Obligation Language for Access Control and Privacy Policies

Muhammed Ali[1], Laurent Bussard[1], and Ulrich Pinsdorf[1]

European Microsoft Innovation Center, Ritterstr. 23, 52072 Aachen, Germany,
{i-alimuh|lbussard|ulrich.pinsdorf}@microsoft.com,
WWW home page: http://www.microsoft.com/emea/emic

Abstract. Defining and enforcing obligations are key aspects of privacy protection. Most of today's access control and data handling languages recognize the importance of obligations and even provide extension points but lack concrete language constructs to actually express obligations. This position paper proposes requirements for a general obligation language spanning access control and usage control. A detailed analysis of our current obligation language and enforcement framework is provided and future extensions are discussed.

1 Introduction

Data handling is an important part of privacy that reaches beyond pure access control. While access control defines *whether* access to data is granted, data handling policies define *what* may happen to the data once the access was granted. Data handling consists of two parts: first, it defines rights of the data consumer to store, process, and share a given piece of data. Second, it defines obligations that the data consumer has to commit to.

We define an *obligation* as: "The *promise* made by the *subject*[1], to the *user*. The subject is expected to fulfill the promise by executing and/or preventing a specific *action* after a *particular event*, e.g. time, and optionally under certain *conditions*".

Obligations play an important role in daily business. Most companies have a process to collect personally identifiable information (PII) on customers and ad-hoc mechanisms to keep track of associated rights and obligations. State of the art mechanisms to handle collected PII accordingly to a privacy policy are lacking expressiveness and/or support for cross-domain obligation definition. Please look at Sect. 5 for a complete evaluation of the state of the art.

We identify four main challenges related to obligations. 1) Service providers must avoid committing to obligations that cannot be enforced. For instance, it is not straightforward to delete data with backup copies. Tools to detect inconsistencies are necessary. 2) Users do care about their privacy and have

[1] In obligation lingo, the subject is the subject of the obligation, i.e. the service provider (or data controller). Do not confuse with the user (subject of the data), generally referred as "data subject" in privacy terminology.

privacy preferences. Preferences may be expressed by ticking check boxes, be a full policy, or even be provided by a trusted third party. Mechanisms to match users privacy preferences and services privacy policies are necessary. 3) Services need a way to express to users obligations, to link obligations and PII, and to enforce them. 4) Users need a way to evaluate the trustworthiness of service providers, i.e. know whether the obligation will indeed be enforced. This could be achieved by assuming that misbehavior impacts reputation, by audit and certification mechanisms, and/or by relying on trusted computing.

While our ongoing work addresses aspects of those four topics, the contributions of this paper are mainly related to the third one. Section 3 describes our proposed general-purpose obligation language. It is expressive enough to specify complex real-world obligations and can be extended with domain specific triggers and actions. This obligation language enables cross-domain scenarios where obligations must be semantically understood on user side and server side. This enables both data handling policies and access control policies to make practical use of obligations. Section 4 gives an overview on the architecture for exchanging and enforcing obligations.

2 Requirements for Obligations

This section describes general requirements for an obligation language and enforcement framework. This list was compiled by looking at scenarios and requirements in [1–5].

Requirement 1: *Independence from policy language.* Obligations can be enforced independently from the embedding policy languages offering the placeholder for the obligation. Thus the obligation framework needs to be technically decoupled from the policy engine. The obligation framework can also be used to enforce policies that are exposed by the service, e.g. P3P [6].

Requirement 2: *Independence from data storage.* The obligation handling must be independent from the concrete data store. The obligation travels with the data and should be stored along with the data so that the reference does not get lost. For instance, the obligation may refer to personal data stored in a database or to documents stored in a file system. Moreover one obligation may refer to multiple pieces of data.

Requirement 3: *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.

Requirement 4: *Support for common obligations.* The obligation handling language should be extensible 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.

Requirement 5: *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 actors.

Requirement 6: *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.

Requirement 7: *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.

Requirement 8: *Support for distributed deployment.* Different deployments of the obligation framework can be envisioned: a corporate-wide obligation framework could cover multiple data bases, a desktop obligation framework could deal with local files, or it could even be provided as a "cloud service". In any cases, only one obligation framework is in charge of a specific piece of data.

Requirement 9: *Support for different trust models.* Users have to trust the service provider, i.e. assume that it will fulfill obligations. The anchor of trust could be based on various technologies, e.g. a trusted stack (certified TPM [7], trusted OS), reputation, or certification by external auditors. The structure of obligations should be independent from the trust model.

Requirement 10: *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.

Requirement 11: *Support for preventive obligations.* The obligation language shall be able to express preventive statements that forbid the execution of an action. For instance, the obligation to store logs six months will forbid deletion of log files.

3 Formal Description of Obligations

Formally, we represent obligation 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 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 is the set of all possible obligations.

Subject Subject s is an identifier for the data processing party that needs to obey an obligation; $s \in S$ where S denotes the set of all existing subject identifiers.

Action 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 $at \in \{proactive, preventive\}$

– *Proactive* actions which require the execution of actions proactively. For example Delete, SendEmail etc.
– *Preventive* actions which 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.

We do not allow actions which can be used as both proactive and preventive within one policy.

An obligation rule contains 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 difficult problem which is part of our future work (see Sect. 6).

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 event locally or externally generated. 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 $\langle ty, d \rangle$ where $ty \in T$, where T denotes the set of all existing trigger types in the system 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.

Application Condition Application condition expressions are boolean equations defining whether a rule is applicable. When an event occurs (e.g. delete resource r), the system takes into account any obligation $\langle s, a, \xi, c, e \rangle$ having triggers ξ registered to such events. 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

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

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 environment variables are specified in the *variable repository*, variable repository is being discussed in Section 4, and values are varied by the administrator. The two product terms are OR-ed together.

Events Optionally obligation rules can have outward events which are generated when a state of the rule is changed. Event of one obligation rule can be a trigger for another rule within the same policy. Through this design we implement the notion of cascaded obligations. Formally, $e \in E$ where E denotes the set of existing events.

3.1 Obligation Policy

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

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

3.2 Inconsistencies

It is of prime importance that the written policy must be consistent. Inconsistencies like self contradiction, infinite cascading including cyclic dependencies etc must be absent from a policy.

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. We use the operator $\bowtie$ to represent *semantic contradiction* between two entities. This is symmetric, non-reflexive and non-transitive relation.

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

1. ρ has an obligation rule which is inconsistent, i.e. $\exists o \in \rho : o\, \textit{is inconsistent}$.
2. 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 plugins are not present within the system. Policies must be validated at the writing time before being deployed in the templates repository.

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 are not overlapping then it is not necessarily a consistency error. We define function $IsConditionOverlap(c_1, c_2) \in \{true, false, undefined\}$ which 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 actions. During action plugin design, we define explicitly which actions are contradicting others within the system. This meta information aid policy writers to write consistent policies.

$$\text{If } a_1 \bowtie a_2 \;\wedge\; s_1 = s_2 \wedge\; \textit{IsConditionOverlap}(c_1, c_2) = true \Rightarrow o_1 \bowtie o_2$$

When it is undecidable to establish the condition overlap relation then we can only raise a warning to the policy writer. We could take *undefined* as a consistency error but that will reduce the expressiveness of the language.

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

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. There is also a possibility 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 existence of policy itself, attached to deleted data, may vanish so the obligation rules which are supposed to be executed after *delete* action may become *redundant* or *non-reachable.* The current implementation does not yet target such complex cases.

Infinite cascading of rules because of the presence of events which can trigger other rules within the same policy is also a problem. It must be ensured that infinite cascading of rules must not happen and cycles are identified at policy writing time.

3.3 Aspects of Obligations

Obligation rules could be subjected to certain generic and temporal conditions which are prerequisite to obligation rule fulfillment. This key aspect is addressed by having *application condition* construct in our proposed obligation rule. Conditions could even be stateful for instance take the statement *subject X commits to Send Account Statement three times a year* where the state of the condition should be tracked to establish the rule applicability.

In case of temporal conditions the time frame in which the rule would be active is specified explicitly which makes the rule as time bounded. Alternatively, we can have only non-temporal conditions but their fulfillment is non deterministic. In the absence of temporal conditions the obligation rule can be time unbounded.

Cyclic or repeating obligations are required to be fulfilled multiple times. This aspect has been incorporated by allowing multiple triggers to be defined for a single obligation rule.

There are some aspects which are not addressed until now but worth mentioniong here. We consider that obligation subject could be more complex than just an identity. We could have an individual entity who has the full responsibility of fulfilling obligations or collection of entities forming a logical subject and the responsibility division to fulfill the obligation in turn could be complex like *All*, *one out of all* etc. In real world we could even have one entity committing something on behalf of another based on some underlying reason e.g *Authority*,*Mutual agreement* etc. *Observability* or *monitoring* of obligation fulfillment is another important aspect as also being discussed in [5]. We do consider that monitoring of obligations is an attribute of the rule as well as dependent on the reference monitor scope whether deployed within the same trust domain or outside. In the current work we have not addressed these problems.

4 Architecture for Enforcement

We have designed and implemented an enforcement architecture for the obligations which are expressed through our proposed language. The core requirements of the architecture were to ensure the enforcement of obligations, to enable customized actions, to facilitate integration with existing systems and to support external systems. The detailed obligation framework architecture is illustrated in Figure 1.

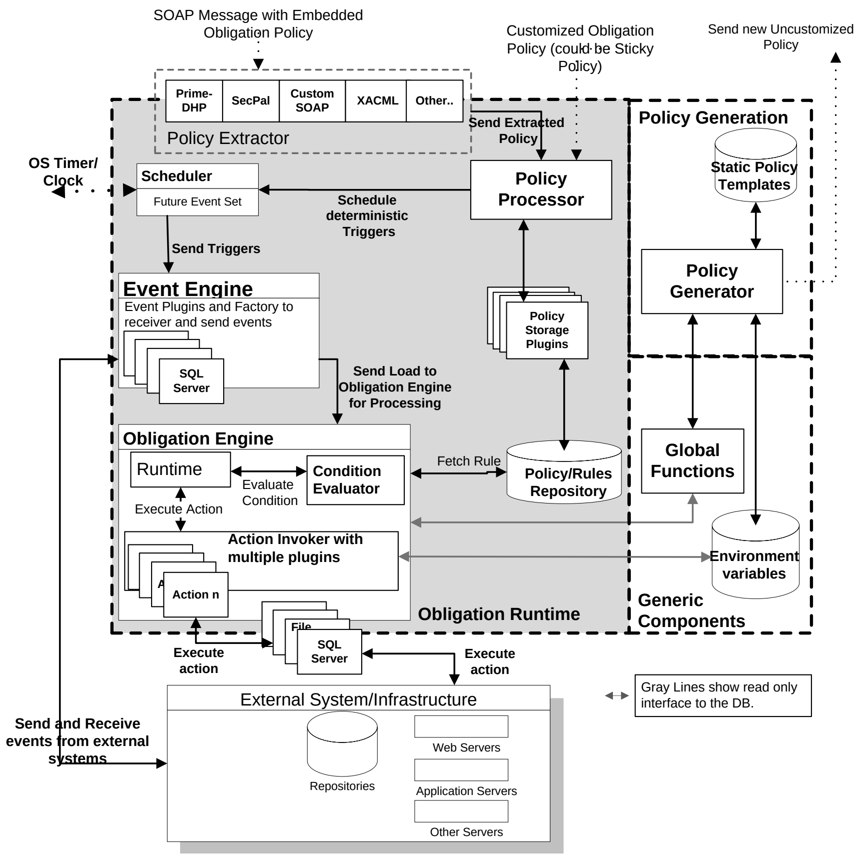

Fig. 1. Obligation Framework Architecture

The key feature of the framework is its flexibility which is achieved through the plugin based design allowing easy integration of new types of obligations and new types of external systems. The framework uses the available plugins 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. As shown in Figure 1, the architecture is separated into three main parts, namely *Policy generation*, *Generic components*, and *Obligation Runtime*.

4.1 Policy Generation Components

They are 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, to be sent to the user, one of those templates is extracted from the repository based on the context of the request and is sent back to the user.

4.2 Generic Components

They are also an optional set of components used to store environmental variables, global functions etc.

4.3 Obligation Runtime Components

This is the core set of components within the architecture. We now discuss each of the subcomponents of the obligation runtime briefly.

Policy Extractor Plugins are used to extract the obligation expression from the incoming message which could be in any format, as long as the required translation/extraction plugin is present. If the obligation policy is embedded within any other container message, the corresponding plugin parses the message and forwards only the obligation policy part to the system. This enables fulfillment of requirements 1, 2 and 3.

Policy Processor The policy is received by the *policy processor* either through an external interface or via any of the *policy extractor plugins.* It processes the policy, check inconsistencies, and schedule deterministic triggers. The initial transaction interplay with the user ends here and the system returns the system wide unique *Policy ID* to the caller which forwards it to the user.

The caller in turn stores PII (Personally Idenfiable Information) somewhere within the infrastructure of subject along with the policy reference. Both data and policy are stored separately but remain connected through cross-references. Thus, the enforcement framework only manages policy templates, policies conntected to some data under the subject ownership and references to that data. Data itself is being managed by systems external to our obligation framework, but within the service provider's trust domain.

Scheduler It is used to initiate time based triggers which are scheduled by other components of the runtime engine. The triggers are being in the form of messages to the event engine. We call scheduled triggers as *future event set.*

Event Engine It 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 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. This design allows us to integrate our framework with existing systems and to enforce preventive obligations which are fulfilled by inhibiting rather performing an action which is our requirement 11.

Obligations Engine It is the main load processing component. It receive the load/triggers from the event engine and process them. On receiving a new trigger, it fetches the policy rules from the policy repository, evaluate conditional statements, finds the respective action plugin 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.

The policies contain *action* with parameters attached to each obligation rule. Each of these actions must match to an available *action plugin* within the obligation engine. The parameters listed within the policy must also match to the parameters required by the actions plugin. This enables fulfillment of requirements 5 and 6.

We propose two layered action plugin mechanism, in the obligation engine component. The upper layer contains the plugins for specific actions e.g *delete*, *notify* and the lower plugin 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 appropriated plugin.

To ensure integrity, the action parameters must satisfy the required parameters for only one lower layer plugin. This design ensures the requirements 4, 6 and 7.

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. This targets our requirement 8 for the enforcement platform. Requirement 9 on trust model and 10 on transparency are not yet covered and need additional work.

5 Comparison with State of the Art

Most of the available policy languages, like XACML [4,8], EPAL [9], Ponder [10], Rei [11] and PRIME-DHP [1], 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.

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. [2] 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. They propose a formal obligation model and provide the framework to enforce obligations. However, they do not offer the notion of preventive obligations (negative obligations) and multiple subjects. As opposed to their policy expressions, we propose a schema which is not modified when domain specific obligations, including new actions, events and triggers, are added. They took the notion of *On violation actions* within a policy rule to express actions which are taken in case of obligation violation. We cover this aspect by defining that obligations rules contain *events* which can be used to trigger another rule within the same policy to invoke a compensatory action. Since this event-based approach allows cascading of rules, we need to ensure the absence of loops, which remains an open issue of our work. Unlike [2], we also do not allow multiple action per rule because of the system integrity problem which arises from the fact that we cannot map fulfillment of a subset of actions in any policy rule as complete fulfillment and we achieve the same behavior through rule cascading without ambiguity.

Irwin et al. [3] proposed a formal model for obligations and define 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 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. So the two research efforts are targeting different problems.

Pretschner et al. [5, 12, 13] worked in the area of distributed usage control. In [5], 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 as an attribute of the reference monitor and not an attribute of the obligation rule. It depends on the scope of the monitor. 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 [12] they have proposed an obligation specification language (OSL) for usage control and presented the translation schemes between OSL and rights expressions languages, e.g. XrML, so the OSL expression could be enforced using DRM enforcement mechanisms. We have tried to fill that gap by implementing the enforcement platform for enforcing obligation policies without translation. In [13], 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 [13].

Katt et al. [14] proposed an extended usage control (UCON) model with obligations and gave a 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 systems domain, while non-controllable objects are outside the systems 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. [15] 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. [16] proposed a concrete obligation model which is an extension of P-RBAC [17]. 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. [18] 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 an 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 an heterogeneous system environment.

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

6 Conclusion and Future Work

This position paper described challenges and requirements to properly address obligations. We presented a general language for obligations which can be used with today's access control and data handling policy languages. We started with reasoning on the requirements of an abstract yet expressive obligation language and presented eleven requirements for design. 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. 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. Finally we showed how our work relates to the state of the art.

Future work will be aligned with challenges described in the introduction of this paper. First we need mechanisms to help authors of privacy policies to check whether a policy can be enforced. This is especially important since violation of obligations impacts reputation and can have legal implications. Next, we need to look at the protocols and matching mechanisms between users (with privacy preferences) and service providers (with privacy policies). Finally, we will also consider distributed services where collected PII is subsequently shared with third parties. From an obligation perspective, the key interest is to look at distributed yet coherent enforcement. We will pursue this work as part of PrimeLife[2] research project.

References

1. Ardagna, C.A., Cremonini, M., De Capitani di Vimercati, S., Samarati, P.: A privacy-aware access control system. J. Comput. Secur. **16**(4) (2008) 369–397
2. Casassa, M., Beato, F.: 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 (June 2007) 51–55
3. Irwin, K., Yu, T., Winsborough, W.H.: On the modeling and analysis of obligations. In: CCS '06: Proceedings of the 13th ACM conference on Computer and communications security, New York, NY, USA, ACM (2006) 134–143
4. Rissanen, E.: OASIS eXtensible Access Control Markup Language (XACML) Version 3.0. OASIS working draft 10, OASIS (March 2009)
5. Manuel Hilty, D.B., Pretschner, A.: On obligations. Computer Security ESORICS 2005 (2005) 98–117
6. Cranor, L., Langheinrich, M., Marchiori, M., Reagle, J.: The platform for privacy preferences 1.0 (p3p1.0) specification. W3C Recommendation (April 2002)
7. : Trusted Computing Platform Alliance (TCPA). Main Specification Version 1.1b, Trusted Computing Group, Inc. (February 22 2002)
8. Moses, T.: OASIS eXtensible Access Control Markup Language (XACML) Version 2.0. OASIS Standard oasis-access_control-xacml-2.0-core-spec-os, OASIS (February 2005)
9. : Enterprise privacy authorization language (epal 1.2) at http://www.w3.org/submission/2003/subm-epal-20031110/
10. Damianou, N., Dulay, N., Lupu, E., Sloman, M.: The ponder policy specification language. In: POLICY '01: Proceedings of the International Workshop on Policies for Distributed Systems and Networks, London, UK, Springer-Verlag (2001) 18–38
11. Kagal, L., Finin, T., Joshi, A.: A policy language for a pervasive computing environment. In: POLICY '03: Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks, Washington, DC, USA, IEEE Computer Society (2003) 63
12. Hilty, M., Pretschner, A., Basin, D., Schaefer, C., Walter, T.: A policy language for distributed usage control. In Biskup, J., Lopez, J., eds.: 12th European Symposium on Research in Computer Security (ESORICS 2007). Volume 4734 of LNCS., Springer-Verlag (2007) 531–546
13. Pretschner, A., Schütz, F., Schaefer, C., Walter, T.: Policy evolution in distributed usage control. In: 4th Intl. Workshop on Security and Trust Management. (06 2008)

[2] The research leading to the results presented in this paper has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) for PrimeLife project. The information in this document is provided "as is" and no guarantee or warranty is given that the information is fit for any particular purpose. The PrimeLife consortium members shall have no liability for damages of any kind including without limitation direct, special, indirect, or consequential damages that may result from the use of these materials subject to any liability which is mandatory due to applicable law.

14. Katt, B., Zhang, X., Breu, R., Hafner, M., Seifert, J.P.: 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, New York, NY, USA, ACM (2008) 123–132
15. Cholvy, L., Garion, C.: Deriving individual obligations from collective obligations. In: AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems, New York, NY, USA, ACM (2003) 962–963
16. Ni, Q., Bertino, E., Lobo, J.: 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, New York, NY, USA, ACM (2008) 133–142
17. Ni, Q., Trombetta, A., Bertino, E., Lobo, J.: Privacy-aware role based access control. In: SACMAT '07: Proceedings of the 12th ACM symposium on Access control models and technologies, New York, NY, USA, ACM (2007) 41–50
18. Gama, P., Ferreira, P.: Obligation policies: An enforcement platform. In: POLICY '05: Proceedings of the Sixth IEEE International Workshop on Policies for Distributed Systems and Networks, Washington, DC, USA, IEEE Computer Society (2005) 203–212