rules to avoid inconsistencies. To establish such set of rules is out of the scope of this thesis and we only designed plug-ins for atomic actions in the architecture.

3.5.2. Safety

The problem of safety mainly arises when we have obligation policies defined at multiple levels of organization and these policies are merged. To define safety issues within a policy, we define the notion of subset relationship among the obligations. Formally, if $s_1 \cap s_2 \neq \emptyset \wedge a_1 \bowtie a_2 \wedge \text{IsConditionOverlap}(c_1, c_2) = \text{true} \Rightarrow$ p is unsafe.

The concept of subject overlapping arises from the vertical subject hierarchies. A policy subject may have many sub-subjects each having its own policy. For example, if we have an organization wide policy and one policy per business unit etc.

The problem of safety arises when there is a difference between the intended purpose of an obligation policy as a whole and actions taken by engine based on the policy. It can arise because of many possible reasons like containment, cyclic dependency, cascaded obligations etc.

Containment

Some of the obligation rules can be a subset or contained within another. If we put less strict conditions on the super set than the subset then the policy becomes unsafe.

Example 18 Inconsistent rules

Bank X says X's credit-card-sales-people will Check your opt-out option before contacting

Bank X says X's Sales people will contact you

In Example 18, the *credit card sales personnel* can contact without even checking the opt-out options of the customer as the second rule give them the right to do so. The above example is unsafe because of superset/subset relation on the intended subject in the rule that is $credit - card - sales - people \subset sales - people$

Cascaded Obligations

Whenever an obligation is fulfilled, it can optionally generate some events which are routed back into the obligation engine which can invoke further obligations. Such a situation is termed as *cascaded obligations* where some of the rules within a policy are dependent on other obligations to be fulfilled.

Now, in this case the second rule has got a condition which is being fulfilled when the first rule will be fulfilled. Such cases are cascaded obligations. We need to ensure non-existence of cyclic dependencies (the cycle can even be a large having 2, 3 or

Example 19 Cascaded rules

Hospital X commits to Delete U's data before <d>
Hospital X commits to Notify(Email=XYZ) when U's data is deleted.
▷ Second rule is dependent on first.

more obligation rules involved) and infinite cascading. It must be ensured that infinite cascading of rules must not happen and cycles are identified at policy writing time.

Redundant Obligations

Following is an example of redundant obligations.

Example 20 Redundant rules

Service X commits to Delete your data on <01/10/2009>.
Service X commits to Delete your data on <01/12/2009>.

The existence of first rule makes the second unreachable or redundant as once the data is deleted on $< 01/10/2009 >$, it may no more be deleted again on $< 01/12/2009 >$.

There are possibilities of having action precedence with some actions which cannot be repeated e.g. *Delete User Data* which is non-repeatable action as once the data is deleted then it cannot be deleted again.

Similarly, after the deletion of data the attached policy is invalid. Thus, many obligation rules which are executed after *delete* action may become *redundant* or *non-reachable*.

Some of these problems can be detected by constructing a dependency relationship tree between rule statements within a same policy. If there would be any cycles then the resultant would be a graph which must be converted into a tree. The tree is then traversed to detect *application condition timeframes* and *action precedence*. Each child within such a tree must be applicable in the timeframe after the fulfillment of its parent. Secondly, action precedence is checked to detect whether action of child can be executed after the execution of parent rule action. In many cases it would be undecidable to detect condition overlaps and it would be policy writers who ensure safety and consistency of policies.

The description of the obligation model ends here however it is still left to be presented that how we formalize the example obligation statements presented throughout this chapter which is the topic of the Section 3.6 below.

3.6. Formalization of Obligation Statements

This section describes the formalization of obligation statements from natural language to the rules expressed in our language. We will take some of the examples already given before and then break the statement into components which forms the obligation. We will start with the simple obligation rule.

Example 21

1: *Hospital X commits to delete patient's history in 1 month.*
2: Policy Object = Patient's History
3: Policy Issuer = Hospital X
4: Subject = Hospital X
5: Action = Delete(Target=Object)
6: Trigger = Time based trigger at the end of 1 month.
▷ This is a proactively enforced obligation.

In Example 21, the rule will be triggered by a time based trigger at the end of one month (in real policies we only use absolute date time value). The action here is *delete* which will be executed on receiving the trigger and the target object is deleted.

Example 22

1: *Hospital X commits not to share patient's history to anyone for 10 years.*
2: Policy Object = Patient's History
3: Policy Issuer = Hospital X
4: Subject = Hospital X
5: Action = Prevent(ActionName=Share)
6: Trigger = Pre-Action(ActionName=Share)
7: Condition = Timeframe(10 Years)
▷ Preventive or negative obligation. Name of action *prevent* is not taken because of the enforcement type.

In Example 22, the Action is *Prevent* with a parameter name of the actual action which is to be prevented. If the *Share* action itself is complex then we can promote it to make it a full action itself (See Appendix B for details). The corresponding action plug-in with the name *Prevent* must be present in the system along with matching parameters. Trigger is of *Pre-Action* type with the same *ActionName* parameter. If the trigger is very complex then we can promote it to form a new trigger type *CanShare* (See Appendix B for details). After 10 years rule will be inactive and will no more enforce prevention. In preventive obligations framework always send a reply to drive the external system. Here rule inactivation means that the data can be shared.

Example 23

1: *Hospital X commits not to use patient's history for any statistical purposes.*
2: Policy Object = Patient's History
3: Policy Issuer = Hospital X
4: Subject = Hospital X
5: Action = Prevent(ActionName=Use, Purpose=Statistics)
6: Trigger = Pre-Action(ActionName=Use, Purpose=Statistics)
▷ Preventive or negative obligation.

Example 23 is same as previous example but now action and trigger are more complex with two parameters. There is no condition defined so enforcement will be done till policy is present.

Example 24

1: *Hospital X commits to notify patient whenever his information is shared If patient is registered at the hospital.*
2: Policy Object = Patient's History
3: Policy Issuer = Hospital X
4: Subject = Hospital X
5: Action = Notify(Email=<XYZ>)
6: Trigger = Post-Action(ActionName=Share)
7: Condition = User must be registered.
▷ Proactively enforced obligation.

In Example 24, a notification is sent to Patient whenever his data is shared. Here the trigger is *Post-Action* with the action name as parameter. When the same trigger with the same parameter is received then the rule is executed and notification is sent.

Example 25

1: *Hospital Y commits to share patient data only 3 times.*
2: Policy Object = Patient's History
3: Policy Issuer = Hospital Y
4: Subject = Hospital Y
5: Action = Prevent(ActionName=Share)
6: Trigger = Pre-Action(ActionName=Share)
7: Condition = Usage > 3
▷ Preventive or negative obligation with condition.

Example 25 is same as Example 23 with the exception that the condition is stateful and updates its state. The condition state will be updated by the framework every time the rule is triggered. When the rule is inactive, it means here that sharing is allowed. So the enforcement will start when *Usage* is greater than 3. We can also implement the same obligation statement with a reverse logic if we provide an action plug-in which evaluates to negative logic as compared to *Prevent* action. Let that action plug-in be *Allow* and the parameter remains the same. Example 26 shows the new formalization.

In Example 26, the action is reversed and thus the condition logic. Now the rule will be active till usage is below 3 and after this the rule will be inactive which means that data cannot be shared now. Thus, it is up to the designer of the actual deployment how they design plug-ins and implement the obligation statements. The possible combinations could result in a wide range of scenarios. It is important to mention that currently we assume that if the trigger comes and there is no obligation rule found for it or the rule is inactive then it implies that there is no obligation to fulfill or prevent and thus the source of the trigger can take actions as per its own discretion.

Example 26

1: *Hospital Y commits to share patient data only 3 times.*
2: Policy Object = Patient's History
3: Policy Issuer = Hospital Y
4: Subject = Hospital Y
5: Action = Allow(ActionName=Share)
6: Trigger = Pre-Action(ActionName=Share)
7: Condition = Usage < 3
▷ This is again preventive obligation like in Example 25 but with a reverse logic.

Example 27

1: *Bank X commits to send account statement to customer C every 3 months.*
2: Policy Object = Customer Data
3: Policy Issuer = Bank X
4: Subject = Bank X
5: Action = Send(Object=Account Statement)
6: Trigger = Absolute Time Trigger Cycle(Period=3 Months)

Example 27 presents a repeating trigger which is fired every 3 months. Action is to send the account statement.

Example 28

1: *Bank X commits to reimburse account holder A $100 in 10 installments in first year after account opening.*
2: Policy Object = Customer Data
3: Policy Issuer = Bank X
4: Subject = Bank X
5: Action = CreditAccount(Amount=$10)
6: Trigger = Absolute Time Trigger Cycle(Period=1 Month, Count = 10 Times)

In Example 28, triggering will be done 10 times and after that there will be no more triggers.

The examples given above only give a brief overview of some of the possibilities and there is no exhaustive list. A lot of new examples, scenarios and corresponding formalization need to be investigated and new possibilities could be discovered. Next chapter presents the architecture and implementation details of the presented obligation framework.

4. Implementation

We have designed and implemented enforcement architecture for the obligations expressed through our proposed language. The detailed obligation framework architecture is illustrated in Figure 4.1.

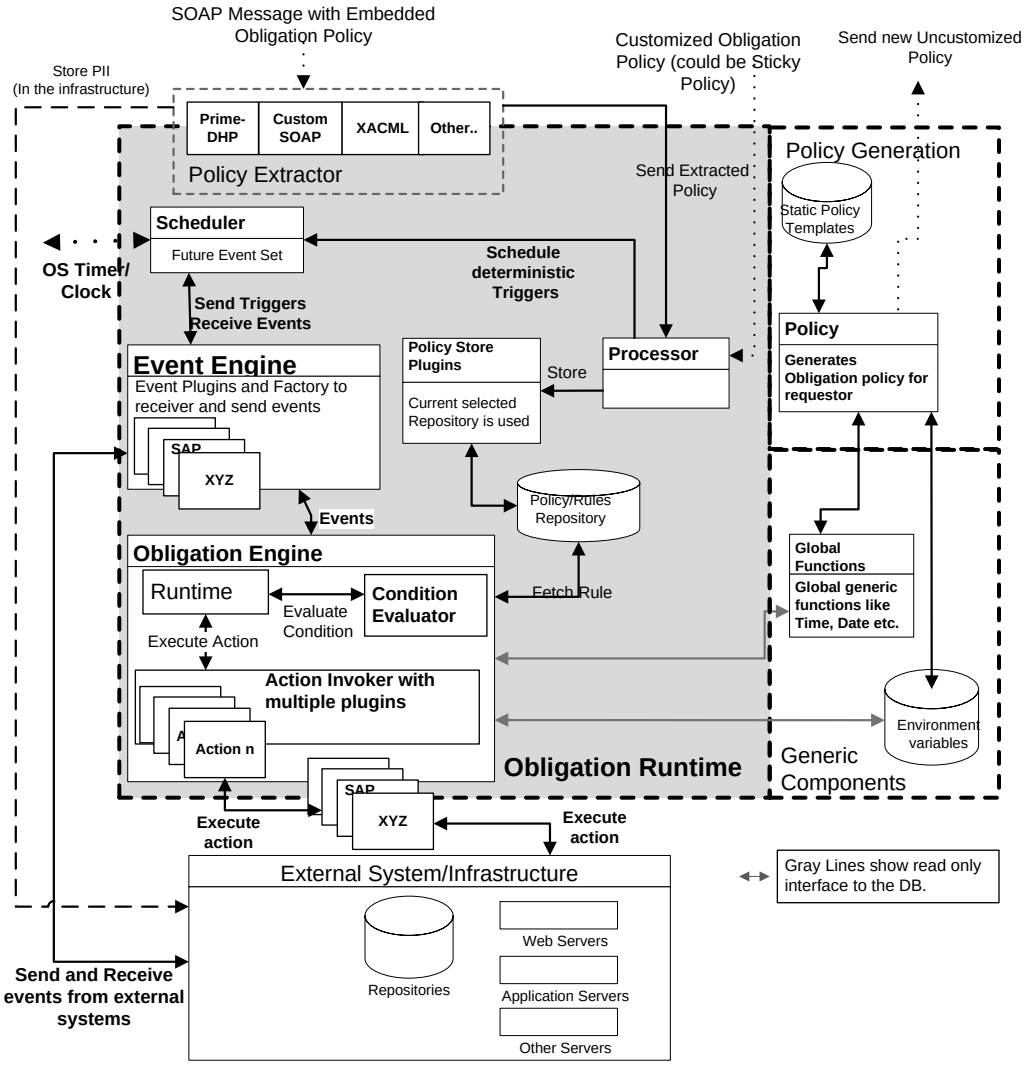

Figure 4.1.: Obligation Framework Architecture

The core requirements of the architecture were to ensure the enforcement of obligations, to simplify the auditing procedures for evaluation and monitoring, to support different deployments, to enable customized actions, to facilitate integration with existing systems, and to support external systems and processes. As shown in Figure 4.1, the architecture is separated into three main parts, namely *Policy generation*, *Generic components*, and *Obligation Runtime*.

We initially had two possible approaches for the enforcement of obligations. First, to allow everything to be managed by the obligation framework including user data and policies but this design would have restricted the integration of framework with existing systems and huge data migration would have been required to move data from existing legacy repositories to new obligation framework controlled repositories. Furthermore, the architecture would not have been independent and decoupled.

Second approach is to design an independent policy management framework which only manages the policies. This design is more flexible as it allows easy integration with existing and new systems. We now discuss each of the three main parts of the architecture in turn.

4.1. Policy Generation Components

Policy Generation section consists of the software components used mainly for policy creation. The underlying idea is to store Obligation policies in the form of *policy templates* with annotated fields. Once the request is received for a new policy one of those templates is extracted from the repository based on the context of the request.

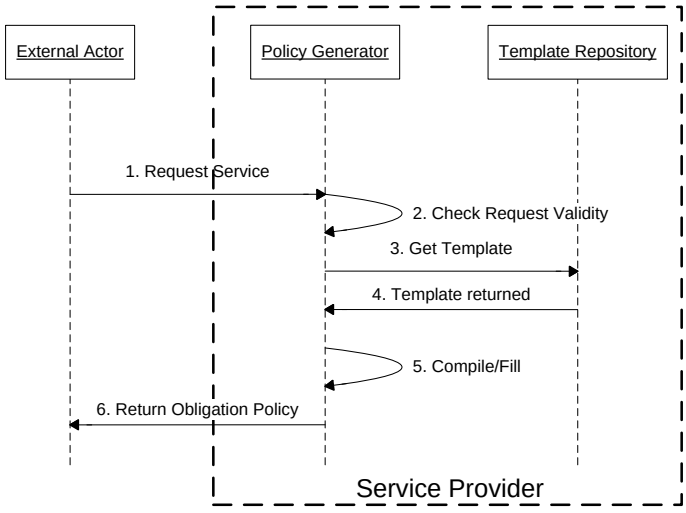

Figure 4.2.: Policy Generation Activity

The templates are processed to fill missing fields based on the environmental variables

etc. At this point, we call it as a *customizable policy* which is then sent back to the requester. The final customization is done at the requester's end which will then return the customized policy. This policy generation methodology is illustrated in Figure 4.2.

Listing 29 shows the proposed form of templates which are received from the repository (Figure 4.2 Step 4). Static Obligation policy templates are same as the actual obligation policies with the exception that they could contain *placeholders* and *IF/ELSE statements* which can be filled at runtime from different sources which could be

- Variable repositories (Database, File, URI, Web Service etc).
- Result of any environment functions (Current Time, Date, Temperature, Aggregating functions etc).
- State of the server.
- Requestor credentials.

At runtime, the conditional statements are evaluated and the corresponding blocks are taken into the final generated customizable policy for the requestor as shown in Figure 4.2 Step[5].

Listing 29 Static Policy Template

```
1: Service X commits to delete data whenever requested by the owner/user.
2: if Requestor.Type == 'Gold User' then
3:     Service X commits to delete data in %Gold.DeletionTime% months.
4:     Service X commits to notify user when data is deleted.
5: end if
6: if Requestor.Type== 'Silver User' then
7:     Service X commits to delete data in %Silver.DeletionTime% month.
8: end if
```

In Listing 29, *%Gold.DeletionTime%* and *%Silver.DeletionTime%* is a place holder and the value of it will be filled at runtime from a variable repository.

The compiled obligation policy for a *requestor* who is also a gold user is shown in Listing 30. Deletion time value *6* came from the environment variables. Thus we could control the policies for different users and could generate flexible policies. Listing 30 shows the final output sent to the user in Step (6). It is important to note that still the policy is customizable as the user may remove some rules from the policy (optional obligations). It may also modify the overridable parameters like value of data retention time which is *6* in the Listing 30 Line 2.

Definition of complete conditional statement constructs are beyond the scope of this thesis report and we have only implemented a schema for policy language. However, the current implementation does allow us to select among different templates, placed in final policy form, and are sent directly to the user without any compilation.

Listing 30 Static Policy Template

```
1: Service X commits to delete data whenever requested by the owner/user.
2: Service X commits to delete data in 6 months.
3: Service X commits to notify user when data is deleted.
```

4.2. Generic Components

Generic components are an optional set of components used to store environmental variables, global functions etc. During policy generation, variables in the template are replaced by their values from these repositories as discussed in Section 4.1. The provision of a variable repository is mainly for the administration of the system. The configuration is being done by the system administrators for effective operation.

4.3. Obligation Runtime

Obligation Runtime is the central part of the architecture and is responsible for the overall obligation enforcement and processing. One of the key features of this framework is its flexibility. The flexibility is achieved through the plug-in based design, which allows integrating new types of obligations and with different types of external systems. The framework uses the available plug-ins to execute different tasks. We assume that the framework is authorized to perform all the obligation actions on the external entities. This is generally achieved by deploying obligation framework and external systems (e.g. databases, email servers) within one single trust domain. This problem has already been discussed in chapter 1. Irwin et.al. [17] also discussed this problem of accountability of obligations when the system does not have sufficient rights to execute an action which should not be considered as a violation of obligation. In the subsequent sections we describe each component within the runtime environment in detail.

4.3.1. Policy extractor and translator plug-ins

The policy extractor plug-ins, as shown in Figure 4.1, could be used either for extraction or translation.

Extraction of the obligation policy from the incoming message, which could be in any format, can be done as long as the required extraction plug-in is present for that specific format. The corresponding plug-in parses the message and forwards only the obligation policy part to the system as shown in Figure 4.3. Alongside extraction the plug-ins create cross references between the message and the extracted policy. The remaining part of the message is returned to the caller for further processing.

The second type is translator plug-ins which, along side extraction, can also translate rules expressed through other obligation specification languages into rules in our language for enforcement as shown in Figure 4.4. However, to design such translation mechanisms we need to conduct additional research for different forms of existing policy languages which could be translated into our language. We would also need to consider

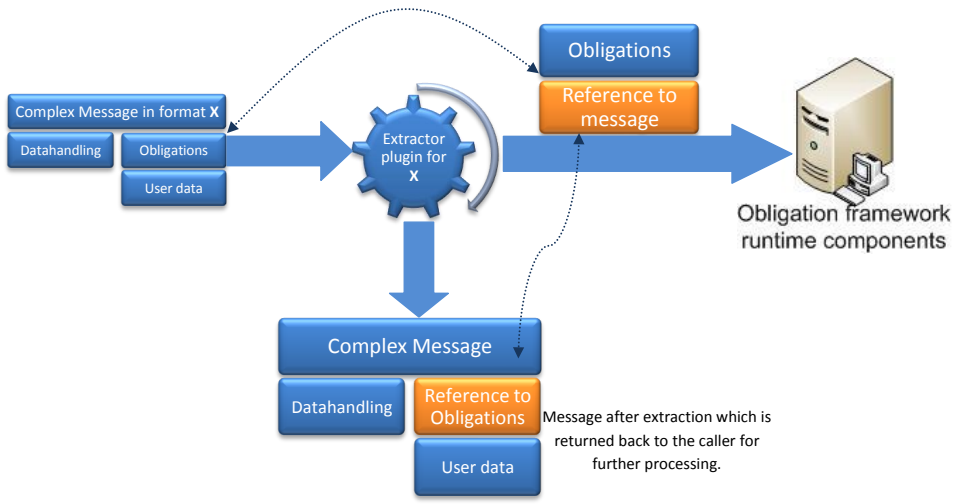

Figure 4.3.: Policy extraction mechanism

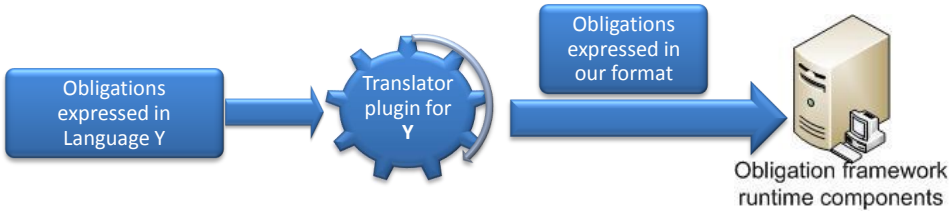

Figure 4.4.: Policy extraction mechanism

the expressiveness of those languages as some of them may be more expressive than our proposed language. For instance, Pretschner et.al. [28] proposed obligation specification language (OSL) and provided translation to rights expression language (REL) to use existing REL enforcement framework. We believe that the same can also be translated into our language and enforced through our framework.

4.3.2. Scheduler

Scheduler is used to initiate time triggers which are scheduled by other components of the runtime engine. The triggers are being sent in the form of messages to the event engine. The scheduler itself contains a repository to store and retrieve *future events* to ensure integrity in the event that the system shuts down. We call scheduled triggers as *future deterministic event set*.

4.3.3. Event Engine

Event engine is the central collection and distribution component. The major goal to have a single point of event receiving and distribution is to ensure integrity. All the external systems, scheduler and obligation engine communicate to other components through the event engine. This component also, like scheduler, keeps track of the received and processed messages. Storing and retrieving these active messages in case of system shut down or malfunction is also the responsibility of this central component. It behaves mainly like a queuing component.

4.3.4. Event Engine Plugins

For integrating the framework with other external systems we have provided an additional set of plugins in the event engine. The event engine exposes an interface to receive notifications from the external systems. It is important to note that *external systems* are external to the framework but within the trust domain of the service provider and must communicate to the obligation framework. Since the obligation framework does not contain any components for access control etc, it is never exposed directly to the entities outside the trust domain where it is deployed.

External systems can send and receive messages either through a generic interface in the format acceptable to the event engine. Alternatively, if the external entity cannot communicate directly with this interface then it can use customized plugins which translate incoming messages in the format acceptable to the framework internally.

4.3.5. Policy Processor

Figure 4.2 showed the activity sequence till the policy is generated and sent back to the client. Figure 4.5 shows the interplay after the client receives the policy.

The policy is received by the *policy processor* either through an external interface or via any of the *policy extractor plugins*, which retrieve obligation policy from a container message. The policy processor processes the policy, checks inconsistencies, and schedules deterministic triggers. The initial transaction interplay ends here and the system returns the system wide unique *Policy ID* to the caller. The caller in turn stores PII somewhere within the IT infrastructure of subject along with the policy reference. Both data and policy are stored separately but remain connected through cross-references as was shown in Figure 4.3 above.

4.3.6. Obligations Engine

Obligations engine receives the messages from the event engine and processes them accordingly. It fetches the obligations rules from the policy repository, checks for obligation applicability by evaluating conditional statements, finds the respective action plug-in and executes the action. After the execution of actions, the obligation engine changes the state of obligation rule and fires the outward events. If the action is executed successfully before the deadline the rule is fulfilled otherwise violated. Being the central processing

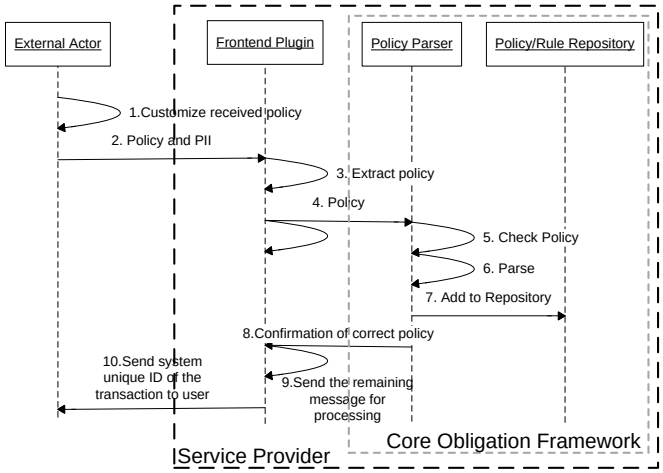

Figure 4.5.: Policy Processing Interplay. This sequence starts after the client receives a valid policy as shown in Figure 4.2

component, it has the most of the processing load. Because of the slow interaction with the external systems during action execution, the framework performance may be degraded. Thus, we can have multiple instances of the engine running in parallel for load balancing purposes. Condition evaluator component inside obligation engine is used to evaluate conditions attached to obligation rule. Based on the result of the condition, the engine decides whether to execute obligation action or not.

4.3.7. Action Plugins

Execution of obligation actions is provided through an additional set of plugins extending the obligation engine. We propose two layered plug-in mechanism in the obligation engine component. The upper layer contains the plugins for specific actions e.g. *delete*, *notify* and the lower plug-in layer contains the implementations for different external systems supporting a set of actions. For instance, delete operation can operate on files or on data in a relational database. Notification to user could be sent via e-mail, fax, or postal mail. Each obligation policy rule in our language specifies a single action with a system-wide unique scope and name, which is used to select appropriate plug-in. To ensure integrity, the action parameters must satisfy the required parameters for only one lower layer plug-in.

Each rule in a policy contains an *Action* with parameters. Each of these actions must match to an available *Action plug-in* within the system. The parameters listed within the policy must also match to the parameters required by the actions plug-in.

During the processing, *action invoker* decides which action to invoke based on the *(scope, name)* tuple received from the rule. Once the action is decided, the parameter values received with the rule decide which lower layer implementation plug-in to call. Finally, the action is invoked and the further processing is being done based on the

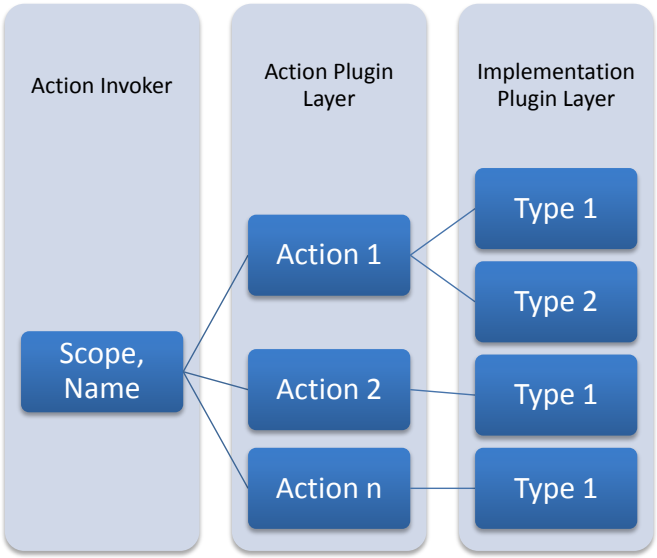

Figure 4.6.: Action Invoke Mechanism

result.

We kept our language independent of schema extensions so the new vocabulary required for domain specific obligations is mainly added by implementing the corresponding plugins each having unique scope and name which are then used within the policy. See Appendix A for the detailed language schema and some sample policies.

4.3.8. Policy Repository

This is the central repository which stores the obligation policies attached to some piece of data/resource. The storage and retrieval is provided through set of policy repository plugins e.g. SQL Server, Volatile Memory etc. The active repository is set through configuration settings and system at runtime dynamically chooses the active repository.

This section covered briefly the role of each component in the architecture. Next we present the internal working of the system.

4.4. Internal Working of the System

In Chapter 3, we presented two forms of enforcement namely *proactive* and *preventive*. It is important to note that both forms of enforcement could involve either synchronous or asynchronous operations. This is the design choice and is up to the action plug-in designer how to implement the action plug-in. Some actions are inherently asynchronous like *Service X commits to ask user U before using his data* where asking would be an asynchronous operation and reply is not expected immediately. Figure 4.7 shows the sequence diagram for synchronous operation.

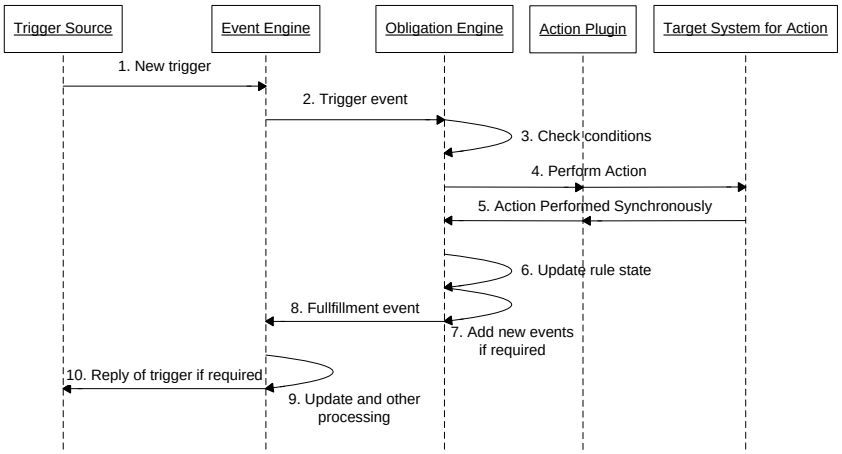

Figure 4.7.: Synchronous operation for obligation enforcement (Proactive or Preventive)

The process of enforcement is listed below.

1. The event engine receives a new message from a trigger source. Types of trigger sources and possible forms of messages are discussed in subsection 4.4.1 and 4.4.2 respectively.

2. The event engine then forwards the message to the obligation engine when it is free to process new load.

3. The obligation engine fetches the targeted policy and rule based on the information received with the trigger.

4. The obligation Engine check conditions within the rule to evaluate whether the rule is applicable or not.

5. If the rule is applicable then the obligation action within the rule is invoked.

6. In case of a synchronous operation, the engine waits till the call is returned otherwise it receives new load for processing.

Based on the result of the operation, the rule state is updated and the result is sent back to the event engine. The event engine forward the result to the trigger source if required otherwise the operation ends here.

Figure 4.8 shows an enforcement with asynchronous operation. The only difference with the synchronous operation is in Step [6]. The operation is completed asynchronously and the result is sent to the event engine by the target system. The integrity of the operation is ensured through unique ids assigned to each trigger by the event engine.

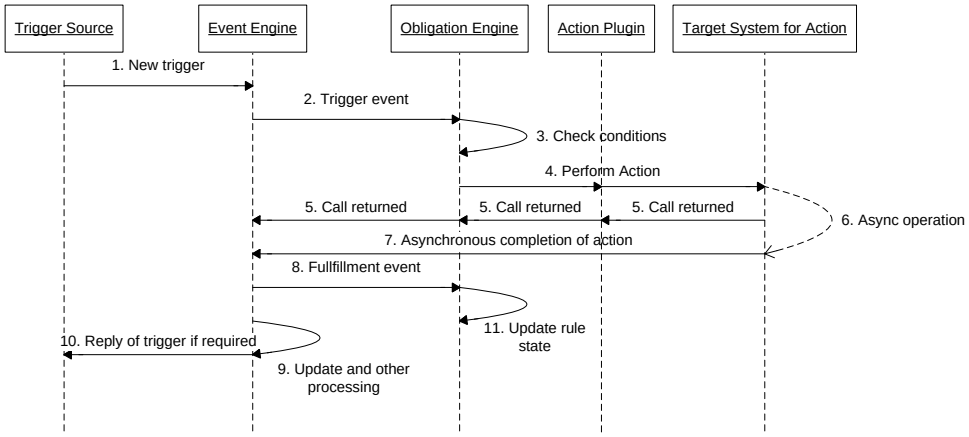

Figure 4.8.: Asynchronous operation for obligation enforcement (Proactive or Preventive)

4.4.1. Trigger Sources

Figure 4.7 and 4.8 show *trigger source* actors. Broadly, we consider two types of trigger sources including

- *Internal Trigger sources*
  - A timer/scheduler generate time based triggers.
  - The obligation engine itself may generate an event because of fulfillment and/or violation of an obligation which is routed back into the engine.
- *External trigger sources* which are a set of entities covering all the external systems interacting with the framework. These only include entities within the subject's trust domain and use framework services to support subject's operations. Users/End Users are not included nor the event engine is exposed to them.
  - DB Engines: SQL Server, Oracle etc sending an event on deletion, modification or addition of data.
  - Sensors: Sending measured data.
  - Web Services/Workflows
  - Humans: May generate an event on completion of some manual work.

4.4.2. Message Types

The event engine could expect the following forms of messages from the trigger sources mentioned in the previous section. These are also implemented through plug-in mechanism so more forms of messages can be added easily. Each type of message has a corresponding processing plugin which processes it accordingly.

- Scheduler Trigger Message: This is the most basic form of message which is a trigger and is sent from scheduler towards obligation engine. These are deterministic triggers.
- External Trigger Message: Any message which is of trigger type message and is received from external source falls in this category. These are non-deterministic triggers and are also directed towards the obligation engine.
- External Trigger Message with Reply: These are the same as external trigger messages but additionally they also have fields for the reply. The corresponding processor sends the reply for these messages too. Preventive obligations always expect a reply and are implemented with these types of messages.
- Schedule Message: These are sent from external sources and are directed towards the scheduler to schedule a new trigger.
- Reply Message: Received from external sources as a reply of an asynchronous operation. It must accompany unique event id which was sent to the external system with the original message.

Beside these, there are other messages which are used between the components of the framework for internal processing of the system. In our design, *Message* is the base entity which is exchanged between internal components and with external entities. Figure 4.9 shows the format of the generic message. A generic message could embed a trigger within it or any other form of information. If the message is of trigger type then it is also important to know the type of the trigger embedded within the message. The next section presents the types of triggers we are covering in our design.

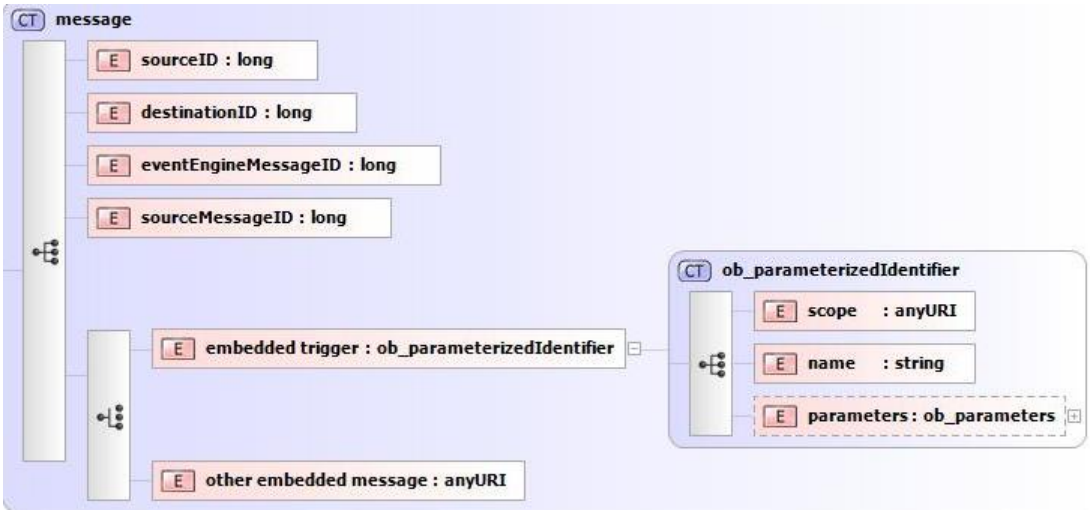

Figure 4.9.: Base message format received by event engine

4.4.3. Forms of Trigger

We have two broad classes of triggers based on the type of trigger source namely *Internal Triggers* and *External Triggers*. Internal triggers are generated from within the framework and are rather simple.

The more complex class of triggers are external triggers which are generated from varying external sources, for various reasons, bring variety of information in the form of parameters and last but not the least is that they could expect a reply from the framework which could itself be different for different trigger types. We handled the problem of external sources by providing plugins in the event engine. What is left is to further classify the triggers and their processing mechanism once they are inside the system. From the existing literature and our research on different scenarios, we identified three reasons to receive an external trigger.

- *For-Action*: These are the basic form of triggers which are received by the event engine to execute some obligation.
- *Pre-Action*: The trigger source wants to execute an arbitrary action and before execution it wants to get permission from the obligation framework to do the action. It may also want the framework to fulfill required obligations before that particular action.
- *Notification/Post-Action*: Alternatively, the source has already executed some arbitrary action and wants to inform framework about it.

These are the reasons because of which a trigger is fired by any external source. We now present some examples and identify the triggering mechanism in them.

Listing 31 For-Action Trigger

1: Subject X commits to delete U's data at $< d >$.
   ▷ Scheduler will raise the event at date $< d >$ and the data will be deleted.
2: Subject X commits to delete U's data when U requests for deletion.
   ▷ U's request is a trigger which will trigger deletion.

Pre-action trigger on the other hand is used to fulfill an obligation before performing some action in which cases the result of pre-obligation condition must be sent back to the source of trigger synchronously or asynchronously. We witnessed that all the preventive obligations are triggered through pre-action triggers as the result of prevention aid the decision making of the external trigger sources.

Notification trigger or post-action trigger is sent by the external system to the deployed obligation framework after performing some arbitrary action. External system does not expect a reply of the trigger. These are mainly generated for the framework to execute obligations which may be connected to those actions committed by the external entity.

Further classification of different types of triggers is presented in Appendix B which answers the question how we handle complex parameter combinations within triggers.

Listing 32 Pre-Action Trigger

1: Subject X commits never sharing U's data with anyone.
▷ X's access control infrastructure will ask Obligation framework *before* sharing U's data with anyone. This is Preventive obligation and is enforced synchronously.
2: Subject X commits to get permission from the user before deleting his data.
▷ X's access control infrastructure will ask Obligation framework *before* deleting U's data. This is preventive obligation and is enforced asynchronously because the framework will send email and deletion is done when the reply is received.

Listing 33 Post-Action Trigger

1: Subject X commits to notify U when U's data is shared.
▷ X's access control infrastructure will tell Obligation framework *after* sharing U's data. This is proactive obligation which is triggered by post-action trigger and is enforced synchronously.

This concludes the discussion on the architecture of the framework. We now present a couple of end-to-end scenarios which are prototyped with our system in the next section.

4.5. Demo Scenarios

Two simple scenarios of enforcement are presented in this section. The first example is of proactive obligation and the second presents a preventive enforcement.The detailed XML representation of the examples presented here are given in Appendix A.

4.5.1. Demo Scenario 1

Let the obligation policy have two rules as presented in Listing 34.

Listing 34 Demo Scenario Policy 1

Service X commits to Delete U's data at $< d >$
Service X commits to Notify(Email=XYZ) when U's data is deleted.

1. The initial interplay occurs and the user data and policy is stored on the service provider side.
2. The scheduler triggers the first obligation of the policy at scheduled time *d*.
3. The framework performs the deletion through action plugins on the PII repository.
4. An event is generated internally on the fulfillment of the first rule which is routed back to framework as trigger.
5. The second rule is triggered and notification is sent to the client.

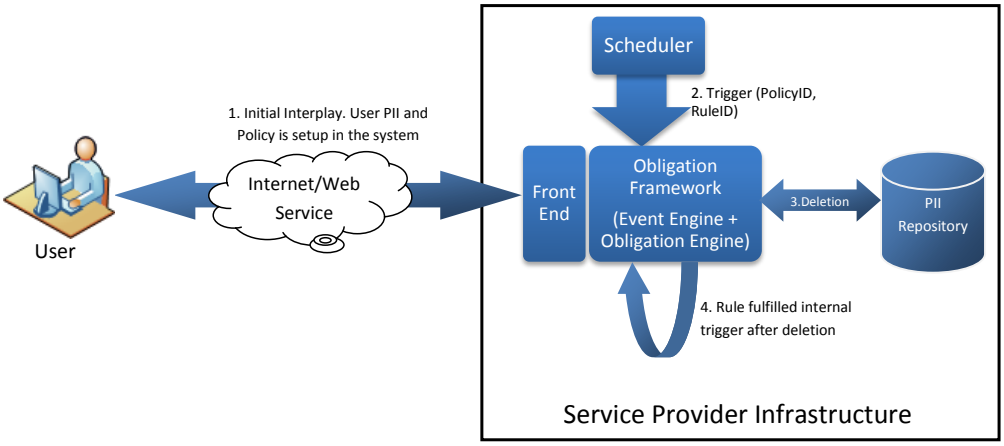

Figure 4.10.: Demo Scenario: Deletion and Notification Obligations cascaded together.

4.5.2. Demo Scenario 2

The corresponding policy is presented in Listing 35.

Listing 35 Demo Scenario Policy 2

Service X commits to Not to Delete U's data.

1. The initial interplay occurs and the user data and policy is stored on the service provider side.
2. The internal operator attempts to delete PII from the repository directly or through some other operator interface.
3. The repository sends Pre-Action Delete trigger to the framework along with additional information like PII ID etc.
4. The framework fetches the policy and finds the rule registered for the trigger. If the rule is found then conditions for prevention are checked.
5. A reply is sent back to the repository which behaves as per the reply of the framework.

In preventive cases like above it is beyond the control of the framework to force external entities to behave as per its decisions. Thus, we assume that all the systems are in the same trust domain for the above scenario to succeed. Let the *repository* deletes without the framework's consent then such cases can be identified during system audits.

Until now we discussed the architecture and internal working of the system. What if we have multiple frameworks deployed?, whether they complement each other or not?.

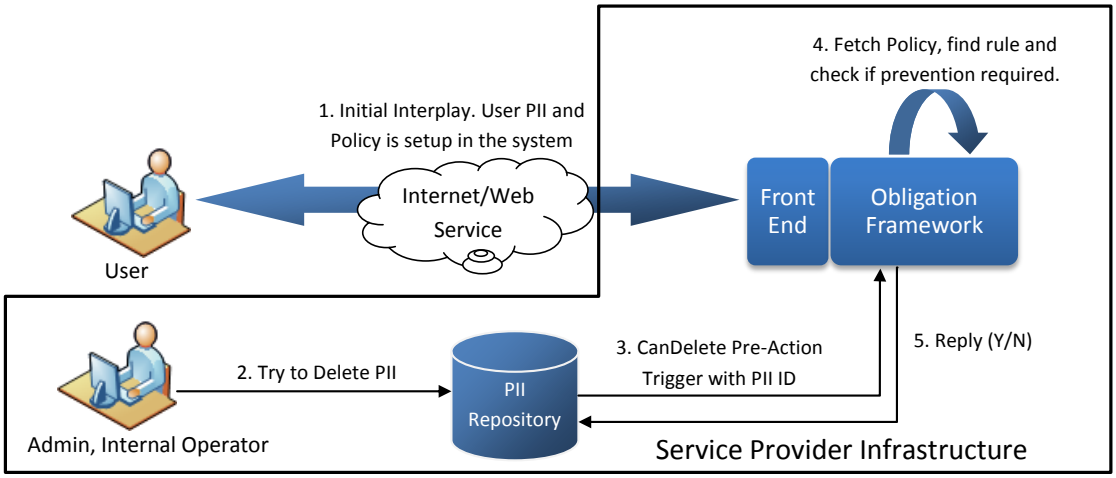

Figure 4.11.: Demo Scenario: Prevention of Deletion.

Thus to understand the environment surrounding the system is as important as understanding the system itself. Before concluding this chapter, we would like to present a brief discussion on different deployment scenarios of the framework.

4.6. Deployments

The framework could be deployed in multiple ways. In the subsequent sections we discuss different deployment scenarios for the obligation framework. The minimal set of components required for the deployment of obligation framework in any environment are

- *Obligation engine* which is the central component responsible for the coordination with other components and execution of actions.
- *Scheduler* which is required to generate timed triggers and future event set.
- *Event engine* is the components which is exposed to the external environment and is used to receive events from all the components and to distribute them.
- *Policy processor* to parse and process the newly received policy.
- Appropriated plugins to support actions, triggers, and external systems.

We consider three major environments where the obligation framework could be deployed which are the topics of the next three subsections.

4.6.1. Desktop framework deployment

This is the simplest deployment with minimal set of components. A very common scenario is when a *user U* download a resource from a server on its desktop machine and in such a case the desktop obligation framework would be responsible to fulfill obligations attached with the downloaded resource. Additionally, user side obligations could be enforced by deploying frameworks on the individual user machines.

For example, a medical doctor may download patient's history (along with obligation policy) from the hospital repository on his machine for review. In this case, the local obligation framework ensures that the data is being used as per the policy received with the data.

4.6.2. Server Deployments

In server side deployment we could have many possible, from simple to complex, scenarios where the obligation framework could be deployed. We start with the simplest deployment scenario. The dimensions we have identified in the deployment are as follows

- Obligation framework deployments indicating how many frameworks are deployed within an infrastructure.
- The number of services exposed by the organization.
- Vertical policy structure which specifies the structure of policies within an organization.

For modeling purposes, we define subject boundary within which the User's/Customer's data will be stored at a single place without duplication.

The obligation framework has a policy repository (containing received sticky policies). If there are multiple instances of framework running based on a single policy repository then we consider it as a single obligation framework. The multiple instances may be used for load balancing purposes but we don't consider them as multiple deployments.

Single framework

In this scenario we have a single organization wide obligation framework deployment. However, the organization can have multiple exposed services each having its own policy or can have a single policy shared across the organization.

When we have a complex policy structure, like in Figure 4.12 (B), then the obligation framework will also be responsible to:

- Check the policy hierarchy
- Merge the policies at different levels in the hierarchy.
- Resolve conflicts.
- Generate final unified policy against the request

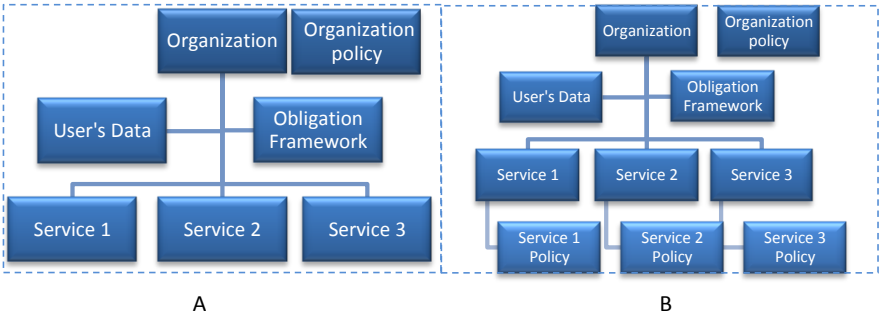

Figure 4.12.: A:Deployment with Single framework and having single organization wide policy B:Deployment with Single framework but each service has its own policy alongside an organization wide policy.

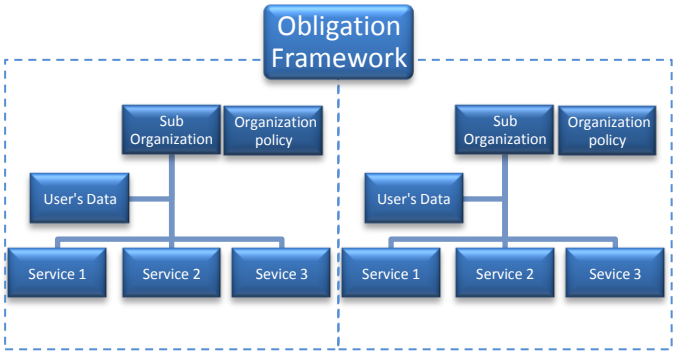

Figure 4.13.: Single framework deployed within an organization having multiple data repositories (each modeled as a single organization)

In Figure 4.13 , we show a scenario where the organization has multiple repositories of User data, with some duplicated data too. Each repository is modeled as a sub-organization but the obligation framework is single and is shared across. The two organizations are under the same trust domain so additional security checks are not considered. It is inherent in our design that a single obligation policy will always point to a single piece of data (data could be very large too in size).

Multi framework

In case of multiple obligation frameworks deployed the problem of integrity may arise. If the two framework references the same user data then it would be difficult to ensure integrity and to avoid incorrect obligation violations. In such cases it is essential to ensure that the frameworks must always reference separate data sets. We propose the

idea of having Policy ID mapping to the obligation frameworks in such cases. While deploying the second framework with its own policy repository. The central framework orchestrator must assign the chunk of Policy IDs to the specific obligation framework. We always assume that only one obligation framework is in charge of a specific piece of data.

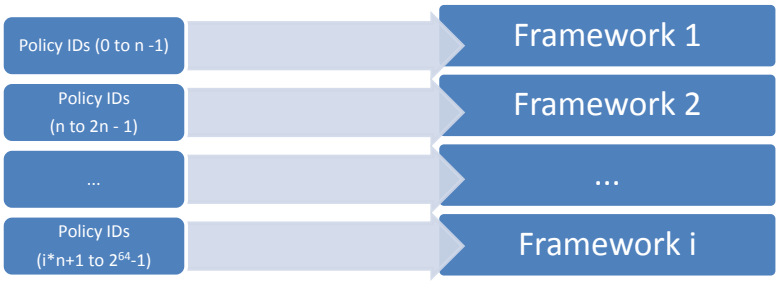

Figure 4.14.: Policy mapping with multiple frameworks deployed

In case, when a new framework is added with data already present for its assigned chunk in the primary repository then we would need to do migration. The policy IDs space will be distributed among the framework as shown in Figure 4.14. To utilize the full policy IDs space for each deployed framework, we can take *Policy IDs* as a combination of *Framework ID + Unique ID*. However from the processing point of view, we then have two integer comparisons to deduce unique system wide Policy ID.

4.6.3. Cloud Deployments

Cloud computing is relatively a new style of computing which incorporates the concepts of *infrastructure as a service (IaaS)*, *platform as a service(PaaS)* and *software as a service(SaaS)*. In cloud computing dynamically scalable resources are provided as a service over the Internet. Users need not have knowledge and expertise over the technology infrastructure in the *cloud* that supports them. In cloud computing both software and data are stored on the third party cloud computing service provider. The subject organization rent the infrastructure on the cloud to support its business. An introduction about the cloud computing can be found in [31].

We perceive that our obligation management system can be deployed on any cloud and is presented as an obligation management service. Customer organizations utilize the obligation management and enforcement services without acquiring the cost of maintenance of software, hardware and human resources. What remains is to correlate our design with the requirements presented in Section 1.2 which is what we present in the next section.

4.7. Correlation with Design Requirements

We presented a set of design requirements earlier. After the discussion on the architecture we would like to relate the requirements with the presented framework. This correlation is being presented in the following list.

- *Independence from policy language*, *data repositories* and *communication protocols* is covered by extraction, translation and repository integration plugins. We have currently implemented repository plugins for volatile memory and SQL Server 2008. The SQL Server plugin highly depends on the DB schema at the back.
- *Support for common,domain specific* and *preventive* obligations is provided by having action plug-in layer. We implemented action plugins for delete, notify and prevent actions.
- *Support for abstraction of actions* is achieved by secondary plug-in layer under the first action plug-in layer. We have provided plugin for SQL Server, RAM and Email notification at this layer.
- *Support for abstraction of triggers* is achieved by having event engine plugins for integration. Currently we have used only the standard event engine interface for receiving external events. For classification of triggers see Appendix B.
- *Support for distributed deployment* is provided by allowing multiple policies to be merged in a single policy. We do allow multiple subjects in a single policy but at the implementation level this feature is missing. Providing placeholder for multiple subject only does not completely address the problem. Downstream integration with the secondary data consumers is one of the open research problems which is out of scope of this thesis.

5. Evaluation

This chapter covers the results of testing and evaluation of our design and prototype implementation. Alongside the results, we compared our system with the sole known obligation management system by HP presented in [7]. We first present this comparison in Section 5.1 which is followed by the evaluation, results of testing and proposed optimizations in Section 5.2.

5.1. Comparison with HP's Obligation Management System

The HP's obligation work was done under the EU FP6 PRIME project while our work is based on the EU FP7 Primelife project which is the continuation of PRIME so the two research efforts are indirectly related.

The overall idea of representing obligation in the form of separate policies is similar in both and some other works related to obligations where the proposition is that existing access control policy languages cannot express all the features of an obligation. Thus, usage control/obligation requirements must be represented by a separate policy language. In the next section we will focus on the key differences in the approach, scenario, representation and enforcement architecture of the two research efforts.

5.1.1. Approach

We took the approach to design the system to achieve more flexibility in terms of obligation representation and user requirements however this approach also brings more resource and processing requirements. HP's approach is based on more of a tradeoff where flexibility/expressiveness is compromised for the sake of lesser resource constraints.

We propose to store policy templates in raw state. When the request for a new policy is received then based on the context a new policy is generated. The generated policy is still customizable by the data owner and returned to the system with data attached to it as depicted in the Figure 5.1. This approach gives the system provider to have multiple varying policies for different users, user groups etc. Through this scheme, we can express more complex scenarios for example

- If we want to have different policies for different types of users. For instance more trusted users may be given leverage as compared to general customers.
- If we need to represent a policy hierarchy where a single organization is having more than one service and each service has its own policy and we also have an organization wide policy. And each obligation policy is the combination of the organization wide policy and the respective service policy rules.

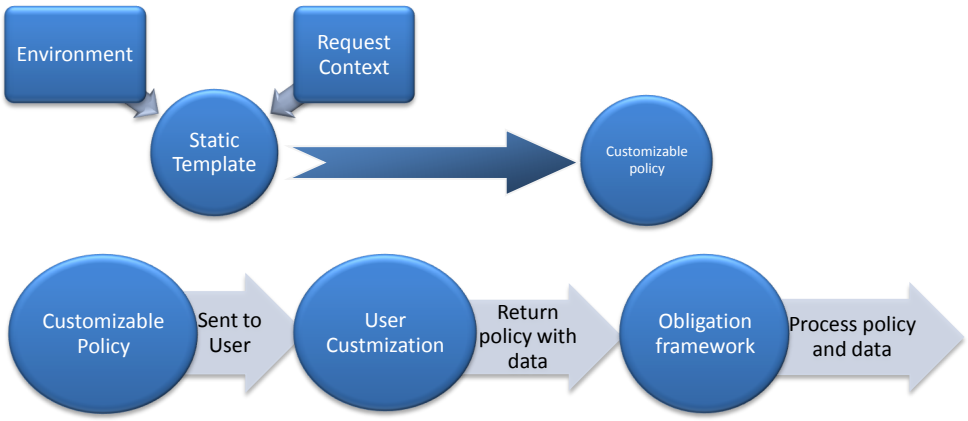

Figure 5.1.: Our Approach

The above scenarios, to the best of our understanding, were not considered in the previous research work. HP does not send the policy on the wire but contrary to that they consume user preferences on the server side. They store policy templates which are when combined with stored user preferences acquired from a specific user constitutes an obligation policy for the user. Their approach would be less resource thirsty as in our approach we need to store and process large amount of obligation policies. This is one of the future directions of work to merge many policies having same semantics attached to different data for optimization purposes.

5.1.2. Scenario

We take the scenario where multiple subjects do exist, both in horizontal plane and vertical hierarchies, each having its own policy. So we introduced subject in obligation rule tuple which is not present in HPs work. Their policy can be applied to a single subject, which may be the system, but cannot be a combination of multiple service providers. The concept that *Who will fulfill the obligation* is missing from the HPs representation of obligations and only system obligations are possible.

In Figure 5.2 each service provider may have its own obligation policy and framework. The user should get the unified view and only interacts with S_1. The unified customizable policy received by the data owner may be a combination of obligation policies from these multiple service providers. Our work/approach is being taken with future extensions in mind where we can introduce constructs in the language for third party obligations and collective obligations. Currently the detailed constructs are not provided. The *Ruleset* construct is being provided to enable definition of multiple subjects within a single policy.

We use the sticky policy paradigm where the policy travels to the customer before getting customized. HP's work focuses on the idea of having obligation policy referencing user preferences and these preferences are travelling on the wire and not the policy. We

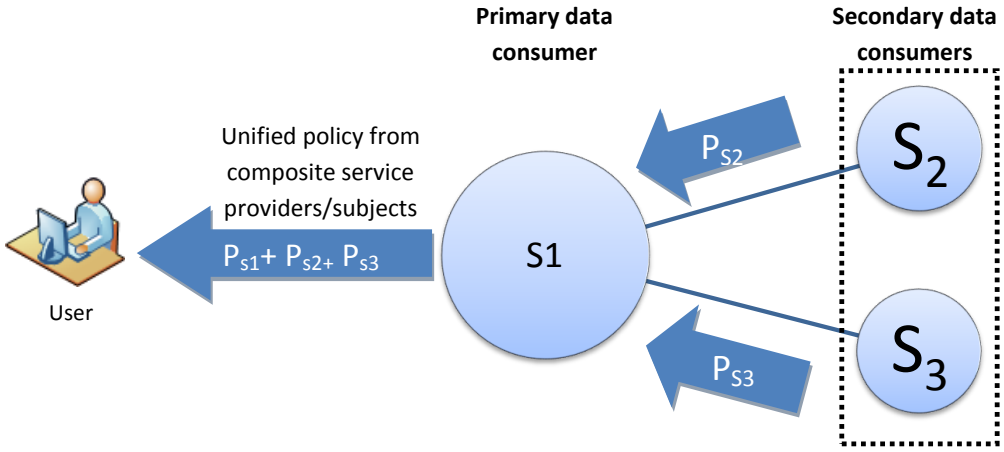

Figure 5.2.: Distributed Composite Service Scenario

provide more flexible user customization including different preferences for different data from the same data owner.

In HP's case, they store the preferences from the user and the same are used for all the data received from the user so their approach is more user oriented and our's is more data oriented and goes down to more granular level.

5.1.3. Representation

The key differences in the representation of obligations are listed below.

- HP represents the obligation policy as a single rule having multiple actions and events. We take policy as a set of rules with each having a single action, which may be a composition of many sub-actions. We take this because we do support negative actions which may be contradicting to each other and it would be complex to reason on consistency of policies having multiple rules each having multiple actions.

- Provision of negative (preventive) obligations in our language which was lacking in the previous work.

- HP provides the idea of having violation actions which is a single level feedback mechanism and only used when the rule is violated. On the other hand we have chosen to have cascaded obligation rules which can be done at any level. An outward event from one obligation rule can be an inward trigger for the other. Outward event can be for fulfillment, violation etc. We have done reasoning on the interrelation of rules for contradiction and safety which was missing in the previous work.

- We have introduced condition statement at the rule level to control its activation using generic conditions. HP allowed conditions at the action level but allows multiple actions within a rule.

- We used the idea of untyped parameters for events and actions which makes the XML representation independent of parameter names. HP XML representation includes the action/event specific language constructs. To introduce a new action they need to extend the XML schema.

5.1.4. Enforcement Architecture

Our architecture is based on dual plug-in layer as mentioned in Chapter 4. For HPs design is not clear how they handle the same action performed on different external systems. Currently, they only allow either RDBMS specified as the target systems. So apparently, if we want to do the same action on a file then it is not possible without modifying the language or extending the language to include such constructs.

It is concluded from the comparison that our system incorporates features which were not supported in the previous system. However the higher flexibility may bring some scalability issues for very large consumer base. System optimization directed towards the more intelligent policy management would be required without compromising the flexibility. The current design allows policies to be attached to resources at the most granular level.

For example if an end user ask for service from the provider multiple times and provide PII to the provider with each transaction. In such a case, we do allow the system to attach separate instances of an obligation policy to each PII (the PII may be same). In real situations, we may have policies without much variation in all the dimensions. Thus, we could develop techniques to detect that the policies attached to two copies of data are same and should be treated as a single policy applied to both the copies. That is moving from the most granular level towards grouping entities in different dimensions and applying policies on the groups for scalability. To judge the performance of the current prototype we conducted validation and scalability tests. The results are presented in the next section.

5.2. System Evaluation

We divided this section into two parts. First, we present the evaluation technique and results and in the second section we present optimizations to the system and design from the testing results.

5.2.1. System Testing

We primarily focused on two aspects of testing that is validation and scalability. The validation testing covers the testing of different components of the system to check their validity and to establish the variation in the intended system and the actual system

prototype. Scalability testing consisted of running system under different kinds of load to see the performance.

Figure 5.3 shows the modified architecture as was also presented in implementation chapter 4. We divided the architecture into two parts. The upper section, with light background, represents the components involved in the initial user interplay when a new data and policy is received and setup. While the internal runtime components involved in enforcement are dark background.

The whole system is designed with clear interfaces and components which communicate through these interfaces. We have marked the testing points too in the Figure 5.3.

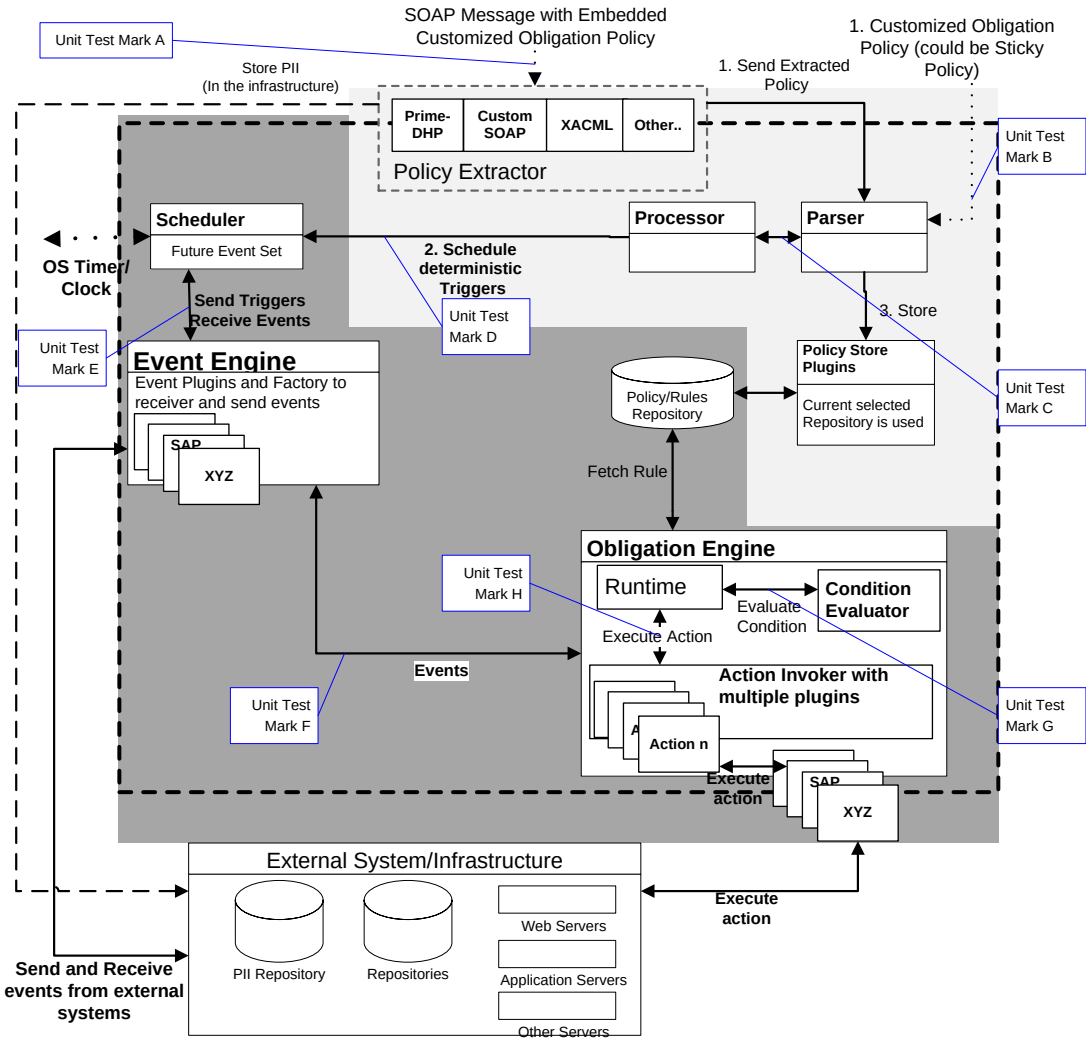

Figure 5.3.: Modified Architecture

For overall testing the system is assumed to have two modes of operations. It is evident from the Figure 5.3 that there are repositories supporting the core components. The first is the policy repository which holds the obligation policies and the second is PII repository which holds the data on which the policies are applied. Both repositories are accessed through plug-in based mechanism and currently we have plugins for Volatile Memory (RAM) and SQL Server DBMS. Based on the type of repository used at the back, we could have the following modes of operation.

- Base Mode: This is the mode when system stores both policies and PIIs in Volatile Memory or RAM. This is called base mode because the access to memory for reading and writing information is fastest. All the performance metrics are first evaluated in this mode and then we compare the base performance with the other modes.
- Operation Mode 1: This is the mode when PII repository is replaced with RDBMS (SQL Server). Though PII repository is outside the system but we wanted to compare the difference in performance when repository is moved to DBMS.

Additional operation modes are established when policy store, event engine and scheduler are also supported by RDBMS.

Validation

The validation of the system is being done by testing each component in isolation with both valid and invalid inputs. Invalid inputs are tested to check the robustness of the system and to provide graceful exit.

The second step was to check the system by cascading different components to see the overall results and to ensure that the systems are performing as was intended and output of one component is acceptable to the next.

We provide a set of unit tests for different components which validate the system. These are normally run after each code revision, addition and bug fixes for validation. The unit tests start the system in Base Mode. *Base Mode* is chosen as modes of operation only impact performance and not validation. Validation unit tests are also independent of the system power and capacity. The test points as mentioned in Figure 5.3 are described below.

A This is the point which is exposed to the client/end user. Thus, unit tests that test the system from this point actually simulate client experience. We currently have one plug-in which receives client message in a custom format and separates the policy and PII part. The overall processing cost in terms of time if measured at Mark A is given in Equation 5.1.

$$t_A = t_{PII\ processing} + t_{policy\ processing} \tag{5.1}$$

$$\begin{aligned} &Where \\ t_{PII\ processing} &= Processing\ PII.\ This\ is\ dependent\ on\ the\ repository\ used. \end{aligned}$$

$$t_{policy\ processing} = \begin{array}{l} For\ RAM\ \approx\ 0 \\ This\ is\ the\ complete\ delay\ taken\ by\ policy\ parser. \\ Elaborated\ in\ next\ item. \end{array}$$

If we want to measure the processing cost at the client side then the equation will have one additional component $t_{network\ overhead}$ which represent the overhead induced because of network delays both directions and web service call overhead like additional serialization/deserialization.

$$t_{client} = t_{network\ overhead} + t_A \tag{5.2}$$

B At this point, we test the parser through the *IParsePolicy* interface. The unit test generates a dummy policy and test the XML parsing. We can also test the complete $t_{policy\ processing}$ cycle from this point including parsing, saving and scheduling of trigger within the policy.

$$t_{policy\ processing} = t_{policy\ deserialize} + t_{policy\ saving} + t_{policy\ processing} \tag{5.3}$$

$$\begin{array}{rcl} Where & & \\ t_{policy\ deserialize} & = & Deserialization\ of\ policy\ \approx 0 \\ t_{policy\ saving} & = & Saving\ PII.\ This\ is\ dependent\ on\ the\ repository\ used.\ For\ RAM\ \approx\ 0 \\ t_{policy\ processing} & = & This\ is\ the\ complete\ processing\ cost\ of\ policy\ processor.\ Elaborated\ in\ next\ item. \end{array}$$

C At this point, we test the policy processing timing using *IProcessPolicy* interface. It is being input by a parsed policy object. Currently it only does one task which is to check the policy object and schedule time based triggers defined in it with policy ID and rule ID. Processing cost of this component $t_{policy\ processor}$ found $\approx$ 0. This is an evolving component and major delay could occur when inserting new events in the scheduler.

D Scheduler is tested using *IScheduleEvent* interface at this point. Currently delays are negligible. At this point the client interplay ends and the call returns.

E At this point, we check the different types of triggers sent to event engine and see whether they are acceptable or not and how the engine behaves in case of invalid triggers.

F The obligation engine is fed with messages to test its behavior and processing.

G At this point, we test condition evaluation components using *IConditionEvaluator* interface. We feed the components with different dummy condition objects. The evaluated reply is compared with the expected one to analyze the behavior.

H This point though marked on the architecture but in reality each action plug-in is tested separately before being integrated with the core engine. Different plugins may have different processing cost based on their design, intended purpose and external system integration. For example deletion plug-in performance may be different than notification plug-in. In turn deletion on RDBMS is more time consuming than a deletion on RAM.

Based on the discussion above the overall time taken to process a single trigger received at event engine is as follows.

$$t_{trigger\ processing} = t_{event\ engine} + t_{obligation\ engine} + t_{condition\ evaluation} + t_{action\ plugin} \tag{5.4}$$

$$\begin{aligned}
&Where \\
t_{event\ engine} &= Event\ engine\ holds\ a\ queue\ of\ messages\ received\ from\ different\ sources. \\
&\quad Depending\ on\ the\ load\ it\ varies \\
t_{obligation\ engine} &= Obligation\ engine\ need\ to\ fetch\ the\ policy\ and\ rule\ from\ repository. \\
&\quad For\ base\ mode\ this\ is\ negligible \approx\ 10^{-6} \\
t_{condition\ evaluation} &= For\ evaluating\ conditions\ within\ the\ rule. \\
t_{action\ plug-in} &= Action\ plugin\ processing\ time.\ This\ must\ be\ evaluated \\
&\quad for\ each\ plug-in\ separately.
\end{aligned}$$

For external triggers there will be an additional delay of deserializing trigger parameters and we represent it is $t_{external\ trigger\ overhead}$. Validation overall resulted in a list of bugs which we fixed in multiple iterations. Next we present the scalability testing results along with the problems we identified.

Scalability

This was the major activity where we load test the system to identify potential scalability problems in real time scenarios. First, we ran test in the base mode to establish a bench mark result and then compared them with results from operation mode 1. Scalability testing is not machine independent and thus the results could be different for different machines. Thus we have presented results in relative terms rather using absolute numbers. We conducted these tests on a machine with the following configuration.

CPU: Intel Core2 Duo 3.16 GHz
RAM: 4GB
OS: Windows Vista
DBMS: SQL Server 2008

The testing is further divided into two parts. First, we estimated t_{client}, t_A and $t_{policy\ processing}$ as presented in Equation 5.2,5.1 and 5.3 respectively. This constitutes the initial transaction interplay processing cost. The second part, dark background, is to estimate $t_{trigger\ processing}$ of Equation 5.4.

We estimate processing cost of each component to find bottlenecks and inefficient code snippets. This is also important to note that the scalability results are highly dependent on the complexity and size of policies. To have the result free from the impact of policy sizes we conducted all the tests with a simple policy having one rule which commits deletion of the user data at a pre-defined time instant.

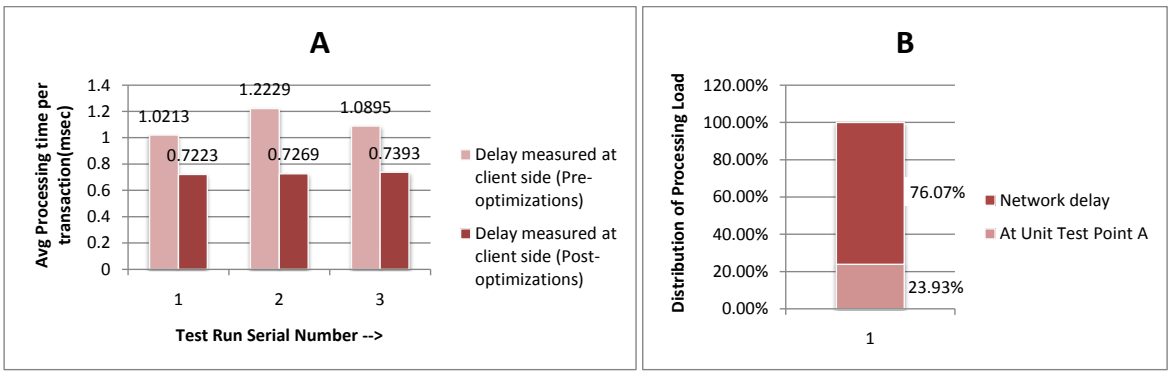

Figure 5.4.: A) Presents the average transaction processing time measured at Client side B) Distribution of processing time between framework and additional WS-Call overhead represented in percentage

Initial Interplay-Base Mode Figure 5.4 Part A present average per transaction processing time measured from the client side. The average is calculated from 1000 transactions fired in each set. The first test was conducted with the initial implementation which helped us to identify different bottlenecks and unoptimized code. Second set of tests were conducted after making certain optimizations in the framework which improved the performance by $\approx 33\%$.

Part B presents the distribution of the processing time between measurements at client side and Mark A on the framework side. These are post optimization distribution from test 2. $t_{policy\ processing}$ is approximately 25% of the total t_A time which is of the order of 200 μsec per transaction. Figure 5.5 presents the complete distribution of processing time for base mode operation.

Initial Interplay-Mode 1 with RDBMS PII Repository Figure 5.6 Part A presents average per transaction processing time measured from the client side for the system in *operation mode 1*. The difference is that the PII repository is moved to SQL Server. In this case, optimizations improved the performance by $\approx 93\%$.

The huge increase was because of the saving of PII in a temporary data structure. Previously, we were saving PII directly into the database in real time. The optimization was to induce a temporary data structure which holds the new PII for a short time

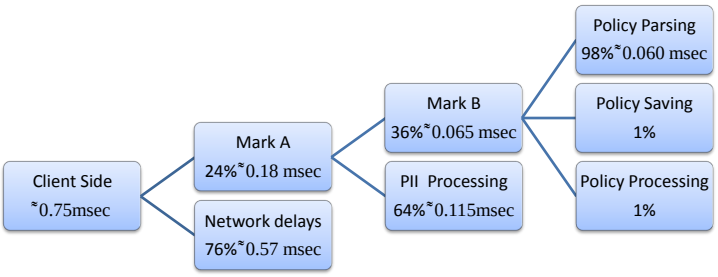

Figure 5.5.: Complete distribution of processing time in base mode.

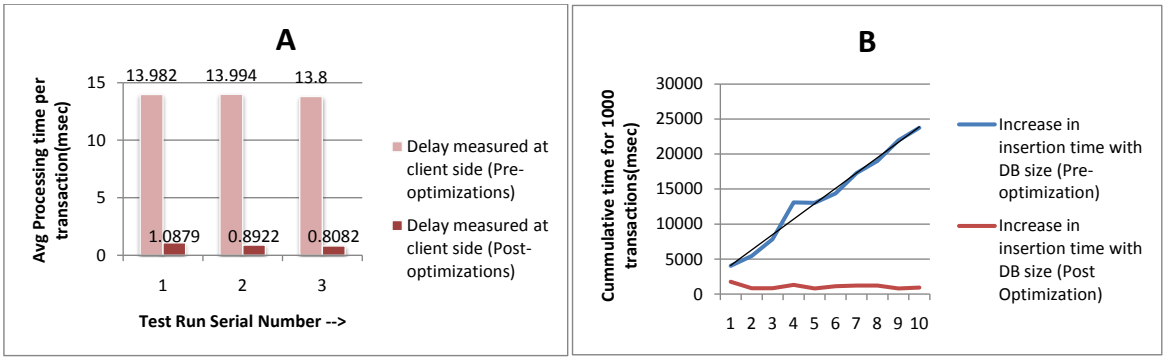

Figure 5.6.: A) Presents the average transaction processing time measured at Client side B) Presents increase in PII insertion processing time because of increase in DB.

and then the queries are run for a bulk insertion in secondary thread. Though the PII processing/extractor plugins are outside the scope of the system but we considered them to show the impact realized on the client side. Generally, the complexity of DB insertion is of the order of O(Log N) where N is the number of records in the respective table.

Part B presents the impact of DB size. The increasing line shows that the insertion time is proportional to size of DB. With the optimization we also solved this problem. The flat straight line at the bottom (Figure 5.6 Part [B] Post optimization) is realized when taking measurements on the main execution thread which is now free from the impact of DB insertions.

Figure 5.7 presents the complete distribution of processing time for operation mode 1.

Once the policies are into the system. The next phase is the actual enforcement. In this phase the load is proportional to the number of triggers expected per unit time. This trigger count per unit time in turn is proportional to the number of policies within the system. To evaluate this part, we scheduled triggers to be fired at the same time

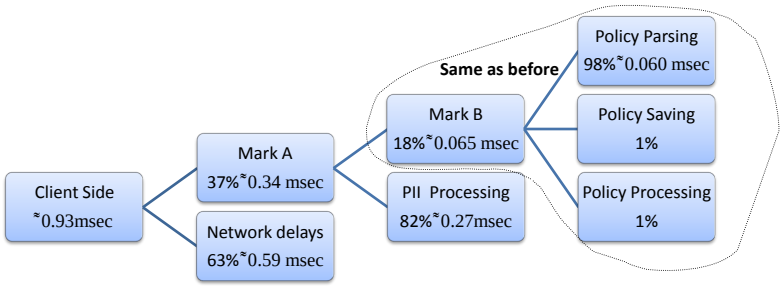

Figure 5.7.: Complete distribution of processing time in mode 1.

and calculated the processing time for them. This is important to note that the actual system would be having triggers processed along side with the new service requests. However, we divided the system into two parts and conducted tests separately to avoid results which are polluted by the impact of each other.

Internal Processing-Base mode We first scheduled triggers in different numbers from 10^3 to 10^5 and fired them at the same time. It is found that the processing time per trigger was of the order of 10 μsec. The waiting in the queue on event engine was dependent on the performance of the action plug-in at the end. In case of exception the performance degrades because of IO operations required for logging.

Internal Processing-Mode 1 with RDBMS PII Repository We performed the same tests with PII repository on the SQL server (Mode 1). The performance was severely reduced because of the deletion operation executed by the delete action plug-in as a synchronous operation on SQL Server. Most of the time is consumed in waiting for the reply from DBMS $\approx 10^{-3}$ sec. Because of the delay on the plug-in side, the waiting time in the queue increased and the overall per trigger processing too which found to be of the order of 10^{-3} sec. The exact processing time in turn depends on the size of DB.

The conclusion we draw is that the system itself does not induce unnecessary delays or overheads. The important requirement is to measure the performance of the action plug-in at the end of the operation which in turn depends on the performance of the external systems it is connected to.

The design can be considered as a classical queuing system. The trigger sources push new work units into the queue (event engine) and servicing engine (obligation engine with plugins) retrieve those jobs from the queue and process. If the rate of service provider is slower than the rate of incoming jobs then the queue will increase in size and ultimately crash. The rate of service provider in our case depends on the performance of action plugins which could block the thread in which they are executed.

Impact of policy size on performance As mentioned before, to keep the measured parameters free of impact of different policy sizes we perform testing with a policy having

one rule which is in turn triggered only once. The action used in the testing was *deletion* with two different repository types. The size of a policy impacts the performance in two ways (leaving the network overhead induced because of higher data transfer).

- The larger the policy size (means number of rules per policy) accompanying the data the more would be the processing cost for parsing, processing and saving the whole policy but processing cost per rule will not be impacted.
- Second would be the cost of processing more rules and in turn the number of times each rule is triggered.

We estimated processing costs per trigger and processing costs per policy each having one rule (implying that the estimator for processing cost per policy is equivalent to processing cost per rule as there was only one rule per policy). This was deliberately taken as these estimators themselves are independent of policy sizes.

The size of the policy impact the overall load on the system. For instance, let the system is capable of storing and processing 1000 rules. This capacity can be consumed if we have 1000 policies with 1 rule each or 250 policies with 4 rules each etc. To calculate the required capacity of the system hardware, storage and other infrastructure we would need to know the number of end users/policies to process and average policy size per rule.

Next we discuss some optimizations of the system which we identified during evaluation.

5.2.2. Optimizations

For optimal performance of the system during production operation we would need to have optimizations from three perspectives which include design, policy and administrative optimizations. Brief discussion on each is as follows.

Design Optimizations Design optimizations comprises of the optimizations made in the overall system design and the written code to make the system as efficient as possible. We already detailed some of the improvements made in the plugins to improve performance.

For the internal processing part, we plan to introduce multiple threads within the obligation engine. The current engine uses a secondary worker thread to post messages to corresponding action plug-in and wait for the reply. However as discussed in previous section, if the plug-in is blocked because of the synchronous operation then the whole engine is blocked too. We propose to introduce multiple threads (or a thread pool) to execute each action executed inside obligation engine.

Another important aspect which is out of scope of this thesis work is to introduce intelligence in the system. For example same trigger is to be fired for n obligation rules of the same type and at the same time. In such cases we can fire a single trigger that apply to n rules. The operation could be accomplished in bulk and using secondary threads which would be faster than deleting n time in the primary thread.

Optimized Policies Design though improve the performance but still cannot cover all aspects. The second important aspect is to write consistent and unambiguous policies. Unenforceable policies would generate exceptions and degrade performance.

Additionally policies could also help to improve the performance too. For example we have n policies in execution each having 1 trigger. Assume that all n triggers are scheduled to fire at the same time. To avoid this, we could write a set of static policies targeted towards different end user segments (like critical users, normal users etc). When these policies are in the system for execution, the corresponding triggers would be distributed in time.

Another proposition is to introduce additional constructs in the policies and intelligence in the engine which allows flexibility and load distribution during execution. For example, we write the policy rule to be triggered at 1200 PM with an additional tolerance level of ± 8 hours. Additionally, assume that all the triggers are scheduled to fire at 1200 PM. Thus, the engine should be able to distribute scheduled triggers using tolerance level information and the processing load it has.

Administration Last is the efficient administration of the system. For instance, if we turn on the tracing at highest level then the performance is degraded. Thus it is an administrative task to control such performance parameters like tracing level, timeouts, activating and deactivating plugins, backup and recovery etc.

We left out Mode 2, with policy repository on RDBMS, because of the problem of saving policies efficiently in the DB. If we store whole policy XML chunk then additional overhead of serialization and deserialization would be there along with the cost of accessing DB. The other extreme is to store the policy in a DB having multiple tables and policy is distributed across tables. However, we identified that this would also be costly to run multiple insertion queries in separate tables. The second option though allow multidimensional analysis of policies in the repository.

Thus we will go with the second option but refrain from storing policies directly into the RDBMS. Instead, we would propose to use the current volatile memory policy storage plug-in as a cache memory between framework and RDBMS. The synchronization between RDBMS and RAM data structures should be done in idle time and in secondary worker threads.

To analyze the cost of accessing DB vs. cost of serialization and to identify the most optimal DB schema to map onto our policy language XML schema is part of the future design enhancements to the current implementation.

We end this chapter with the note that the prototype allowed us to verify our language and architecture design and we have been able to enforce different obligations through the system. However, there are lot of enhancements still required and problems need to be addressed which surround the system. We now move to the conclusion of our work and present the future direction of work.

6. Conslusion and Future Work

We presented a general language for expressing obligations which can be integrated with or used in conjunction of today's access control and data handling policy languages. The thesis presented various aspects of the language so that our work can serve as a contribution to other research activities in this field. We started with the background of problem scenario then presented the reasoning on the requirements of an abstract yet expressive obligation language and presented requirements for design. We showed how our work relates to state of the art and positioned our work. Next we presented an abstract notion of an obligation language fulfilling those requirements. We described important design aspects and formal structure of the obligation language. The language offers basic actions, triggers and terms which are rich enough to cover a broad range of scenarios. In addition the language can be extended with domain specific actions, terms and events to adapt it to specific application domains.

The next major contribution of the work is to present a design of the framework to enforce obligations. This was something missing and we addressed this problem to fill the gap. We correlated our design aspects with the requirement of the design and gave reasoning on different components and features of the architecture. We verified our work with an implementation of an obligation framework which features both the requirements and the proposed language design. This allowed us to make practical comments on the implementation aspects and drawbacks. The current implementation provides a subset of the features discussed in the language and design.

We compared our system with one of the known closely related system and highlighted the new features and differences. We also conducted initial evaluation testing of the system to verify the feasibility of such a design and implementation in real time environments. Optimizations to the systems along with results from the testing has been presented.

The current structure of the language is able to express wide range of obligations and is extendible. The idea to have multiple subjects in a single policy makes it extendible in distributed scenarios with multiple subjects. The plugin based design aid to this extendibility of the design and expression of obligations.

Based on the results from the evaluation of current prototype of the system, we conclude that the design of the core enforcement platform itself is efficient but the performance is highly dependent on the design and efficiency of plugins inserted into the system. It would also be dependent on the integrated external systems with which the framework is integrated. For synchronous operations, unoptimized action/event plugins may degrade the performance of the overall engine. Furthermore, ill written policies could take the system to ambiguous states.

Retreival from and storage into RDBMS repositories took the most time because of

IO operation. It would be better to have advanced methods like having cache memories instead of directly retriving/storing information on demand. The object form of the policy to save in the repository impacts the performance as well.

The work accomplished in the thesis is part of a EU FP7 Primelife project and we addressed one problem of the user privacy which is to express and enforce obligations. This work is in its initial stage and certainly there is much room for further improvements. However, we beleive that the proposed architecture and language if taken forward could serve as an enforcement platform to cater business scenarios. From the overall perspective there are open research problems which were beyond the scope of the thesis work but could be addressed as part of an ongoing effort. Next we present some of these known open problems identified during the work and areas of further research.

One of the most important part is the establishment of *trust* and support for different trust models. It is generally hard to prove to a third party that an obligation is fulfilled. Hence, users have to trust the data processor, i.e. assume that the data processor will fulfill obligations. The anchor of trust could be based on various technologies, e.g. a trusted stack (certified TPM [2], trusted OS), on reputation, or on certification by external auditors. The structure of obligations should be independent from the trust model.

Complementary to the problem of trust is the observability of obligations and transparency of data handling. The obligation enforcement as well as mechanisms to load policies should be comprehensive so that data processors and auditors can easily check whether a specific deployment is compliant with a given specification. This is a prerequisite to enable data-handling transparency toward end users.

The problem of *Rights to fulfill obligations* as discussed by Ni et. al. [25] and others in the literature. It is important to distinguish between violation of an obligation because of the enforcement platform's capability to fulfill it or because of the inappropriate rights assigned to it. Such cases could arise when the platform does not have rights to perform certain actions to fulfill the obligation. Our assumption was that the framework would always have appropriate rights to perform actions on the external systems within the same trust domain.

Usability for the end users is another important aspect. Surveys has been done to evaluate the user privacy concerns and behavior on the internet. Earp et. al [13] classified users into three categories with the bulk of them willing to show their private data under certain conditions. Mont Cassasa et. al. [27] presented the results of usability testing of their obligation management system and identified key problem areas.

Another important problem to be addressed is the distributed scenario with the data travelling through multiple hops and the policy attached to it is evolved. We could have multipe possible schemes like

- Before sending an obligation policy to the data owner, we collect the policies from possible secondary data consumers and send a unified policy to the data owner. This also covers the delegation of obligation scenario. The policy received at the data owner side will contain obligations committed by mutiple subjects (see Figure 5.2).

- Alternative scheme could be that the primary data consumer only send its policy to the data owner. Later, on request from secondary consumers for sharing user data, the primary consumer behaves as the data owner. It compares the sticky policy attached to the requested data with the policy of the requestor. The sharing of requested data on the downstream depends on such a negotiation mechanism.
- Hybrid scheme combining multiple techniques.

Investigating composite actions would increase the flexibility of the language and help users to structure domain specific dialects of the obligation language in a more efficient and error-preventing way. However, the downside is that composite actions introduce a level of complexity that makes it harder to detect contradictions inside an obligation rule. Perhaps a set of design rules defining how action may be composed could help to deal with this complexity.

We would also like to implement a policy validity and consistancy tool which input a policy and detect inconsistancies within it. Validity checking is to ensure that policy will be enforceable and consistancy checking is to ensure that the policy is free from contradictions.

At the design and implementation level, we would like to investigate and implement schemes to save and retreive policies from RDBMS efficiently. Research has already been done in the area of databases which will aid in our problem resolution. Additionally, policies should be saved in a way that multidimensional analysis of the policy could be performed. For example administrator would like to have a report showing

- All the policies in the system having Action X within one of its rules.
- All the rules which were violated last month.
- All the triggers fired between <start time> and <end time>.

Such analysis are only possible if we have policies saved in a normalized database with a well designed schema. Additionally, the DB schema should also support efficient insertions and updates as it would be an online transaction processing system. The incorrect or sub-optimal database design will impact the performance. We left the results with policy repository on RDBMS as they were highly dependent on the dynamics of the backend RDBMS repository and would not have given any valuable information to further optimize the obligation framework core components. Factors like size of DB, DB schema, performance tuning of DB (e.g Logs, query buffers, backups, indexes etc) and DB administration were more important than anything inside the framework.

We conclude the thesis here. Complete schema of the XML language and trigger classification is being presented in the appendices.

Bibliography

[1] Microsoft windows rights management services at http://www.microsoft.com/rms/.

[2] Trusted Computing Platform Alliance (TCPA). Main Specification Version 1.1b, Trusted Computing Group, Inc., February 22 2002.

[3] Anne H. Anderson. A comparison of two privacy policy languages: Epal and xacml. In *SWS '06: Proceedings of the 3rd ACM workshop on Secure web services*, pages 53–60, New York, NY, USA, 2006. ACM.

[4] C. A. Ardagna, M. Cremonini, S. De Capitani di Vimercati, and P. Samarati. A privacy-aware access control system. *J. Comput. Secur.*, 16(4):369–397, 2008.

[5] Paul Ashley, Satoshi Hada, Gnter Karjoth, Calvin Powers, and Matthias Schunter. Enterprise privacy authorization language (epal 1.2) at http://www.w3.org/submission/2003/subm-epal-20031110/.

[6] Adam Barth, John C. Mitchell, and Justin Rosenstein. Conflict and combination in privacy policy languages. In *WPES '04: Proceedings of the 2004 ACM workshop on Privacy in the electronic society*, pages 45–46, New York, NY, USA, 2004. ACM.

[7] M. Casassa and F. Beato. On parametric obligation policies: Enabling privacy-aware information lifecycle management in enterprises. *Policies for Distributed Systems and Networks, 2007. POLICY '07. Eighth IEEE International Workshop on*, pages 51–55, June 2007.

[8] Marco Casassa Mont. Hpl-2005-180: A system to handle privacy obligations in enterprises at http://www.hpl.hp.com/techreports/2005/hpl-2005-180.html, 2005.

[9] Marco Casassa Mont, Siani Pearson, and Pete Bramhall. Towards accountable management of identity and privacy: Sticky policies and enforceable tracing services. In *Proceedings of 14th International Workshop on Database and Expert Systems Applications (DEXA'03)*, pages 377 – 382, 2003.

[10] David W. Chadwick and Stijn F. Lievens. Enforcing "sticky" security policies throughout a distributed application. In *Proceedings of the 2008 workshop on Middleware security table of contents*, pages 1–6, New York, NY, USA, 2008. ACM.

[11] Laurence Cholvy and Christophe Garion. Deriving individual obligations from collective obligations. In *AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 962–963, New York, NY, USA, 2003. ACM.

[12] Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Obligations and their interaction with programs. In *ESORICS*, pages 375–389, 2007.

[13] J.B. Earp, A.I. Anton, L. Aiman-Smith, and W.H. Stufflebeam. Examining internet privacy policies within the context of user privacy values. *Engineering Management, IEEE Transactions on*, 52(2):227–237, May 2005.

[14] Pedro Gama and Paulo Ferreira. Obligation policies: An enforcement platform. In *POLICY '05: Proceedings of the Sixth IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 203–212, Washington, DC, USA, 2005. IEEE Computer Society.

[15] Manuel Hilty, Alexander Pretschner, David Basin, Christian Schaefer, and Thomas Walter. A policy language for distributed usage control. In Joachim Biskup and Javier Lopez, editors, *12th European Symposium on Research in Computer Security (ESORICS 2007)*, volume 4734 of *LNCS*, pages 531–546. Springer-Verlag, 2007.

[16] R. (ed.) Iannella. Open digital rights language - version 1.1 (august 2002). odrl.net/1.1/ODRL-11.pdf.

[17] Keith Irwin, Ting Yu, and William H. Winsborough. On the modeling and analysis of obligations. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 134–143, New York, NY, USA, 2006. ACM.

[18] Basel Katt, Xinwen Zhang, Ruth Breu, Michael Hafner, and Jean-Pierre Seifert. A general obligation model and continuity: enhanced policy enforcement engine for usage control. In *SACMAT '08: Proceedings of the 13th ACM symposium on Access control models and technologies*, pages 123–132, New York, NY, USA, 2008. ACM.

[19] Pranam Kolari, Li Ding, Shashidhara Ganjugunte, Lalana Kagal, Anupam Joshi, and Tim Finin. Enhancing web privacy protection through declarative policies. In *Proceedings of the IEEE Workshop on Policy for Distributed Systems and Networks(POLICY 2005)*, June 2005.

[20] Emil Lupu, Nicodemos Damianou, Naranker Dulay, and Morris Sloman. The ponder policy specification language. In *POLICY '01: Proceedings of the International Workshop on Policies for Distributed Systems and Networks*, pages 18–38, London, UK, 2001. Springer-Verlag.

[21] Emil C. Lupu and Morris Sloman. Conflicts in policy-based distributed systems management. *IEEE Trans. Softw. Eng.*, 25(6):852–869, 1999.

[22] David Basin Manuel Hilty and Alexander Pretschner. On obligations. *Computer Security ESORICS 2005*, pages 98–117, 2005.

[23] Pietro Mazzoleni, Bruno Crispo, Swaminathan Sivasubramanian, and Elisa Bertino. Xacml policy integration algorithms. *ACM Trans. Inf. Syst. Secur.*, 11(1):1–29, 2008.

[24] Tim Moses. OASIS eXtensible Access Control Markup Language (XACML) Version 2.0. OASIS Standard oasis-access_control-xacml-2.0-core-spec-os, OASIS, February 2005.

[25] Qun Ni, Elisa Bertino, and Jorge Lobo. An obligation model bridging access control policies and privacy policies. In *SACMAT '08: Proceedings of the 13th ACM symposium on Access control models and technologies*, pages 133–142, New York, NY, USA, 2008. ACM.

[26] Qun Ni, Alberto Trombetta, Elisa Bertino, and Jorge Lobo. Privacy-aware role based access control. In *SACMAT '07: Proceedings of the 12th ACM symposium on Access control models and technologies*, pages 41–50, New York, NY, USA, 2007. ACM.

[27] John Sören Pettersson, Simone Fischer-Hubner, Marco Casassa Mont, and Siani Pearson. How ordinary internet users can have a chance to influence privacy policies. In *NordiCHI '06: Proceedings of the 4th Nordic conference on Human-computer interaction*, pages 473–476, New York, NY, USA, 2006. ACM.

[28] A. Pretschner, F. Schütz, C. Schaefer, and T. Walter. Policy evolution in distributed usage control. In *4th Intl. Workshop on Security and Trust Management*. 06 2008.

[29] Erik Rissanen. OASIS eXtensible Access Control Markup Language (XACML) Version 3.0. OASIS working draft 10, OASIS, March 2009.

[30] William H. Stufflebeam, Annie I. Antón, Qingfeng He, and Neha Jain. Specifying privacy policies with p3p and epal: lessons learned. In *WPES '04: Proceedings of the 2004 ACM workshop on Privacy in the electronic society*, pages 35–35, New York, NY, USA, 2004. ACM.

[31] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, 2009.

[32] Janice Warner and Soon Ae Chun. A citizen privacy protection model for e-government mashup services. In *dg.o '08: Proceedings of the 2008 international conference on Digital government research*, pages 188–196. Digital Government Society of North America, 2008.

A. Policy Schema and Example Policies

This appendix gives the detailed XML schema of our proposed language which is then followed by two example policies.

A.1. XML Schema of the Language

```
<xs:schema elementFormDefault="qualified"
  targetNamespace="http://www.microsoft.com/emic/primelife/obligations"
  xmlns:pl_ob="http://www.microsoft.com/emic/primelife/obligations"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="obligationPolicy" type="pl_ob:obligationPolicy" />
  <xs:complexType name="obligationPolicy">
    <xs:sequence minOccurs="1" maxOccurs="1">
      <!--Name: optional part and not required in sticky policy, only for DO if it stores it-->
      <xs:element minOccurs="0" maxOccurs="1" name="name" type="xs:anyURI" />
      <!--Description: optional part and not required in sticky policy, only for DO if it stores it-->
      <xs:element minOccurs="0" maxOccurs="1" name="description" type="xs:string" />
      <xs:element name="rules" type="pl_ob:ob_rules" />
    </xs:sequence>
    <!--ID: this is the id of the policy issued by the issuer for its record.
      will come back from user with the data, the issuer will ensure the state of the policy using this id-->
    <xs:attribute name="id" type="xs:ID" use="required" />
  </xs:complexType>
  <xs:complexType name="ob_rules">
    <xs:sequence minOccurs="1" maxOccurs="1">
      <!--Ruleset is introduced to avoid redundancy by defining subject with each and every rule-->
      <xs:element minOccurs="1" maxOccurs="unbounded" name="ruleset" type="pl_ob:ob_ruleset" />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="ob_ruleset">
    <xs:sequence>
      <!--Simple cartesian product of subject and rule will give us the set of complete obligation rules-->
      <xs:element minOccurs="1" maxOccurs="unbounded" name="subject" type="pl_ob:ob_subject" />
      <xs:element minOccurs="1" maxOccurs="unbounded" name="rule" type="pl_ob:ob_rule" />
    </xs:sequence>
    <!--must be unique with in a policy-->
    <xs:attribute name="id" type="xs:integer" use="required" />
  </xs:complexType>
  <!--subject-->
  <xs:complexType name="ob_subject">
    <xs:sequence>
      <xs:element minOccurs="1" maxOccurs="1" name="name" type="xs:anyURI" />
      <xs:element minOccurs="0" maxOccurs="1" name="description" type="xs:string" />
      <!--Will be extended in future to introduce other features-->
    </xs:sequence>
    <!--must be unique with in a ruleset-->
    <xs:attribute name="id" type="xs:integer" use="required" />
  </xs:complexType>
  <!--Obligation rule-->
  <xs:complexType name="ob_rule">
    <xs:sequence>
      <xs:element minOccurs="1" maxOccurs="1" name="action_expression" type="pl_ob:ob_aexpr" />
      <!--triggers are inward to the rule, rule must be active w.r.t application condition-->
      <xs:element minOccurs="1" maxOccurs="1" name="triggers" type="pl_ob:ob_triggers" />
      <!--defines additional application condition of this rule-->
      <xs:element minOccurs="1" maxOccurs="1" name="application" type="pl_ob:ob_cexpr" />
      <!--events are outward and are sent to event engine when rule is violated/fulfilled-->
      <xs:element minOccurs="0" maxOccurs="1" name="events" type="pl_ob:ob_events" />
    </xs:sequence>
    <!--must be unique with in a ruleset-->
    <xs:attribute name="id" type="xs:integer" use="required" />
    <xs:attribute default="true" name="IsMandatory" type="xs:boolean" />
  </xs:complexType>
```

Obligation Policy Schema

```
<!--Action-->
<xs:complexType name="ob_aexpr">
  <xs:sequence>
    <!--To identify action plugin-->
    <xs:element minOccurs="1" maxOccurs="1" name="action" type="pl_ob:ob_parameterizedIdentifier" />
    <!--This is optional, default value can be set in the system-->
    <xs:element minOccurs="1" maxOccurs="1" name="retry_interval" type="xs:duration" />
    <!--This is optional, default value can be set in the system-->
    <xs:element minOccurs="0" maxOccurs="1" name="retry_count" type="xs:integer" />
  </xs:sequence>
</xs:complexType>
<!--Condition Expression-->
<xs:complexType name="ob_cexpr">
  <xs:sequence>
    <xs:element minOccurs="1" maxOccurs="unbounded" name="pterm" type="pl_ob:ob_prodterm" />
  </xs:sequence>
</xs:complexType>
<!--Product term-->
<xs:complexType name="ob_prodterm">
  <xs:sequence>
    <xs:choice minOccurs="1" maxOccurs="unbounded">
      <xs:element minOccurs="0" maxOccurs="1" name="condition" type="pl_ob:ob_condition" />
      <xs:element minOccurs="0" maxOccurs="1" name="condition_expression" type="pl_ob:ob_cexpr" />
    </xs:choice>
  </xs:sequence>
</xs:complexType>
<!--Condition - Result will always be boolean-->
<xs:complexType name="ob_condition">
  <xs:sequence>
   <xs:element minOccurs="1" maxOccurs="1" name="scope" type="xs:anyURI" />
   <xs:element minOccurs="1" maxOccurs="1" name="name" type="xs:string" />
   <xs:element minOccurs="1" maxOccurs="unbounded" name="parameters" type="pl_ob:ob_condition_parameters"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="ob_condition_parameters">
  <xs:sequence>
    <xs:choice minOccurs="1" maxOccurs="unbounded">
      <xs:element name="literal" type="pl_ob:ob_condition_literal" />
      <xs:element name="variable" type="pl_ob:ob_condition_var" />
      <!--function output can be other than boolean-->
      <xs:element name="function" type="pl_ob:ob_condition_function" />
    </xs:choice>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="ob_condition_var">
  <xs:sequence>
    <xs:element minOccurs="1" maxOccurs="1" name="type" type="xs:string" />
    <xs:element minOccurs="1" maxOccurs="1" name="scope" type="xs:anyURI" />
    <xs:element minOccurs="1" maxOccurs="1" name="name" type="xs:string" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="ob_condition_literal">
  <xs:sequence>
    <xs:element minOccurs="1" maxOccurs="1" name="type" type="xs:string" />
    <xs:element name="value" type="xs:string" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="ob_condition_function">
  <xs:sequence>
   <xs:element minOccurs="1" maxOccurs="1" name="type" type="xs:string" />
   <xs:element minOccurs="1" maxOccurs="1" name="scope" type="xs:anyURI" />
   <xs:element minOccurs="1" maxOccurs="1" name="name" type="xs:string" />
   <xs:element minOccurs="1" maxOccurs="unbounded" name="parameters" type="pl_ob:ob_condition_parameters"/>
  </xs:sequence>
</xs:complexType>
<!--Triggers-->
<xs:complexType name="ob_triggers">
 <xs:sequence>
  <xs:element minOccurs="0" maxOccurs="unbounded" name="absolute" type="pl_ob:ob_trigger_absolute" />
  <!--Trigger function will be used for iterative obligations e.g periodic triggers-->
  <xs:element minOccurs="0" maxOccurs="1" name="function" type="pl_ob:ob_parameterizedIdentifier" />
  <!--External triggers define only the type, the type hierarchy is application dependent-->
  <xs:element minOccurs="0" maxOccurs="unbounded" name="external" type="pl_ob:ob_parameterizedIdentifier"/>
 </xs:sequence>
</xs:complexType>
<xs:complexType name="ob_trigger_absolute">
  <xs:sequence>
    <xs:element minOccurs="1" maxOccurs="1" name="fireAt" type="xs:dateTime" />
    <xs:element minOccurs="0" maxOccurs="1" name="timeout" type="xs:duration" />
```

Obligation Policy Schema

```
137     </xs:sequence>
138     <xs:attribute name="id" type="xs:integer" use="required" />
139   </xs:complexType>
140   <!--Events-->
141   <xs:complexType name="ob_events">
142     <xs:sequence>
143       <xs:element minOccurs="0" maxOccurs="unbounded" name="event" type="pl_ob:ob_parameterizedIdentifier" />
144     </xs:sequence>
145   </xs:complexType>
146   <!--parameterized object-->
147   <xs:complexType name="ob_parameterizedIdentifier">
148
149     <xs:sequence>
150       <xs:element minOccurs="1" maxOccurs="1" name="scope" type="xs:anyURI" />
151       <xs:element minOccurs="1" maxOccurs="1" name="name" type="xs:string" />
152       <xs:element minOccurs="0" maxOccurs="1" name="parameters" type="pl_ob:ob_parameters" />
153     </xs:sequence>
154     <xs:attribute name="id" type="xs:integer" use="required" />
155   </xs:complexType>
156   <xs:complexType name="ob_parameters">
157     <xs:sequence>
158       <xs:element minOccurs="1" maxOccurs="unbounded" name="parameter" type="pl_ob:ob_parameter" />
159     </xs:sequence>
160   </xs:complexType>
161   <!--Parameter-->
162   <xs:complexType name="ob_parameter">
163     <xs:sequence>
164       <xs:element minOccurs="1" maxOccurs="1" name="type" type="xs:string" />
165       <xs:element name="name" type="xs:anyURI" />
166       <xs:element name="value" type="xs:string" />
167     </xs:sequence>
168     <xs:attribute name="IsRequiredByUser" type="xs:boolean" use="required" />
169   </xs:complexType>
170 </xs:schema>
```

Obligation Policy Schema

A.2. Example: Deleted and Notify Proactive Obligations

Below we give an example obligation expressed in our obligation language. It defines two rules. The first rule (7) states that the data shall be delete at a given point in time (16). When the rule is fulfilled, the obligation engine shall fire an event (20). The second rule (26) shall notify the data owner at a given e-mail address (35). The engine tries this twice (41) with a delay of one day (40). The rule is invoked via the event (45) that was defined in the first rule. In other words, when the deletion is successful the data owner will be notified. If we remove rule (26) from the policy then only deletion will be performed and no notification will be sent.

```
1  <obligationPolicy id="1">
2    <rules>
3      <ruleset>
4        <subject id="1">
5          <name>http://www.microsoft.com/emic</name>
6        </subject>
7        <rule id="1">
8          <action_expression>
9            <action id="1">
10             <scope>http://www.microsoft.com/emic/obligations/base/actions</scope>
11             <name>Delete</name>
12           </action>
13         </action_expression>
14         <triggers>
15           <absolute id="1">
16             <fireAt>2009-03-18T18:05:00</fireAt>
17           </absolute>
18         </triggers>
19         <events>
20           <event>
21             <scope>http://www.microsoft.com/emic/obligations/base/triggers</scope>
22             <name>/ruleFulfilled#</name>
23           </event>
```

```
24          </events>
25        </rule>
26        <rule id="2">
27          <action_expression>
28            <action id="1">
29              <scope>http://www.microsoft.com/emic/obligations/base/actions</scope>
30              <name>Notify</name>
31              <parameters>
32                <parameter>
33                  <type>String</type>
34                  <name>http://base/parameter/target/emailAddress</name>
35                  <value>alice@contoso.com</value>
36                </parameter>
37              </parameters>
38            </action>
39            <parameter_type>EmailNotifyParameters</parameter_type>
40            <retry_interval>P0Y0M1DT0H0M0S</retry_interval>
41            <retry_count>2</retry_count>
42          </action_expression>
43          <triggers>
44            <external id="1">
45              <scope>http://www.microsoft.com/emic/obligations/base/triggers</scope>
46              <name>/ruleFulfilled#</name>
47              <parameters>
48                <parameter>
49                  <type>System.Int64</type>
50                  <name>http://base/parameter/ruleID</name>
51                  <value>1</value>
52                </parameter>
53              </parameters>
54            </external>
55          </triggers>
56        </rule>
57    </ruleset>
58    </rules>
59  </obligationPolicy>
```

A.3. Example: Preventive Obligation

Listing below gives an example of obligation rule which is enforced through prevention. Action of this obligation is *preventAction* in line (10) which is the action to prevent a particular action. In our case this action is *delete* which is expressed as first parameter of action in line 14. The trigger, as defined in line (26), is of type *pre-action external*. It illustrates that a trigger will be sent by some external entity to the framework before doing some particular action. This action is same as the first parameter of action that is *delete* which is expressed in line (30).

The requirement to enforce such obligations is that the framework is integrated with the data repositories and other infrastructure system. When someone would try to delete the user data directly from the data repository, the repository will generate a trigger to the framework to ask and the framework will decline based on this rule. We prototyped this scenario by writing a *DELETE* trigger procedure in T-SQL on SQL server 2008 DB. The SQL Server trigger procedure called the Event Engine web service in our framework and based on the result it either completed the deleted operation or completely rolled back the transaction.

```
1  <?xml version="1.0" encoding="utf-8" ?>
2    <obligationPolicy id = "3" xmlns="http://www.microsoft.com/emic/primelife/obligations">
3      <rules>
4        <ruleset>
5          <subject id="1">
6            <name>http://www.microsoft.com/emic</name>
7          </subject>
8          <rule id="1">
9            <action_expression>
```

```
10              <action id="1" >
11                <scope>http://www.microsoft.com/emic/primelife/obligations/base/actions</scope>
12                <name>preventAction</name>
13                <parameters>
14                  <parameter IsRequiredByUser="False">
15                    <type>String</type>
16                    <name>http://base/parameter/name</name>
17                    <value>Delete</value>
18                  </parameter>
19                </parameters>
20              </action>
21              <parameter_type>GenericPreventionParameters</parameter_type>
22              <retry_interval>P0Y0M0DT0H1M0S</retry_interval>
23              <retry_count>2</retry_count>
24            </action_expression>
25            <triggers>
26              <external id="1">
27                <scope>http://www.microsoft.com/emic/primelife/obligations/base/triggers</scope>
28                <name>/pre-actionExternal#</name>
29                <parameters>
30                    <parameter IsRequiredByUser="False">
31                      <type>String</type>
32                      <name>http://base/parameter/name</name>
33                      <value>Delete</value>
34                    </parameter>
35                <parameters>
36              </external>
37            </triggers>
38          </rule>
39        </ruleset>
40      </rules>
41    </obligationPolicy>
```

B. Classification of Triggers

We discussed the three forms of triggers in Section 4.4.3. Though the classification given in Section 4.4.3 is at the top. However, we could encounter many different types of triggers under each of the three major forms. Figure B.1 below present the inheritance hierarchy of triggers which are currently in place in the prototype.

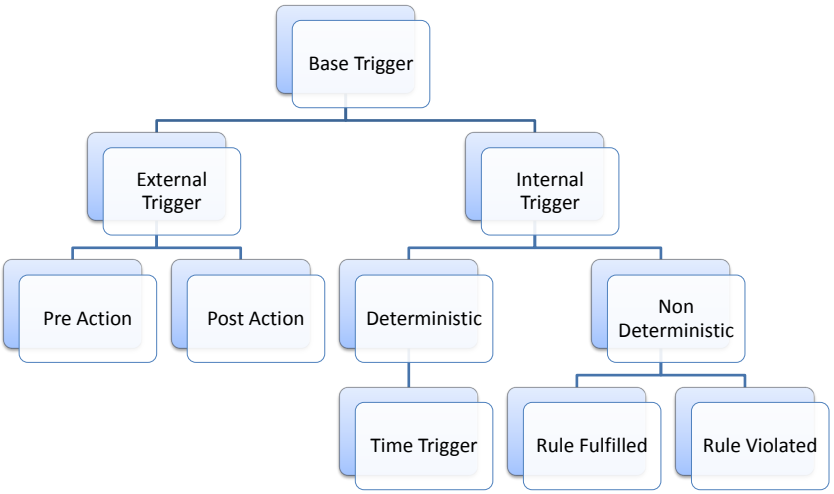

Figure B.1.: Hierarchy of Triggers

The first layer of classification corresponds to trigger sources. Internal triggers are further classified as *deterministic* and *non deterministic*.

External triggers on the other hand are classified based on the forms as was discussed in Section 4.4.3. Now the problem remains to further classify the triggers.

For example a pre-action trigger could come before *Sharing* action or a *Deletion* action. If we start creating separate plug-ins for each of these then it would not be feasible as only the name of the action is changing and not the dynamics or semantics of the pre-action trigger. Thus, we followed the scheme of having parameterized triggers.

With this scheme the pre-action trigger will accompany a set of named parameters like *pre-action(ActionName=Share, PII ID=1)*. This is the simplest form of pre-action trigger having only one parameter and the handling can be done by the default pre-action plug-in. But let's assume we receive *pre-action(ActionName=Share, PII ID=1, Purpose=X, With=Bob)* which is a complex form of trigger which is asking framework the question *Can I share PII with ID 1 with Bob for purpose X*. Though the XML parsing

would be done easily as our schema is not extended for new trigger and actions but the default plug-in may not be able to handle the semantic of this complex trigger.

For such cases, we could promote the parameters of a trigger as a new trigger type and add the plug-in within the system. For the above given complex example we could have a new trigger type named *CanShare* and now the received trigger would be *CanShare(PII ID=1, Purpose=X, With=Bob)*.

If we still receive a complex trigger in a different dimension then we can further promote one parameter and make a new one. For example, if we now receive a trigger *CanShare(PII ID=1, Purpose=Fax, AtNumber = X, With=Bob)* which semantically mean *Can I share PII, with ID 1, for faxing at number X*. In this case, we can further promote the parameter *Purpose* to form a new trigger name *CanShareForFax* and the trigger will now become *CanShareForFax(PII ID=1, With=Bob, Purpose=Fax)*. After these new plug-ins, the hierarchy could become like in Figure B.2 below.

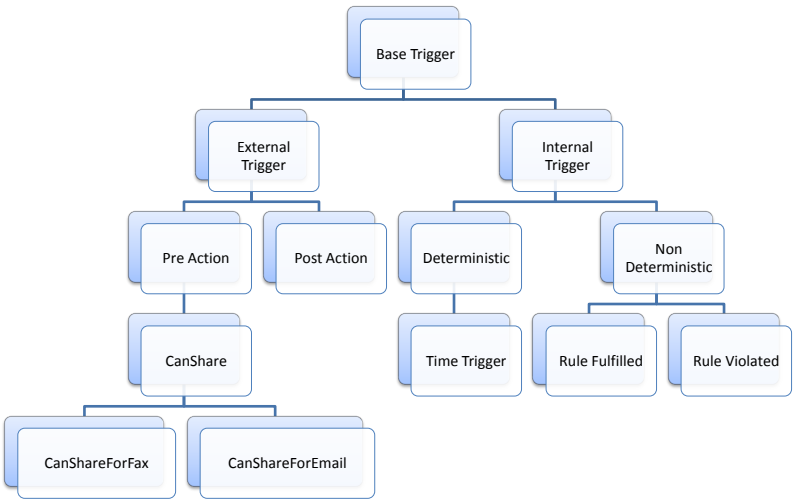

Figure B.2.: Hierarchy with additional plug-ins

The same scheme can be applied to actions as the two are equally complex and analogous to each other. At the top level, actions are classified based on enforcement mechanism. Further plug-ins could be added as shown above for triggers.