Enforcing UCON Policies on the Enterprise Service Bus

Gabriela Gheorghe[1], Paolo Mori[2], Bruno Crispo[1], and Fabio Martinelli[2]

[1] University of Trento, Italy
first.last@disi.unitn.it
[2] IIT CNR Pisa, Italy
first.last@iit.cnr.it

Abstract. In enterprise applications, regulatory and business policies are shifting their semantic from access to usage control requirements. The aim of such policies is to constrain the usage of groups of resources based on complex conditions that require not only state-keeping but also automatic reaction to state changes. We argue that these policies instantiate usage control requirements that can be enforced at the infrastructure layer. Extending a policy language that we prove equivalent to an enhanced version of the UCON model, we build on an instrumented message bus to enact these policies.

Keywords: Usage control model, message bus, policy, enforcement, SOA.

1 Introduction

Modern organizations need to face business conditions that change much more frequently than ever before. Two main drivers for such changes are first, the pressure to reduce costs and the time to market for new services and applications; and second, the need to comply with regulatory requirements that, in recent years, have grown very rapidly in terms of both numbers and complexity.

To support such organizational dynamism, IT systems had to go through a profound transformation. The software-as-a-service paradigm started to be adopted together with SOAs. Among the several potential advantages of SOA, interoperability, modularity and scalability are the first ones organizations are already experiencing. While existing SOA technologies address the first of the two drivers we mentioned (i.e. by supporting standard service interfaces that make outsourcing much easier to implement), not much work as been done until now to address the problem of regulatory compliance. To be suitable, any solution must be: 1) integrated with SOAs, 2) able to easily adapt to regulatory changes (i.e. due to a department relocation in a new country or simply due to the new version of the current regulations) since often the same regulatory requirements apply to many services and departments of the organizations; 3) inclusive of a wide range of policies that may control not only who accesses some services but also how these services are used.

This paper proposes a novel solution that apart from meeting these three requirements, extends the role of reference monitors in enforcing the applicable policies. We build on an existing reference monitor at the *message bus* level [4], which is a common requirement in all service-oriented architectures. By positioning the policy enforcement

R. Meersman et al. (Eds.): OTM 2010, Part II, LNCS 6427, pp. 876–893, 2010.

within the message bus, the security logic is decoupled from the business logic, making it simpler to enforce new regulations that have as scope several services of the organization. Furthermore, different from existing SOA solutions, our extension to the reference monitor supports the UCON model along with classic access control models (i.e. RBAC). This is achieved by using an extended version of the POLPA [10,11] policy language, that we prove able to express any UCON policy.

Overall, to make use of all the properties of the message bus as mediator of all service invocations, we extended the semantics of traditional reference monitors. We propose a first step towards the possibility to design reference monitors which act proactively on service invocations that do not comply with the policy. These actions consist of applying a set of predefined transformations to try to make the invocations compliant rather than resulting in service rejection. Our contributions are:

- A language for expressing SOA message-level requirements and the proof of its equivalence with the UCON model. In this way, we can correctly express any constraint on the usage of the enterprise infrastructure (Sect. 5).
- An extended UCON semantics to a reference monitor for large scale applications. The usability of the concept resides in separating policy specification, enforcement mechanism and enforcement semantics (Sect. 6).
- A prototype for message-flow enforcement that considers cross-service UCON-related policies. Existing message-level security solutions do not consider this kind of restrictions (Sect. 4).

The remainder of this paper is organized as follows: after presenting the need of message-level policy enforcement in a real-world scenario (Sect. 2), we give an overview of the UCON model and the enterprise bus (Sect. 3). Then we describe our proposal of a three-step enforcement process (Sect. 4), and introduce a UCON language that we prove is able to express the policies we aim to enforce (Sect. 5). We present the implementation of the model within the enhanced service bus (Sect. 6), and conclude by discussing the advantages and limitations of our approach (Sect. 8).

2 Motivating Example

Our work is motivated by a case study provided by the Hong Kong Red Cross organization, which manages blood donations for a number of public hospitals. The Red Cross works as an intermediary between several actors: the blood donors, blood receivers, and public hospitals. We concentrate on several of Red Cross's components that relate to data gathering and processing (see Figure 1): a Data Collector Service (DC), a Data Submitter service (DS), a Notification Service (NS), a Logger Service (LS) and a Donation Processing Service (PD). The data collector service is shared between the Red Cross (who gets donor data) and public hospitals. The service can be deployed on several locations, and we assume there is metadata to specify these locations. This service will wrap a standard electronic form and will send it to the DS. The data submitter service DS inserts data into a database. For the Red Cross, this database contains the donor data. This service has a public operation called submit_donor_data(). PD is a service for processing possible donation cases, that wraps a business process that decides to allow

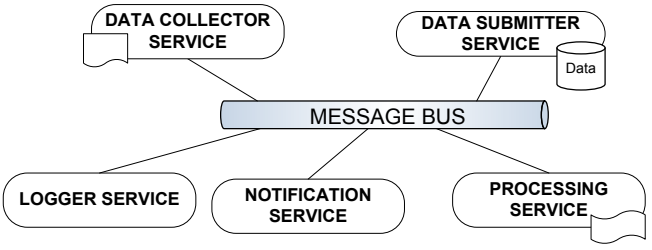


Fig. 1. Web Services used by the Hong Kong Red Cross

or deny the blood donation. The notification service NS sends an email to the Red Cross security administrator. Blood donation data is subject to the internal or even regulatory policies:

(P1). New electronic entries of blood donor data can *only* be issued from the Red Cross branches throughout Hong Kong. This requires the DS to record an entry to the donor database only if the DC has some metadata certifying its location.

(P2). Once a potential donor fills in a paper form, the donation volunteering case will be processed for approval by a nurse within 3 days. This requires a mechanism to monitor any incoming requests to the PD service that contain a reference to a recent donor_id. If such requests are not reported within the given time frame, a notification will be sent to the NS.

(P3). The Red Cross must keep record of all data and operations onto the system. This is both for statistical purposes and for controlling what happens to the data once it reaches its points of interest. This constraint requires an additional logging action (invoking the Logger Service) every time a message to a data-processing service is received.

(P4). Donor data can be kept in the hospital data base for a maximum of 300 days. After this period, a service that deletes the expired donor data will be invoked.

The Red Cross is trusted by all actors in the system to provide assurance that such constraints are complied with. Hence the Red Cross needs a centric view over the general application design, data release and usage. In order to manage a common infrastructure to which the hospitals and government organizations connect, the Red Cross employs the Enterprise Service Bus (ESB) to integrate application components. Using the ESB helps to enact restrictions as P1-P4, as the ESB can distinguish between types of services – data sources and data consumers, as well as map actual endpoints to business logic endpoints. With the help of the ESB, the Red Cross can ensure that what reaches these services is only the *right data*, used on the *right terms*. However, since the restrictions can change depending on external organizational requirements, the Red Cross system needs to offer flexibility and assurance in dealing with the issues described above.

3 Background

This section overviews the two building blocks of our approach. Firstly, UCON comes as a promising way of modeling security concerns in realistic distributed applications.

Secondly, the enterprise service bus has been a well-established vehicle of application integration. The model we use, xESB [4], combines these two concepts and can give guarantees in enforcing policies such as those described above.

3.1 The UCON Model

The Usage Control (UCON) model is a next generation access control model designed for modern and distributed environments. The main novelties of UCON are that attributes of subjects and objects are mutable over time, and hence the decision process and the enforcement of the security policy is performed continuously during access time. This means that an access that has been authorized and that is currently in progress could be revoked because some factors have changed and the access right does not hold any more. The UCON core components are subjects, objects and their attributes, rights and authorizations, conditions and obligations, as described in [12,19].

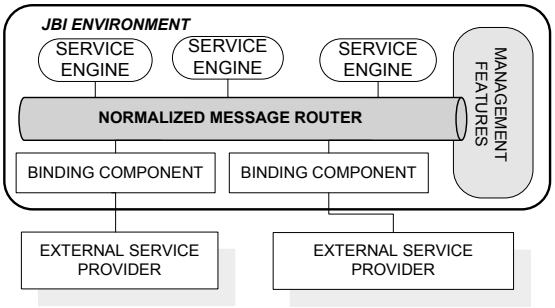
3.2 The ESB in SOA

The ESB is a middleware placed between the various services of an SOA application. It offers a layer of communication and integration logic in order to mitigate technology disparities between communication parties, ranging from intelligent routing to XML data transformation. For example, if the communication partners use different communication protocols, the ESB mediator will perform the protocol conversion; the same if a business consumer needs to aggregate data from several providers, to change the format of its data, or to route it to different services. The ESB increases connectivity by facilitating application integration.

Java Business Integration (JBI) [16] standardizes deployment and management of services deployed on an ESB. It describes how applications need to be built in order to be integrated easily. The generic JBI architecture is shown in Fig. 2 (top). Since the primary function of the ESB is message mediation, the Normalized Message Router (NMR) is the core of the JBI-ESB and provides the infrastructure for all message exchanges once they are transformed into a normalized form (an example is shown in Fig. 2 (bottom)). Components deployed onto the bus can be either service engines, which encapsulate business logic and transformation services; or binding components, which connect external services to the JBI environment.

4 The Enforcement Model

This section examines in more detail the prerequisites of enforcing the policies previously described. Motivating the connection with the usage control model, we argue that enacting these requirements is separate from application logic and can be best performed at the mediation layer provided by the ESB. To that end, we present the enhanced semantics of a message-level reference monitor that enforces Red Cross's usage control constraints.




```
InOnly[
 id: ID:192.168.233.83-1228375ebcb-8:0
 status: Active
 role: provider
 service: {http://www.microsoft.com}acceptor1
 endpoint: acceptor1
 in: <?xml version=1.0" encoding="UTF-8"?><example id="123"/> ]
```

Fig. 2. The architecture of the JBI system (top) and example of a normalized message in an InOnly exchange (bottom)

4.1 A New Vision of Enforcement

Policies such as Red Cross's P1-P4 require an enforcement layer that is independent from any data processing or data management logic. The Red Cross *mediates* all interactions between individual users and the system, and between the system and hospital or government applications. This mediation controls *how* the data is used and how the different actors use the system. We argue that this mediation logic can be naturally integrated with the mediation framework presented by Gheorghe et al. [4]. Note here that no matter the Web services concerned, their states and life cycle are transparent to the messages they use to communicate; the service bus and any extra logic would only handle these messages.

The policies that the Red Cross wants enforced pertain to usage control. They respectively relate to: (P1) location-based usage of a piece of code, (P2) and (P4) time-bound obligations of certain roles or entities, (P3) a system-wide obligation to log application events. P1 refers to a pre-authorization that evaluates an environment condition – the code can only be invoked if the location attribute is a Red Cross branch. P2 is an attribute-based authorization for groups of subjects. P3 is an ongoing obligation that may require management of data or system states.

Designing a *versatile* enforcer of usage control policies at the ESB level requires some refinement and additions to the original UCON model. First, UCON enforcement is binary: it either allows or disallows the usage of an object by a subject. This might be a draconic solution in most applications, when the impossibility to execute an application step might have side-effects over an entire flow of actions. To bridge this gap, a solution is to allow the system to interfere with the original invocation or with any other connected state. This interference has several aspects: what entity performs it, when it should happen, how can it happen. First, we envision two possibilities of what trusted

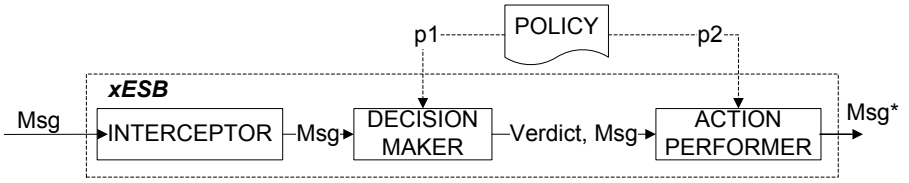


Fig. 3. The enforcement process behind xESB. The interceptor captures a message, the decision maker decides if the message complies with the p1 part of a policy; if not, the action performer enacts the p2 part of a policy, that describes the action to be taken.

entity can interfere with the enforcement process: the system or a trusted third-party (designated in advance). Second, this interference can be performed in two ways: *preventively* or *reactively*, that is before or after the UCON authorization decision. Third, once settled that a usage policy may specify an action to be performed when a condition holds, this action can either be delegated to a subsystem or performed on the fly. These three aspects lead to the following *additional semantics of a UCON-based enforcer*: separation between system and subject obligations as well as between pre- and post-obligations (also noticed in [7]), an active mechanism that can both execute an action or delegate it and observe the results (termed *executor*), the separation between subject actions and consequences of those actions onto the system.

4.2 The Three Steps in Enforcement

As shown in Fig. 3, the enforcement process has three basic steps [4]. **Interception** refers to capturing a usage request as specified by a policy. This assumes the existence of a filtering mechanism that can separate between requests that are interesting from those that are not. The second is the **decision** phase, where based on the same policy, the request just captured is deemed allowable or forbidden by the system, based on its internal state keeping. Making this evaluation implies reasoning about previous actions and obligations. The result is a decision seconded by a set of actions to be executed before of after allowing or forbidding the usage request. Thirdly, the **action performing** means the actions provided in the decision phase will be performed either by the system, or by a trusted third-party entity. In the case of a fault occuring anytime along these steps of the process, a fault message is sent back to the requesting entity and the default action is performed on the original message. In case an external actor performs these actions, the evidence of these actions will be sent back to the usage control enforcer, and they will have to be trusted. Having this split design of the enforcement process helps in achieving good flexibility: if a mechanism implementing any of these steps changes, the others can remain unmodified.

5 Policy Language

The language we use to express security policies is POLPA, an operational *PO*licy *L*anguage based on *P*rocess *A*lgebra [10,11]. This language represents the allowed behaviour of users by defining the sequences of actions that are allowed on the system,

and which authorizations, conditions and obligations must hold before, during and after the execution of each action. Compared to XACML, POLPA ensures continuous usage control: resource accesses can be interrupted if the factors that authorized them before are not valid any more. Let us suppose that $\alpha(x)$ is an action and $p(x)$ is a predicate whose evaluation returns a boolean result; a security policy P is obtained by composing actions according to the following grammar:

$$P ::= \bot \parallel \top \parallel \alpha(x).P \parallel p(x).P \parallel x := e.P \parallel P_1 or P_2 \parallel P_1 par_{\alpha_1,\ldots,\alpha_n} P_2 \parallel \{P\} \parallel Z$$

The informal semantics is the following:

- $\bot$ and $\top$ are the *deny-All* and *allow-All* operators;
- $\alpha(x).P$ is the *sequential operator*, and represents the possibility of performing an action $\alpha(x)$ and then to behave as P;
- $p(x).P$ behaves as P in the case the predicate $p(x)$ is true;
- $x := e.P$ assigns the values of expressions e to variables x and then behaves as P;
- $P_1 or P_2$ is the *alternative operator*, and represents the nondeterministic choice between P_1 and P_2;
- $P_1 par_{\alpha_1,\ldots,\alpha_n} P_2$ is the *synchronous parallel operator*. It expresses that both P_1 and P_2 policies must be simultaneously satisfied. This is used when the two policies deal with actions (in $\alpha_1, \ldots, \alpha_n$);
- $\{P\}$ is the *atomic evaluation*, and represents the fact that P is evaluated in an atomic manner, that once started must be completed. P here is assumed to have at most one action, along with predicates and assignments. It allows the testing or update of variables prior or after an action;
- Z is the constant process. We assume that there is a specification for the process $Z \doteq P$ and Z behaves as P.

As usual for (process) description languages, derived operators may be defined. For instance, $P_1 par P_2$ is the *parallel operator*, and represents the interleaved execution of P_1 and P_2. It is used when the policies P_1 and P_2 deal with disjoint actions. The policy sequence operator $P_1; P_2$ may be implemented using the policy language's operators and control variables; see [6]. It allows to put two process behaviours in sequence. By using the constant definition, the sequence and the parallel operators, the iteration and replication operators, `i(`P`)` and `r(`P`)` resp., can be derived. Informally, `i(`P`)` behaves as the iteration of P zero or more times, while `r(`P`)` is the parallel composition of the same process an unbounded number of times.

This language is able to naturally represent even complex execution patterns. As an example, given that α, β and γ represent actions and `p`, `q` and `r` represent predicates, the following policy:

`(p(`x`).`$\alpha(x)$`.q(`y`).`$\beta(y)$`) par (r(`z`).`$\gamma(z)$`)`

allows the execution of the action α when its parameters x satisfy predicate `p`, followed by the execution of the action β when its parameters y satisfy the predicate `q`. Hence, this rule defines an ordering among the actions α and β, because β can be executed only after α. Even if the sequential operator determines the precedence

between the two actions α and β, it does not require that β is executed immediately after α. As a matter of fact, after the execution of α and before the execution of β, the policy allows the execution of γ, provided that z satisfies the predicate τ.

As in [10,11], the set of actions of the policy has been defined following an approach inspired by Zhang et al.[19]. However, here we introduce a further action: *execute(c)*. Given that the triple (s,o,r) represents the access performed by a user s to execute the operation r on the object o, the following is the list of the actions that can be used in the policy:

- $tryaccess(s,o,r)$: performed when the subject s requests the access (s,o,r).
- $endaccess(s,o,r)$: performed when the access (s,o,r) ends.
- $permitaccess(s,o,r)$: performed by the system to grant the access (s,o,r).
- $denyaccess(s,o,r)$: performed by the system to reject the access (s,o,r).
- $revokeaccess(s,o,r)$: performed by the system to revoke the access (s,o,r) previously granted while it is in progress.
- $update(s,a)$: performed by the system to update the attribute a of subject or object s.
- $execute(c)$: performed by the system to execute the command c. This action is an addition to the approach in [19] and extends the original POLPA obligations model. In our implementation, it piggybacks the command that the decision component delegates to the action performer.

For example, the following policy allows the user to access the web site *www.siteA.it*, but before this access the system requires that he should wait some time.

```
1 tryaccess(user, net, get_site(url)).
2 [(url == "www.siteA.it")].
3 execute(delay).
4 permitaccess(user, net, get_site(url)).
5 endaccess(user, net, get_site(url)).
```

In particular, line 1 of the policy refers to the access request for the A web site, and line 2 includes the authorization predicate that checks whether the URL of the site is *www.siteA.it*. In line 3 the policy asks the system to execute a command that generates a delay before actually allowing the access with the *permitaccess* action in line 4.

5.1 The Correspondence of POLPA to UCON Model

The authors of UCON defined a set of several core models that differ by the temporal position of the UCON components with respect to the execution of the access. As an example, in the $preA_3$ authorization model, the authorization phase is executed before the access, while the update of the attributes is performed after the usage. Since the POLPA language is able to define complex sequences of actions by exploiting its composition operators (i.e. sequential, alternative, parallel, iterative and replication), it is natural to encode the UCON core models in POLPA policies. It was showed [10] that POLPA's expressiveness is sufficient to model the basic features of UCON model by encoding all the basic UCON core models. In what follows we will show the example of the preAuthorization with preUpdate because is very general, as well as two widely used onAuthorizations models.

PreAuthorization with preUpdate ($PreA_1$). The preAuthorization model with preUpdate of the attribute a of the subject s is represented by the following policy.

$$tryaccess(s,o,r).$$
$$p_A(s,o,r).$$
$$update(s,a).$$
$$permitaccess(s,o,r).$$
$$endaccess(s,o,r)$$

In this policy we can see that both the evaluation of the authorization predicate $p_A(s,o,r)$ and the $update(s,a)$ action are executed by the system before issuing the $permitaccess(s,o,r)$ action, i.e. before the actual execution of the required operation.

OnAuthorization without Update (OnA_0). The following policy represents the ongoing authorization model without update. Here the predicate $\overline{p_A}$ denotes the negation of the predicate p_A, expressing the authorization condition. This policy states that after granting the permission to execute an access, if the authorization condition does not hold any more this permission is revoked even if the access is still in progress.

$$tryaccess(s,o,r).$$
$$permitaccess(s,o,r).$$
$$(endaccess(s,o,r)\ or\ (\overline{p_A}(s,o,r).revokeaccess(s,o,r)))$$

In this case, the system authorizes the access as soon as it receives the request, because the $permitaccess(s,o,r)$ action immediately follows the $tryaccess(s,o,r)$ one. However, while the access is in progress, i.e. before the $endaccess(s,o,r)$ action has been received by the system, the predicate $\overline{p_A}(s,o,r)$ is repeatedly tested, and if it is satisfied the $revokeaccess(s,o,r)$ action is executed by the system, i.e. the access is interrupted before it naturally ends.

OnAuthorization with preUpdate (OnA_1). The following policy encodes the onAuthorization model with preUpdate.

$$tryaccess(s,o,r).$$
$$update(s,a).$$
$$permitaccess(s,o,r).$$
$$(endaccess(s,o,r)\ or\ (\overline{p_A}(s,o,r).revokeaccess(s,o,r)))$$

In this policy the $update(s,a)$ action follows the $tryaccess(s,o,r)$ action and precedes the $permitaccess(s,o,r)$ action in the policy. Hence, the update is executed as soon as the access request is received, i.e. before granting the access to the resource and before evaluating the authorization predicate.

5.2 Expressing Usage Control Policies

Policies P1-P4 are expressed in POLPA in Table 1. Lines from 2 to 6 allow the execution of the security relevant action *submit_donor_data()* that requests the registration of

the donor *name* in the data base, only if the predicate in line 3 is satisfied. This predicate checks that the location of the remote service that sent the message to the Data Submitter (DS) service is equal to "Hong Kong". If it is satisfied, the registration request is allowed by the *permitaccess(source,DS,submit_donor_data())* control action in line 4, and the *execute(writelog(submit_donor_data()))* command in line 5 asks the PEP to add an entry in the "shadow" log service for this action (P3). Hence, the PEP implements the *writelog* command by invoking the log service and communicating it the data about the action received by the PDP. After the registration of the donor data has been executed (line 6), the policy either allows the request for the confirmation of the previous registration, or asks the PEP to execute the *send_notification(donor_id)* command. Lines from 10 to 14 of the policy allow the execution of the confirmation action only if the donor id paired with the confirmation request is the same of the registration action previously allowed, as stated by the predicate in line 10. If the confirmation of the registration is allowed, the *execute* command in line 13 asks the PEP to invoke the log service to add a new entry in the log file for the current action (P3). However, if the confirmation request is not received within 3 days, the predicate in line 7 of the policy, that checks the elapsed time from the registration request arrival, is satisfied and the *send_notification(donor_id)* command is sent to the PEP. The PEP implements this command by invoking the Notification Service (NS) and communicating it the id of the donor whose registration has not been confirmed. The registration request arrival time is saved in the variable *req_date*, while the current time is stored in the *system.cur_date* system attribute. Since the alternative operator (or) in line 9 allows only one between the two previous actions, if the confirmation request is not received within 3 days from the registration, the system executes the notification command, and the confirmation action is not allowed any more.

If the confirmation action has been executed (line 14), the policy waits for 300 days before requesting the cancellation of the donor data from the data base (P4). In fact, line 15 of the policy includes a predicate that is satisfied when the time elapsed from the registration action is greater than 300 days. In this case, line 16 of the policy asks the PEP to execute the command *clean(donor_id)* that invokes the data cancellation service to delete the data of the donor *donor_id* from the data base.

Finally, the *rep* operator in line 1 of the policy allows the parallel execution of any number of instances of the policy from line 2 to line 17. This means that the policy can handle any number of registration and confirmation requests in parallel.

5.3 Extending POLPA

The reason why the original language had to be extended was the need for more verdicts. As mentioned before, all current access control and usage control monitors produce a binary decision: a service request can either be allowed, because the policy does not forbit it, or can be blocked, because it is a violation of the policy. We argued that these two options are no longer sufficient in a real system - it may be that *the bad thing* cannot be prevented. If the policy writer is astute to refer to the visible effects of a violation, then there is a chance of correction. The original POLPA had only allow/deny verdicts, hence we needed to extend it with the possibility to *execute* an external trusted action. We can support the features described by Gheorghe et al. [4], by allowing

Table 1. POLPA policy showing policies P1-P4

```
01 rep(
02    tryaccess(source, DS, submit_donor_data(location,req_date,name,donor_id)).
03    [(location = "Hong Kong")].
04    permitaccess(source, DS, submit_donor_data(location,req_date,name,donor_id)).
05    execute(writelog(submit_donor_data(location,req_date,name,donor_id))).
06    endaccess(source, DS, submit_donor_data(location,req_date,name,donor_id)).
07    (  (  [(system.cur_date - req_date > 3days)].
08          execute(send_notification(donor_id)))
09    or
10       (  tryaccess(source, PD, confirm_donor_data(confirm_id)).
11          [(donor_id = confirm_id)].
12          permitaccess(source, PD, confirm_donor_data(confirm_id)).
13          execute(writelog(confirm_donor_data(confirm_id)).
14          endaccess(source, PD, confirm_donor_data(confirm_id)).
15          [(system.cur_date - req_date > 300days)].
16          execute(clean(donor_id)))
17    )
18 );
```

variable granularity to this action: it can be an atomic action (delay a message, modify message payload or metadata, etc.) or a set of ordered actions. For modularity, these actions can be expressed in a language that the PDP is agnostic of. Overall, this POLPA extension preserves its capabilities to encode the UCON model.

6 Design and Implementation

6.1 Design

The design of xESB merges the enforcement process described above with the capabilities of POLPA. As shown in Fig. 4, xESB consists of an instrumented message router that mediates the interactions between service (or resource) clients and providers. These interactions are governed by POLPA policies that are enforced by the modified NMR. For the implementation, we chose Apache Service Mix 3.3[1] as an open-source ESB platform and instrumented its NMR by adding three modules corresponding to the three basic enforcement steps.

The interceptor captures XML messages within the ESB. The POLPA policies dictate what message types and parameters to look at: message destination, source, size, or metadata like annotation information. In xESB, this functionality is implemented by extending a listener interface immediately after an XML message is fired – this allows pre-updating, pre-authorization, pre-obligation enforcement – and immediately before a message is received by a client or provider – this allows for post-updating and post-obligation management. Our implementation captures every message on the ESB, but we are aware a prefiltering mechanism is important. Figure 2 (right) shows the format of a message on the NMR; because of the normalized format, split into structured metadata and payload, the message destination and the direction of the message can be easily extracted and compared against a filter. This mechanism would separate between policy relevant and non-relevant messages.

[1] `http://servicemix.apache.org/home.html`

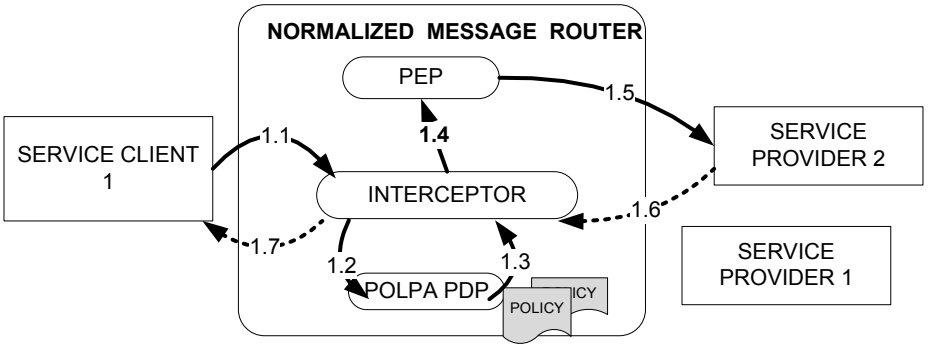


Fig. 4. The xESB design

The POLPA PDP makes the enforcement decision. It gets a message from the interceptor and evaluates it against the applicable policy (see Fig.3). In our implementation the PDP examines all policies in the policy base (or policy repository) against the current message, until the first one applies, but we are aware there is room for optimization. Once the applicable policy is found, evaluation is done by comparing message context and actual parameters (e.g., destination service, source service, message type, etc) with the conditions required by the POLPA policy. The output of the decision phase is called the *verdict* and zero or more actions. The enforcement decision is binary: the request is either allowed or rejected. While the first case implies no consequences onto the message flow, the second case calls for one or more enforcement actions that the PEP must perform. These actions are a form of $execute(c)$ as expressed above. In our implementation, the PDP cannot interpret and execute these actions.

The PEP interprets and enacts the actions specified in the previous phase. At the ESB level, we envision several custom mechanisms for usage control enforcement: **The acceptor** accepts whole messages. If the verdict does not indicate any policy violation, then the acceptor is invoked, with the effect that the message is allowed without any modification. No effective action is supplied. **The blocker** rejects whole messages. The blocker mechanism is invoked to react to a policy violation by rejecting the entire message and sending back an error message. In xESB it is implemented by routing the XML message to an internal endpoint while sending back an error message. **The executor** describes enforcement actions other than accept or block. The reaction to a violation can be either modifying, duplicating or postponing a message. *Modifying* a message acts on the fly so that it conforms to the given policy. The action specifies the pairs of field-new values. In this way, a policy like "anonymize patient id when calling a statistics service" can be enacted automatically. *Duplicating* creates a clone of each intercepted message and forwards it to a predefined logging endpoint. *Delaying* is postponing a message until a condition is satisfied. This mechanism maps to the idea of obligation enforcement, where an actor would only be allowed to perform an action only after a condition has been verified. A policy like "delay patient data deletion for 10 days" can be implemented by delaying the deletion of call traces until the arrival of a message that signals the end of the tenth day.

6.2 Experimental Results

Enforcing POLPA usage control policies introduces a delay in message forwarding, and this section evaluates this overhead within our research prototype.

As depicted in Figure 4, the xESB interceptor calls the PDP to determine the if the action included in every message can be executed. Then the PEP enforces the PDP decision either by forwarding the message to the destination service or by discarding it. Moreover, the PEP executes the obligation commands triggered by the security policy when asked by the PDP. The time to reach an enforcement decision mainly depends on two factors: the time required to exchange data between the xESB interceptor and the PDP, and the complexity of the policy to be evaluated, since complex policies will take more time to be checked than simple ones. In particular, the time to evaluate the policy to determine the right to forward the message that requests the execution of given action α, depends on the number of *tryaccess* commands concerning α in the policy, and on the number and complexity of predicates in-between these tryaccesses and the related permitaccess commands.

In fact, the PDP searches in the security policy for all the *tryaccess* commands that refer to α, and for each of them evaluates all the following conditions in the policy until the related permitaccess is reached. For example, take the policy in Appendix A. To determine whether to allow the forwarding to the Data Submitter (DS) service of the message concerning the action *submit_donor_data*, the PDP evaluates the *tryaccess* command in line 2 of the policy, and the related *permitaccess* command is in line 4. Hence, the PDP evaluates one predicate only (line 3) for *submit_donor_data*, that compares the value of a parameter of this action with a constant string. The obligations that the PDP asks the PEP to execute may cause additional delays. For example, the *writelog(..)* function in line 5 of the policy represents an obligation that is triggered by the PDP but is executed by the PEP.

In order to evaluate the overhead, we set up a number of experiments against our research prototype. Table 2 reports the results. In our tests, we let xESB forward two messages originating from the Data Collector (DC) service. The first message invokes the *submit_donor_data* method on the Data Submitter (DS) service, the second invokes the *confirm_donor_data* method on the Process Donor service (see the first and second lines of Table 2, respectively).

To show how the complexity of the security policy impacts on the overhead introduced by the UCON authorization system, for the messages previously described, we measured the decision time enforcing three distinct policies: the policy shown in Table 1 (Policy 1), and two other derived policies (Policies 2 and 3, respectively). Policy 2 is shown in Appendix A, and it has been obtained adding two new clauses to Policy 1 for the actions *submit_donor_data* (lines 21 to 25 and lines 40 to 44) and *confirm_donor_data*, (lines 10 to 14 and lines 48 to 52). The purpose of Policy 2 is to increase the complexity of Policy 1 by increasing the number of tryaccess commands that the PDP has to evaluate to decide whether to allow the *submit_donor_data* and *confirm_donor_data* actions. In fact, to decide whether to allow the forwarding of the message including the *submit_donor_data* action, the PDP should evaluate three tryaccess commands, that are in lines 2, 21 and 40, and hence the related predicates in lines 3, 22 and 41.

The purpose of policy 3, instead, is to show that the time required to evaluate the security policy for an action α depends only on the tryaccess commands that refer to α, and it is not affected by the number of tryaccess commands related to other actions. Hence, policy 3 is obtained by adding to policy 1 ten tryaccess commands that concern other actions than *submit_donor_data* and *confirm_donor_data*. For the sake of brevity we don't show policy 3 here.

Table 2. Average round-trip time for message forwarding (μs)

action name	policy 1	policy 2	policy 3
submit_donor_data	5205 (22)	5166 (24)	4939 (22)
confirm_donor_data	4036 (29)	4108 (31)	4104 (29)

We obtained the figures in Table 2 by installing the PDP on a dedicated machine on the same local network as xESB. We measured both the average decision time with network delay including time for serialization and deserialization (the first figure), and without network delay (the second figure, in parentheses). Hence the second figure measures the average authorization system overhead due to the decision process only.

We observe that the maximum delay introduced by the authorization system is around 5000 μs, and this delay is greater for the *submit_donor_data* message than for the *confirm_donor_data* message, because the first message needs the transmission of four parameters compared to only one parameter for the second message.

We also observe that the PDP decision time is negligible when compared to the delay introduced by the local network. In fact, in the case of the *submit_donor_data*, Policy 1 is evaluated in about 22 μs, while the overhead including the network delay is about 5205 μs. Hence, placing the PDP on the same machine as xESB considerably reduces the overhead.

The experimental results also confirm that the PDP decision time depends on the complexity of the security policy to be enforced. As a matter of fact, the evaluation of Policy 2 takes more time than the evaluation of Policy 1 for the two messages because for each of them, policy 2 requires the evaluation of three tryaccess commands, while Policy 1 requires the evaluation of 1 tryaccess command only. Finally, we observe that the evaluation of Policy 3 takes the same time of Policy 1. This confirms that, to authorize a given action, the PDP does not evaluate all the tryaccess commands in the policy, but it only evaluates the tryaccess commands that concern that action.

7 Related Work

Usage Control. The complexity of usage control over access control makes elaborate access control languages (e.g., Ponder [2], EPAL [1], SPL [15]) unable to express obligations as per usage control, nor decisions other than allow or block. The work of Pretschner et al. [13,14] describes a formalized usage control language and the mechanisms to enforce it, but do not cover an enforcement model for SOA. Our solution

reuses these enforcement mechanisms but applies them for the first time to the ESB level. Another related approach is that of Minsky [8], but just as [2], they consider generic distributed dedicated entities to realize the law. Our approach uses the ESB as a centralized mechanism that either performs runtime enforcement or delegates it.

Message-level standards. There are several XML security standards for message authentication, authorization, encryption and confidentiality: OASIS's WS-Security, SAML, and other WS-* specifications. They deal with narrow issues of SOA communication, and are orthogonal to our solution: they can bring generic guarantees on the security of the payload of the message and of the communicating ends. Our solution uses the message metadata for enforcing requirements on message mediation (and not only communication ends), which are adapted to the business logic.

Message-level enforcement language. Research in policy language capabilities for message-level enforcement retains open problems. Svirskas et al. suggest an XACML architecture for role-based access control enforcement [17]. Their solution employs ESB capabilities, but there is no stress on interception or mediation. Likewise, languages like Ponder2 [3] focus on authorizations without ongoing conditions, but Ponder2's obligations cannot update subject and target attributes. xDUCON tries to solve that problem with a shared data space model, but the solution remains purely abstract and stays away from a concrete policy language. Other approaches that discuss message-level enforcement do not discuss policy language capabilities [9,18,5]. Our solution uses POLPA, a process algebra policy language, within an instrumented ESB, and is shown to support ongoing usage control and obligations that affect attributes of any system user or asset. Our work is based on an existing approach [4], but while Gheorghe et al. use a customized policy language, we focus on formal enforcement capabilities, and prove the usefulness of POLPA when enforcing UCON policies.

8 Discussion and Conclusions

With the need of enforcing usage control policies at the enterprise message level, we model and implement an enterprise reference monitor enacting an extended semantics of the UCON concepts. An important contribution is to use a formally proved usage control language to express and enforce usage policies. This language allows for a set of *enforcement primitives* on secure messaging (the blocker, the modifier, the delayer, the executor). Because implementing these primitives might have different semantics from application to application, we argue that our design caters for a *customizable* ESB enforcement framework because it offers a semantic-independent wiring between the components.

As future work we plan to further extend obligations in POLPA and refine the subclasses of UCON policies enforced at the message bus level. In terms of our prototype, we plan to evaluate our implementation against different policy languages to the extend of their expressivity and against different policy complexities. Also, we plan to investigate if the extension of POLPA, initially unable to fully model all the Red Cross policies, warrants a conceptual extension of the UCON model.

Acknowledgement

The work published in this article has partially received funding from the European Community's 7th Framework Programme Information Society Technologies Objective under the MASTER project[2] contract no. FP7-216917.

References

1. Backes, M., Pfitzmann, B., Schunter, M.: A toolkit for managing enterprise privacy policies. In: Snekkenes, E., Gollmann, D. (eds.) ESORICS 2003. LNCS, vol. 2808, pp. 162–180. Springer, Heidelberg (2003)
2. Damianou, N., Dulay, N., Lupu, E., Sloman, M.: The ponder policy specification language. In: Sloman, M., Lobo, J., Lupu, E.C. (eds.) POLICY 2001. LNCS, vol. 1995, pp. 18–38. Springer, Heidelberg (2001)
3. Damianou, N., Dulay, N., Lupu, E., Sloman, M., Tonouchi, T.: Tools for domain-based policy management of distributed systems. In: NOMS, pp. 203–217 (2002)
4. Gheorghe, G., Neuhaus, S., Crispo, B.: xESB: An Enterprise Service Bus for access and usage control policy enforcement. In: 4th IFIP WG 11.11 International Conference on Trust Management (2010)
5. Goovaerts, T., Win, B.D., Joosen, W.: A flexible architecture for enforcing and composing policies in a service-oriented environment. In: Indulska, J., Raymond, K. (eds.) DAIS 2007. LNCS, vol. 4531, pp. 253–266. Springer, Heidelberg (2007)
6. Hoare, C.: Communicating sequential processes. Communications of the ACM 21(8), 666–677 (1978)
7. Katt, B., Zhang, X., Breu, R., Hafner, M., Seifert, J.: A general obligation model and continuity: enhanced policy enforcement engine for usage control. In: Proc. 13th ACM Symposium on Access Control Models and Technologies, SACMAT 2008, pp. 123–132. ACM, New York (2008)
8. Lam, T., Minsky, N.: A collaborative framework for enforcing server commitments, and for regulating server interactive behavior in soa-based systems. In: Proc. 5th Intl. Conf. on Collaborative Computing: Networking, Applications and Worksharing, pp. 1–10 (2009)
9. Maierhofer, A., Dimitrakos, T., Titkov, L., Brossard, D.: Extendable and adaptive message-level security enforcement framework. In: ICNS 2006, p. 72 (2006)
10. Martinelli, F., Mori, P.: On usage control for grid systems. In: Future Generation Computer Systems (to appear 2010)
11. Martinelli, F., Mori, P., Vaccarelli, A.: Towards continuous usage control on grid computational services. In: Proc. Intl. Conf. Autonomic and Autonomous Systems and International Conference on Networking and Services 2005, p. 82. IEEE Computer Society, Los Alamitos (2005)
12. Park, J., Sandhu, R.: The UCON$_{ABC}$ usage control model. ACM Trans. Inf. Syst. Secur. 7(1), 128–174 (2004)
13. Pretschner, A., Hilty, M., Basin, D., Schaefer, C., Walter, T.: Mechanisms for usage control. In: Proc. of 2008 ACM Symposium on Information, Computer and Comm. Sec., ASIACCS 2008, pp. 240–244. ACM, New York (2008)
14. Pretschner, A., Schütz, F., Schaefer, C., Walter, T.: Policy evolution in distributed usage control. In: 4th Intl. Workshop on Security and Trust Management (June 2008)

[2] `http://www.master-fp7.eu`

15. Ribeiro, C., Zúquete, A., Ferreira, P., Guedes, P.: Spl: An access control language for security policies with complex constraints. In: Proceedings of the Network and Distributed System Security Symposium, pp. 89–107 (1999)
16. Sun, Java Community Process Program: Sun JSR-000208 Java Business Integration, http://jcp.org/aboutJava/communityprocess/final/jsr208/index.html
17. Svirskas, A., Isachenkova, J., Molva, R.: Towards secure and trusted collaboration environment for european public sector. In: Intl. Conf. on Collaborative Computing: Networking, Applications and Worksharing, pp. 49–56 (November 2007)
18. Verhanneman, T., Piessens, F., Win, B.D., Joosen, W.: Uniform application-level access control enforcement of organizationwide policies. In: ACSAC 2005, pp. 431–440. IEEE Computer Society, Los Alamitos (2005)
19. Zhang, X., Parisi-Presicce, F., Sandhu, R., Park, J.: Formal model and policy specification of usage control. ACM Trans. on Information and System Security, 351–387 (2005)

A Appendix: Policy P2 Expressed in POLPA

```
01rep(
02    (  tryaccess(source, DS, submit_donor_data(location,req_date,name,donor_id)).
03       [(location = "Hong Kong")].
04       permitaccess(source, DS, submit_donor_data(location,req_date,name,donor_id)).
05       execute(writelog(submit_donor_data(location,req_date,name,donor_id))).
06       endaccess(source, DS, submit_donor_data(location,req_date,name,donor_id)).
07       (  (  [(system.cur_date - req_date > 3days)].
08             execute(send_notification(donor_id)))
09          or
10             (  tryaccess(source, PD, confirm_donor_data(confirm_id)).
11                [(donor_id = confirm_id)].
12                permitaccess(source, PD, confirm_donor_data(confirm_id)).
13                execute(writelog(confirm_donor_data(confirm_id)).
14                endaccess(source, PD, confirm_donor_data(confirm_id)).
15                [(system.cur_date - req_date > 300days)].
16                execute(clean(donor_id))
17             )
18       )
19    )
20    or
21    (  tryaccess(source, DS, submit_donor_data(location,req_date,name,donor_id)).
22       [(location = "Italy")].
23       permitaccess(source, DS, submit_donor_data(location,req_date,name,donor_id)).
24       execute(writelog(submit_donor_data(location,req_date,name,donor_id))).
25       endaccess(source, DS, submit_donor_data(location,req_date,name,donor_id)).
26       (  (  [(system.cur_date - req_date > 3days)].
27             execute(send_notification(donor_id)))
28          or
29             (  tryaccess(source, PD, confirm_donor_data(confirm_id)).
30                [(donor_id = confirm_id)].
31                permitaccess(source, PD, confirm_donor_data(confirm_id)).
32                execute(writelog(confirm_donor_data(confirm_id)).
33                endaccess(source, PD, confirm_donor_data(confirm_id)).
34                [(system.cur_date - req_date > 600days)].
35                execute(clean(donor_id))
36             )
37       )
38    )
39    or
40    (  tryaccess(source, DS, submit_donor_data(location,req_date,name,donor_id)).
41       [(location = "UK")].
42       permitaccess(source, DS, submit_donor_data(location,req_date,name,donor_id)).
43       execute(writelog(submit_donor_data(location,req_date,name,donor_id))).
44       endaccess(source, DS, submit_donor_data(location,req_date,name,donor_id)).
45       (  (  [(system.cur_date - req_date > 3days)].
46             execute(send_notification(donor_id)))
47          or
48             (  tryaccess(source, PD, confirm_donor_data(confirm_id)).
49                [(donor_id = confirm_id)].
50                permitaccess(source, PD, confirm_donor_data(confirm_id)).
51                execute(writelog(confirm_donor_data(confirm_id)).
52                endaccess(source, PD, confirm_donor_data(confirm_id)).
53             )
54       )
55    )
56);
```