A Flexible Architecture for Enforcing and Composing Policies in a Service-Oriented Environment

Tom Goovaerts, Bart De Win, and Wouter Joosen

DistriNet Research Group, Katholieke Universiteit Leuven
Celestijnenlaan 200A, 3001 Leuven, Belgium
{tom.goovaerts,bart.dewin,wouter.joosen}@cs.kuleuven.be

Abstract. Service Oriented Architectures (SOA's) enable powerful application and end user service composition from independently defined services. The effective deployment of such composed services requires adaptation of and interoperability between services. This challenge can be approached by specifying service composition in policies, and by enforcing these policies in a sophisticated run-time architecture.

In this paper, we present an open architecture for enforcing and composing complex policies that can depend on the available services in the environment. Complex polices have typically been studied in the context of policy languages, yet they have never been fully supported in a SOA-based execution environment. We have created a flexible run-time architecture that maximizes interoperability, adaptability and evolution. We have prototyped our architecture on an Enterprise Service Bus and we illustrate how our solution supports realistic and complex policies.

1 Introduction

Services are the fundamental building blocks of software systems when applying the Service-Oriented Computing (SOC) paradigm [17]. Services expose a well-defined behavior in independent units of business logic and are used and deployed in complex compositions to create distributed business applications. The applications that emerge in service oriented architectures can become large and fairly complex, and can be interconnected with services from various organizations and stakeholders. In this context, describing and enforcing acceptable (correct, permitted, manageable, affordable, etc.) compositions have become key challenges.

Policy languages and subsystems that ensure the enforcement of policies have become an increasingly important sub domain in distributed systems and middleware as they deal with the above mentioned challenges. Systems need to be able to comply with an ever growing set of business rules and regulations that are subject to continuous change. Policies are rules that specify choices in the behavior of a system [4]. By specifying these rules separately from the applications, the behavior can be changed dynamically by modifying the policy rules

J. Indulska and K. Raymond (Eds.): DAIS 2007, LNCS 4531, pp. 253–266, 2007.

without affecting any application code. Policy-based systems most often adhere to the XACML dataflow model [16], which consists of policy decision points (PDPs), policy enforcement points (PEPs) and optionally policy information points (PIPs). A PDP focuses on how policy rules are evaluated given a certain state of the system. A PEP handles the provisioning of system state information to the PDP and the execution of the correct semantics of the policy decisions that are returned by the PDP. The PIP provides additional context information to support the decision making process of the PDPs. The most familiar types of policies are probably in the area of authorization. Other examples of policies include user preference policies that govern user-configurable behavior of a system, privacy policies that contain privacy rules and SLA/SLO policies that deal with elements of quality of service.

In most cases, policies are defined and enforced at the level of individual resources such as objects or components. However, a SOA introduces an abstraction layer of services that are indirectly related to the underlying resources. Therefore, new types of policies arise that make use of services. Moreover, the underlying resources that are being interconnected may be implemented on different heterogeneous systems that are unaware of each other. For these reasons, the emerging policies cannot exclusively be enforced at the level of the underlying systems, or at the level of underlying resources.

In other words, the kinds of policies that can be supported by straightforward adoption of existing policy technology are often restricted in several ways. First: it is hard to enforce application-level policies that require information that is contained in multiple distinct services. Secondly, policy enforcement mechanisms are often tightly coupled with both the specific middleware platform and/or with specific policy-related technologies. Due to the openness and very frequent evolution of a service oriented environment, it needs to be able to interface with a range of policy languages, policy servers, message formats and functional services.

This paper addresses the gap between existing message oriented service platforms and known policy systems by offering policy enforcement as a service. A message oriented service platform is a specific type of message oriented middleware that is based on message interception capabilities. Such architecture therefore is agnostic on specific message formats and policy languages. Our architecture maximizes the reuse of policy decision logic and of enforcement logic. Due to its flexibility, the architecture can be used to implement fine-tuned policy enforcement points in multiple operational contexts.

The main contribution of this paper is an open architecture for enforcing and composing complex policies that can depend on the available services in a SOA. To the best of our knowledge, complex polices have typically been studied and supported in the context of policy languages, yet they have never been fully supported in a SOA-based execution environment. We have created a flexible run-time architecture that maximizes interoperability, adaptability and evolution. We have prototyped our architecture on an Enterprise Service Bus (ESB). The prototype demonstrates policy enforcement for SOAP messages in a

telecom-centric ESB. We show how our solution improves interoperability and flexible adaptation in a service oriented environment.

The rest of the paper is structured as follows. Section 2 elaborates on the problem domain by presenting some representative policies. Section 3 then summarizes the requirements for a policy enforcement architecture that can manage complex polices. Section 4 presents our solution: the architecture, an illustration of policy enforcement in the architecture and the prototype implementation. We evaluate our solution in Section 5 and compare with related work in Section 6. Then we conclude.

2 Motivating Example

We illustrate the kinds of high level policies that need to be enforced by means of a concrete policy set. Suppose we have a set of three services: an Address Book service that keeps track of a contact list, a Call service that can be used to setup phone calls and a Location service that can be used to lookup the current location of a given user. Consider the following policy set:

1. Everyone can view all address book records, but one can only modify the contact information of its own record.
2. Address book records can only be modified when the user is at its desk. When a user tries to modify its address book record when he/she is at home, deny this and audit the attempt.
3. Only allow calls to someone's work phone during office hours. If someone calls a user on its work phone during office hours, but the user is not located in the office, reroute the call to the user's mobile phone.

The first rule illustrates the fact that policies can be based on the contents of a message. In this case, the message will contain an argument that determines the target record that will be modified. The second rule illustrates the fact that the enforcement of rules that concern one service (in this case the Address Book service) may need information contained in other services (the Location service). Moreover, the second rule illustrates that policies might specify complex results that can contain obligations. Obligations are tasks that need to be fulfilled by the system upon enforcement of a policy decision. The third rule shows that a high level policy rule might actually consist of different kinds of policies that need to be combined. The first part of the policy rule specifies an authorization while the second part is a typical business rule. Therefore, this policy rule will normally be split up (or at least, it may be implemented) in different languages and with different decision mechanisms.

These services are loosely coupled to each other: they could be implemented by a different underlying platform or could even belong to an external party. The implementations of the services are fully unaware that they are being integrated with specific other services. For these reasons, the enforcement of such policies needs to be performed at the level of the platform that hosts the services.

3 Requirements

This section describes the most important characteristics that drive the architectural design of our policy enforcement solution. We assume that policies are contained in *policy services* that support the making of policy decisions and thus function as PDPs.

Advanced policy support. Advanced service-level policies such as the example policies from Section 2 should be supported. What is characteristic for these policies, is that their enforcement might need the invocation of other services. This consists of:

- *Information provisioning* ensures that policy services have access to all the information they need in order to make a policy decision. This information can be contained in the message itself or it can be contained in functional services.
- *Decision execution* is the execution of the decision(s) that are returned by a policy service. In contrast to an authorization decision that declares a binary allow/deny result, the execution of a general policy decision can be a complex operation. Policy decisions can also contain obligations that need to be executed by the system in addition to the decision itself.

Interoperability with policy services and message formats. A SOA interconnects a set of heterogeneous systems that may use different messages and formats. Moreover, it is important that different policy languages and engines can be supported. Therefore, a policy enforcement solution needs to be interoperable with multiple message formats and with multiple policy services.

Flexibility. Because the operational environment is subject to frequent evolution, it is necessary that a policy enforcement solution is able to be adapted to these changes. More specifically:

- *Changing and combining policy services* It should be possible to easily change policy services that are specific to one language. Moreover, it is possible that policy enforcement for one message requires the combination of decisions from multiple policy services. Therefore, an enforcement solution should allow policy services to be changed and combined with each other.
- *Flexible binding with the operational environment* The policy services and their policies should be made independent of all environment-specific aspects. Therefore, it must be possible to change the binding of the policies with the environment. This binding consists both of information sources containing policy-relevant data and of the execution logic of the policy decisions and their obligations.

Performance. Since policy enforcement needs to operate on messages, it may become an unacceptable performance bottleneck. Therefore, the runtime enforcement overhead should be minimized.

The first and last requirements ensure that policy enforcement is possible and feasible in practice. The second and third requirements make sure that the

architecture can deal with changes in the environment. These requirements are used as the basis for the architectural design that is discussed in Section 4.

4 Architecture

In this section, the policy enforcement architecture is presented. Since the service is the basic building block in the environment, we have chosen to offer the policy enforcement functionality itself as a service, which we call the 'policy enforcement service' or simply the 'enforcement service'. In Section 4.1, we elaborate on the architectural design that was driven by the requirements from Section 3. In Section 4.2, the architecture is applied to the example policies from Section 2 and in Section 4.3 we discuss our prototype.

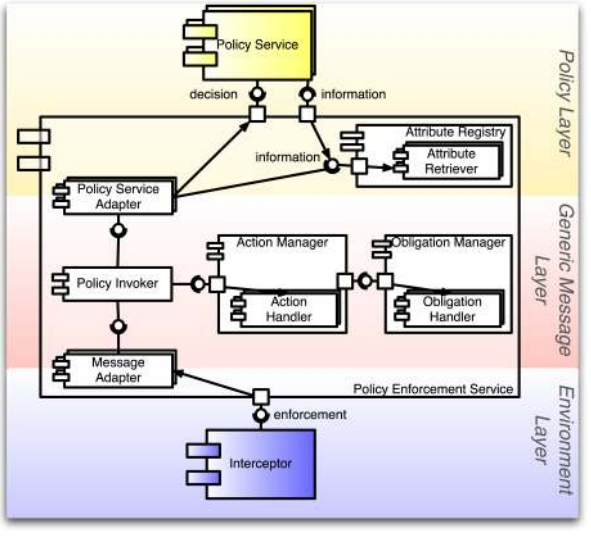


Fig. 1. Overview of the architecture

4.1 Architectural Design

Figure 1 depicts the main structure and the components of the architecture. The architecture is structured in three layers. From top to bottom, the *Policy Layer* focuses on managing and reasoning about policy rules. The *Generic Message Layer*, which is the most important layer of the architecture, is responsible for connecting execution contexts with high-level policy services (by means of a generic message format), for coordinating the invocation of policy services and for executing policy decisions. Finally, the *Environment Layer* represents the operational environment. Since each layer operates with different representations, adapter components are introduced to bridge between the different layers.

From a high-level perspective, the architecture operates as follows. All messages in the environment that require policy enforcement are forwarded to the policy enforcement service by means of an *Interceptor* component. *Policy services* contain the effective policies and return policy decisions over a native protocol. For each relevant policy service, the enforcement service creates a policy request based on the contents of the message, sends it to the policy service and obtains a policy decision. Subsequently, the policy decision is enforced and the next policy service is consulted and so on. The final output of the enforcement service will always be another message, which is usually a transformation of the incoming message.

In terms of the XACML model, the combination of the Interceptor and the Policy Enforcement Service function as PEP, the Policy Enforcement Service functions also as PIP and the policy services function as PDPs. The core components of the architecture, contained in the architectural component as indicated on Figure 1, will now be discussed in more detail.

Policy Layer

Policy Service Adapters. The integration of (possibly external) policy services is enabled by *Policy Service Adapters*. A Policy Service Adapter inspects an incoming message and sends a request to its policy service based on the contents of the message. Subsequently, it returns the result of the policy service to the architecture in the form of a *policy decision*. A policy decision consists of a set of *actions* that each can contain a set of *obligations*. Actions and obligations are abstract task descriptions that are defined by an identifier and a set of arguments. Obligations are tasks that should be enforced in conjunction with the enforcement of an action. For instance, an 'audit' obligation can be attached to a 'deny' action, indicating that a negative authorization should be audited.

Attribute Registry. Policy services sometimes require access to contextual information that is not contained in the message itself. Examples of such information are the location of an end-user or the uptime of a service. For this purpose, the Attribute Registry is introduced. The Attribute Registry is a simple attribute repository that can be queried by the Policy Service Adapters or by Policy Services themselves. It actually binds the abstract information that is used in the policies with the concrete environment. The retrieval logic for a particular attribute is contained in Attribute Retrievers and can be plugged in upon integration with a particular environment.

Generic Message Layer

Generic Message Model. The policy enforcement service needs to bridge a variety of message formats on the one hand, and different types of requests for policy services on the other hand. In order to deal with this N-to-M mapping, a common message representation, the *generic message format*, is introduced. A generic message consists of the subject responsible for sending the message, the action that is being targeted by the message and the target service for the

message. For instance, if user 'X' requests an address on an Address Book service, the generic message will consist of subject X, action 'getAddress' and target 'Address Book'. Each of these elements is represented by an identifier and a set of key-value pairs called attributes. The architecture assumes that all messages in the environment at least contain identifiers for these three concepts.[1] The generic message also contains a reference to the original message. Within the architecture, each generic message is wrapped in a *Message Context*, which is used, among others, to maintain state over the invocation of multiple policy services.

Policy Invoker. The Policy Invoker expects an incoming generic message and is responsible for coordinating the invocation of the different policy services. The Policy Invoker holds a sequential chain of Policy Service Adapters to determine the order in which the policy services are consulted and their decisions are enforced. After a Policy Service Adapter returns a decision, the Policy Invoker passes it to the Action Manager. When the actions are enforced, the next Policy Service Adapter gets to process the message and the process is repeated.

If multiple policy services are chained, one policy service might need to use information that has been generated by a decision of a previous policy service. While the architecture does not provide support for the semantics of such metadata, it does allow a Policy Service Adapter to influence the decisions of its successors in the following ways:

1. By modifying the original message. This can result in new or modified attributes of the subject, action or target service of the message.
2. By adding attributes to the generic message. These modifications only live as long as the enforcement service handles the message.
3. Through the Message Context. The Message Context consists of a set of key-value pairs that can contain arbitrary metadata that is not related to the subject, action or target service.

Action Manager & Obligation Manager. Actions and obligations in a policy decision specify *what* functionality needs to be enforced. The architecture needs to know *how* to enforce these actions and obligations in a concrete environment. This is the responsibility of the *Action Manager*. Action execution logic is delegated to *Action Handlers*: the Action Manager associates action identifiers to the Action Handlers that are responsible for executing them. Modification of the original message (e.g., for an 'encrypt' action) is supported through the generic message's reference to the original message. Obligations are handled similarly: an *Obligation Manager* consisting of a set of *Obligation Handlers* executes the obligations contained in each action.

[1] While this consideration makes sense at a conceptual level, it is possible that these elements are not explicitly represented in the actual messages. In these cases, adaptations to services or middleware infrastructure should take care of attaching the appropriate metadata to the messages.

Environment Layer

Message Adapters. Messages from the environment layer in a specific format are converted into generic messages by *Message Adapters*. It is possible that the original message is changed by a policy decision. If this happens, the Message Adapter needs to synchronize the generic message with the original message in order to represent the potentially altered subject, action and target attributes.

4.2 Enforcement of Example Policies

We illustrate the architecture by describing the enforcement of the three example policies from Section 2. Figure 2 illustrates the execution flows in the architecture. Since the example policies contain authorization rules as well as business rules, we assume that there are two different policy services. The authorization service returns 'allow'/'deny' decisions (potentially including an 'audit' obligation) and the business rule service returns a 'redirect' decision. Therefore, the Action Manager is configured with 'allow', 'deny' and 'redirect' Action Handlers and the Obligation Manager is configured with an Audit Obligation Handler. Furthermore, the Attribute Registry is configured with two Attribute Retrievers: one that fetches the current time and one that contacts the Location Service for getting the location of a user with a given identifier.

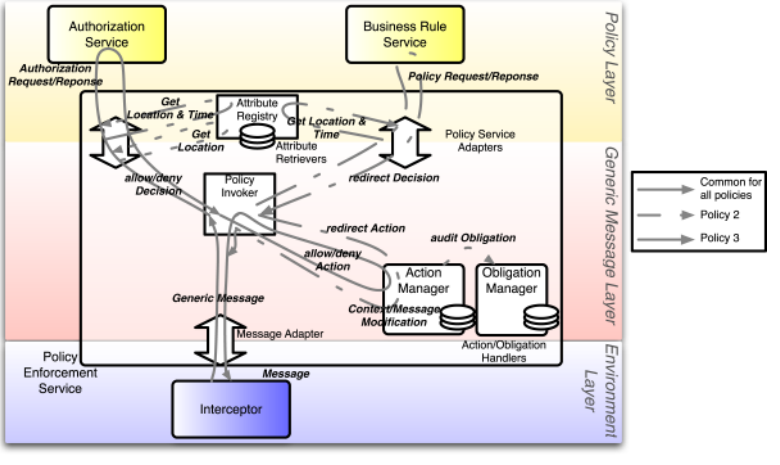


Fig. 2. Execution flow throughout the architecture applied to the example policies

The solid gray arrows show the minimal flow that is followed for enforcing all three policies. First of all, the incoming message from the bus is translated into a generic message by the message adapter. In our case, the target of the generic message can be either the Address Book service or the Call service.[2] Then,

[2] The Location service is only used within the policies for gathering the location attribute for the subject.

the generic message is forwarded to the Policy Invoker that contains a chain of two Policy Service Adapters. The generic message is wrapped in a Message Context and is sent to the Authorization Service Adapter. For the first rule, no additional attributes have to be gathered from the Attribute Registry, but both the second and the third rule do require the Attribute Registry. In the example, the adapter fetches these attributes proactively. After the adapter has created and sent a policy request from the generic message, the policy service returns a decision. In this case, the decision will contain either an 'allow' or 'deny' action without parameters and optionally an 'audit' obligation with the audit message as parameter. The decision is returned to the Invoker, which forwards it to the Action Manager. The Action Handlers execute the decision, potentially modifying the Message Context or the message itself. If an obligation is included, the Action Manager invokes the Obligation Manager.

For the third rule, an additional Adapter is required. The invocation of the Policy Service is similar to the previous case, except that the decision can be a 'redirect' action with the new telephone number as argument. Again, the invoker sends this action to the Action Manager which executes it and has the potential to modify the target telephone number in the original message.

4.3 Prototype

We have implemented a prototype of the architecture and we have validated it on an ESB-based telecom service platform in the context of the T-CASE project [9]. An Enterprise Service Bus (ESB) is a message oriented middleware for the integration of enterprise applications that in itself is architected in a service-oriented way. The ESB is used to mediate SOAP messages between a set of simple Web Services, such as an Address Book service, a Jabber service and a Calendar service. The interception logic is implemented by a message routing service that is called the Content-Based Router. The policy enforcement service itself is integrated by means of a SOAP interface.

It is expected that the SOAP messages on the ESB at least contain the following information:

- The authenticated identity of the user represented by a SAML [14] authentication assertion in a WS-Security [15] header.
- The action and the target service are represented in a WS-Addressing [23] header as `<Action>` and `<To>` elements respectively.

Since the ESB only mediates SOAP messages, a single SOAP Message Adapter has been implemented. For efficiency reasons, the body of the SOAP message is not processed, but the full SOAP message does get attached to the generic message so that it can be used later on.

Two Policy Service Adapters have been implemented. The first one verifies the SAML assertion that is included with the message. A separate authentication server is used to authenticate users in advance and generate these assertions. The second adapter wraps a rule-based policy service and returns authorization decisions that contain either 'permit' or 'deny' actions. The 'deny' action is

enforced by a SOAP-specific Action Handler that replaces the original message by a SOAP Fault message that is directed to the original sender.

The implementation of the Attribute Registry consists of a set of Attribute Retrievers that are responsible for looking up attributes for demonstration purposes. Attribute Retrievers can inspect the SOAP message, for instance to lookup information that is contained in the body.

5 Evaluation and Discussion

In this section each of the requirements of the architecture is evaluated and some interesting points of discussion are put forward.

Advanced Policy Support. The information that can flow to the policy services consists of two categories: information that is pushed towards the policy services and the information that is pulled from the environment. The former is realized by the generic message model and the Message Adapters and the latter is realized by the Attribute Registry. Some information such as the parameters of a message can be pushed as well as pulled. The choice for pulling such an attribute from the Attribute Registry is mainly driven by performance reasons, since pushing an attribute introduces an overhead for every single message.

Interoperability with Policy Services and Message Formats. Interoperability is achieved by inserting the adapter components (Message Adapters and Policy Service Adapters) that translate back and forth between the generic format and native formats. In our prototype, we chose to focus on SOAP messages. However, we are confident that, as long as messages contain the right set of metadata, it is feasible to write a Message Adapter for them. Concerning policy service interoperability, we have implemented one adapter for an authentication policy service and one adapter for a business rule-based policy service.

Flexibility

- *Changing and Combining Policy Services* The combination of multiple policy services is supported by sequentially chaining Policy Service Adapters, which may cause conflicts. The architecture has no explicit support for conflict detection or resolution: we assume that conflicts are solved at the policy layer. If two or more Policy Service Adapters have a large semantic overlap, they should be integrated in a single adapter that is capable of resolving conflicts.
- *Flexible Binding with the Operational Environment* The policy enforcement service effectively binds heterogeneous policy services with functional services by presenting a generic policy-centric view of the environment to the policy services. This view consists of an information part, which is realized by the generic message format and the Attribute Registry, and of an enforcement part, which is realized by the Action Manager and Obligation Manager. Flexibility of the mapping between this abstract view and the concrete semantics – *how* to enforce an action, *how* to retrieve an attribute, etc.

– is supported by isolating this logic in replaceable components (Message Adapters, Attribute Retrievers, Action Handlers and Obligation Handlers). Throughout the development of the prototype, this flexibility proved to be very useful: numerous transitions and additions of logic were made to support new kinds of policies.

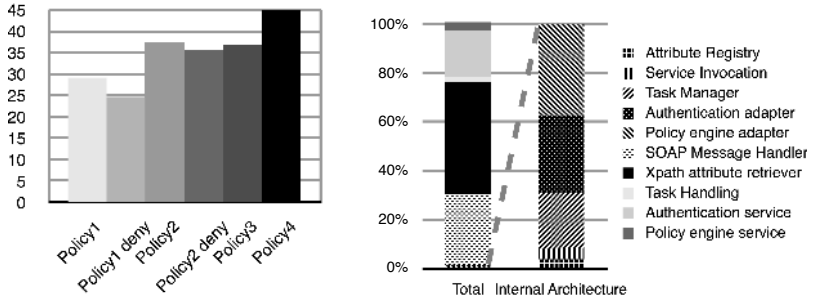
If the semantics depend on specific functional services (eg. send a warning message to a user's mobile phone or get the bank account number of a user), new instances of these components have to be written and need to be plugged into the enforcement service. The architecture currently has no support for automating this problem and thus dealing with it in a fully generic way.

Performance. The performance of the prototype has been evaluated at two levels. At a macroscopic level, the average response times have been measured for a set of incoming messages. A set of 4 policy rules of increasing complexity has been created that allowed us to trigger the execution of specific and predictable components. The test has been performed in six phases in which the response times for a specific and known message has been measured. The first two policies have also been tested for a 'deny' decision. The results of the macroscopic test are shown in Figure 3(a). The enforcement of a 'deny' result is slightly more efficient since it does not induce duplication of the original message. The biggest performance hit is introduced by Policy2 and Policy4 and is caused by the introduction of additional Attribute Retrievers.

Figure 3(b) gives an insight into the microscopic performance overhead. The left bar shows the percentage of the total time spent in each of the components during the execution of all six test phases. The major performance overhead is introduced by two components that operate on the SOAP messages: the SOAP Message Adapter and the XPath Attribute Retriever. The second bar zooms in on the lower part of the left bar and shows that the internal components of the enforcement service (including the policy service adapters) only account for less than 5% of the total overhead. This means that there is still much room for improvement for optimized implementations of the adapters and Atribute Retrievers.

The current design of the architecture has some limitations as well, which will be discussed next. Note that addressing these limitations involves a trade-off that depends on the deployment environment, as they increase the complexity and, hence, the execution time of the architecture.

The chaining of Policy Service Adapters only supports the combination of policy services in a static way such that chains cannot be changed dynamically at runtime. The latter could be useful however, among others, to support scenarios in which the outcome of one policy service can have an impact on the other services to be consulted. The support for dynamic policy service combination requires the extension of the Policy Invoker component with a meta-policy that specifies the effective sequence of policy services based on runtime information. The Policy Invoker needs to enforce this meta-policy before each invocation of a Policy Service.



(a) Macroscopic: Average response times over 1000 messages (in ms) for increasingly complex policies.

(b) Microscopic: Average relative time spent (in %) in each of the components.

Fig. 3. Performance measurements

Related to this, the composition of policy services is currently purely sequential. Other useful composition strategies exist [11] such as parallel or hierarchical composition. The former could improve the execution speed of the architecture and the latter could be used to support the semantic composition of policy rules. While these strategies are realizable at the level of the enforcement architecture in the Policy Invoker, they are often also supported at the level of individual policy services, in which case they are fully transparent for the enforcement architecture.

6 Related Work

In the field of policy languages, some languages such as Rei [10] and XACML [16] put more focus on the language features itself than on the enforcement aspect. Some notable languages that do consider enforcement in detail are Ponder [4,7] and KAoS [22]. These languages offer excellent support for specifying and combining advanced policies, but they assume that a single policy language governs the whole environment. Our work is capable of integrating multiple languages.

We take a centralized approach to policy enforcement: policies are enforced near the services they govern. Some kinds of policies such as refrain policies [4] require client-side enforcement. If interceptor components can be placed at the client side, these kinds of policies can also be supported. In the context of large scale SOA's such as Web Services or Grid systems, policy enforcement is often decentralized [19,5]: effective properties of an interaction are negotiated at runtime by the semantic matching of client requirements with service offerings. Our architecture can enforce the outcome of these policies at each peer once they are negotiated.

When enforcing security policies for isolated applications, the PDP and PEP are often merged and integrated by instrumenting the application's code [20,1].

While this approach is very efficient, it is difficult to apply it to an open environment where policies and applications evolve rapidly.

Evolution and openness requires the strict separation of PEPs and PDPs. In the access control field, there are two major directions that promote this separation: PDPs can be offered through a uniform API (for example, the Authorization (AZN) API from the Open Group [8] and the Java Authorization Contract for Containers (JACC) [21]) or they can be offered as a distributed service (such as the Resource Access Decision Facility (RAD) [2] and Tivoli Access Manager [12]). Pulling decision logic out of the application increases interoperability and flexibility, at the expense of making it harder to enforce advanced kinds of application-level policies. Our work is a first step towards bridging this gap.

Message interception is a well known technique for separating policy logic from application logic that is generic and flexible (for example, see [3,18]). Other authors have used Aspect-Oriented Programming (AOP) [13] for integrating security and policies with applications [6,24]. AOP is situated at a higher level of abstraction but is less generic, which makes it hard to apply it in a heterogeneous environment.

7 Conclusion

In this paper, we have presented an open architecture for enforcing advanced policies in a service-oriented environment. The architecture can be used to instantiate flexible policy enforcement points; it can handle realistic policies that are required by state-of-the-art Service Oriented Architectures. Our solution supports adaptation and evolution of the platform, the service composition and the specific policies. We have prototyped and validated our architecture on a telecom-centric ESB.

Future work includes further validation of our architecture. More specifically, the interoperability with existing environments and policy languages will be studied in more detail. In addition, the challenge of managing the information flow of policy-relevant data will be studied.

References

1. Bauer, L., Ligatti, J., Walker, D.: Composing Security Policies with Polymer. In: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, pp. 305–314 (2005)
2. Beznosov, K., Deng, Y., Blakley, B., Burt, C., Barkley, J.: A Resource Access Decision Service for CORBA-based Distributed Systems. In: Proceedings of the 15th Annual Computer Security Applications Conference, p. 310 (1999)
3. Damiani, E., De Capitani di Vimercati, S., Paraboschi, S., Samarati, P.: Securing SOAP e-services. International Journal of Information Security 1(2), 100–115 (2002)
4. Damianou, N., Dulay, N., Lupu, E., Sloman, M.: The Ponder Policy Specification Language. Lecture Notes in Computer Science 2001, pp. 18–38 (2001)

5. Dan, A., Dumitrescu, C., Ripeanu, M.: Connecting Client Objectives with Resource Capabilities: an Essential Component for Grid Service Managent Infrastructures. In: Proceedings of the 2nd International Conference on Service Oriented Computing, pp. 57–64 (2004)
6. D'Hondt, M., Jonckers, V.: Hybrid Aspects for Weaving Object-Oriented Functionality and Rule-Based Knowledge. In: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development, pp. 132–140 (2004)
7. Dulay, N., Lupu, E., Sloman, M., Damianou, N.: A Policy Deployment Model for the Ponder Language. Integrated Network Management Proceedings, 2001 IEEE/IFIP International Symposium on, pp. 529–543 (2001)
8. The Open Group. Authorization (AZN) API. Open Group Technical Standard C908 (2000)
9. Interdisciplinary Institute for BroadBand Technology. T-CASE Project (Technologies and Capabilities for Service-Enabling) (2005) `https://projects.ibbt.be/tcase/`
10. Kagal, L.F., Joshi, T.A.: A Policy Language for a Pervasive Computing Environment. Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop on, pp. 63–74 (2003)
11. Kanada, Y.: Taxonomy and Description of Policy Combination Methods. In: Proceedings of the International Workshop on Policies for Distributed Systems and Networks, pp. 171–184 (2001)
12. Karjoth, G.: Access Control with IBM Tivoli Access Manager. ACM Transactions on Information and System Security 6(2), 232–257 (2003)
13. Kiczales, G.: Aspect-Oriented Programming. ACM Computing Surveys 28, 232–257 (1996)
14. OASIS. Security Assertion Markup Language Specification, Version 1.1 (2003)
15. OASIS. Web Services Security: SOAP Message Security, Version 1.0 (2004)
16. OASIS. eXtensible Access Control Markup Language (XACML) Version 2.0 (2005)
17. Papazoglou, M., Georgakopoulos, D.: Service-Oriented Computing: Introduction. Communications of the ACM, vol. 46(10) (2003)
18. Ritter, T., Schreiner, R., Lang, U.: Integrating Security Policies via Container Portable Interceptors. IEEE Distributed Systems Online, vol. 7 (2006)
19. Schlimmer, J., et al.: Web Services Policy Framework Specification, Draft Version (2004)
20. Schneider, F.B.: Enforceable Security Policies. ACM Transactions on Information and System Security 3(1), 30–50 (2000)
21. Sun Microsystems. Java Authrozation Contract for Containers (JACC) Version 1.0 (2003)
22. Uszok, A., Bradshaw, J., Jeffers, R., Suri, N., Hayes, P., Breedy, M., Bunch, L., Johnson, M., Kulkarni, S., Lott, J.: KAoS Policy and Domain Services: Toward a Description-logic Approach to Policy Representation, Deconfliction, and Enforcement. Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop on, pp. 93–96 (2003)
23. W3C. Web Services Addressing, W3C Member Submission (2004)
24. De Win, B.: Engineering Application-level Security through Aspect-Oriented Software development. PhD thesis, Katholieke Universiteit Leuven (2004)