ANR programme ARPEGE 2008

Systèmes embarqués et Grandes Infrastructures

Projet SELKIS : Une méthode de développement de systèmes d'information médicaux sécurisés : de l'analyse des besoins à l'implémentation.

ANR-08-SEGI-018

Février 2009 - Décembre 2011

Specification of translation rules from a PIM to a PSM for access control policies models

Livrable 5.1

Michel Embe Jiague
Jérémy Milhau
Régine Laleau
Frédéric Gervais

LACL

February 2011

Agence Nationale de la Recherche GIP ANR

Abbreviations

AAST *annotated abstract syntax tree*

AC Access Control

ASTD Algebraic State Transition Diagram

ATL *ATLAS Transformation Language*

BPCL *Business Process Constraint Language*

BPEL Business Process Execution Language

CSP Communicating Sequential Processes

DynAC Dynamic Access Control

EB^{3}SEC EB^{3}Secured

ESB Enterprise Service Bus

HTTP Hypertext Transfer Protocol

IS Information System

IS *information systems*

LOTOS Language Of Temporal Ordering Specification

MDA *Model Driven Architecture*

OASIS Organization for the Advancement of Structured Information Standards

OCL Object Constraint Language

PAP Policy Administration Point

PDP Policy Decision Point

PEM Policy Enforcement Manager

PEP Policy Enforcement Point

PIP Policy Information point

PIM Platform Independant Model

PSM Platform Specific Model

RBAC Role Based Access Control

SAC Static Access Control

SELKIS SEcure heaLth care networKs Information Systems

SOA Service Oriented Architecture

SOAP Simple Object Access Protocol

SoD *separation of duty*

SoDA *separation of duty algebra*

WS Web Service

WSDL Web Services Description Language

WS Web services

WS-BPEL Web Services Business Process Execution Language

XACML eXtensible Access Control Markup Language

XML eXtensible Markup Language

XPath XML Path Language

XSD XML Schema Document

Contents

Abbreviations 2

List of Figures 5

I Platform Specific Model (PSM) Metamodel 6

1 Architecture and Target Platform 6

2 Class diagrams 8
- 2.1 The message hierarchy . . . 9
- 2.2 The SAC package . . . 10
- 2.3 The DynAC package . . . 10

3 Sequence diagrams 11

II Platform Independant Model (PIM) Metamodel - B for Access Control 12

III From ASTD Access Control Policies to WS-BPEL Processes Deployed in a SOA Environment 14

4 Introduction 14

5 Expressing Security Rules with an ASTD 15

6 Architecture and Target Platform 16

7 Transforming an ASTD Access Control Specification into a BPEL Process 17
- 7.1 The BPEL Process Language . . . 17
- 7.2 Transformation Rules from ASTD to BPEL . . . 18
- 7.3 $\mathcal{L}1$: An Intermediate Language between ASTD and BPEL . . . 18
- 7.4 From ASTD to $\mathcal{L}1$ . . . 20
- 7.5 From $\mathcal{L}1$ to BPEL . . . 20
- 7.6 Generating the WSDL Interface . . . 21
- 7.7 From $\mathcal{L}1$ to XSD Type Definitions . . . 21
- 7.8 Implementation of Transformations with ATL . . . 22

8 Related Work 23

9 Conclusion 24

References 26

List of Figures

1 A typical SOA application . . . 6
2 PDP abstract internal view . . . 7
3 PEP and PDP security schema . . . 7
4 Subset of the PEM class diagram . . . 8
5 The PDP uses filters . . . 8
6 PSM message hierarchy . . . 9
7 The base access control parameters of the PSM . . . 9
8 Static AC filtering classes . . . 10
9 Dynamic AC filtering classes . . . 10
10 PEP sequence diagram . . . 11
11 PDP sequence diagram . . . 11
12 Static AC PIM Metamodel . . . 12
13 Dynamic AC PIM Metamodel . . . 13
14 Algebraic State Transition Diagram (ASTD) examples . . . 15
15 ASTD pattern for permission . . . 16
16 ASTD pattern for obligation . . . 16
17 A typical SOA application . . . 16
18 PDP abstract internal view . . . 17
19 PEP and PDP security schema . . . 17
20 Permission BPEL code skeleton . . . 18
21 Obligation BPEL code skeleton . . . 18
22 Construction of the *annotated abstract syntax tree* (AAST) . . . 19
23 *Message* . . . 20
24 *Sequence* . . . 20
25 WSDL document for the PDP interface . . . 21
26 WSDL code for an event signature . . . 22
27 XSD code for an enumeration . . . 22
28 XSD code for an interval . . . 22
29 Transforming a model M_1 into a model M_2 using ATL . . . 23

Introduction

In order to provide an implementation of access control policies in the *Model Driven Architecture* (MDA) approach, a PIM and a PSM metamodels are required. Then we can express translation rules between the two levels. Our PIM language for static access control rules is UML/B and for dynamic access control rules is the ASTD notation.

In this report, we present the PSM metamodel of our target architecture for the PSM level. Then we present a PIM metamodel combining static and dynamic B access control specifications. Finally we introduce a translation mechanism from ASTD specifications to Business Process Execution Language (BPEL) processes following the PSM metamodel.

Part I

PSM Metamodel

In the WP4 of the SELKIS project proposal, Information Systems (ISs) are implemented using Web services (in the broad sense). Security features are implemented in the Policy Enforcement Manager (PEM). Those services rely on data available in relational databases or eXtensible Markup Language (XML) based files. Our security framework is based on two mains actors: the Policy Enforcement Point (PEP) and the Policy Decision Point (PDP). They are responsible of intercepting client application's requests to services and applying security policies on those requests. There are two other actors to consider: the Policy Administration Point (PAP) which allows to manage the security policies in a policy repository and the Policy Information point (PIP) which provides additional information on request's subjects (roles, actions/services, environment, ...) when required by the PDP.

1 Architecture and Target Platform

Figure 17 depicts a typical IS and its interaction with a client application. In this particular view, the client application sends a request to a WS using standard protocols such as HTTP, WSDL and SOAP. The request goes through an Enterprise Service Bus (ESB) acting as a middleware for the environment and a routing point for secure exchanges of messages between communicating partners. In our projects,

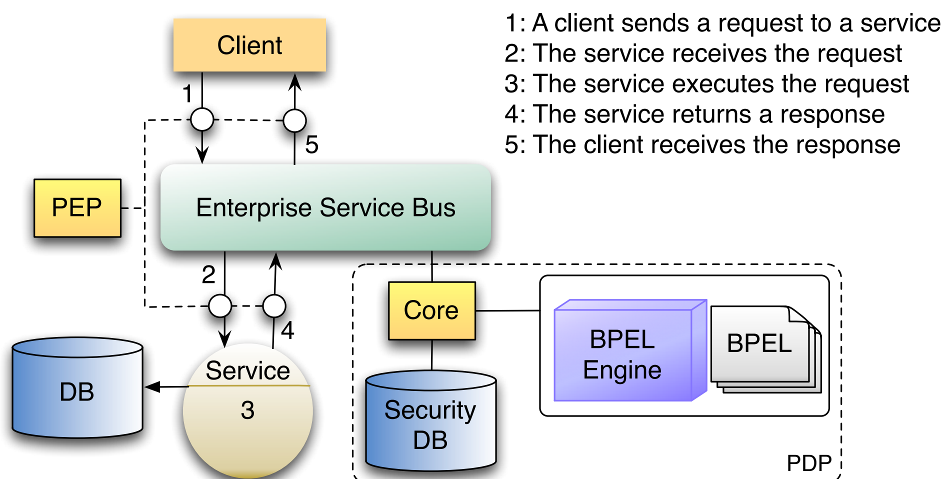


Figure 1: A typical SOA application

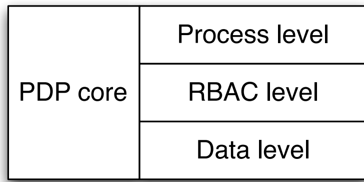


Figure 2: PDP abstract internal view

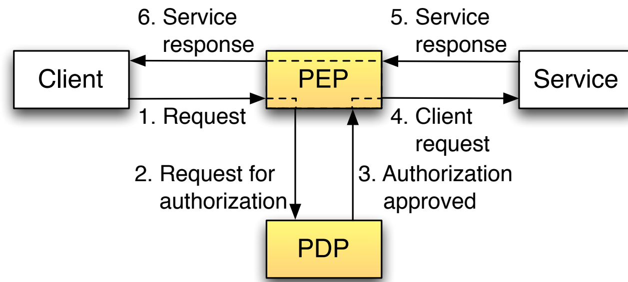


Figure 3: PEP and PDP security schema

the PEM complies with the one specified in the XACML standard from OASIS [22]. It is based on two main components: the PEP and PDP. Together, they are responsible for intercepting requests from client applications to services and providing authorization control w.r.t. access decisions for these requests based on security policies. There are two other auxiliary components to consider: the PAP that provides facilities for the management of a policy repository and the PIP that supplies additional information closely related to requests (e.g., roles, actions/services, environment) when required by the PDP.

The PDP has a key role to play in the PEM, since it takes approval/denial decisions based on security policies. In order to ensure security with a high level of granularity, decisions are based on three different levels of functional security as shown in Figure 18. The work described in this paper focuses on enforcement of functional security rules associated with the third level, called *process level*. The reader is referred to a companion paper for a presentation of aspects related to the two other levels [6]. Functional security rules defined at the third level concern business processes (collections of related, structured atomic services). They describe rules that depend on the state of the system (e.g., on the history of the past events accepted by the system) and are specified at an abstract level using ASTD [9]. Generally an ASTD takes into consideration a set of security rules, which defines an Access Control (AC) policy. Indeed, the security rules are put together in the same ASTD by using the parallel composition operator. At the implementation level, the decision-making task is realized with the aid of a BPEL engine that enforces security rules from a BPEL process derived from an ASTD. Therefore, the rules are not attached to actions or services to secure, nor to entities (e.g., roles, actors) involved in the IS. For example, a specialist can consult a patient's health record only when this patient has been referred to him by the treating physician.

Figure 19 details the interaction between a client and a service through the PEP as well as the interaction between the PEP and the PDP. In a typical scenario, a client sends a request to a service or a component of a distributed application (1) along with some user information (identification and role). The request is intercepted by the PEP, which extracts user information and then formulates an authorization request for approval/denial by the PDP (2). The PDP takes a decision on whether to approve or deny the client request (in this scenario the request has been approved by the PDP). This decision, centralized at the Core component of the PDP, is based on a check performed on the user information database Security DB (identifiers and roles) and the response from the BPEL engine to a specific request formulated by the Core. The authorization is reported back to the PEP (3). If the request is allowed by the PEP, then the PEP allows the original request to reach the requested service (4), which may perform specific business validations before executing the request (e.g., checking that an account has sufficient funds before initiating an electronic fund transfer). The response goes through the PEP (5) so that the policy repository or PAP (if there is any) can be updated with respect to the recently executed request. Finally, the response is redirected to the client (6). The case in which the request is denied by the PDP is similar, except that the steps (4) and (5) are superfluous, since an authorization denied response is returned immediately by the PEP upon a reject from the PDP. In both cases, messages must be sent through secure channels in order to guarantee confidentiality and integrity of the communication between all the partners. It should be noted that this schema is a simplification of the security data-flow diagram

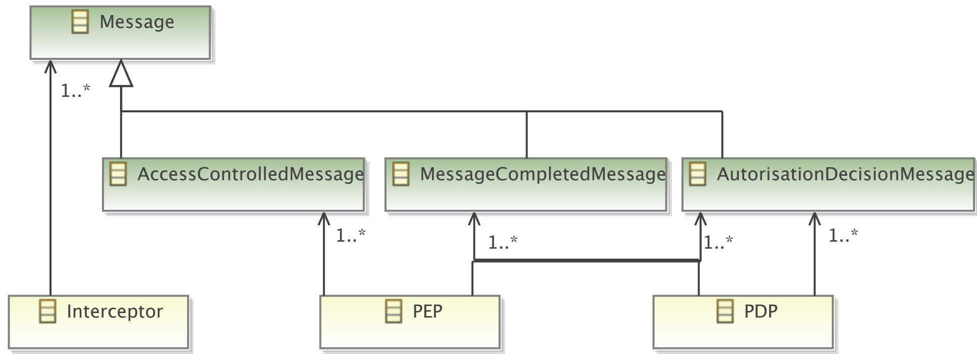


Figure 4: Subset of the PEM class diagram

described in the XACML standard.

2 Class diagrams

Figure 4 depicts the metamodel of the PEM metamodel. The active components in the diagram are `Interceptor`, `PEP` and `PDP`. The class `Message` is used by user application to request services (or information) from the IS. The `Interceptor` class is part of the user application and is responsible to add the required AC parameters (`AccessControlParameter`) to the `Message`, thus making it an `AccessControlledMessage`. In the prototype Service Oriented Architecture (SOA)-oriented application, the `Interceptor` is instantiated as a Simple Object Access Protocol (SOAP) handler. It acts between the Web Service (WS) proxy in the client application. The main goal of this action is to inject into the SOAP message of the form $E(\vec{x})$ (E is the requested service and $\vec{x}$ the parameters list) the AC parameters: the user identifier u, its role r and organization o previously gathered through required authentication. In this example, AC is exercised based on these three parameters which are instances the subclasses of the class `AccessControlParameter` (see Figure 7). The new SOAP message of the form $E(\langle u, r, o \rangle; \vec{x})$ exiting from the handler is an instance of the `AccessControlledMessage`. Any model derived from the metamodel may consider other AC parameter such as time.

The `PEP` is an active component of the PEM framework. It enforces AC on service requests routed from the user application to service implementations. The service requests are received as instances of the class `AccessControlledMessage`. Control is then exercised by instances of the class `PEP` in a process described by the sequence diagram of Figure 10.

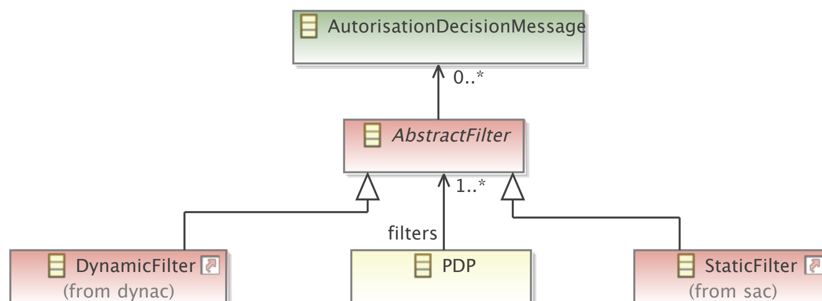


Figure 5: The PDP uses filters

AbstractFilters are used by the PDP as depicted in Figure 5. They receive an instance of the AutorisationDecisionMessage class, which contains elements from the initial user request to a service (class Message) with actual values of the parameters (class AbstractParameter), and also values of the AC parameters — that is parameters on which AC parameters are based. The PDP class synthesizes the AC decision based on the response of its filter collection. The synthesis of the AC decision is generally based on a simple conjunction function but more sophisticated behaviors may be possible. For example an algorithm that prioritizes a policy over another in case of emergency might be implemented by the PDP.

2.1 The message hierarchy

Messages are the mean by which actives components of the PEM framework exchange information or request services. Message is the base class of all messages. Each message has a collection of AbstractParameters. A AbstractParameter may hold an input value or an output as a result of a service request or method call. A Message instance can then be:

- *one way* i.e. the Message has only input parameters and the sender does not expect a response;
- *two way* i.e. the Message has input and output parameters

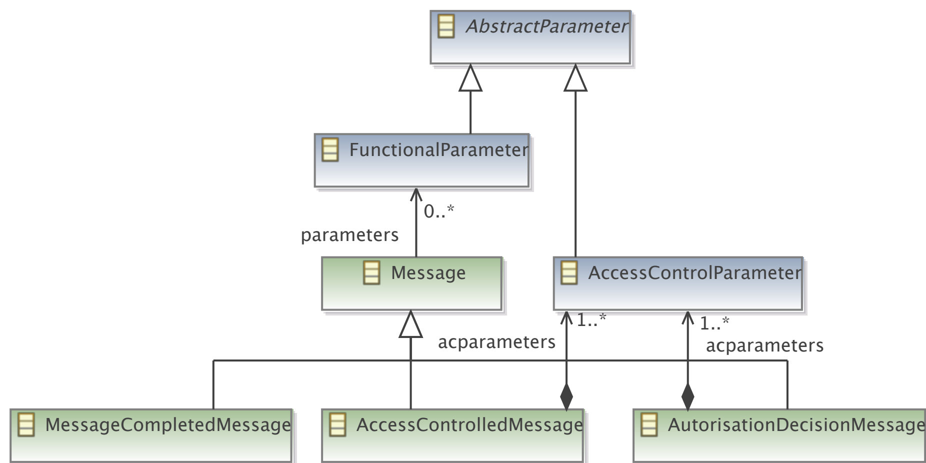


Figure 6: PSM message hierarchy

As already mentioned earlier, the AccessControlParameters are the base on which AC is exercised. In most implementations, notably RBAC-based ones — RBAC stands for Role Based Access Control, these parameters are derived into User, Role, Organization classes, most of which are self-explanatory.

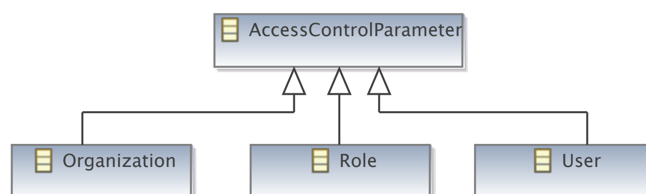


Figure 7: The base access control parameters of the PSM

2.2 The SAC package

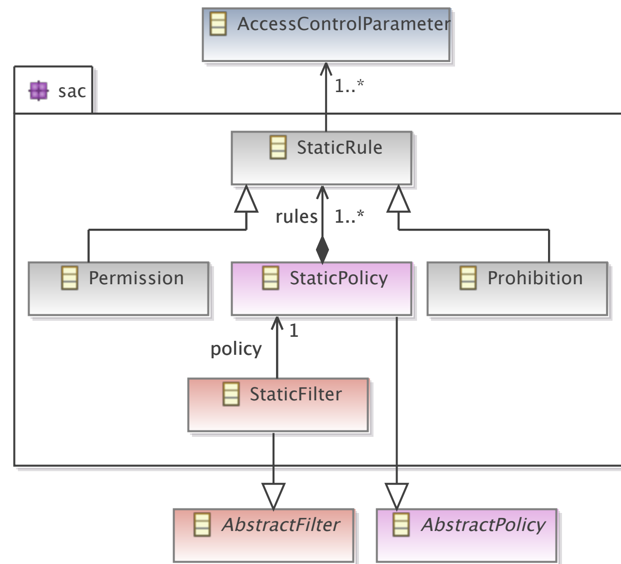


Figure 8: Static AC filtering classes

The sac package depicted in Figure 8 contains that allow AC decision making based on Static Access Control (SAC) policies. A SAC `StaticPolicy` is a set of `StaticRule`s. Each `StaticRule` materializes an association between AC parameters. Most implementation deals with two types of `StaticRule`:

- `Permission` allows execution of a service if the actual values of AC parameters in the `AccessControlledMessage` matches the values specified by the `Permission` rule;
- `Prohition` fobits execution for values of `AccessControlParameter` defined by the rule.

A SAC `StaticFilter` — subclass of the abstract class `AbstractFilter` in the upper package — uses a SAC `StaticPolicy` and infers an AC decision when it receives a `AutorisationDecisionMessage`. The response of the sac `StaticFilter` to the `PDP` is positive — *access granted* — (or negative — *access denied* — depending on the `StaticRule` type) when the AC parameters provided by the `Interceptor` matches the values specified by the `StaticRule`.

2.3 The DynAC package

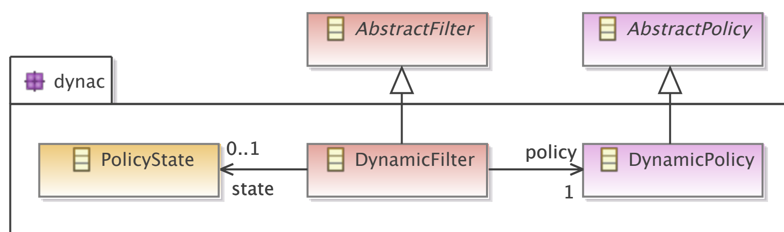


Figure 9: Dynamic AC filtering classes

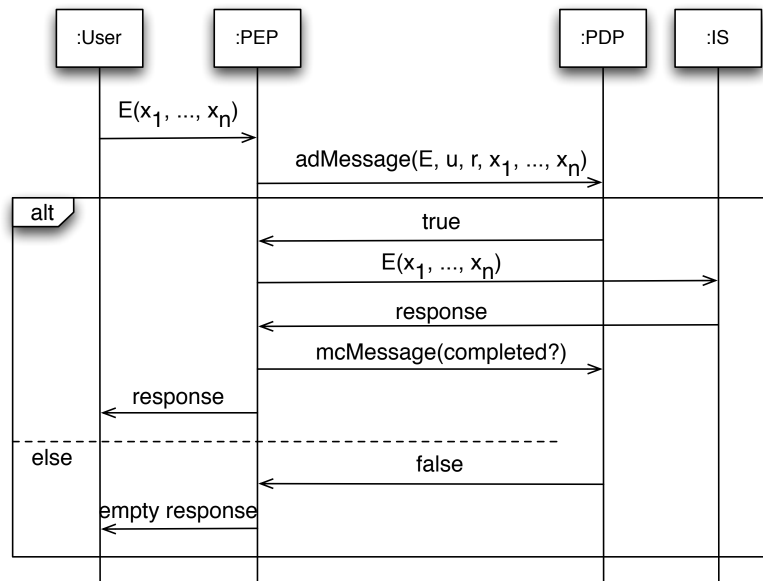


Figure 10: PEP sequence diagram

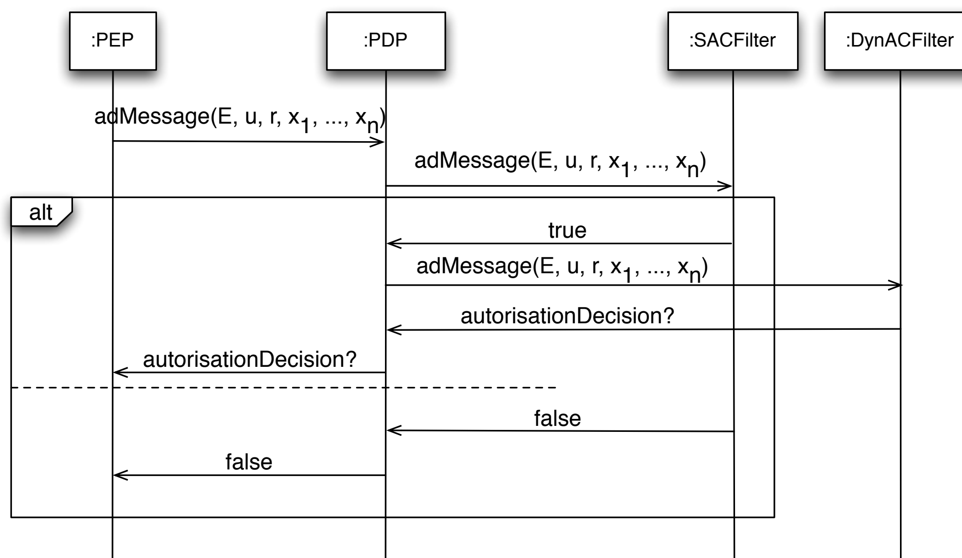


Figure 11: PDP sequence diagram

In Figure 9, classes that perfom Dynamic Access Control (DynAC) are depicted. A DynAC `DynamicFilter` — subclass of the abstract class `AbstractFilter` in the upper package — uses a DynAC `DynamicPolicy` and its current state (class `PolicyState`) to infer AC decisions. Instances of the `DynamicPolicy` class express constraints at the process level. The constraints may mandate a specific ordering on the service requests based on the values of the AC parameters. Regarding the update of the policy state, the class `MessageCompletedMessage` is used by the `PEP` to keep the `PDP` up to date on whether the IS has successfully executed the initial request or not.

3 Sequence diagrams

Part II

PIM Metamodel - B for Access Control

In the following, we present a draft of a B metamodel at the PIM level in order to express AC policies. In this metamodel, AC rules can be either static or dynamic. According to Neumann and Strembeck in [21], dynamic rules are defined as rules that can only be evaluated at execution time according to the context of execution. Static rules are defined as rules that are not dynamic.

Fig. 12 details an early version of the static part of the metamodel by linking it to the PSM architecture defined in the previous part and to the B metamodel. This part of the specification results from the translation of UML into B designed by the LIG team.

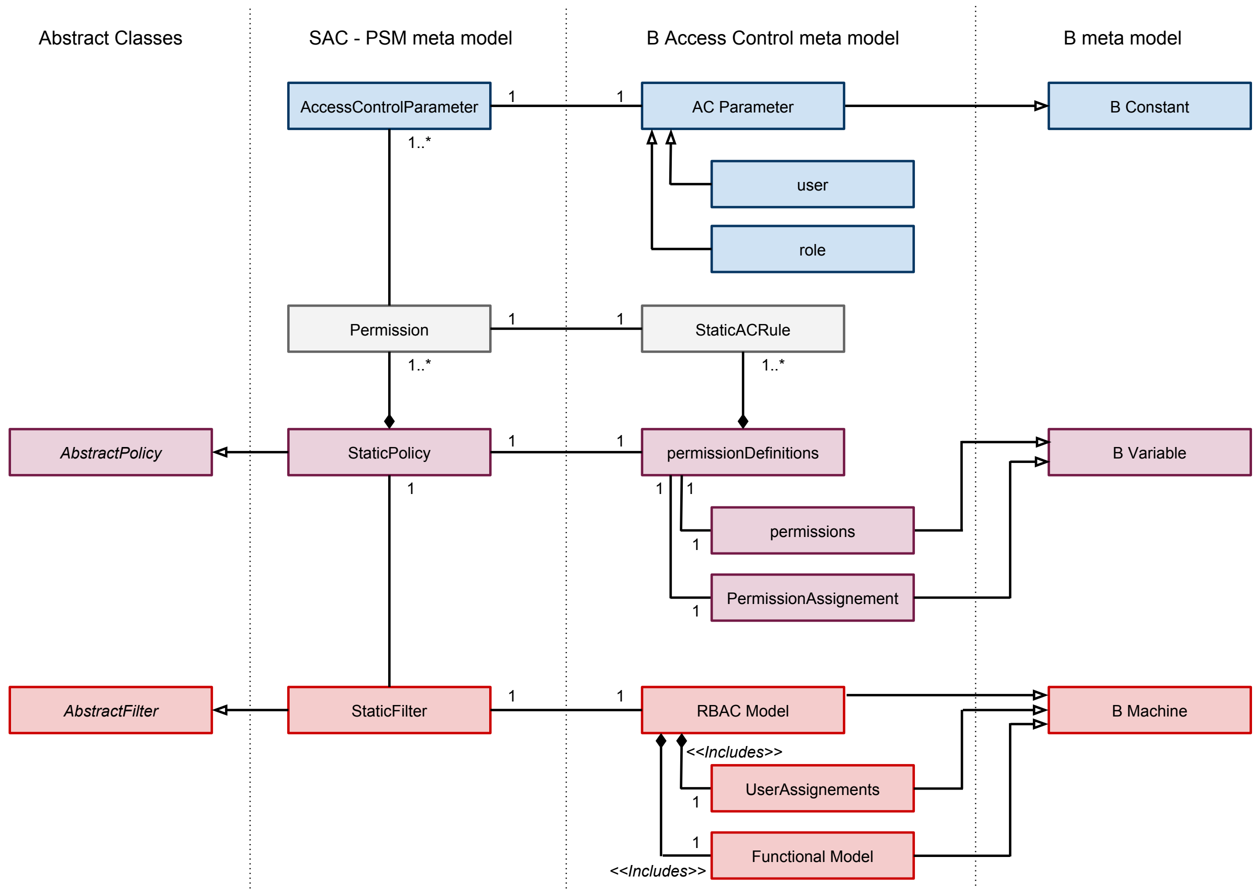


Figure 12: Static AC PIM Metamodel

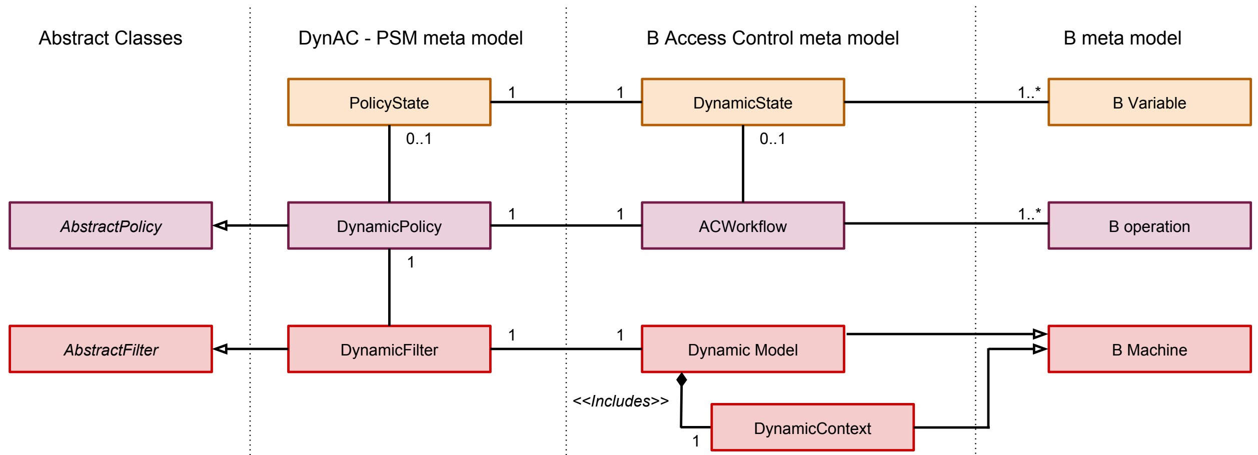


Figure 13: Dynamic AC PIM Metamodel

Fig. 13 details an early version of the dynamic part of the metamodel by linking it to the PSM architecture defined in the previous part and to the B metamodel. This part of the specification results from the translation of ASTD into B designed by the LACL and GRIL teams.

Part III

From ASTD Access Control Policies to WS-BPEL Processes Deployed in a SOA Environment

Controlling access to Web services of public agencies as well as private corporations primarily depends on specification and deployment of functional security rules in order to satisfy strict regulations imposed by governments, particularly in financial and health sectors. This paper focuses on one aspect of the SELKIS and EB³SEC projects related to security of Web-based information systems, namely the automatic transformation of security rules, instantiated from security rule patterns written in a graphical notation with a denotational semantics close to statecharts, into WS-BPEL (or BPEL for short) processes. The latter are executed by a BPEL engine integrated into a policy decision point, a component of a policy enforcement manager similar to the one proposed in the XACML standard. The main results described in this part are presented in [7].

4 Introduction

In some business sectors, *information systems* (IS) are governed by internal organization policies and government laws. To enforce such policies as well as to prevent data confidentiality and integrity security breaches, access control is widely used in IS. More precisely, user access to data and functionalities are filtered, based on well defined policies. *Role based access control* (RBAC), a methodology which associates user identities with the data and/or functionalities through their role, is the most implemented solution. However, it does not solve new problems found in todays common SOA applications. These applications are not "one user centric" only and implement workflows that may involve interactions with different users. With respect to workflows, RBAC has little expressiveness power. As an example, RBAC frameworks cannot implement *separation of duty* (SoD) properties.

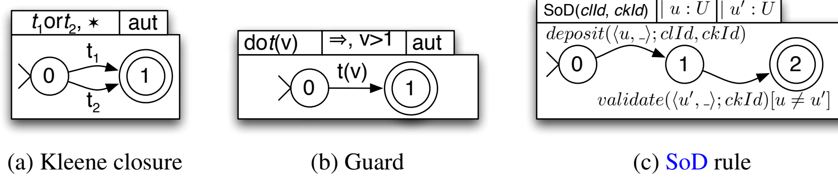
A substantial part of the EB³SEC[1] and SELKIS[2] [3] projects consists in developing a prototype of a PEM for distributed IS executed in a SOA environment as Web services (WS). Our approach focuses on three identified levels of access control: the data level, the RBAC level and the process level. In this paper, we propose an automatic implementation of a significant part of the enforcement framework derived from an access control policy expressed in a high level language. This high level language is formal, powerful enough to implement common properties encountered in security policies and can also express many sort of constraints. The implementation relies on a two-steps translation algorithm which produces an executable BPEL process from a formal specification of an access control policy. Overall, the enforcement framework follows architectural guidelines proposed by the XACML standard.

The rest of this paper is organized as follows. Section 5 provides an overview of the formal notation (ASTD) used for specifying security rules and presents patterns for *permission* and *obligation*. The other patterns, such as *separation of duty* and *prohibition*, are only mentioned. The ASTD notation allows for the combination of state transition diagrams using process algebra operators. This high-level notation is appropriate for specifying security rules at the process level and is independent from any implementation environment. Section 6 describes the architecture of SOA applications our project targets as well as the three level of granularity we have identified previously. Enforcement framework components are also depicted as well as two typical message exchange scenarios between the principal components. Section 7 details a translation schema that transforms an ASTD specification into a BPEL process along with its WSDL interface and XSD type definitions, which are deployed in a SOA environment so that they constitutes the core of the PDP. This translation schema is mechanizable as far as the security

[1] EB³SEC stands for EB³Secured.

[2] SELKIS is an acronym for SEcure heaLth care networKs Information Systems.

[3] Project ANR-08-SEGI-018 in France http://lacl.univ-paris12.fr/selkis/



(a) Kleene closure (b) Guard (c) SoD rule

Figure 14: ASTD examples

rules obey to the aforementioned patterns. An error-prone development phase is then replaced by a safe translation procedure. Section 8 describes strongly related work and points out differences with some aspects developed in this paper. Section 9 concludes this paper with ongoing and future aspects of this work.

5 Expressing Security Rules with an ASTD

In most security frameworks, a security policy is a combination of many security rules. Researchers and security practitioners [4, 16, 17, 28] have considered the following categories for security rules:

- *permission* which authorizes actions to be executed;
- *prohibition* which forbids actions to be executed;
- *separation of duty* which expresses the fact that a set of tasks cannot be executed by the same users or roles;
- *obligation* which forces a user to perform an action sometimes in the future after he has performed a specific action, in other words two distinct actions must be performed by the same user.

Such rules can be expressed by using ASTD which is a graphical and formal notation initially created to design IS. It has been inspired from statecharts [10] and process algebras like CSP and LOTOS. Since the ASTD notation is formal, a wide range of verification and validation tools can be used with ASTD-based models. Especially, Milhau et al.[20] devised a transformation from ASTD to Event-B [1] which has solid and well established tools for proofs and formal verification. The ASTD notation can also be used to define security rules that restrict the free behavior of IS so that it does not violate organizational access control policies.

An ASTD is a hierarchical transition system. The dynamics is based on transitions labeled with events of the form $t(\vec{x})[\phi]$, where $\vec{x}$ is a list of parameters (which can be empty) and ϕ is an optional guard that must hold to enable the transition to fire. States are typed. Possible types are *elementary state*, *automaton*, *guard*, *choice*, etc. A special feature of the ASTD notation is a quantified version of parameterized synchronization and choice operators. In addition, each non-elementary state may carry parameters as events do. Figure 14a shows a Kleene closure ($*$) in which, either t_1 or t_2 will be executed repetitively. Figure 14b shows a guard that executes the action t only if $v > 1$ where v is the parameter of the ASTD. The complete definition of the ASTD notation and its formal operational semantics is available in [8]. Concerning expression of security policies with ASTD, transitions are augmented with two parameters to take into account the user identity and his role while executing an action: $\langle u, r \rangle$ where u is the user identifier and r his role.

Due to space limitation, only *permission* and *obligation* patterns are detailed in the sequel. Figure 15a shows the ASTD pattern for *permission*. It enables the execution of action t_1 (to t_n respectively) with parameters $\vec{x_1}$ (to $\vec{x_n}$ respectively) by user u_1 (to u_n respectively) acting with role r_1 (to r_n respectively). A Kleene closure ($*$) is used, since an action can be repeatedly executed. Figure 15b illustrates an instance of this pattern. `Adrian` and `Boris` have permission to execute actions `deposit` and `cancel` when acting with roles `clerk` and `banker` respectively. The symbol "–" denotes a don't care value for a

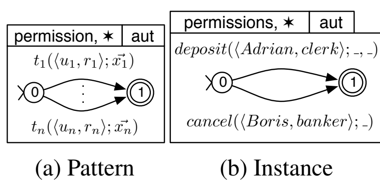
parameter (account number and amount in the case of a deposit). Figure 16a includes the ASTD pattern for *obligation*. With such a rule, actions t_1 to t_2 must be executed by the same user u. As illustrated in Fig. 16b, deposits from a client must be always executed by the same `cashier`, unless the operation is done by the `head office`. Figure 14c illustrates a simple SoD constraint which is an instance of the SoD pattern. It states that actions `deposit` and `validate` must be executed by two different users ($u \neq u'$).

It should be noted that limiting security rules to patterns makes it possible to derive BPEL processes from these particular forms of rules. Indeed, automatic translation regardless of the ASTD structure represents a complex and challenging problem which is out the scope of our projects.

6 Architecture and Target Platform

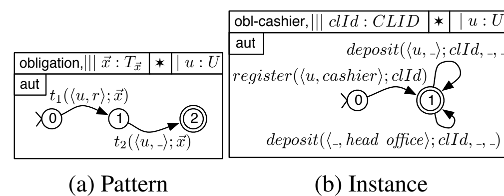
Figure 17 depicts a typical IS and its interaction with a client application. In this particular view, the client application sends a request to a WS using standard protocols such as HTTP, WSDL and SOAP. The request goes through an ESB acting as a middleware for the environment and a routing point for secure exchanges of messages between communicating partners. In our projects, the PEM complies with the one specified in the XACML standard from OASIS [22]. It is based on two main components: the PEP and PDP. Together, they are responsible for intercepting requests from client applications to services and providing authorization control w.r.t. access decisions for these requests based on security policies. There are two other auxiliary components to consider: the PAP that provides facilities for the management of a policy repository and the PIP that supplies additional information closely related to requests (e.g., roles, actions/services, environment) when required by the PDP.

The PDP has a key role to play in the PEM, since it takes approval/denial decisions based on security policies. In order to ensure security with a high level of granularity, decisions are based on three different levels of functional security as shown in Fig. 18. The work described in this paper focuses on enforcement of functional security rules associated with the third level, called *process level*. The reader is referred to a companion paper for a presentation of aspects related to the two other levels [6]. Functional security rules defined at the third level concern business processes (collections of related, structured atomic services). They describe rules that depend on the state of the system (e.g., on the history of the past events accepted by the system) and are specified at an abstract level using ASTD [9]. Generally an ASTD takes into consideration a set of security rules, which defines an access control policy. Indeed, the security rules are put together in the same ASTD by using the parallel composition operator. At the



(a) Pattern (b) Instance

Figure 15: ASTD pattern for permission



(a) Pattern (b) Instance

Figure 16: ASTD pattern for obligation

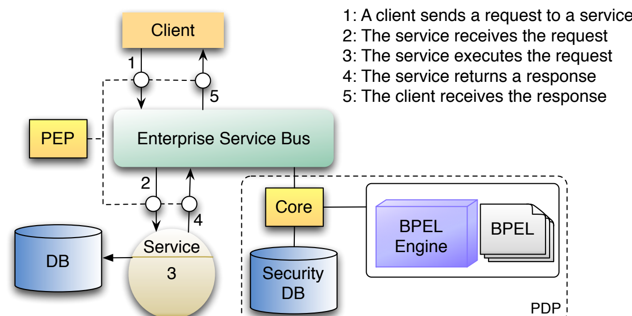


Figure 17: A typical SOA application

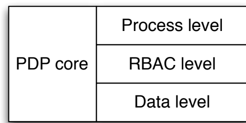


Figure 18: PDP abstract internal view

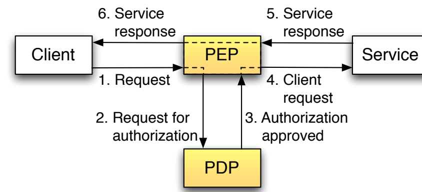


Figure 19: PEP and PDP security schema

implementation level, the decision-making task is realized with the aid of a BPEL engine that enforces security rules from a BPEL process derived from an ASTD. Therefore, the rules are not attached to actions or services to secure, nor to entities (e.g., roles, actors) involved in the IS. For example, a specialist can consult a patient's health record only when this patient has been referred to him by the treating physician.

Figure 19 details the interaction between a client and a service through the PEP as well as the interaction between the PEP and the PDP. In a typical scenario, a client sends a request to a service or a component of a distributed application (1) along with some user information (identification and role). The request is intercepted by the PEP, which extracts user information and then formulates an authorization request for approval/denial by the PDP (2). The PDP takes a decision on whether to approve or deny the client request (in this scenario the request has been approved by the PDP). This decision, centralized at the Core component of the PDP, is based on a check performed on the user information (identifiers and roles in Security DB) and the response from the BPEL engine to a specific request formulated by the Core. The authorization is reported back to the PEP (3). If the request is allowed by the PEP, then the PEP allows the original request to reach the requested service (4), which may perform specific business validations before executing the request (e.g., checking that an account has sufficient funds before initiating an electronic fund transfer). The response goes through the PEP (5) so that the policy repository or PAP (if there is any) can be updated with respect to the recently executed request. Finally, the response is redirected to the client (6). The case in which the request is denied by the PDP is similar, except that the steps (4) and (5) are superfluous, since an authorization denied response is returned immediately by the PEP upon a reject from the PDP. In both cases, messages must be sent through secure channels in order to guarantee confidentiality and integrity of the communication between all the partners. It should be noted that this schema is a simplification of the security data-flow diagram described in the XACML standard.

7 Transforming an ASTD Access Control Specification into a BPEL Process

The algorithm, that translates an ASTD access control specification into a BPEL process, includes two phases since an intermediate representation in an AAST form is more convenient for computation purposes. For a technical reason introduced in Sect. 7.8 (the use of a specific tool for implementing the transformation), the AAST notation is considered as an intermediate language called $\mathcal{L}1$. The algorithm generates a BPEL process along with its WSDL interface and XSD type definitions. The transformation is *event-based* in the sense that the BPEL process mimics the flow of events described by the ASTD.

7.1 The BPEL Process Language

WS-BPEL stands for Web Service Business Process Execution Language. It is basically an XML language for designing business processes independently from enactment engines. BPEL is an OASIS standard [23] and makes intensive use of other standards. Because tasks are often automated through WS, it uses WSDL, the standard for describing service interfaces, and XPath, for navigation in XML variables.

```
<repeatUntil>
 <pick>
  <onMessage operation="authDect1"...>
   ...
  </onMessage>
  ...
  <onMessage operation="authDectn"...>
   ...
  </onMessage>
 </pick>
 <condition>false()</condition>
</repeatUntil>
```

Figure 20: Permission BPEL code skeleton

```
<repeatUntil>
 <sequence>
  <scope>
    <!-- t1 BPEL code -->
  </scope>
  <scope>
    <!-- t2 BPEL code -->
  </scope>
 </sequence>
 <condition>false()</condition>
</repeatUntil>
```

Figure 21: Obligation BPEL code skeleton

A BPEL specification defines one process through the XML root element **process**. Such a process can be directly run by an enactment engine. It is thus referred to as an *executable process*. BPEL provides various basic activities, such as arrival of a message (element **receive**), reply to a message (element **reply**) and invocation of a WS (element **invoke**). The standard includes more elaborated constructs like scopes (**scope**) for variable declaration, loops (e.g., **repeatUntil**) and conditionals (e.g., **if**). Furthermore, complex parallel processing is possible with the **flow** element, combined with **link** to create dependencies or synchronization points between concurrently running activities.

Our industrial partners both in Canada and France are replacing their legacy systems using SOA platforms. As BPEL integrates very well within such environments, it is an appropriate choice to enact security processes. Since engines deployed in a distributed environment are bundled with technical security characteristics (e.g., encryption protocols, reliable messaging, scalability), any proper BPEL engine would provide such functionalities at no additional cost. Another motivation behind the choice of BPEL is the work done to verify properties on BPEL processes [3], which allows to check properties on the final process using Event-B [1] and the Rodin platform [2].

7.2 Transformation Rules from ASTD to BPEL

The proposed transformation is tailored for the security rules patterns expressed with the ASTD notation. Each pattern is transformed into a set of adequate BPEL elements. This section provides a skeleton overview of these BPEL elements for the *permission* and *obligation* patterns.

Intuitively, pattern permission is implemented by a **repeatUntil** BPEL activity. As shown in Fig. 15a, the events t_1 to t_n have the same outgoing state 0, thus introducing a **pick** activity (see Fig. 20). For this pattern, the interaction between the PEP and the PDP regarding a *permission* goes shortly as following: when the part of the BPEL code corresponding to the pattern is testing, a request of the form authDec$t(\langle u, r\rangle, \vec{x})$ (where t is the event, u and r the user and role respectively, $\vec{x}$ the event's parameters) is received by the **pick** activity. If $t \notin \{t_1, ..., t_n\}$, then the request is immediately denied since the **pick** element cannot process it. Otherwise, the processing goes further in the corresponding **onMessage** element where the user identity and role as well as event's parameters are tested against the values encoded in the BPEL code from the formal specification. Whether or not the match fails, the test will loop due to the upper **repeatUntil** element.

In the same way, an *obligation* rule is implemented by a **repeatUntil** BPEL element. Events t_1 and t_2 in Fig. 16a are transformed into a set of elements wrapped into a **sequence** BPEL element. Figure 21 is a skeleton of the transformation. Table 1 summarizes the mappings used to guide the transformation rules.

7.3 $\mathcal{L}1$: An Intermediate Language between ASTD and BPEL

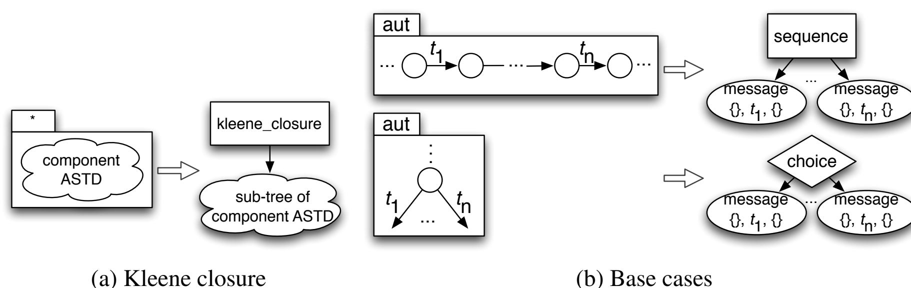
$\mathcal{L}1$ is a notation for AAST. Particular data fields are added in the nodes in order to calculate <**link**/> BPEL elements required to manage dependencies that arise when dealing with the parameterized synchronization construct. Leafs have a node type, *empty*(*left_links*, *right_links*) or *message*(*left_links*, *t*, *right_links*),

where *left_links* and *right_links* represent control flow dependencies between activities of a BPEL process, and t a transition from an ASTD automaton. Node type *empty* are useful to create synchronization points when dealing with ASTD synchronization. Internal nodes have also a node type in accordance with the ASTD operators:

1. $sequence(activity_1, ..., activity_n)$ represents the sequence execution flow of $activity_1$ to $activity_n$ (parameters named *activity* represent subnodes of the tree);
2. $synchronization(\Delta, activity_1, activity_2)$ represents the synchronization between the threads associated to processes $activity_1$ and $activity_2$ over the common events set Δ;
3. $kleene_closure(activity)$ represents the repetition of *activity*;
4. $prohibition_choice(activity_1, ..., activity_n)$ represents the choice between activities that prohibit execution of some events;
5. $guard(p, activity)$ represents the execution of *activity* if the predicate p holds;
6. $choice(branches)$ represents the choice of a thread to execute based on the first incoming message, the other choices are discarded (a branch can be viewed as a pair $\langle message(l, t, r), activity\rangle$);
7. $quantified_choice(v, T, activity)$ represents the execution of one thread of *activity* for a value of $v \in T$;
8. $parameterized_synchronization(v, T, activity)$ represents the interleaved execution of threads of *activity* for each value of $v \in T$.

Table 1: Mapping between ASTD and BPEL constructs

	ASTD	BPEL
Sequence in automaton		<**sequence**/>
Choice in automaton		<**flow**/> with <**link**/> to manage dependencies between the first action and the *remainder* of each branch
Kleene closure	*	<**repeatUntil**/> with condition set to `false`
Guard	⇒ p	Add the predicate p to all the first possible events in the sub-ASTD
Prohibition choice	* |	<**flow**/>
Quantified choice	| v: T	<**scope**/> with two <**variable**/> based on v to manage the choice



(a) Kleene closure (b) Base cases

Figure 22: Construction of the AAST

```
<scope>
<variables>
 <variable name="tAuthDecIn" type="authorizationDecisiontRequest"/>
 <variable name="tAuthDecOut" type="authorizationDecisionResponse"/>
 <variable name="tServExecIn" type="serviceExecutedRequest"/>
</variables>
<sources>
 <!-- Transforming r to actual source links -->
</sources>
<targets>
 <!-- Transforming l to actual target links -->
</targets>
<repeatUntil>
 <receive variable="tAuthDecIn" operation="authDect".../>
 <assign>
  <!-- $tAuthDecOut.access =... -->
 </assign>
 <reply variable="tAuthDecOut" operation="authDect".../>
  <if>
   <condition>$tAuthDecOut.access = granted</condition>
   <receive variable="tServExecIn"
    operation="servExect".../>
  </if>
 <condition>$tServExec.exec</condition>
</repeatUntil>
</scope>
```

Figure 23: *Message*

```
<sequence>
 <!-- activity_1 -->
 ...
 <!-- activity_n -->
</sequence>
```

Figure 24: *Sequence*

7.4 From ASTD to $\mathcal{L}1$

To transform an ASTD specification into a BPEL process as a first step, we use some straightforward rules, as defined in an attribute grammar, to produce a $\mathcal{L}1$ tree. These rules are used recursively when encountering a component ASTD in a parent ASTD operator, as illustrated for the Kleene closure ASTD in Fig. 22a. Other internal nodes are created in the same usual way. For instance, a *quantified_choice* node is created, with fields for the quantified variable v and its type T, and recursively a node for the *activity*, when a quantified choice ASTD is encountered. The base cases are shown in Fig. 22b. For an automaton ASTD, a *message* is created for each transition in the automaton and inserted as a leaf, which is linked to an internal node (*sequence* or *choice*) w.r.t. the transition function of the automaton. The sets *left_links* and *right_links* are initially empty and are filled later by a helper function that deduces dependencies when encountering synchronization ASTD. Also, in accordance with the security rule patterns, there is no cycle in the automaton avoiding generation of loops in the BPEL process with **link** crossing their boundary (this is the cross-boundary restriction of the BPEL standard).

7.5 From $\mathcal{L}1$ to BPEL

Much like the conversion of an ASTD into $\mathcal{L}1$, the conversion of a $\mathcal{L}1$ model into a full BPEL process follows a set of transformation rules. To be able to execute the process in a BPEL enactment engine, its WSDL interface and XSD type definitions are also generated using other suitable sets of transformation rules as described in the next two sections.

A $\mathcal{L}1$ $message(l,t,r)$ is converted into a sequence of BPEL constructs that must perform altogether the arrival of a request for authorization decision, the effective computation of the access decision, the response to request for authorization and if required the arrival of a message which indicates if the initial target service, when authorized, has been effectively executed. This translation schema is given in Fig. 23 for a simple message which is not a choice child node. For a message under a choice node, the corresponding BPEL code encompasses an **onMessage** element rather than a simple **receive**.

A $\mathcal{L}1$ $sequence(activity_1, ..., activity_n)$ is converted into a BPEL **sequence** and the transformation rules are called recursively to transform the subnodes $activity_1$ to $activity_n$ of the *sequence* as shown in Fig. 24. A $\mathcal{L}1$ *choice* node is transformed into a BPEL **flow** with a **link** creating a flow dependency between the

```
1  <definitions name="SecurityFilterProcessInterface">
2   <message name="authorizationDecisionResponse">
3    <part name="access" type="AccessType"/> <!-- granted or denied -->
4   </message>
5   <message name="serviceExecutedRequest"/>
6    <part name="exec" type="boolean"/>
7   </message>
8   <!-- Place for message definitions -->
9   <portType name="PDPPortType">
10   <!-- Place for operation definitions -->
11  </portType>
12  <partnerLinkType name="PDPPartnerLink">
13   <role name="securityFilter" portType="PDPPortType"/>
14  </partnerLinkType>
15 </definitions>
```

Figure 25: WSDL document for the PDP interface

first message and the remaining activity for each branch making up the *choice*. The *choice* is implemented by first enabling execution of the first action — which is a leaf of the type $message(l, t, r)$ — of each branch. Then a **link** is used to enabled the execution of the remainder of the branch as soon as the first action is done running. As it is a choice that has to be made between branches, the same **link** mechanism is used to disable to other branches that have not been chosen. Due to space limitation, the corresponding BPEL code is omitted.

A *kleene_closure* node is transformed into a BPEL **repeatUntil** with its condition set to `false`. Again, the subnode is processed recursively using the rules. A $\mathcal{L}1$ *guard* is not transformed into an enclosing BPEL element. However, its predicate p is added to the execution condition (the `AccessType`-typed `access` part in the output message) of all the first events of the sub-ASTD. A $\mathcal{L}1$ *prohibition_choice* node is transformed into a **flow** containing the transformation of each subtree node wrapped in a **repeatUntil**. A $\mathcal{L}1$ *quantified_choice*$(v, T, activity)$ node is transformed into an enclosing BPEL **scope** with two declared **variable**s to manage the choice. The first **variable** holds the chosen value for v and the second one holds a Boolean that keeps track of when a value has been chosen or not for v. An additional **variable** may be required in order to validate the value passed to the BPEL process by the PEP according to legal values in T.

7.6 Generating the WSDL Interface

The ASTD notation does not provide clauses to define the signature of events that label transitions. Nonetheless, elements of the form $\langle t, \langle p_1, T_1\rangle, \ldots, \langle p_n, T_n\rangle\rangle$, where t is an event name, p_i a parameter name and T_i a type name, can be inserted into a list which is attached to the corresponding ASTD. A WSDL document for the PDP interface, which includes messages, operations and port types, is generated from the event list of the ASTD specification according to the skeleton given in Fig. 25. In this skeleton, the first two message types are independent from events. The message type `authorizationDecisionResponse` is used when the PDP transmits an access decision, which can be `granted` or `denied`, to the PEP (point 3 in Fig. 19). The message type `serviceExecutedRequest` is sent by the PEP to the PDP in order to update its state w.r.t the fact that the requested service has been or not executed by the IS. WSDL code snippets are inserted in appropriate places in this document for each event in the list. For instance, lines 1–7 in Fig. 26 provide the code snippet for the message used when the PEP makes a request for authorization to execute the service t (point 2 in Fig. 19). The code snippets associated with messages are inserted at line 8 in Fig. 25. Similarly, code snippets associated with event signatures are generated according to the schema in Fig. 26 (lines 10 to 16) and inserted at line 10 in the skeleton.

7.7 From $\mathcal{L}1$ to XSD Type Definitions

In ASTD quantified operators, a variable appears explicitly and its value must belong to a predefined set of values, which can be expressed in two forms. The first form is an enumeration, $T = \{x_1, \ldots, x_n\}$. The

```
1  <message name="authorizationDecisionιRequest"/>
2   <part name="u" type="UserType"/>
3   <part name="r" type="RoleType"/>
4   <part name="p1" type="T1"/>
5   ...
6   <part name="pn" type="Tn"/>
7  </message>
8
9  <portType name="PDPPortType">
10  <operation name="authDecι">
11   <input name="inputAuthDecι" message="authorizationDecisionιRequest"/>
12   <output name="outputAuthDecι" message="authorizationDecisionResponse"/>
13  </operation>
14  <operation name="servExecι">
15   <input name="inputServExecι" message="serviceExecutedRequest"/>
16  </operation>
17 </portType>
```

Figure 26: WSDL code for an event signature

resulting XSD code is a simple type enumerating all the values in T as shown in Fig. 27. The name `baseT` (line 2) is a base type (`integer`, `float` or `string`) of the simple type (which can be seen as a restriction of a base type). The base type is determined by an helper function based on the values that are included in the original set T. The second form is an interval of values, $T = [x_l, x_u]$. The resulting XSD code is still a simple type with a range defined from the lower and upper bounds of the interval (see Fig. 28). However, `baseT` is necessarily `integer`, since it is the only range base type presently supported. All XSD code snippets are inserted in the same file which is then imported in the main BPEL process file.

7.8 Implementation of Transformations with ATL

In our projects, the principles of model driven engineering are adopted. Therefore, access control policy specifications are abstract models, BPEL processes, along with WSDL interfaces and XSD type definitions, are concrete models and $\mathcal{L}1$ trees are intermediate models. To implement a transformation from a model into another sort of model, as described in Sects. 7.4 to 7.7, the *ATLAS Transformation Language* (ATL)[4] is used, because its framework is well-suited for conversion of models written in formal languages described with metamodels. The ATL framework is built on the Eclipse platform[5] and provides both a language to express transformation rules and a toolkit to execute those rules. Since, the metamodels for BPEL and WSDL already exist in the form of XSD models, only the metamodel has been defined for the ASTD notation.

In Fig. 29 a model M_1 is transformed into a model M_2. This is done by providing a transformation model (i.e., a set of transformation rules) $\mathcal{M}_1 2\mathcal{M}_2$ which maps elements of M_1's metamodel $\mathcal{M}_1$ to elements of M_2's metamodel $\mathcal{M}_2$. The metamodels $\mathcal{M}_1$ and $\mathcal{M}_2$ as well as ATL are instances of the *Ecore* metamodel.

[4] http://www.eclipse.org/m2m/atl/
[5] http://www.eclipse.org/

```
1 <simpleType name="T">
2   <restriction base="baseT">
3     <enumeration value="x1">
4     ...
5     <enumeration value="xn">
6   </restriction>
7 </simpleType>
```

Figure 27: XSD code for an enumeration

```
1 <simpleType name="T">
2   <restriction base="baseT">
3     <minInclusive value="xl">
4     <maxInclusive value="xu">
5   </restriction>
6 </simpleType>
```

Figure 28: XSD code for an interval

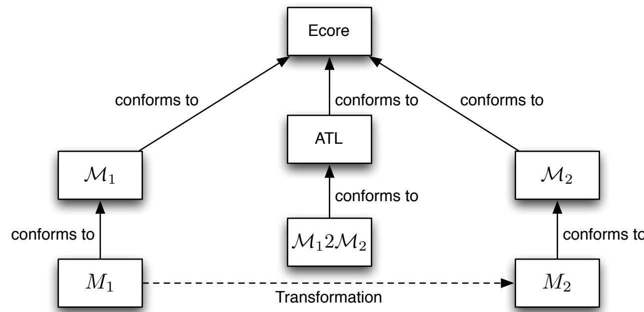


Figure 29: Transforming a model M_1 into a model M_2 using ATL

8 Related Work

The ORKA[6] project includes a practical approach to specify, develop and deploy access control policies. In a first step, UML and OCL are used to define RBAC-like access control policies and the associated constraints, respectively (see [13] for more information on the Role Based Access Control standard). The constraints express usual notions like SoD and *delegation/revocation*. The authors advocate that the UML class diagram of access control policies and OCL constraints used in conjunction with the tool USE are well-suited to validate, *à la* model-checking, security rules against different possible RBAC *configurations*. Inconsistency or lack of completeness can then be detected. In the second step, RBAC policies are expressed in CASL[7] with the aid of linear temporal logic formulas. The proofs are carried out by the theorem prover *Isabelle*[8]. The latter step is more comprehensive than the former, since it involves formal proofs. Therefore, this approach is an attempt to fill the gap between practical design for enforcement of RBAC access control policies and the use of formal methods to verify them. Even if ORKA, SELKIS and EB^{3}SEC projects share similar goals, SELKIS and EB^{3}SEC go one step further. Contrary to OCL, the ASTD notation has powerful constructs to take into consideration *history* of activities as required to deal reasonably well with constraints like SoD. Furthermore, since ASTDs can be automatically translated into BPEL processes under some assumptions, the implementation approach adopted in our projects seem more appropriate in situations where such constraints are frequently used. In addition, since ASTD is the only notation used by designers to specify and verify security policies, there is no need to rewrite constraints in another language if formal proof is required, thus avoiding possible errors. Another important difference is at the implementation level. The integration of our enforcement framework into existing applications is a matter of configuration of the middleware to route messages from the client to the targeted services through corresponding handlers for the PEP to work correctly. A change in the interface of a service may require modifications in the access control policy but no modification at all in the PEP. In the specific PEP mentioned in [25], such a change would require an update (even if it is a little one) to reflect a new interface version.

The ASTD notation is not the only one that has been adapted to specify RBAC-like access control policies. Access right constructs have been recently added to the CaSPiS (Calculus of Services with Pipelines and Sessions) notation [15], which is a calculus to specify Web services by explicitly defining sessions (conversations between clients and servers) and properties like *graceful termination* [5]. CaSPiS in its original version provides a denotational semantics, which has been extended to accommodate access rights. To the best of our knowledge, no further work has been done to exploit this notation in a practical framework.

In our projects, transformations are mainly used to obtain implementation code from a high-level specification. Transformations can also be used in the modeling phase to derive secrecy models from a base (UML) model by iteratively applying *property preserving* transformation operations as proposed in [11]. This work is still in its early stages. The main drawback seems related to the expressiveness of UML

[6]The ORKA Consortium http://www.orka-projekt.de/index-en.htm
[7]http://www.informatik.uni-bremen.de/cofi/wiki/index.php/CASL
[8]http://www.cl.cam.ac.uk/research/hvg/Isabelle/

models, which are limited to the RBAC level of Fig. 18.

Bassin et al. use in [4] *separation of duty algebra* (SoDA), an algebra for SoD developed by Li et al. in [18], to implement access control policies in a framework where workflow are modeled with CSP [12]. This methodology cannot be used to express other type of constraint like *obligation*, as it is possible in our projects. Other various work have been proposed to implement access control on WS. Bertino et al. in [24] have developed a framework to apply RBAC policies to WS using an XACML encoding. Since RBAC itself is not tailored for SoD constraints, their framework uses *Business Process Constraint Language* (BPCL) to express not only SoD rules but a broader range of constraints. In [28], Yao et al. present an architecture for access control around services with support for formal RBAC-like policies which cannot take into account the history of actions. Jajodia et al. propose in [14] a language that supports multiple access control policies for a single system with a focus on conflict resolution. They provide the notion of *history* through a history table. All those frameworks have the same drawback: when they do not support at all constraints like SoD, they use different notations or mechanisms to overcome this limitation. In our projects, we use ASTD as a unified, intuitive and powerful notation to express permissions as well as various constraints including ordering constraints and SoD in particular.

There are other work around BPEL and formal notations. Gibbons et al. describe in [27] an approach to design workflows using CSP. Their methodology is based on the specification of some control flow and state based patterns originally defined by van der Aalst. The latter has define a formal model of workflows in [26] using Petri nets, enriched later by Massuthe et al. in [19]. Both notations have each their disadvantages w.r.t. access control and verification. On one hand CSP does not support state variables and are not well suited for liveness properties. On the other hand, Petri nets does not support quantification which is an important feature when dealing with IS.

9 Conclusion

The aim of the method presented in this paper is to deploy access control policies in a PDP automatically. Security managers are able to specify such policies at the process level using a rigorous notation. The policies are enacted in a BPEL engine as part of an access right enforcement framework. In future work, PEP and PDP will be integrated into a larger framework where additional functionalities (e.g., policy edition for the three levels in Fig. 18) will be provided. The drawback of the approach presented in this paper is linked to the transformation method. Indeed the BPEL process derived by applying transformation rules reproduces the flow of events specified by an ASTD. When the policy is updated, the BPEL process and thus its current execution state have to be recreated from scratch. A way to avoid this is to record ASTDs as XML documents and exploit a BPEL engine as an interpreter for those documents. Specifically, an ASTD specification would be transformed into an XML variable in the BPEL process, and so would be the current state of the ASTD. Since BPEL is not suitable for data manipulation (the `assign` functionality is rather rudimentary), such an interpreter would require a BPEL engine that provides extension points to deal with adequate XML data manipulation and complex computation. For instance, Oracle BPEL Process Manager[9] and GlassFish ESB v2.1[10] support data manipulation through Java and Javascript code respectively.

[9] http://www.oracle.com/technology/products/ias/bpel/index.html

[10] https://glassfish.dev.java.net/

Conclusion

We have presented a combined PIM metamodel, a PSM metamodel and translation rules for a BPEL implementation of access control policies.

As future work we will present a B0 PSM metamodel and refinement steps from the combined PIM B metamodel. We will also compare the different implementations of the access control filter (BPEL, B and using iASTD).

References

[1] Jean-Raymond Abrial. *Modeling in Event-B*. Cambridge University Press, 2010.

[2] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *Software Tools for Technology Transfer*, 12(6):447–466, 2010.

[3] Idir Aït-Sadoune and Yamine Aït-Ameur. Stepwise design of BPEL Web services compositions, an Event B refinement based approach. In *8th ACIS International Conference on Software Engineering Research, Management and Applications*, pages 51–68, 2010.

[4] David A. Basin, Samuel J. Burri, and Günter Karjoth. Dynamic enforcement of abstract separation of duty constraints. In *14th European Symposium on Research in Computer Security*, pages 250–267, 2009.

[5] M. Boreale, R. Bruni, R. Nicola, and M. Loreti. Sessions and pipelines for structured service programming. In *10th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems*, pages 19–38, 2008.

[6] Michel Embe Jiague, Marc Frappier, Frédéric Gervais, Pierre Konopacki, Jérémy Milhau, Régine Laleau, and Richard St-Denis. Model-driven engineering of functional security policies. In *International Conference on Enterprise Information Systems*, volume 3, pages 374–379, 2010.

[7] Michel Embe-Jiague, Richard St-Denis, Marc Frappier, Frederic Gervais, and Regine Laleau. From astd access control policies to wsbpel processes deployed in a soa environment. In *1st International Symposium on Web Intelligent Systems and Services (WISS 2010)*, LNCS. Springer-Verlag, 2010. To be published. 15 pages.

[8] M. Frappier, F. Gervais, R. Laleau, and B. Fraikin. Algebraic state transition diagrams. Technical Report 24, Département d'informatique, Université de Sherbrooke, 2008.

[9] M. Frappier, F. Gervais, R. Laleau, B. Fraikin, and R. St-Denis. Extending statecharts with process algebra operators. *Innovations in Systems and Software Engineering*, 4(3):285–292, 2008.

[10] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[11] W. Hassan, N. Slimani, K. Adi, and L. Logrippo. Secrecy UML method for model transformations. In *2nd International Conference ABZ Short Papers*, pages 16–21, 2010.

[12] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[13] INCITS. *Role Base Access Control*. ANSI, 2004.

[14] Sushil Jajodia, Pierangela Samarati, Maria Luisa Sapino, and V. S. Subrahmanian. Flexible support for multiple access control policies. *ACM Transactions on Database Systems*, 26(2):214–260, 2001.

[15] M. Kolundzija. Security types for sessions and pipelines. In *5th International Workshop on Web Services and Formal Methods*, volume LNCS 5387, pages 175–190, 2009.

[16] P. Konopacki, M. Frappier, and R. Laleau. Expressing access control policies with an event-based approach. Technical Report TR-LACL-2010-6, LACL (Laboratory of Algorithms, Complexity and Logic), University of Paris-Est (Paris 12), 2010.

[17] P. Konopacki, M. Frappier, and R. Laleau. Modélisation de politiques de sécurité à l'aide d'une algèbre de processus. *RSTI - Ingénierie des systèmes d'information*, 15(3), 2010.

[18] Ninghui Li and Qihua Wang. Beyond separation of duty: an algebra for specifying high-level security policies. In *13th ACM Conference on Computer and Communications Security*, pages 356–369, New York, NY, USA, 2006. ACM.

[19] Peter Massuthe, Wolfgang Reisig, and Karsten Schmidt. An operating guideline approach to the SOA. *Annals of Mathematics, Computing & Teleinformatics*, 1:35–43, 2005.

[20] Jérémy Milhau, Marc Frappier, Frédéric Gervais, and Régine Laleau. Systematic translation rules from ASTD to Event-B. In Dominique Méry and Stephan Merz, editors, *Integrated Formal Methods*, volume 6396 of *Lecture Notes in Computer Science*, pages 245–259. Springer Berlin / Heidelberg, 2010.

[21] Gustaf Neumann and Mark Strembeck. An approach to engineer and enforce context constraints in an rbac environment. In *Proceedings of the eighth ACM symposium on Access control models and technologies*, SACMAT '03, pages 65–79, New York, NY, USA, 2003. ACM.

[22] OASIS. *eXtensible Access Control Markup Language (XACML) Version 2.0.* OASIS, 2005.

[23] OASIS. *Web Services Business Process Execution Language Version 2.0.* OASIS, 2007.

[24] Federica Paci, Elisa Bertino, and Jason Crampton. An access-control framework for WS-BPEL. *International Journal of Web Services Research*, 5(3):20–43, 2008.

[25] K. Sohr, T. Mustafa, X. Bao, and G.-J. Ahn. Enforcing role-based access control policies in Web services with UML and OCL. In *24th Annual Computer Security Applications Conference*, pages 257–266, Washington, DC, USA, 2008. IEEE Computer Society.

[26] W. M. P. van der Aalst. The application of Petri nets to workflow management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[27] Peter Y. H. Wong and Jeremy Gibbons. A process-algebraic approach to workflow specification and refinement. In *Software Composition*, pages 51–65, 2007.

[28] Walt Yao, Ken Moody, and Jean Bacon. A model of OASIS role-based access control and its support for active security. In *6th ACM Symposium on Access Control Models and Technologies*, pages 171–181, New York, NY, USA, 2001. ACM.