ORKA – Organizational Control Architecture

Deliverable AP 2.6

Transformation of OPL/XML into Enforceable Formats

Aug 31, 2008

Christopher Alm, University of Passau
Robert Fies, SAP AG
Roland Illig, University of Passau
Mathias Kohler, SAP AG
Wanja Johannes Slawski, University of Passau

ORKA is funded by the German Ministry of Education and Research (BMBF) as part of its Software Engineering 2006 program.

Gefördert vom
Bundesministerium
für Bildung
und Forschung

© 2006 ORKA Consortium

Internal document information:

`$Id: del-ap3.5.tex 1017 2008-03-31 08:43:56Z kohler $`

Contents

1 Introduction **4**

2 PDC Architecture **4**

3 Transformation OPL - DROOLS Policy Objects **5**
3.1 Overview 5
3.1.1 Component Overview of the Transformation Framework 6
3.1.2 Package Overview 7
3.2 Transformation Process Description 8
3.2.1 General Outline of the Transformation Process 8
3.2.2 Resolving System-Specific Process and Task Identifiers 12
3.3 Adding new Policy Modules 14

4 Transformation OPL - XACML Policy Objects **15**
4.1 Overview 15
4.2 Translation of Policy Modules 15
4.2.1 RBAC-Core 15
4.2.2 RBAC with Role Hierarchy 17
4.2.3 Context Constraints 17
4.3 Package Overview 18

5 Java OPL Engine **18**
5.1 Overview 18
5.2 The Modularization 19

6 Performance Tests **20**

7 Conclusions **20**

1 Introduction

In the previous workpackages of SPEC, we have developed a policy language for the ORKA system (Ap2.4) that is on the one hand able to express a wide range of organizational control principles. On the other hand, it takes into account the requirements coming from enforcement, administration, and validation. This development was based on an extensive requirements analysis (Ap2.2) and on an analysis of existing access control models, systems, and policy languages (Ap2.1, Ap2.3). A programming interface has been developed with which we can administer all aspects of the policy language such as adding users, assigning roles, and placing new constraints (Ap2.5). Finally, in order that OPL can be used by the ORKA system, a mechanism is required that is able to evaluate OPL policies and to compute access decisions accordingly. This is the goal this workpackage.

The idea is to make use of existing techologies—namely Drools and XACML—which are promising in terms of policy evaluation. In particular, XACML provides a dedicated access control policy evaluation engine. Furthermore, this document also introduces a policy engine implemented in Java that is able to evalutate OPL policies natively. As a result, we obtain three policy engines that can be compared in a performance test.

The organization of this document is straightforward: in Section 2 we explain the role of the PDCs in the overall ORKA architecture. The Sections 3, 4, and 5 introduce the Drools, XACML, and the native engine, respectively. Finally, in Section 6 we briefly refer to the performance test which is carried out in Ap3.7. Section 7 draws conclusions and summarizes our main results.

2 PDC Architecture

A policy decision engine (PDC) is a class that inherits from the abstract class `PolicyDecisionComponent`. Such an engine can evaluate policy objects and make access decisions. An instance of a policy decision component has loaded all the policy objects it is responsible for and every time it evaluates a request the appropriate policy object is selected.

The methods declared through the abstract class are the following:

- `initialize()` is invoked once when the policy decision component is created. Establishing a database connection is an example that can take place at this point.
- (**Transformation**) `registerPolicyObject()` is invoked in order to load a policy object into the policy decision component. Note that a each policy decision engine needs to maintain an internal list of policy objects according to which it can perform access decisions. At this point all transformations are carried out that are necessary to transform the policy from its XML representation (which is given in the policy object) to the internal representation of the policy decision engine. There we have three different representations for our engines:
  - XACML for the XACML engine
  - Drools facts for the Drools engine

  - A Policy-Object-Tree for the Java native engine.
- `unregisterPolicyObject()` is invoked in order to clear a policy object from the internal list of policy objects.
- `performAccessDecision()` is the actual access decision maker. This method is invoked in order to respond to an access request. Special about this method is that besides the access requests it gets a context manager and a policy object as a parameter. The latter is given so that the policy decision engine can identify the policy object from its internal list according to which the given request should be evaluated. The former is used to provide the appropriate context manager which should be used to perform context lookups for this request. In particular, in this way the context manager is able to provide information related to the given request such as the current workflow instance and the current task instance.

Access decisions may depend on context information which needs to be retrieved dynamically. This can be done by calling the `lookup()` method provided by the context manager. As a parameter `lookup()` requires a string in the form

`contextprovider.function(parameter, ...)`

In particular, nesting is allowed. For example

```
wfms.getTasks(parameters.workflowinstance)
```

Here `wfms` and `parameters` are the context providers.

3 Transformation OPL - DROOLS Policy Objects

3.1 Overview

The general function of the Drools-based PDC is to enforce access control request evaluation according to both the policies specified in the OPL/XML format, and the general enforcement semantics of the policy modules which are specified as Object Z definitions in AP 2.4.

In order to fulfill this function, the Drools evaluation engine needs two relevant parts: the Rules that comprise the general evaluation semantics, and the Facts on which these rules are to be enforced.

The rules for evaluation are generic and independent from any specific OPL-based policy object, that is, they are derived from the Object Z semantics of the policy modules they are to enforce and are valid for all access control requests and policy objects concerning these modules. Therefore, a set of PERMIT and DENY rules is written only once for each policy module according to its specification, and these rules can subsequently be used to evaluate any combination and number of policy objects and access control requests concerning these modules.

The fact objects, however, are not generic but contain the specific information of each OPL policy object. Therefore, they have to be created as instances whenever a new OPL policy object is to be enforced. These fact object classes need to be structured in such a way that they contain

all the relevant information specified in the OPL/XML policy object definitions, and they have to be designed in such a way that the Drools rules can be enforced on them for each request efficiently. For each policy object, a transformation from OPL to instances of fact objects has to be performed. This is the responsibility of the OPL/Drools Transformation Framework that is described in this chapter.

3.1.1 Component Overview of the Transformation Framework

The Transformation Framework consists of several interdependent components:

- DroolsPolicyObjectTransformator: The central class for the Transformation Framework. It can transform a complete OPL policy object into a policy object that contains a number of fact objects for Drools evaluation that can then be put into the Drools PDC's working memory. This way, it can encapsulate and hide the entire complexity of the transformation process and provide a single, clear interface for the complete transformation. To these ends, it implements the PolicyConversionInterface that can also be implemented by other Transformation Frameworks (e.g. for OPL/XACML).
- ACObject and its derived subclasses: The class hierarchy for the Drools constraint classes. It comprises a result String (e.g. "PERMIT" or "DENY") and a Target that can specify a power set of Subjects, Actions and Resources for which the constraint should be relevant. This Target is used by the Drools rules for matching Requests to constraint fact objects. Additionally, the subclasses contain all the necessary information from their respective OPL elements. The subclass hierarchy of ACObjects loosely follows the Object Z class hierarchy and shares some naming similarities with it. However, since it was specifically designed to facilitate an efficient enforcement by Drools rules, the class hierarchies are not structured entirely identically. One of the main differences is that the fact object hierarchy comprises of more finely-grained class definitions that can efficiently enforce semantically independent constraints within and in between Object Z class definitions.
- TransformationStrategy and its implementations: The Transformator itself does not implement the entire algorithmic complexity of the full policy object transformation. Instead, a Strategy pattern is used to facilitate modularization and flexibility. Each OPL constraint element that can be transformed into a specific ACObject subclass is registered with the Transformator and correlated with its own TransformationStrategy implementation class. Each such implementation algorithmically encapsulates the transformation code for its respective OPL element. For those cases in which an element is fundamentally or partially dependent on other independent elements that are transformed in their own strategy implementations (e.g. RBACCoreM - WFCoreM), references can be provided via a global Map that can be accessed by all TransformationStrategy implementations in the Transformator.
- Fact classes: While the ACObject class and its derived classes implement the constraints specified in OPL elements, even finer-grained fact classes are used to represent single entities within the policy. Examples of such entities are Action, Resource, Subject, Role or Permission. These fact objects comprise the fundamental core content of each policy object. In order to represent the OPL definitions fully, they can reference each other, and they are generally referenced by the ACObject constraint classes. These references and

cross-references in their entirety represent the OPL content in the working memory, and in this sense, they are at the heart of the enforcement by Drools.

3.1.2 Package Overview

Several Java packages are relevant for the OPL/Drools Transformation Framework. The following list provides a short overview over these packages and their organizational structure.

- org.orka.technologies.drools.constraints: This package contains the constraint objects that are inserted into the Drools working memory as facts, to be evaluated according to the rules. There is one basic class, ACObject, from which a number of subclasses is derived. These subclasses correspond to the constraints specified in the OPL/XML elements, and a set of their instances is the result of the transformation by the Transformation Framework. Every constraint subclass inherits the target and result fields. The target is used to match Requests to relevant constraints by the rules according to a combination of Subjects, Actions and Resources. The result is a String field that starts as "NEUTRAL" and is subsequently evaluated to "PERMIT" or "DENY" when relevant rules fire for a constraint in the working memory.
- org.orka.technologies.drools.facts: A package comprising the finely-grained fact classes. These classes mostly represent data entities like Subjects, Actions, Permissions, Roles and so on, and their main use is to be aggregated, referenced and mutually assigned to each other in constraint subclasses, so they can provide the semantic content for evaluating specific requests matching specific constraints according to general rules.
- org.orka.technologies.drools.transformation: The most important class in this package is the Transformator itself: DroolsPolicyObjectTransformator. This Transformator class publicly provides the complete transformation of OPL policy objects into Drools policy objects as a single method. In order to accomplish this, it registers a number of TransformationStrategy implementation classes. TransformationStrategy is an interface that is also contained within the transformation package, along with all of its implementing classes. Each such implementation algorithmically encapsulates the transformation of one specific OPL element to one specific constraint class. The Transformator recognizes all transformable OPL elements in a complete document and transforms them by delegating each specific transformation to the respective TransformationStrategy implementation. This Strategy pattern design vastly improves modularity and flexibility.
- org.orka.technologies.drools.rules: This package is not really a part of the Transformation Framework. Actually, it is not even really a Java package. However, it is still important when considering transformation and DroolsPDC enforcement because in this folder, all the .drl rules files are contained, and the rules are at the heart of the evaluation of the transformed policy objects. Consequently, in order to introduce new OPL elements for transformation and enforcement, new rules have to be written and added to this folder and registered in the DroolsPDC as well.

3.2 Transformation Process Description

3.2.1 General Outline of the Transformation Process

This section provides a general description of the Transformation process that occurs when an OPL-based PolicyObject is to be transformed into Drools constraint objects that are to be inserted into the working memory for request evaluation. This whole process is performed by calling the conversion method provided by the DroolsPolicyObjectTransformator class.

The DroolsPolicyObjectTransformator is at the core of the OPL/Drools Transformation Framework. It implements the PolicyConversionInterface, which provides one method to convert a PolicyObject into another kind of PolicyObject.

Listing 1: PolicyConversionInterface

```
1
2 package org.orka.enforce.interfaces;
3
4 import org.orka.enforce.db.PolicyObject;
5
6 /**
7  * basic policy object conversion functions
8  *
9  * @author Stefan Kowski
10  */
11 public interface PolicyConversionInterface
12 {
13         /**
14          * convert policy object
15          *
16          * @param policy
17          * @return
18          */
19         public PolicyObject convert(PolicyObject policy);
20 }
```

By implementing this conversion method, the DroolsPolicyObjectTransformator encapsulates the complete Transformation logic of OPL-based PolicyObjects into PolicyObjects that include Drools constraint objects to be inserted into the working memory for Drools evaluation.

There is a specific PolicyObject subclass for the DroolsPDC. It maintains a List of ACObjects, which are the transformed constraint objects that are derived from the constraint specifications provided by the OPL/XML. These constraint objects are to be put into the working memory in the DroolsPDC before any request can be evaluated.

Listing 2: DroolsPolicyObject

```
1 package org.orka.enforce.lang.drools;
2
3 import java.util.List;
```

```
4
5 import org.orka.enforce.db.PolicyObject;
6 import org.orka.technologies.drools.constraints.ACObject;
7
8 /**
9  * PolicyObject subclass for Drools enforcement.
10  * It contains the constraint elements as a List of ACObjects (to be used as constraint facts
       by Drools).
11  *
12  * @author Robert Fies, SAP Research
13  *
14  */
15 public class DroolsPolicyObject extends PolicyObject
16 {
17         /**
18          * Default Constructor.
19          */
20         public DroolsPolicyObject()
21         {
22                 super();
23         }
24
25         /**
26          * Convienience Constructor.
27          *
28          * @param facts, a List of ACObjects (drools constraint facts for policy constraints)
29          */
30         public DroolsPolicyObject(List<ACObject> facts)
31         {
32                 super();
33                 this.facts = facts;
34         }
35
36         private List<ACObject> facts;
37
38         public List<ACObject> getFacts()
39         {
40                 return facts;
41         }
42
43         public void setFacts(List<ACObject> facts)
44         {
45                 this.facts = facts;
46         }
47
48 }
```

The DroolsPolicyObjectTransformator is responsible for converting an OPL-based PolicyObject into such a DroolsPolicyObject that includes the constraint fact objects. In order to fulfill this requirement, it needs knowledge of the OPL elements that are to be transformed, the algorithmic steps that are to be taken in order to transform each element, the interdependencies that may exist between certain elements, and the constraint class hierarchy that, in its entirety, provides a data structure that is suitable to represent the semantic content of the OPL definitions.

The knowledge of OPL elements that can be transformed into ACObjects (the constraint objects

for the working memory) is provided in a static Map that maps the OPL element identifiers to PolicyModuleTypes, which are provided in a public enum. It is important to note that the Transformation framework is not able to read the OPL DTD or any other form of semantic or syntactic definition. It recognizes only the elements that are registered within the Transformator and mapped to a PolicyModuleType.

The knowledge of how these elements have to be transformed into constraint objects is registered in a second static Map that maps the PolicyModuleTypes from the enum to TransformationStrategy implementations. Each such implementation algorithmically encapsulates the complete transformation of its respective OPL element into a constraint object.

Listing 3: Mappings

```
1        /**
2         * This Map is used to find the name of each starting element in an OPL file for each
              kind of policy module type.
3         * The mapping is hard-coded and should be changed if OPL specs or policy module types
               should ever be changed or expanded.
4         */
5        public static Map<String, PolicyModuleType> ALL_MODULE_ELEMENT_NAMETYPES = new HashMap
             <String, PolicyModuleType>();
6
7        /**
8         * This Map holds all Strategy objects referenced by Module types:
9         */
10       public static Map<PolicyModuleType, TransformationStrategy> ALL_STRATEGIES = new
             HashMap<PolicyModuleType, TransformationStrategy>();
11
12       // initialize mappings:
13       static
14       {
15               ALL_MODULE_ELEMENT_NAMETYPES.put("module_rbac_core_policy",
16                               PolicyModuleType.RBACCoreM);
17               ALL_MODULE_ELEMENT_NAMETYPES.put("task_role_assignments",
18                               PolicyModuleType.WFCoreM);
19               ALL_MODULE_ELEMENT_NAMETYPES.put("module_wf_core_policy",
20                               PolicyModuleType.WFCoreTPAM);
21               ALL_MODULE_ELEMENT_NAMETYPES.put("critical_tasks",
22                               PolicyModuleType.WFSepDutyM);
23               ALL_MODULE_ELEMENT_NAMETYPES.put("hdsodtp_partitions",
24                               PolicyModuleType.WFSepDutyTPM);
25               ALL_MODULE_ELEMENT_NAMETYPES.put("hdsodtpcc_partitioning",
26                               PolicyModuleType.WFSepDutyTPCCM);
27               ALL_MODULE_ELEMENT_NAMETYPES.put("context_constraint",
28                               PolicyModuleType.ContextConstraint);
29               ALL_MODULE_ELEMENT_NAMETYPES.put("pcc", PolicyModuleType.PCC);
30               ALL_MODULE_ELEMENT_NAMETYPES.put("rcc", PolicyModuleType.RCC);
31               ALL_MODULE_ELEMENT_NAMETYPES.put("pacc", PolicyModuleType.PACC);
32
33               ALL_STRATEGIES.put(PolicyModuleType.RBACCoreM,
34                               new RBACCoreTransformationStrategy());
35               ALL_STRATEGIES.put(PolicyModuleType.WFCoreM,
36                               new WFCoreTransformationStrategy());
37               ALL_STRATEGIES.put(PolicyModuleType.WFCoreTPAM,
38                               new WFCoreTPATransformationStrategy());
39               ALL_STRATEGIES.put(PolicyModuleType.WFSepDutyM,
```

```
40                                  new WFSepDutyTransformationStrategy());
41                  ALL_STRATEGIES.put(PolicyModuleType.WFSepDutyTPM,
42                                  new WFSepDutyTPTransformationStrategy());
43                  ALL_STRATEGIES.put(PolicyModuleType.WFSepDutyTPCCM,
44                                  new WFSepDutyTPCCTransformationStrategy());
45                  ALL_STRATEGIES.put(PolicyModuleType.ContextConstraint,
46                                  new ContextConstraintTransformationStrategy());
47                  ALL_STRATEGIES.put(PolicyModuleType.PCC, new PCCTransformationStrategy());
48                  ALL_STRATEGIES.put(PolicyModuleType.RCC, new RCCTransformationStrategy());
49                  ALL_STRATEGIES.put(PolicyModuleType.PACC,
50                                  new PACCTransformationStrategy());
51
52          }
```

Using this information, the Transformator can perform the complete transformation of an OPL document into a List of constraint objects. In order to accomplish this transformation, the following steps are performed:

1. The OPL is read and converted into a JDOM document. JDOM is an XML parsing framework that provides methods for traversing an XML document, extracting attributes and searching child and parent nodes. All elements that are recognized as transformable are extracted as JDOM Element objects.

2. For each such element, the appropriate TransformationStrategy is looked up in the Map. There is a Strategy implementation for each OPL element that can be transformed by the Transformator. Each such strategy encapsulates the transformation of one element into a constraint object. The Strategy‘s transform method is called on the Element by the Transformator, which has no detailed knowledge of the transformation algorithms for each Element. The resulting constraint objects are gathered by the Transformator.

3. The list of ACObjects that are provided by the TransformationStrategies is assembled, and a new DroolsPolicyObject is created that includes that list. The DroolsPolicyObject is then returned by the Transformator as a result value of its conversion method, therefore providing the complete end result of the transformation.

Listing 4: TransformationStrategy

```
1 package org.orka.technologies.drools.transformation;
2
3 import java.util.Map;
4
5 import org.jdom.Element;
6 import org.orka.technologies.drools.constraints.ACObject;
7
8 /**
9  * This interface is the basic interface for all OPL/XML policy object transformation
       strategies.
10 * There is an strategy implementation for each type of policy object that algorithmically
       encapsulates the relevant kind of transformation.
11 *
12 * @author Robert Fies, SAP Research
13 *
```

```
14  */
15 public interface TransformationStrategy
16 {
17
18         /**
19          * This is the default action name for WF task assignment actions. It is used whenever
                there is no specified action for a task in the OPL.
20          */
21         public static final String TASK_ASSIGNMENT_ACTION = "assign";
22
23         /**
24          * Neutral default result String.
25          */
26         public static final String NEUTRAL = "NEUTRAL";
27
28         /**
29          * This method encapsulates the transformation of the opl input into an adequate
               policy module object for Drools and returns that object.
30          *
31          * @param oplPolicyElement - a JDOM Element with the XML/OPL module input to be
               transformed
32          * into an ACObject - this Element is supposed to contain the basic XML element for
33          * this transformation, e.g. <module_rbac_core_policy> or <module_wf_sep_duty_policy>.
34          *
35          * @return the ACObject carrying the transformed policy information.
36          *
37          */
38         public ACObject transform(Element oplPolicyElement,
39                         Map<String, Object> globals)
40                 throws PolicyTransformationException;
41
42 }
```

3.2.2 Resolving System-Specific Process and Task Identifiers

One requirement for the transformation from OPL/XML to a specific enforcement representation like Drools facts is the system-specific resolution of process and task identifiers. In workflow-related OPL constraints, the resource in question for a given access control request often is a task or process template. A common example is the restriction of task assignments according to Separation of Duty constraints. Similarly, the creation of a process instance for a given process template could be defined as an access control-relevant action, for which permissions are assigned in the RBAC module in the OPL policy.

Whenever a process or task is defined as a potentially restricted resource in any OPL constraint module, the workflow engine that is connected to the ORKA architecture may reference the process or task in question as a resource in access control requests, and for this reference, it may use system-internal identifiers. It cannot be guaranteed that these system-specific identifiers are identical to the ones that are specified in the OPL policy. Instead, it is useful to specify identifiers in the OPL that can be resolved in a system-specific way during the transformation process using calls to context providers that are connected to the workflow engine and implement the knowledge of what kind of system-specific identifiers are used to represent a given process or

task. This way, the specification of the tasks and processes in the OPL policy is as system-independent as possible while still providing the full expressional power that ORKA makes available for administrators to restrict and permit user actions on the workflow level.

If a given resource that is specified in an OPL constraint represents a process or task, it is marked with one of the `process:` and `task:` prefixes in its respective OPL/XML element. Such elements include `object_id` or `task_id`, e.g. in permission definitions in the RBAC module, or in Separation of Duty constraints. When these elements are transformed to Drools fact objects, the prefixes are recognized by the transformation framework. Then, the actual identifiers are resolved using context providers that are connected to the workflow engine.

Of course, the transformation framework needs knowledge of which context providers to use, and what methods to call on them. This knowledge is provided by specific `<wfms_attribute>` elements in the `<wfms>` element that is specified as part of the `<module_wf_core_policy>` module. A `<wfms_attribute>` consists of a key and value pair, and the transformation framework recognizes the defined keys `process_name_to_id` and `process_task_name_to_id` that signify the context providers and their methods that can be called to resolve system-specific process and task identifiers, respectively.

Since the actual transformation of each constraint module is performed in strategy implementations that algorithmically encapsulate the transformation steps for each module, the Transformator provides these implementations with an instance of an `ProcessAndTaskIdResolver` helper class, which has access to the Context Manager and the knowledge from the `<wfms_attribute>` elements about context providers and their resolution methods. Whenever a transformation strategy transforms an element that may be a process or task resource, the identifier can be resolved with a simple call to the `ProcessAndTaskIdResolver`'s `checkAndResolveIds()` method that expects the element name as an input parameter. Inside this method, the Resolver component checks for the `process:` and `task:` prefixes and, if necessary, resolves the identifiers using context lookups.

By using this design, the transformation framework can create system-specific constraint facts even though the process and task names in the OPL definitions may be system-independent. Note that, since tasks always belong to process definitions, they are specified by a composition of their parent process identifier and a unique task identifier while the processes are simply specified by plain identifiers.

Listing 5: example WFMS definition with resolution information

```
1 <wfms>
2   <wfms_attribute key="process_name_to_id" value="wfs.getProcessId(processName)"/>
3   <wfms_attribute key="process_task_name_to_id" value="wfs.getTaskId(processName|taskName)"/>
4   <wfms_attribute key="user_task_history" value="wfs.getTaskHistory(parameters.subject)"/>
5   <wfms_attribute key="user_current_tasks" value="wfs.getCurrentTasks(parameters.subject)"/>
6 </wfms>
```

3.3 Adding new Policy Modules

Due to the modularized approach to OPL/Drools policy object transformation, OPL elements can be transformed independently from each other to a high degree, which enables the framework to provide its basic functionality without requiring that every theoretically possible OPL constraint be transformed and enforced. As of yet, the transformation is implemented mainly for those constraints which are relevant in the Banking Process reference scenario. However, it is easily possible to add other policy modules to the transformation and Drools enforcement. The possibilities are potentially limitless: Apart from the modules currently specified in Object Z that are not yet implemented, completely new constraint modules can be added at any time. By introducing new OPL elements and fact classes, constraints that include completely new semantics can potentially be enforced as well.

Here is an outline of all the necessary steps that have to be undertaken if a new policy module is to be transformed and enforced:

1. A new OPL element has to be introduced that can express the constraint content in a specific OPL/XML-based policy object. If that element is already included in the OPL language specification, and only its transformation is not implemented yet, this step can be omitted.

2. In the rare case that a completely new fact class should be introduced by the constraint element (i.e. a completely new entity class like Role, Resource, Action, Permission etc.), that class has to be defined in the org.orka.technologies.drools.facts package.

3. One or more ACObject subclasses have to be introduced and implemented. These classes provide the constraint facts for the Drools working memory and should be designed in such a way that they contain all the constraint information that is specified in the respective OPL element. Mostly, they will include data fields that can hold references to relevant fact objects like mutually exclusive Resources, assignments of Roles to Permissions, or any such semantic information that can be extracted from the OPL element. The class names of the new ACObject subtypes have to be added to the public PolicyModuleType enum that contains each type so they can be used by the Transformator for mapping constraint types to OPL elements and TransformationStrategy implementations.

4. A new TransformationStrategy implementation class has to be implemented for each new ACObject subclass. In its transform() method, the complete transformation of the OPL policy element into a new constraint object has to be implemented. In addition, the object‘s Target should be filled with a selection of Subject, Action and Resource objects depending on the OPL element‘s content in such a way that only relevant Requests will be matched to the Target during Drools enforcement. In the rare case that other OPL elements are to be made dependent on the content of a new element, the necessary fact object references have to be put into the globals Map that is provided by the Transformator to each Strategy implementation, with the fact element‘s unique id String as a key. In the Transformator, it is vital that the OPL elements on which other elements rely be transformed before all others. This can be asserted by simple type checking in the Transformator‘s extractModuleElements() method. This is a special requirement that is only relevant for modules on which other modules are dependent.

5. Both the Strategy implementations and the OPL elements have to be mapped to the new

constraints‘ enum types in the Transformator‘s static Maps.

6. A new set of Drools rules has to be manually written. These rules should express the matching of Requests to the ACObjects‘ Targets, and the semantic evaluation logic that leads to "PERMIT" or "DENY" results for each such request/constraint object match. Once written into .drl files, they also have to be registered in the DroolsPDC so they can be considered for evaluation.

4 Transformation OPL - XACML Policy Objects

4.1 Overview

The XACML PDC translates OPL policies into XACML policies, which are then stored as Java objects in memory. They are currently not written to disk, but could be if the need arises. The main entry point of this transformation component is the XacmlPdc class, which implements the PolicyDecisionComponent interface.

XACML is a rule-based policy description language, written in XML. An XACML ⟨Request⟩ consists of a set of attributes, each having an identifier and one or more values. These attributes are then analyzed by the policies written in the XACML policy language. Its main components are ⟨Rule⟩s, which evaluate to either Permit, Deny, NotApplicable or Indeterminate. The next bigger components are ⟨Policy⟩s, which combine ⟨Rule⟩s, and ⟨PolicySet⟩s, which combine ⟨Policy⟩s and other ⟨PolicySet⟩s.

4.2 Translation of Policy Modules

Each OPL policy module is translated into a corresponding XACML policy. These policies are later combined to form a “complete policy”. In each rule, the evaluation result is determined by applying a standard set of functions such as string-equal, and or any-of to the attributes. Out of these function applications, arbitrarily complex expressions can be produced and encoded into an XACML ⟨Condition⟩ element. For the purpose of allowing simple optimizations in the XACML policy evaluation component, each ⟨Rule⟩, ⟨Policy⟩ and ⟨PolicySet⟩ has a ⟨Target⟩ element, which works similar to a ⟨Condition⟩ and is used for quickly checking whether the ⟨Target⟩’s parent element is applicable to the current request. The target has a simplified, constrained structure that doesn’t allow arbitrary expressions. It is used in the translation of the OPL policies whenever possible to make use of these possible optimizations. The remaining things that don’t fit into the ⟨Target⟩ structure are then encoded as a ⟨Condition⟩.

4.2.1 RBAC-Core

The CheckAccess function of the RBAC-Core module can be implemented as follows (shown in the Python programming language):

Listing 6: The CheckAccess function of the RBAC-Core module, highlevel code

```
def checkAccess(s, op, ob):
    roles = SR.lookup(s)
    perms = PA.lookup(roles)
    return perms.contains((op, ob))
```

The *PA* relation is available at translation time, while the current roles from the *SR* relation have to be resolved at evaluation time. The code that needs to be translated to XACML would look like the following, this time expressed in the Scheme programming language:

Listing 7: The CheckAccess function of the RBAC-Core module, intermediate code

```
(define (check-access role op ob)
  (or
    (and (string-equal role "role1") (string-equal op "op1") (string-equal ob "ob1"))
    (and (string-equal role "role2") (string-equal op "op2") (string-equal ob "ob2"))
    (and (string-equal role ... ) (string-equal op ... ) (string-equal ob ... ))
    (...)))
```

This representation can be easily translated into the final XACML form. First, each of the and nodes is translated into an XACML ⟨Rule⟩, and then all these rules are combined using a permit-overrides algorithm. The following XACML listings, although quite lengthy, are not shown in real XACML but rather in an abbreviated variant of it, to save space and keep it legible.

Listing 8: XACML fragment for one tuple (r, ob, op) from the *PA* relation

```
⟨Rule Effect="Permit"⟩
 ⟨Target⟩
  ⟨Subjects⟩
   ⟨Subject⟩
    ⟨SubjectMatch MatchId="string-equal"⟩
     ⟨AttributeValue⟩r⟨/AttributeValue⟩
     ⟨SubjectAttributeDesignator AttributeId="orka:role"/⟩
    ⟨/SubjectMatch⟩
   ⟨/Subject⟩
  ⟨/Subjects⟩
  ⟨Resources⟩
   ⟨Resource⟩
    ⟨ResourceMatch MatchId="string-equal"⟩
     ⟨AttributeValue⟩ob⟨/AttributeValue⟩
     ⟨ResourceAttributeDesignator AttributeId="resource-id"/⟩
    ⟨/ResourceMatch⟩
   ⟨/Resource⟩
```

```
18    </Resources>
19    <Actions>
20     <Action>
21      <ActionMatch MatchId="string-equal">
22       <ActionAttributeDesignator AttributeId="action-id"/>
23       <AttributeValue>op</AttributeValue>
24      </ActionMatch>
25     </Action>
26    </Actions>
27   </Target>
28  </Rule>
```

The attribute names resource-id and action-id are defined in XACML, while the name orka:role is application-specific. These attributes will have to be provided to the XACML PDP by the ORKA system.

All these rules ($r\ldots$) are then combined to the final RBAC-Core policy as follows:

Listing 9: The complete XACML policy for RBAC-Core

```
1  <Policy RuleCombiningAlgorithmId="permit-overrides">
2   <Target/>
3   r...
4  </Policy>
```

4.2.2 RBAC with Role Hierarchy

Once the RBAC-Core module is translated, translating the RBAC with Role Hierarchy module becomes trivial. The only thing that changes is that the roles for the subject may be more than before. Since that information is never used in the translation process, the translated XACML policy is almost the same. The only exception is that the policy will not query the orka:role attribute, but orka:transitive-role instead.

4.2.3 Context Constraints

Each context constraint (CC) represents a function call with custom parameters. These function calls are translated into calls of the XACML standard functions. The mapping is relatively straight-forward, since the OPL data types are all supported by XACML. When translating a context function parameter (CFParam) into an XACML expression, there are two possible cases:

- The CFParam is a literal parameter. In that case, the literal's value is translated in a straight-forward way to an XACML literal of the corresponding data type.
- The CFParam has type "context". In that case, the CFParam is translated into an ⟨EnvironmentAttributeDesignator⟩, and the value of the CFParam is encoded in the attribute name. This parameter will be resolved later by the same instance that also resolves the orka:role attribute.

Special care has been taken to translate the Context Constraints module exactly as specified. An intuitive approach, which also would have been easy to transform, might take the XACML policy from the RBAC-Core module and augment each ⟨Rule⟩ with the CCs that are assigned to the role, the permission and the role permission assignment. This approach would not match the specification, which states that even roles that are not necessary for a specific access are subject to CC evaluation. So when a CC of such a role fails, the access must be denied.

Therefore, the generated policy has two parts: The first checks if any of the applicable CCs fails, and if so, the access is denied. Only if all the applicable CCs succeed, the outcome of the CC policy is the same as that of the RBAC-Core module.

4.3 Package Overview

The transformation to XACML is implemented in the org.orka.enforce.lang.xacml Java package, which contains (among others) the following Java classes:

XacmlPdc: The main entry point of the XACML transformation.

OrkaAttributeFinder: The component that is responsible for resolving the dynamic attributes like orka:role or the context function parameters.

modules.RbacCoreTranslator: The translator for the RBAC-Core module.

modules.RbacRoleHierarchyTranslator: The translator for the RBAC with Role Hierarchy module.

transform.CCTranslator: The translator for the Context Contraints module.

transform.*: Helper classes for the Context Constraint translator.

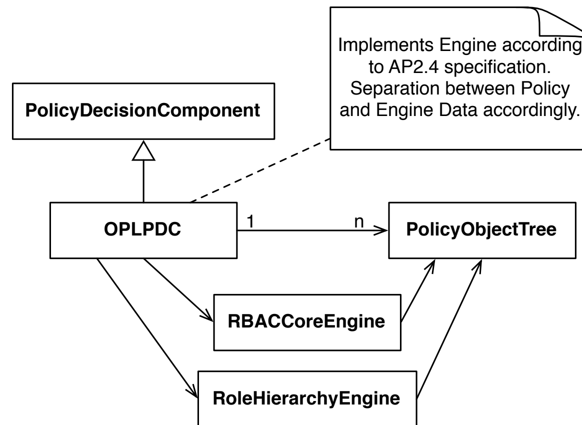
util.*: Generic helper classes.

5 Java OPL Engine

5.1 Overview

The Java OPL Engine is implemented according to the AP2.4 specification. Each authorization module such as RBACCorePolicy or ExoContextPolicy has its own enforcement module. For example the enforcement of the RBACCorePolicy authorization module is implemented in the according RBACCoreEngine module.

As shown by in the UML diagram depicted below in this section, all engine modules are aggregated in the OPLPDC class. This class, in turn, inherits from the abstract class PolicyDecision Component described in Section 2.



5.2 The Modularization

The enforcement system is modularized in such a way that each access control principle has a dedicated engine responsible for it. The modules are aggregated and managed by the OPLPDC class.

The access decision process is stepwise:

- Each PolicyObject has to be registered at the OPLPDC. This step initializes the set of required engine modules specified in the PolicyObject. This set is then appended to a list which also contains the PolicyObject identifiers accordingly. In this way, the appropriate set of engine modules can be found for an incoming access request, because an access request always comes together with a policy object.
- A PEP component can now perform an access decision by using the OPLPDC. As parameters, it has to pass a ContextManager, a AccessDecisionRequest and the PolicyObject to the OPLPDC.
- If an access decision is requested, the OPLPDC uses the PolicyObject identifier to load the appropriate engine modules. Now these engine modules can perform the access decision with their specific enforcement policy.
- For example, an engine module such as RBACCoreEngine gets a PolicyObjectTree from the OPLPDC during initialization containing the suitable policy. If an access decision is called, the engine module receives the ContextManager and the AccessDecision Request to obtain all information needed. The engine module can now perform a decision and return it.
- The OPLPDC collects the decisions made by the queried engine modules. The single decisions are combined with a deny overrides algorithm, which correspondes to the logical and in the Ap2.4. specification. A AccessDecisionResponse containing the result is returned to the PEP component afterwards.

6 Performance Tests

The performance analysis of the policy decisions components is based on the Japex micro-benchmarking framework [1]. The development and the design of the Japex test bed is carried out as part of Ap3.7.

7 Conclusions

Java Native Engine: Using the Java programming language directly is a straightforward and easy-to-understand way to implement an ORKA policy engine.

Our Java implementation is based on the same modularization as the specification. In doing so, we keep the promise of the extensibility. It should be noted, however, that in contrast to the specification level the modularization at implementation level is not based on multiple inheritance. It turned out, that multiple inheritance is not the appropriate tool at this level.

We expect the Java engine to perform better than the XACML and the Drools engine because no “additional” engine needs to be called in order to evaluate a policy (cf. Ap3.7 for the results).

XACML Engine: The transformation from OPL to XACML can be done efficiently, and the generated policies can be optimized by a good XACML implementation. It is expected that such an implementation can evaluate simple queries not much slower than the native engine. More complicated queries will need more time, since the XACML implementation will have to ask the ORKA system for every attribute that is not provided directly in the request. Additionally, the XACML implementation that is used by ORKA does not fall into the category of an optimizing XACML implementations, so the results of the performance tests should be viewed with caution. It is probable that a change of the XACML engine could result in great speedup, especially for large policies.

Drools Engine: The transformation from OPL/XML to Drools Objects (called Drools Facts) is straight forward as for each policy module a specific Drools Fact is defined which allows a direct mapping between modules and facts.

Further, the developed transformation framework allows an easy extension for additional modules if necessary in case the OPL definition is extended.

We assume that the Drools Engine itself reflects a scalable decision engine, especially with respect to the size of the policy. The reason for this assumption is that with increasing policy, only the facts increase (which is the content of a policy object) whereas the evaluation semantic (given as rules) remains the same. The performance test will also be conducted and described in AP 3.7.

References

[1] Japex micro-benchmark framework. https://japex.dev.java.net/ (2008-08-19).

[2] DROOLS 4.0, 2008. http://downloads.jboss.com/drools/docs/4.0.5.19064.GA/html/index.html.