FACPL Documentation

Release 2.0

Andrea Margheri

Jul 11, 2018

Contents:

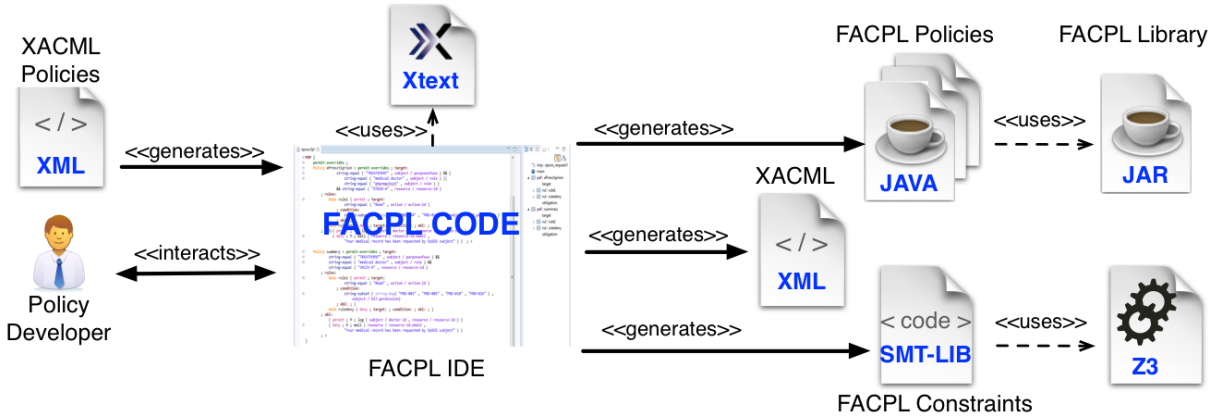
CHAPTER 1

FACPL at a glance

FACPL: Specifying, Analysing and Enforcing Access Control Policies

The FACPL language is a formal, easy-to-use language that permits specifying access control policies. FACPL is the basis of a feasible and effective approach for defining access control systems. Various applications have been proposed, varying from e-Health to autonomic computing domains.

FACPL is equipped with a powerful Integrated Development Environment (IDE) and a Java library, supporting access control system developers in the tasks of specifying, analysing and enforcing FACPL policies. Figure 1 shows the toolchain enabling the use of the language.

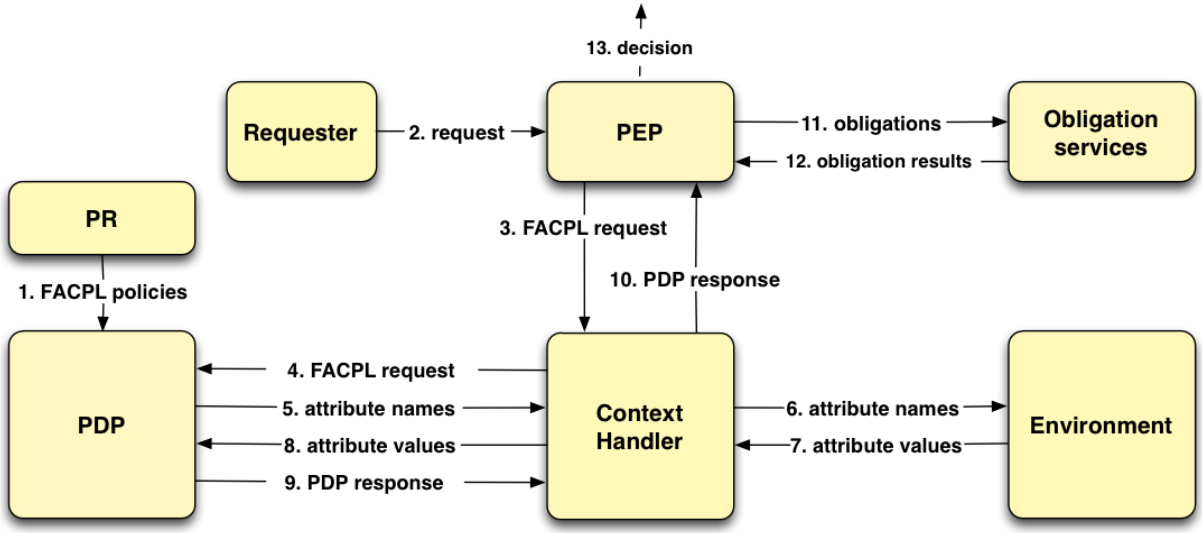


Developers can use the IDE, in the form of an Eclipse plugin, for specifying the desired policies in FACPL syntax, by taking advantage of the supporting features provided by the environment. The IDE automatically produces a set of Java classes enforcing the FACPL policies and of SMT-LIB files enabling the automatic analysis of policies. The Java FACPL library provides the compile- and run-time support for validating and enforcing the generated Java policies in real systems. The use of the SMT-LIB code and of the Z3 constraint solver offers effective analysis means. Furthermore, the toolchain offers a (partial) interoperability with the XACML standard, commonly used to deploy

real-world access control systems. See Section 9.1 of this FACPL paper for further details on XACML vs. FACPL interoperability.

1.1 FACPL Evaluation Process

Policies control system resources by means of a particular evaluation process, which relies on two main components: the *Policy Decision Point* (PDP) and the *Policy Enforcement Point* (PEP). The former calculates the authorization decision for an access request, and the latter enforces such decision in the system. Figure 2 shows the FACPL evaluation process.



Each controlled resource is paired with one or more FACPL policies, which define the access control rules expressing the credentials necessary to gain access to the resource. These policies are stored within the Policy Repository (PR) that makes them available to the PDP (step 1), which has the task of deciding whether to grant access to resources or not. The evaluation of a request is organized in the following steps.

- A request to access a resource is received by the PEP (step 2) and it is encoded as a FACPL request containing the credentials expressed as attribute elements (step 3). An attribute is a pair *(name, value)* representing a security-relevant information.
- The context handler sends the request to the PDP (step 4) and can add environmental attributes to the request, as e.g. the request receiving time, which may be needed for the evaluation process.
- The PDP computes the PDP response for the request by checking the attributes, that may belong either to the request or to the context (steps 5-8), against the controls contained in the policies. The PDP response contains an authorization decision and, possibly, some obligations.
- The PDP response is sent to the PEP, that, by appropriate obligation services, must discharge all possibly present obligations (steps 9-11).
- On the basis of the result of obligations discharge, the PEP computes the final decision (steps 12-13). This decision, that could differ from the PDP one, is the overall outcome of the evaluation process.

Notably, obligations are additional actions connected to the access control system and might correspond to, e.g., updating a log file, sending a message, generating an event or executing an action.

CHAPTER 2

Getting started

A set of FACPL examples are available in the GitHub repository together with the corresponding Java-translated policies, in the code examples repository. The binaries and source code of the Java library and its unit tests can be downloaded from the repository as well.

2.1 Eclipse installation

Note: The Eclipse plugin is provided by means of the Eclipse p2 repository (the current stable version is the 2.0.5). The repository is available in .zip format as part of the last release in GitHub here.

By using the well-known procedure "Install new software..." from the Eclipse's toolbar menu, the FACPL plugin can be easily installed. Note that it is required to accept the *Eclipse Public License* in order to complete the installation. The plugin installation requires:

- Eclipse for Java and DSL Developers version 4.* or higher version
- Xtext framework plugins
- Java 8

If the Xtext plugins are missing, they will be automatically added through the standard Eclipse update site.

Note: The plugin has been successfully tested by using the Eclipse DSL Release Neon

2.1.1 Using the tool

When the installation of the plugin has completed, we can create a FACPL project to start coding, analysing and evaluating FACPL policies. In the *Usage guide*, all the needed details.

2.2 Java Library

Basic examples of FACPL Java code is available in the Java code examples.

Note: FACPL is not intended to be used directly from Java, but via its high-level syntax (whose IDE is available in the Eclipse plugin). Therefore, the way a policy is constructed is more *friendly* for an automated code generator than a programmer.

2.2.1 Library Structure

The high-level type structure of FACPL policy is

```
| IEvaluablePolicy
| ├── FacplPolicy
| │   ├── PolicySet
| │   └── Rule
```

FacplPolicy abstracts *obligation* and *target* field of the PolicySet and Rule. The corresponding type structure of the two fields are

```
| IObligationElement
| ├── Obligation
```

```
| ExpressionBooleanTree
| ├── ExpressionFunction
```

where the tree structure organises the functions with boolean operators. Comparison and arithmetics functions are organised with a Factory pattern according to the input types.

Therefore, *PolicySet* and *Rule* provide the abstract structure and the evaluation methods of the FACPL policy elements. Specifically, *PolicySet* includes the combining algorithm (whose specification is given by *IEvaluableAlgorithm*) and the list of enclosed elements (either PolicySet or Rule), while *Rule* contains the decision (viz. *PERMIT* or *DENY*).

2.2.2 Creating a FACPL policy

PolicySet and *Rule* are abstract classes, hence to create a FACPL policy is needed to extended the corresponding class and use the 'setter' methods to add the internal elements.

We report here some Java code from the examples.

Let's start with a policy enclosing a single rule

```
public class PolicySet_pName extends PolicySet {
        public PolicySet_pName() {
                addId("pName");
                // Algorithm Combining
                addCombiningAlg(new it.unifi.facpl.lib.algorithm.
↪PermitOverridesGreedy());
                // PolElements
                addPolicyElement(new Rule_rule1());
        }

        private class Rule_rule1 extends Rule {
```

(continues on next page)

(continued from previous page)

```
                Rule_rule1() {
                        addId("rule1");
                        // Effect
                        addEffect(Effect.PERMIT);
                }
        }
}
```

A more complex target can be added by using a tree structure with the following code

```
addTarget(new ExpressionBooleanTree(ExprBooleanConnector.AND,
        new ExpressionBooleanTree(new ExpressionFunction(new it.unifi.facpl.lib.
↪function.comparison.Equal(),
                                "John", new AttributeName("subject", "id"))),
        new ExpressionBooleanTree(new ExpressionFunction(new it.unifi.facpl.lib.
↪function.comparison.In(),
                                new AttributeName("action", "id"), new Set("read",
↪"seek")))));
```

the corresponding target expression is `equal(subject/id,"John") && in(action/id,{"read", "seek"})`.

To add obligations to either the rule of the policy, the following code has to be added

```
addObligation(new Obligation("compress", Effect.PERMIT, ObligationType.O, null));
```

According to the chosen obligation actions (in this case *compress*), a list of arguments can be inserted in place of `null`. By default, the available obligation actions is

- *mailTo*: to send an email to a given address and text
- *log*: to create a log file with a given text
- *compress*: to zip a given text

Here an example of a *log* obligation

```
addObligation(new Obligation("log", Effect.DENY, ObligationType.M, "Subject: ",
                new AttributeName("subject", "id"), new AttributeName("subject", "name
↪")));
```

the use of *AttributeName* as obligation arguments allows to retrieve at the policy evaluation time the actual input for discharging the action.

Note: To add additional obligation action, just implement the interface *IPepAction* and provide the class with the corresponding name in the PEPAction class. Details below on its usage.

2.2.3 Evaluating a policy

The evaluation of FACPL Policy correspond to invoke the method `evalute` given an access request in input. The method is

public AuthorisationPDP evaluate(ContextRequest cxtRequest, Boolean extendedIndeterminate)

where `extendedIndeterminate` set to `true` means that the extended evaluation of the indeterminate values (see XACML semantics).

An access request is defined by a list of attributes, grouped by category, and a link to a context stub that can be used to dynamically access to external information. A simple request is

```
public class ContextRequest_Name {

        private static ContextRequest CxtReq;

        public static ContextRequest getContextReq() {
                if (CxtReq != null) {
                        return CxtReq;
                }
                // create map for each category
                HashMap<String, Object> req_action = new HashMap<String, Object>();
                req_action.put("id", "READ");

                Request req = new Request("Name");
                req.addAttribute("action", req_action);

                // context stub: default-one
                CxtReq = new ContextRequest(req, ContextStub_Default.getInstance());
                return CxtReq;
        }
}
```

which is formed by a single attribute named `id` and with category `action`; together represented as `action/id`.

The enforcement procedure is completed by the two key components PDP and PEP described in the *Introduction*. Their structure is defined in the library and can be instantiated as follows

```
this.pdp = new PDP(new it.unifi.facpl.lib.algorithm.PermitUnlessDenyGreedy(),␣
↪policies, false);
this.pep = new PEP(EnforcementAlgorithm.DENY_BIASED);
```

where the PDP gets the combining algorithm to use (in this case *PermitUnlessDenyGreedy* for the evaluation of the list of *policies*; the last boolean sets the use of extendedIndeterminate. The PEP just requires the enforcement algorithm to use for discharging the obligations.

To add additional obligations to the PEP we can use

```
this.pep.addPEPActions(PEPAction.getPepActions());
```

where the template of the class PEPAction is defined as

```
public class PEPAction{

    public static HashMap<String, IPepAction> getPepActions() {
                /*
                 * Set your own pep action e.g. HashMap<String,new ***** class Action␣
↪extending IPepAction***()
                 *
                 * pepAction = new HashMap<String,IPepAction>();
                 * pepAction.put("action", Action.class); return
                 * pepAction;
                 */
                return null;
        }
```

(continues on next page)

(continued from previous page)

```
}
```

All together, the Eclipse plugin generates a *MainFACPL.java* file that create a main method for the evaluation of selected requests. Here an example

```
public class MainFACPL{

        private PDP pdp;
        private PEP pep;

        public MainFACPL() {
                // defined list of policies included in the PDP
                LinkedList<IEvaluablePolicy> policies = new LinkedList
↪<IEvaluablePolicy>();
                policies.add(new PolicySet_PSet());
                this.pdp = new PDP(new it.unifi.facpl.lib.algorithm.
↪PermitUnlessDenyGreedy(), policies, false);

                this.pep = new PEP(EnforcementAlgorithm.DENY_BIASED);

                this.pep.addPEPActions(PEPAction.getPepActions());
        }

        /*
        *ENTRY POINT FOR EVALUATION
        */
        public static void main(String[] args){
                //Initialise Authorisation System
                MainFACPL system = new MainFACPL();

                //log
                StringBuffer result = new StringBuffer();
                //request
                LinkedList<ContextRequest> requests = new LinkedList<ContextRequest>
↪();
                requests.add(ContextRequest_Name.getContextReq());
                for (ContextRequest rcxt : requests) {
                        result.append("-----------------------------------------------
↪----\n");
                        AuthorisationPDP resPDP = system.pdp.doAuthorisation(rcxt);
                        result.append("Request: "+ resPDP.getId() + "\n\n");
                        result.append("PDP Decision=\n " + resPDP.toString()+"\n\n");
                        //enforce decision
                        AuthorisationPEP resPEP = system.pep.doEnforcement(resPDP);
                        result.append("PEP Decision=\n " + resPEP.toString()+"\n");
                        result.append("-----------------------------------------------
↪----\n");
                }
                System.out.println(result.toString());
        }


        public PDP getPdp() {
                return pdp;
        }
```

(continues on next page)

(continued from previous page)

```
        public PEP getPep() {
                return pep;
        }

}
```

2.3 FACPL Java Code Generator and Parsers

FACPL polices can be generated starting from FACPL code (aka the one used in the Eclipse plugin), instead of directly using the Java library.

The (parser and) code generators are available standalone by the Eclipse plugin in the latest release. This example project reports practical examples of the code generation, given a FACPL file, of Java, XACML and SMT_LIB code.

By way of example, given the following FACPL code

```
PolicySet patientConsent { permit-overrides
        target: equal ( "Alice" , resource / patient-id )
        policies:
        PolicySet ePre { permit-overrides - all
                target:equal("e-Prescription",resource/type)
                policies:
                Rule writeDoc ( permit target: equal ( subject / role , "doctor" )
                        && equal ( action / id , "write" )
                        && in ("e-Pre-Write" , subject / permission )
                        && in ( "e-Pre-Read" , subject / permission ) )
                Rule readDoc ( permit target: equal ( subject / role , "doctor" )
                        && equal ( action / id , "read" )
                        && in ( "e-Pre-Read", subject / permission ) )
                Rule readPha ( permit target: equal ( subject / role , "pharmacist" )
                        && equal ( action / id , "read" )
                        && in ("e-Pre-Read" , subject / permission ) )
                obl-p:
                [ M log ( system / time , resource / type , subject / id , action /␣
↪id ) ]
        }
        Rule denyRule ( deny )
        obl-d:
        [ M mailTo ( resource / patient-id.mail , "Data requested by unauthorized␣
↪subject" ) ]
}
```

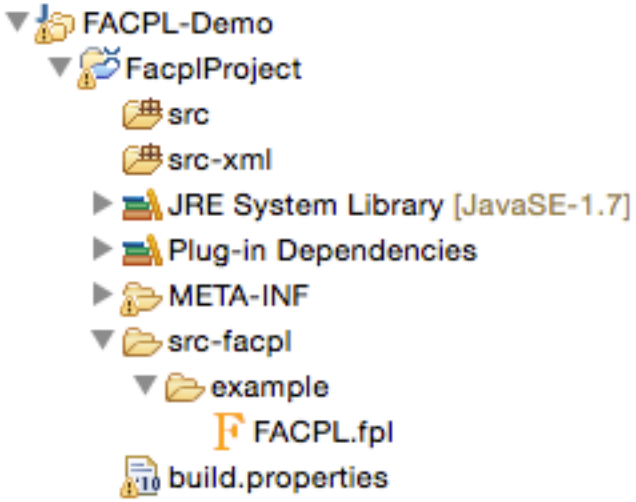
The code corresponding to the PolicySet *ePre* and *parientConsent* is generated.

CHAPTER 3

Usage guide

3.1 Setting Up a FACPL Project

A FACPL project can be created from the project menu *"File -> New Project . . . "*, where the customised wizard *FACPL Development Project* is available. After choosing a project name, the wizard creates a new Java Plugin-Development Project that contains all the required libraries for the coding and evaluation tasks; note that the project name cannot contain any blank space.



The generated FACPL project is like the one reported in Figure 3. FACPL files are generic text files having the "*.fpl*" extension and, for practical convenience, are placed in the *src-facpl* folder; a policy demo is added to the auto-generated project. The FACPL Java-translated policies and requests are automatically placed in the *src* folder. Instead, the *src-xml* folder contains the generated XML files and the *src-smtlib* folder contains the generated SMT-LIB files.

A new FACPL file can be created either as a new generic file with extension "*.fpl*" or by using the *FACPL File* wizard from the command *File -> New. . .* in the menu. The wizard permits specifying the container of the file (by selecting

it from the projects available in the workspace), the name of the file, and some basic code examples to add to the new file.

3.2 Policy Specification

A FACPL file is composed of three different parts (for which the new file wizard provides basic templates):

- *Policy declarations*: define the access control policies and the algorithms used for calculating and enforcing authorisation decisions.
- *Request declarations*: define the attributes values modeling an access attempt. The requests will be evaluated with respect to the available policies to obtain the corresponding authorisation decisions.
- *Main*: defines the Policy Authorisation System (PAS), i.e. the PEP and PDP, and some options for the generation of Java code and for request evaluation. More details on this part are presented in *Plugin Commands and Facets*.

An access control policy is hierarchically structured in terms of *rules* and *policy sets*, where a rule is a basic element for specifying access controls, while a policy set is a collection of other policies.

A **rule** specifies a name, the positive or negative decision of its successful evaluation (i.e., permit or deny), and a target expression for checking the applicability with respect to a request.

A **target** is a boolean expression defining the conditions deciding if the enclosed policy has to authorised an incoming request. The expressions are formed by basic relational and arithmetic operators. Such opertors define conditions on requests by means of *attribute name*. The available operators and some special attribute names (e.g. to get the current time) are provided by the auto-completion feature (e.g., for Mac/s users +Space) of the plugin. Attribute names are of the form *Identifier/Identifier*, where the first identifier stands for a category name and the second for an attribute name. For example, the name *action/action-id* represents the value of the attribute *action-id* within the category *action*. Notably, the plugin provides a type inference system checking that the expressions are correctly typed.

A **policy set** specifies a name, the combining algorithm to be used for combining the results of the contained policies, and a target expression for defining its applicability. The available combining algorithms are: *permit-overrides, deny-overrides, permit-unless-deny, deny-unless-permit, first-applicable, only-one-applicable, weak-consensus and strong-consensus*. The behaviour of each of them is presented in *Policy Evaluation*. Each algorithm is paired with a fulfilment strategy, i.e. *all* or *greedy*, leading its evaluation (see below). In addition, if different behaviours are requested, it is also possible to specialise the *custom-algorithm*. Furthermore, the command **include** permits to add, by means of name reference, a policy set to another one.

Each of the previous elements can also include a list of obligations. An **obligation** specifies an effect, i.e. permit or deny, for the applicability of the obligation, a type, i.e. M for Mandatory and O for Optional, and the identifier of an action with its argument. These arguments are generic expressions possibly containing attribute names, while the set of action identifiers understood by the PEP can be chosen, from time to time, according to the specific application.

The definition of the policy authorisation system (PAS), in addition to the access control policies defining the PDP, defines the top-level combining algorithm for the PDP (i.e., one among the algorithms already mentioned) and the enforcement algorithm for the PEP (i.e., one among *base*, *permit-biased* and *deny-biased*).

The following figure reports an example of policy declaration from an e-Health case study.

```
PolicySet patientConsent {permit-overrides
     target:equal("Alice",resource/patient-id) policies:
     PolicySet ePre { permit-overrides
          target:equal("e-Prescription",resource/type)
          policies:
          Rule readDocPre ( permit target: equal("doctor",subject/role) &&
               equal("read",action/id)
               && in("e-Pre-Read",subject/permission))

     obl-p:[M log(system/time,resource/type,subject/id,action/id)]
    }
    PolicySet eDis { permit-overrides
          target:equal("e-Dispensation",resource/type)
          policies:
          Rule readPhaDis ( permit target: equal("pharmacist",subject/role) &&
               equal("read",action/id)
               && in("e-Dis-Read",subject/permission))
         obl-p:[M log(system/time,resource/type,subject/id,action/id)]
    }
    Rule rule_deny (deny)

    obl-p:[O compress()]
    obl-d:[M mailTo(resource/patient-id.mail, "Data requested by unauthorized subject")]
}
```

The policy manages all the requests for the management of the *e-Prescription* service of the patient named 'Alice'. The rules checks the credentials exposed by the requester (i.e., the permission) and the requested actions.

We briefly comment part of the reported policy. The policy named "ePre" checks, by means of its target, if the requested service is "e-Prescription", then the internal rules check the exposed credentials according to the requested actions. By way of example, the rule named "writeDoc" authorises with permit (i.e., a positive authorisation) a subject whose role is doctor (i.e., by using attribute *subject/role*) and whose permissions contain both the permissions "e-Pre-Read" and "e-Pre-Write". Notably, the rules are evaluated in the same order as they appear within the policy. Thus, since the chosen combining algorithm is permit-overrides (see below), if the first rule evaluates correctly (i.e. it returns permit) then the second rule is not evaluated. Finally, the obligation *log* is used to record in the system the authorised access. The other rules are similarly defined, as well as the obligation *mailTo*.

```
Request:{ Request1
(subject/id,"Dr. House")
(resource/patient-id,"Alice")
(resource/type,"e-Prescription")
(subject/role,"doctor")
(subject/permission,"e-Pre-Access","e-Pre-Create")
(action/id,"write")
}
```

Figure 5 reports an example of FACPL request. Specifically, it represents the "doctor" with id "Dr. House" and credentials "e-Pre-Read" and "e-Pre-Write", willing to "write" an "e-Prescription" for the patient with id "Alice". This request is authorised to permit by the previous policy.

3.3 Policy Evaluation

The evaluation of a request with respect to a policy generates one among the following *authorization decisions*:

- **permit**: the request is granted;
- **deny**: the request is not granted;
- **not-applicable**: there is no policy that applies to the request;
- **indeterminate**: some errors occurred in the evaluation.

When the resulting authorisation decision is **permit** or **deny** some obligations can possibly be present.

The **evaluation of a policy** with respect to a request starts by checking its applicability to the request, which is done by evaluating the expression defining its target. Evaluating expressions amounts to apply operators and to resolve the attribute names occurring within, that is to determine the value corresponding to each such name. If this is not possible, i.e. an attribute with that name is missing in the request and cannot be retrieved through the context handler, the special value is returned. This value can be explicitly managed by the various operators. The evaluation of a policy has indeed the following cases:

- Let us suppose that the applicability holds, i.e. the expression evaluates to *true*. In case of rules, the rule effect is returned. In case of policy sets, the result is obtained by evaluating the contained policies and combining their evaluation results through the specified algorithm. In both cases, the evaluation ends with the fulfilment of the enclosed obligations.
- Let us suppose now that the applicability does not hold. If the expression evaluates to *false* or , the policy evaluation returns *not-applicable*, while if the expression returns an error or a non-boolean value, the policy evaluation returns *indeterminate*.

Clearly, a policy with target expression true (resp., false) applies to all (resp., no) requests. The evaluation process of rules and policy sets is summarised, respectively, in Tables 1 and 2.

Target	Obligation	**Rule Result**
true	*fulfilled*	*rule effect* + *FO*
true	*fulfilment error*	*indeterminate*
false or	•	*not-applicable*
error or non-boolean value	-	*indeterminate*

Table 1. Rule evaluation (where FO stands for 'fulfilled obligations')

Target	Combining Algorithm	Obligation	**Policy Set Result**
true	*permit* (resp., *deny*)	*fulfilled*	*permit* (resp., *deny*) + FO
true	*not-applicable*	•	*not-applicable*
true	*indeterminate*	•	*indeterminate*
true	*permit* (resp., *deny*)	*fulfilment error*	*indeterminate*
false or	•	•	*not-applicable*
error or non-boolean value	-	•	*indeterminate*

Table 2. Policy set evaluation (where FO stands for 'fulfilled obligations')

Concerning the **evaluation of expressions**, it takes into account the types of the operators arguments, and possibly returns the special value and *error*. In details, if the arguments are of the expected type, the operator is applied, else, i.e. at least one argument is *error*, *error* is returned; otherwise, i.e. at least one argument is and none is error, is returned. The expression operators *and* and *or* enforce a different treatment of these special values. Specifically, *and* returns *true* if both operands are *true*, *false* if at least one operand is *false*, if at least one operand is and none is *false* or *error*, and *error* otherwise (e.g. when an operand is not a boolean value). The operator *or* is the dual of and. Hence, and and or may mask and *error*. Instead, the unary operator *not* only swaps values *true* and *false* and leaves and *error* unchanged. The other expression operators have the expected semantics (e.g., operator *equal* checks if the arguments are equal) and enforce the management strategy for the special values and *error* possibly resulting from the evaluation of their arguments. Indeed, they establish that *error* takes precedence over and is returned every time the operator arguments have unexpected types; whereas is returned when at least an argument is and there is no *error*.

The evaluation of a policy includes the **fulfilment** of the enclosed **obligations** whose applicability effect coincides with the decision calculated for the policy. The fulfilment of an obligation consists in evaluating all the expression arguments of the enclosed action. If an error occurs, the policy decision is changed to indet. Otherwise, the fulfilled obligations are paired with the policy decision to form the PDP response.

The behaviour of the **combining algorithms** available in the plugin is as follows:

- **deny-overrides** (specular to **permit-overrides**): if the processing of a policy returns *deny*, then the result is *deny*. In other words, deny takes precedence, regardless of the result of processing any other policy. Instead, if at least a policy returns *permit* and all others return *not-applicable* or *permit*, then the result is *permit*. If all policies return *not-applicable*, then the result is *not-applicable*. In the remaining cases, the result is *indeterminate*.
- **deny-unless-permit** (specular to **permit-unless-deny**): this algorithm gives precedence to permit over deny, but never returns *not-applicable* or *indeterminate* because, if a request is not evaluated as permit, then it is evaluated as deny.
- **first-applicable**: in this case, the combined result is the same as the result of processing the first policy in the sequence of policies whose target is applicable to the request, if such result is either *permit*, *deny* or *indeterminate*. If all policies return *not-applicable*, then the result is *not-applicable*.
- **only-one-applicable**: this algorithm ensures that one and only one policy is applicable by virtue of its target. If no policy applies, the algorithm returns *not-applicable*, while if more than one policy is applicable, it returns *indeterminate*. When exactly one policy is applicable, the result of the algorithm is that of the applicable policy.
- **weak-consensus**: this algorithm returns *permit* (resp., *deny*) if some policies return *permit* (resp., *deny*) and no other policy returns *deny* (resp., *permit*); if both decisions are returned by different policies in the sequence, the algorithm returns indeterminate. If only *not-applicable* and *indeterminate* decisions are returned, *indeterminate* takes precedence. When all policies return *not-applicable* then the result is *not-applicable*.
- **strong-consensus**: this algorithm is the stronger version of the previous one, in the sense that to obtain *permit* (resp., *deny*) all policies have to return *permit* (resp., *deny*), otherwise *indeterminate* is returned. If all policies return *not-applicable* then the result is *not-applicable*.

Each algorithm is paired with a **fulfilment strategy**, i.e. one between *all* and *greedy*.

- The *all* strategy requires evaluation of all the occurring policies and returns the fulfilled obligations pertaining to all decisions.
- The *greedy* strategy prescribes that, as soon as a decision is obtained that cannot change due to evaluation of subsequent policies in the input sequence, the execution halts. Hence, the result will not consider the possibly remaining policies and only contains the obligations already fulfilled. Therefore, the fulfilment strategies mainly affect the amount of fulfilled obligations possibly returned.

The greedy strategy may significantly improve the evaluation performance of a sequence of several policies.

Finally, the **custom-algorithm** doesn't implement any behaviour; when the Java code is generated, it only returns a "template" for implementing a customised combining algorithm.

The authorisation decision resulting from the PDP evaluation is then enforced by means of the chosen enforcement algorithm according to the results of the execution of obligations. The behaviour of each enforcement algorithm is as follows:

- **base**: it allows (resp. forbids) access only if the decision is permit (resp. deny) and all obligations are successfully discharged, otherwise it enforces indeterminate;
- **deny-biased**: if the decision is permit and all obligations are successfully discharged, the access is granted, otherwise it is forbidden;
- **permit-biased**: if the decision is deny and all obligations are successfully discharged, the access is forbidden, otherwise it is granted.

Notably, errors possibly occurring while discharging optional obligations are ignored, so that they do not affect the enforcement process.

3.4 Policy Analysis

To analyse FACPL policies, it is used an approach based on constraints. The automatic verification of such constraints is obtained through an SMT solver, like, e.g., Z3. For additional details on how such constraints are generated see this FACPL paper The type of properties we can check on policies by means of such constraints are:

- **Authorisation Properties** These properties permit to statically reason on the result of the evaluation of a policy with respect to a specific request. Additionally, the properties MAY and MUST permit also to take into account the role of *additional attributes* that can be possibly introduced in the request at run-time and that might lead to unexpected authorisations. The properties are
  - *EVAL*: check if a policy evaluates a request to a certain decision.
  - *MAY*: check if a policy evaluates a request and ANY of its possible extensions (i.e., where additional attributes are present) to a certain decision.
  - *MUST*: check if a policy evaluates a request and ALL its possible extensions (i.e., where additional attributes are present) to a certain decision.
- **Structural Properties** These properties permit to statically reason on the whole set of authorisations enforced by one or more policies. The properties are
  - *COMPLETE*: a policy is complete if it applies to all requests, i.e. it does not return *not-applicable*
  - *DISJOINT*: two policies are disjoint if there is no request for which both policies evaluate to *permit* or *deny*
  - *COVER*: a policy *p* covers a policy *p'* if the for each request for which *p'* evaluates to *permit* or *deny*, the policy *p* evaluates such requests to the same decision.

3.5 Plugin Commands and Facets

The FACPL plugin offers many facets to support policy development, from the organisation of code to commands for generating Java and XML code.

Navigation and formatting. The multi-page editor highlights FACPL keywords and policies' structure defining various formatting layouts for policy elements (i.e., combining algorithms, keywords, effects, and literals), and an auto-indentation command for FACPL code. The latter command can be invoked by using the classical Eclipse shortcut *+Shift+F* (or *Ctrl+Shift+F* for Window's users). Furthermore, the structure of policies can be also navigated by means of the *Outline View* specifically designed for FACPL specifications.

Scope and Import. The scope of a file is the set of requests and policies defined inside the file. The scope is used to check the references of requests and policies in the *Eval Request* option and in the *include* command, respectively.

The plugin allows the developers to split the code in different modules and, by using import commands, to create cross-file scope for policies and requests. The import is defined as the command *import 'name_file.fpl'* and can access all the FACPL files in the current folder. Therefore, the scope of the file where the import is defined is extended with the scope of the imported files. Specifically, all requests and policies defined in the imported file are also visible in the current file.

Name checks. For policies and policy sets it is ensured the uniqueness of names. This check is performed among policy items together with policy set ones, because both of them can be used in an include command. Moreover, when an import command is present, the name check verifies uniqueness of local items with respect to the imported ones.

Generation parameters. The meaning of the attributes defined in the *Main Attributes* section of the FACPL code is as follows:

- *Combined Decision* (optional): if multiple requests have to be evaluated, we can require that only one combined decision will be returned.
- *Extended Indeterminate*: it activates an additional features for the management of *indeterminate*; we advice to put this option to false.
- *Java Package*: it specifies the Java package where the generated Java-translated policies and requests will be placed (if empty, it is assigned the default Java package).
- *Requests To Evaluate*: it defines the name of the requests to evaluate (each request name must be visible within the file scope).

When these options are properly selected, the generation of Java code defines, in the PEP Java class, the main method for running requests' evaluation.

Generation of Java Code. To generate the corresponding Java code of a FACPL specification, the IDE provides the command *Generate Java Code from FACPL* in the pop-up menu (right click in the editor or on the specific file in the package explorer view) and in the FACPL toolbar menu. The resulting Java classes will be included in the package defined in the main attributes. If there are one or more imported files, the generation command is recursively executed on those FACPL files.

Generation of XACML (XML) policies. From the FACPL code it is also possible to generate the corresponding XACML files written as XML code. The command *Generate XACML Code from FACPL* in the pop-up menu or in the FACPL toolbar menu generates the corresponding XML files into the *src-xml* folder.

Generation of SMT-LIB. From the FACPL code it is also possible to generate the corresponding SMT-LIB code. The command *Generate SMT-LIB Code from FACPL* in the pop-up menu or in the FACPL toolbar menu generates the corresponding SMT-LIB file into the *src-smtlib* folder. This file can then pass as input to an SMT solver like, e.g., Z3.

Policy Analysis. The menu commands *Create Authorisation Property...* and *Create Structural Property...* provide a guided interface to create the SMT-LIB file needed to check the satisfiability of the chosen authorisation and structural property, respectively.

3.6 FAQ

- **Which additional action are available for FACPL obligations?** The PEP implementation provides by default *log* and *mailTo* actions. Other actions can be easily defined by using the Java class *PEPAction* that results from the generation of Java code.
- **May I code with FACPL directly in Java?** Yes, the Java libraries can be found on the web-site and they can be easily added as additional reference libraries to a Java project.

- **How can I update the Eclipse plugin?** The Eclipse plugin can be automatically updated (if a new version will be available) by using the Eclipse command *Check for Updates*.

Note: For any problem or questions, add an issue to the GitHub repository or mail to margheri.andrea@gmail.com.
