Service Oriented Computing and Applications

Developing and Enforcing Policies for Access Control, Resource Usage, and Adaptation

--Manuscript Draft--

Manuscript Number:	
Full Title:	Developing and Enforcing Policies for Access Control, Resource Usage, and Adaptation
Article Type:	S.I . : WS-FM and 4PAD
Keywords:	Policy-based languages; semantic-driven development tools; cloud computing
Corresponding Author:	Andrea Margheri Università di Firenze Firenze, ITALY
Corresponding Author Secondary Information:	
Corresponding Author's Institution:	Università di Firenze
Corresponding Author's Secondary Institution:	
First Author:	Andrea Margheri
First Author Secondary Information:	
Order of Authors:	Andrea Margheri
	Massimiliano Masi
	Rosario Pugliese
	Francesco Tiezzi
Order of Authors Secondary Information:	
Abstract:	Policy-based computing systems are nowadays widely exploited to regulate different aspects of systems' behavior, such as access control, resource usage, and adaptation. Although, several languages and technologies have been proposed, as e.g. the standard XACML, developing real-world systems using such approaches is still a tricky task, being them complex and error-prone. To overcome such difficulties, we advocate the use of FACPL, a formal policy language inspired to but simpler than XACML. FACPL has an intuitive syntax, a mathematical semantics and easy-to-use software tools supporting policy development and enforcement. We illustrate potentialities and effectiveness of our approach through a case study from the cloud computing domain.

Noname manuscript No.
(will be inserted by the editor)

Developing and Enforcing Policies for Access Control, Resource Usage, and Adaptation

A. Margheri, M. Masi, R. Pugliese and F. Tiezzi

the date of receipt and acceptance should be inserted later

Abstract Policy-based computing systems are nowadays widely exploited to regulate different aspects of systems' behavior, such as access control, resource usage, and adaptation. Although, several languages and technologies have been proposed, as e.g. the standard XACML, developing real-world systems using such approaches is still a tricky task, being them complex and error-prone. To overcome such difficulties, we advocate the use of FACPL, a formal policy language inspired to but simpler than XACML. FACPL has an intuitive syntax, a mathematical semantics and easy-to-use software tools supporting policy development and enforcement. We illustrate potentialities and effectiveness of our approach through a case study from the cloud computing domain.

Keywords Policy-based languages, semantic-driven development tools, cloud computing

This work has been partially sponsored by the EU project ASCENS (257414) and by the Italian MIUR project CINA, PRIN 2010-2011.

A. Margheri, R. Pugliese
Università degli Studi di Firenze, Viale Morgagni, 65 - 50134 Firenze, Italy
E-mail: {andrea.margheri,rosario.pugliese}@unifi.it

A. Margheri
Università di Pisa, Largo Bruno Pontecorvo, 3 - 56127 Pisa, Italy E-mail: margheri@di.unipi.it

M. Masi
Tiani "Spirit" GmbH, Guglgasse, 6 - 1110 Vienna, Austria
E-mail: massimiliano.masi@tiani-spirit.com

F. Tiezzi
IMT Advanced Studies Lucca, Piazza S. Francesco, 19 - 55100, Lucca, Italy E-mail: francesco.tiezzi@imtlucca.it

1 Introduction

Today's ICT systems are becoming more and more complex, as they consist of several distributed components interacting with each other, as well as with other systems or humans in open-ended environments. Significant examples of such systems are cloud infrastructures, social networks, and smart grids. A key challenge they pose is that of programming and managing different aspects of their behaviour, ranging from access control to resource usage and, whenever needed, to strategies for re-configuring their architecture according to their own state and to the environmental conditions.

To face the above challenges, considerable attention has been given recently to the use of policies, i.e. sets of rules specifying users' credentials needed to access resources. The declarative nature of policies is in fact suitable to express system's behaviours, requirements, constraints, guidelines, strategies, etc. in a form that is accessible not only to developers and system architects, but also to other professionals (e.g., lawyers, medical doctors) and, possibly, to systems end-users (e.g., citizens, patients). We are in particular interested in languages for access control policies [22]. These languages allow security administrators to express policies regulating access to resources in terms of some *attributes*, i.e. sets of properties describing the entities that must be considered for the authorization decision. The policy-based approach has many advantages [22] with respect to other access control approaches, as e.g. the role-based one. Indeed, it overcomes scalability problems raised by large distributed systems, enables a more fine-grained control, and permits a dynamic, context-aware authorization process, since the decision may depend also on environment attributes, not only on subject/resource ones.

Many languages for defining access control policies have been proposed in the literature, among which the standard eXtensible Access Control Markup Language (XACML) [25] has emerged as a de-facto standard. However, developing real-world systems using policy-based approaches is still hard because designing, writing, testing and maintaining policies are difficult and error-prone tasks. On the one hand, policy languages, especially those defined at industry level, have an XML syntax that makes defining and managing policies awkward. For example, XACML has been conceived to write policies by directly using the XML syntax defined in the corresponding specification document. Moreover, they are equipped with intricate features, whose semantics may be not completely clear to developers, since it is often informally defined in natural language. On the other hand, despite the large number of proposals, only a few of them come with software tools for policy enforcement and, which is even more rare, dedicated developing environments. Finally, most of such languages are devised to only deal with specific aspects of systems, like e.g. user authorization. Hence, different languages are needed to address different aspects of the considered system.

To overcome the difficulties mentioned above, in this paper we propose the use of a policy language capable of dealing with different systems' aspects through a user-friendly, uniform, and comprehensive approach. Indeed, we have designed a policy language, called FACPL [16], that intentionally takes inspiration from XACML but is much simpler and usable. Although FACPL is not more expressive than XACML, it differs for the higher abstraction level at which policies can be written. Differently from XACML, FACPL has a compact and intuitive syntax and is endowed with a formal semantics based on solid mathematical foundations, which make it easy to learn and use. FACPL can express access control policies as well as policies dealing with other systems' aspects, e.g. resource usage and adaptation. Moreover, the development and the enforcement of FACPL policies is supported by practical software tools: an Eclipse-based development environment and a Java library supporting the policy evaluation process. The policy engineer can use the dedicated environment for writing the desired policies in FACPL syntax, by taking advantage of the supporting features provided by the development environment. Then, according to the rules defining the language's semantics, the development environment automatically produces a set of Java classes implementing the FACPL policies. The generated classes can be compiled using the FACPL Java library. The resulting (byte)code can be then executed for computing the authorization decision corresponding to a given request and can be also integrated as a module into the enclosing application.

The main contributions of this work are *(i)* a policy development methodology based on FACPL, *(ii)* the implementation of the software tools supporting it, and *(iii)* their application to a significant case study. Indeed, for illustrating potentialities of the FACPL language and effectiveness of the proposed development approach, we consider a case study from the cloud computing domain. Specifically, we use FACPL to define policies for access control, resource-usage and adaptation of a cloud IaaS provider, and we use FACPL's tools for implementing a policy-based cloud manager. The integration of this manager with a well-established hypervisor is supported by FACPL through obligations, i.e. additional actions specified in the policies that must be successfully executed at enforcing time.

Structure of the paper. The rest of the paper is organized as follows. In Section 2, we introduce the IaaS cloud computing case study used throughout the paper to illustrate our policy-based approach. In Section 3, we describe FACPL's syntax in a step-by-step fashion while commenting upon some policies in force in the case study. We also give a glimpse of the FACPL formal semantics. In Section 4, we discuss the main differences between FACPL and XACML. In Section 5, we illustrate FACPL's software tools and the implementation of the case study. We discuss more strictly related work in Section 6 and sketch some future work in Section 7.

This paper is an extended and revisited version of [17]. Here, we provide a wider overview of cloud systems and extend the implementation of the case study by integrating our FACPL-based cloud manager with XEN hypervisors. We also provide a short introduction to XACML and extend the discussion of its relationship and interoperability with FACPL. Finally, we briefly present the formal semantics of FACPL, whose complete definition is out-of-scope in this paper and given in [16].

2 Cloud computing systems

Cloud computing [20] has emerged in recent years as a new computing paradigm that aims at providing easy and low-cost access to externalized IT resources. The significant economic and administrative benefits brought by cloud computing make it more and more attracting for industries, academia, public institutions, and private customers.

Cloud computing fosters three well-known approaches to provide computational services:

Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS) and Service-as-a-Service (SaaS). Each of them is characterized by the offer provided to the clients, ranging from the low-level access provided by IaaS, where clients require the instantiation of general purpose virtual machines for running their processes, to the high-level access provided by SaaS, where clients directly interact with application services (e.g., storage or mail services) without being aware of details concerning the infrastructure and platform supporting the provision of such services.

These cloud services are offered in a *multi-tenancy* architecture that serves multiple clients by partitioning a virtualized infrastructure among them. From the providers' point of view, this kind of architecture permits to manage resource allocation more efficiently. From the clients' point of view, multi-tenancy and virtualized environments raise security concerns on confidentiality and integrity of data (see, e.g., [5, 28]). This calls for appropriate authentication and authorization mechanisms to secure the access to data and applications within cloud environments.

In order to address such security issues, the use of standardized specifications is advocated. For example, two wide-spread standard languages, i.e. SAML [24] and XACML, can be used for defining authentication credentials and access control policies, respectively. However, their practical usage is hindered by their verbose syntax and lack of formal semantics.

Moreover, the on-demand request of cloud services requires the use of specific *Service Level Agreements* (SLAs) defining the resources assigned to the clients, the client's guarantees and the provider's responsibilities. A SLA is a formal contract between client and provider that must be respected during the whole service duration. In case of violations, the provider has to fulfill specific contractual obligations to, e.g., partially refund the service cost.

The management of SLAs within a cloud platform concerns, on the one hand, the definition and negotiation of a SLA among the various stakeholders, and, on the other hand, the provisioning strategies to follow for enforcing the contract. The former aspect has attracted a large research effort providing various specification languages for SLAs (see, e.g., [15, 9]), whereas the latter aspect is tightly related to the various metrics one needs to achieve (e.g., power consumption, throughput, availability) and their monitoring (e.g., [7]).

To cope with security and resource provisioning issues, we advocate the use of the FACPL language to reconcile the definition of access control, resource allocation and SLAs enforcement. In particular, to ensure the guarantees defined in the SLAs in highly dynamic environments as the cloud ones, we rely on adaptation strategies expressed as policies and enacted through discharge of some obligations.

In the rest of this section, we introduce the cloud case study we use as a running example throughout the paper. We consider a small-size IaaS provider that offers clients a range of pre-configured *virtual machines* (VMs), providing different amounts of dedicated computing capacity in order to meet different computing needs. Each type of VM features a specific SLA that the provider commits to guarantee. The allocation of the right amount of resources needed to instantiate new VMs (while respecting committed SLAs) is a key aspect of the considered IaaS provider. As is common for cloud systems, virtualization is accomplished using an *hypervisor*, i.e., a software entity managing the execution of VMs.

For the sake of presentation, our IaaS provider relies only on two hypervisors running on top of two physical machines. The provider offers strongly defined types of VMs, like most of popular IaaS providers (consider, e.g., the instance types *M1 Small* and *M1 Medium* provided by Amazon EC2[1]). Two types of VMs, namely `TYPE_1` and `TYPE_2`, are in the provider's service portfolio. Each type of VM has an associated SLA describing the hardware resources needed to instantiate the VM (e.g., CPU performance, size of memory and storage) by means of an aggregated measure: `TYPE_1` requires the allocation of one *unit of resources*, while `TYPE_2` requires two resource units.

The two types of VMs have different guarantees when the system is highly loaded. Specifically, if the system does not have enough resources for allocating a new `TYPE_2` VM, an appropriate number of `TYPE_1` VMs already instantiated will be frozen and moved to a queue of suspended VMs. This queue is periodically checked with the aim of trying to reactivate suspended VMs. When a VM is frozen, according to the *Insurance* [32] SLA approach for resource provisioning in cloud systems, the VM's owner will receive a credit that can be used, e.g., for activating new VMs or for paying computational time.

A graphical representation of the data-flow in our implementation of the case study is shown in Figure 1. Clients interacts with the cloud system via a Web portal that, following a multi-tenancy architecture, sends VM instantiation requests to the *cloud manager* through SOAP messages. This means that the manager exposes its functionalities to users by means of a Web service. Then, the manager evaluates the received requests with respect to a set of policies defining the logic of the system. In particular, such policies specify the credentials

[1] `http://aws.amazon.com/ec2/instance-types/`

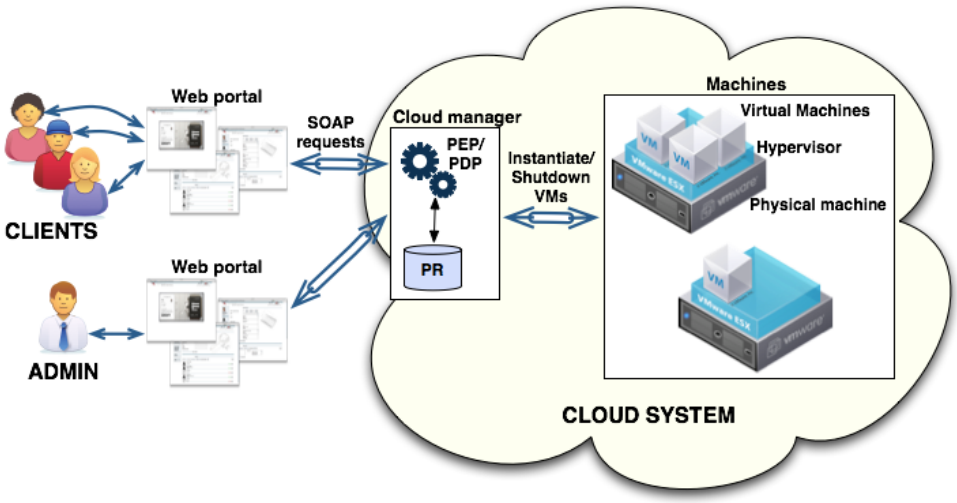

Fig. 1 IaaS provider case study

the clients have to provide in order to access the service (*access control policies*), the resource allocation strategy (*resource-usage policies*), and the actions to be performed to fulfill the requests by also taking into account the current system state, which include the system re-configuration actions in case of high load state (*adaptation policies*). It is worth noticing that all policies are written by using the same policy language (described, together with the policies mentioned above, in Section 3) and are enforced through the same software tools (presented in Section 5). By means of a similar workflow, clients can request the shutdown of VMs, which involves the release of the allocated resources.

The administrator of the cloud system can access a dedicated panel for managing the governing policies. Indeed, he can change at run-time the current policies with other ones, obtaining in this way a fully configurable and adaptable system. The core of the cloud manager is the *Policy Enforcement Point* (PEP), which evaluates client requests according to the available policies in the *Policy Repository* (PR) and the environmental information about the cloud system. The subcomponent *Policy Decision Point* (PDP) has the duty of calculating if a request can be granted or rejected, and determining the actions needed to enforce decisions, such as creation, freezing and shutdown of VMs. The enforcing is executed by the PEP by sending to hypervisors the commands corresponding to the obtained actions. Notably, policies are independent from the specific kind of hypervisors installed on the system, such as XEN[2] or Linux-KVM[3], i.e., the actions returned by the PDP are converted by the PEP into the appropriate commands accepted by the used hypervisors. Thus, in principle, the policy engine we have developed could be integrated with any IaaS system provided that the adequate action translation is also defined. To demonstrate the feasibility of such an integration, besides a demo relying on a mock-up of the cloud system, we have developed a prototype installation of a private cloud provider created upon XEN hypervisors and using FACPL technologies for VMs management. Both these implementations are illustrated in Section 5.

[2] XEN - `http://www.xen.org`
[3] Linux-KVM - `http://www.linux-kvm.org`

3 The FACPL language

FACPL is an easy-to-learn, tiny language, with a mathematically defined semantics, for writing access control policies and requests. In this paper, we introduce how to exploit FACPL for managing resource-usage and adaptation, in addition to more traditional access control.

Formal syntax and semantics of the language are illustrated in [16]. Here, we present the syntax accepted as input by the FACPL supporting tools and informally describe its semantics. Notably, the tool-based syntax differs from the formal one for the notation of some constructs, in order to be machine-readable, and for the addition of a few commands aiming at increasing code modularization.

3.1 Syntax and informal semantics

We will gently introduce FACPL's syntax and informal semantics in a step-by-step fashion while commenting upon some policies in force in the case study described in the previous section. In particular, to present the

language, we will focus on the policy for managing creation of `TYPE_2` VMs. We will show then the complete syntax of the language.

When a client requests a service, a FACPL *request* is generated; in addition to its identifier, this latter request contains the client credentials and the specification of the type of the requested service. It is worth noticing that, before a request is received by the PEP, authentication controls are performed on its content. Hence, in the request evaluation process, the credentials provided by the client can be safely considered reliable. Such controls can rely on standardized authentication technologies, as e.g. SAML; hence, they are out-of-scope in this work.

The information in the client request are organized as *attributes*, i.e., pairs name-value. Names are structured in the form `Id1/Id2`, where `Id1` stands for a category name (as, e.g., `subject`, `resource`, `action`, and `system`) and `Id2` for an attribute name (as, e.g., `profile-id` and `action-id`). A sample request for creating a `TYPE_2` VM follows.

```
Request:{ Create_Type2_VM
  (subject/profile-id, "P_2")
  (resource/vm-type, "TYPE_2")
  (action/action-id, "CREATE")
}
```

When a request arrives, the PEP calls the PDP which starts the evaluation procedure on the basis of the policies stored within PR. Notably, the PDP performs the evaluation as if it has to evaluate the request with respect to a single policy set consisting of the set of available policies/policy sets and of a *combining algorithm* specifying the decision to be returned in case multiple policies apply and return different decisions. The decision resulting from the PDP's evaluation is one among `permit`, `deny`, `not-applicable`, and `indeterminate`. The meaning of the first two decisions is obvious, while the third means that the PDP does not have any policy that applies to the request and the fourth means that the PDP is unable to evaluate the request (because some errors occurred during the evaluation process). Moreover, the first two decisions can be paired with a set of *obligations*, i.e., actions that must be successfully performed for enforcing the decision, such as creation, freezing and shutdown of VMs.

When the PDP completes the decision, the resulting authorization, which can include obligations, is sent to the PEP for the enforcement. To enforce the authorization decision, the PEP must understand and discharge all the obligations possibly included in the PDP decision. If some obligations cannot be discharged, the decision taken depends on the *enforcement algorithm* chosen for the PEP.

In the policy considered below, the PEP uses the enforcing algorithm `deny-biased`[4] and the PDP uses the combining algorithm `permit-overrides`[5], and includes a policy for creating VMs and another one for shutting them down.

```
{
 pep: deny-biased;
 pdp: permit-overrides
      include Create_Policies
      include Release_Policies
}
```

Notably, in the PDP, the policies are included through a cross name reference, which simplifies code organization. In this case, the policies `Create_Policies` and `Release_Policies` are specified into two separated files. More specifically, `Create_Policies` is a *policy set* that uses the combining algorithm `permit-overrides` and specifies two policies, `SLA_Type1` and `SLA_Type2`, and a *target* determining the requests to which the policy set applies.

```
PolicySet Create_Policies
{ permit-overrides
  target: equal("CREATE",action/action-id)
  policies:
    Policy SLA_Type1
    < deny-unless-permit
      target: ...
      rules: ...
      obl: ...
    >
    Policy SLA_Type2
    < deny-unless-permit
      target: ...
      rules: ...
      obl: ...
    >
}
```

The policy set processing starts from the boolean expression in the target. This expression consists of *matching functions*, i.e., functions comparing literal values and attribute values, composed through conjunctive (`&&`) and disjunctive (`||`) boolean operators. For example, the function `equal` in the target of `Create_Policies` checks the equality of the string `CREATE` with the value, retrieved from the request, of attribute `action/action-id`. Note that an attribute name may refer to a specific attribute of the request, e.g. subject or action identity, or of the environment, e.g. hypervisor load. In the latter case, the value of the attribute can be retrieved from the environment by the

[4] The algorithm `deny-biased` states: if the decision is `permit` and all obligations are successfully discharged, then the PEP grants access, otherwise it forbids access.

[5] The algorithm `permit-overrides` states: if any policy among the considered ones evaluates to `permit`, then the decision is `permit`; otherwise, if all policies are found to be `not-applicable`, then the decision is `not-applicable`; in the remaining cases, the decision is `deny` or `indeterminate` (see [16]).

context handler. From a practical point of view, the context handler is an application-dependent software entity interacting with the environment of the PDP/PEP. In our case study, the environment consists of the hypervisors forming the cloud system. As we will clarify in Section 5, the FACPL supporting tools automatically generate a stub Java class corresponding to the context handler.

The processing of the target expression determines if the policy set applies to the request and produces one of values `match`, `no-match`, and `indeterminate`. The first two values have an obvious meaning, the last one is returned in case of errors. If the target does not match the request, the value resulting from the policy set processing is `not-applicable`, while if the target evaluates to `indeterminate` then this is the value returned (together with some information on the error occurred). Instead, if the target matches, the enclosed policies are processed and the results are combined according to the combining algorithm specified in the policy set. The combining algorithm also determines how conflicts and special situations, e.g., all policies are not applicable, are dealt with.

The syntax, as well as the semantics, of a *policy* is similar to that of a policy set, but comprises a set of rules instead of policies, as the previous listing shows.

A *rule* specifies the intended consequence of a positive evaluation (the allowed values are `permit` and `deny`), a target, a *condition*, which is a boolean expression that may further refine the applicability of the rule, and a set of obligations. Notably, target, condition and obligations may be missing. For example, the rule `hyper_1_freeze`, which is part of policy `SLA_Type2`, is applied to freeze some `TYPE_1` VMs running on the first hypervisor when a request for creating a new `TYPE_2` VM arrives and all the hypervisors are fully loaded. The rule inherits the target of the enclosing policy and specifies the effect `permit`, a condition refining the applicability of the rule to a request, and a set of obligations.

```
Rule hyper_1_freeze
(permit
 condition:
  or(and(equal(0,system/hyper1.availableResources),
         less-than-or-equal(2,system/
                            hyper1.vm1-counter)),
     and(equal(1,system/hyper1.availableResources),
         less-than-or-equal(1,system/
                            hyper1.vm1-counter)))
 obl:
  [permit M freeze("HYPER_1",
                   subtract(2,system/hyper1.
                            availableResources),
                   "TYPE_1")]
  [permit M create("HYPER_1",system/vm-id,"TYPE_2")]
)
```

A condition is a boolean expression formed by functions that operate on values and attributes. It is more general than a target expression, which just consists of conjunctions and disjunctions of matching functions. In fact, to define condition expressions, FACPL provides the main logical, relational, and arithmetic operators, which operate on boolean, string, integer, double, and date values. Moreover, bags (i.e., unordered collections that may contain duplicated values) of strings are also supported. The condition in the listing above checks if there are enough `TYPE_1` VMs to freeze for increasing the available resources in the hypervisor and granting the request; recall that each `TYPE_1` VM requires 1 resource unit, while each `TYPE_2` VM requires 2 resource units.

It is worth noticing that the `system` attributes occurring in the condition above are not present in the client request. They rather represent environmental information about the cloud system, needed to evaluate a request, that are retrieved by the context handler.

Finally, 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 action, with the corresponding expression arguments, to be performed by the PEP. Notably, the set of action identifiers understood by the PEP can be chosen according to the specific application.

To fulfill an obligation, the PDP must successfully evaluate all expression arguments of the action. For example, the first obligation of the previous rule requires the hypervisor `HYPER_1` to `freeze` the appropriate number of `TYPE_1` VMs. This number is calculated by means of an arithmetic expression in term of the currently available resources, retrieved through the attribute `system/hyper1.availableResources`, and considering that the creation of a `TYPE_2` VM needs 2 resource units. In this way, the fulfilled actions executed at run-time by the PEP guarantee the SLA contract for the `TYPE_2` VMs.

The complete syntax of FACPL is reported in Table 1. As usual in EBNF-like grammars, the symbol ? stands for optional items, $*$ for (possibly empty) sequences, and $+$ for non-empty sequences. To ease code writing and reading, whenever a sub-term of a policy is missing, the corresponding keyword can be omitted. Thus, e.g., the rule (`deny target : condition : obl :`) can be simply written as (`deny`).

Besides the features already illustrated, we want to point out that the language is equipped with a set of predefined matching functions (e.g., `not-equal`, `greater-than`, ...) and combing algorithms (e.g., `deny-unless-permit`, where a `permit` takes precedence over the other decisions, and decisions `not-applicable` and `indeterminate` are never returned). In particular, FACPL's developers can define their own com-

Specification ::= { pep : *PEP* pdp : *PDP* }

PEP ::= base | deny-biased | permit-biased

PDP ::= *Alg* *Policy*$^+$

Alg ::= deny-overrides | permit-overrides | deny-unless-permit
| permit-unless-deny | first-applicable | only-one-applicable | custom-algorithm

Policy ::= Policy *ID* < *Alg* target : *Target*$^?$ rules : *Rule*$^+$ obl : *Obligation** >
| PolicySet *ID* { *Alg* target : *Target*$^?$ policies : *Policy*$^+$ obl : *Obligation** }
| include *ID*

Rule ::= Rule *ID* (*Effect* target : *Target*$^?$ condition : *BoolExpr*$^?$ obl : *Obligation**)

Effect ::= permit | deny

Target ::= *MatchId*(*Value*, *Name*) | *Target* && *Target* | *Target* || *Target*

MatchId ::= equal | not-equal | greater-than | less-than
| greater-than-or-equal | less-than-or-equal

Name ::= *Identifier*/*Identifier*

Obligation ::= [*Effect* *Type* *PepAction*(*Expression**)]

Type ::= M | O

PepAction ::= log | mailTo | ...

Table 1 Syntax of FACPL policies

bining strategies by exploiting the combining algorithm `custom-algorithm`. Similarly, they can easily add to the language other actions besides the predefined *PEPAction* for logging and sending email notifications.

3.2 Policies for the IaaS case study

The IaaS case study presented in Section 2 defines a simple SLA contract for the `TYPE_2` VMs, that could be implemented following various provisioning strategies. By means of FACPL policies we can integrate the access control of the system with provisioning strategies devoted to ensure the SLA. In particular, we have developed two different approaches for managing, instantiating and releasing requests. The first one concentrates the workload on hypervisor `HYPER_1`, while hypervisor `HYPER_2` is only used when the primary one is fully loaded. Thus, by keeping the secondary hypervisor in stand-by mode until its use becomes necessary, energy can be saved. The second approach, instead, balances the workload between the two hypervisors.

Some excerpts of the *energy saving* policies have been previously commented, other relevant ones are presented below (for the sake of completeness, the whole specification is reported in Listing 1 in the Appendix). This specification defines a PDP containing a policy set, for supervising instantiation requests (specifying action `CREATE`), and a policy, for supervising release requests (specifying action `RELEASE`). The policy set contains a policy for each type of VM that, in its own turn, achieves the prioritized choice between the two hypervisors by specifying the combining algorithm `deny-unless-permit` and by relying on the rules order. The policy managing the instantiation of `TYPE_1` VMs is as follows:

```
Policy SLA_Type1
< deny-unless-permit
  target:
    (equal("P_1", subject/profile-id)
      ||equal("P_2", subject/profile-id))
    && equal("TYPE_1", resource/vm-type)
  rules:
    Rule hyper_1
    ( permit
      target:
        less-than-or-equal(1,system/hyper1.
                                availableResources)
      obl:
        [permit M create("HYPER_1",system/vm-id,
                         "TYPE_1")]
    )
    Rule hyper_2 ( ... )
  obl:
    [deny O warning("Not enough available resources
                     for TYPE_1 VMs")]
>
```

The policy's target indicates that instantiation of `TYPE_1` VMs can be required by clients having `P_1` or `P_2` as profile. The policy's combining algorithm evaluates the enclosed rules according to the order they occur in the policy; then, if one of them evaluates to `permit`,

the evaluation terminates. Rule `hyper_1` evaluates to `permit` only if the hypervisor `HYPER_1` has at least one unit of available resources and, in this case, returns an obligation requiring the PEP to create a VM in this hypervisor. Rule `hyper_2`, governing VMs creation on `HYPER_2`, is similar. If no rule evaluates to `permit`, then the combining algorithm returns `deny` and, hence, the policy's (optional) obligation will be executed by the PEP to notify the cloud administrator that there are not enough resources in the system to instantiate a new `TYPE_1` VM. In this way, the administrator can decide to upgrade the system by adding new resources (e.g., a new physical machine).

The policies for the *load balancing* follows a similar approach as before, but we need to ensure the load balance among hypervisors. The complete code is reported in Table 2. For implementing this second strategy, a condition (highlighted with a blue color in the code) on the hypervisors' load is added to each instantiation rule. This condition permits applying a rule for a certain hypervisor only if its amount of available resources is greater than or equal to the amount of available resources of the other hypervisor. The policy for the releasing of VMs is instead unchanged.

3.3 A glimpse of the formal semantics

The processing of FACPL policies follows their tree-like hierarchical structure: the leaf nodes, i.e. rules, returns a 'starting' decision, `permit`, `deny`, `not-applicable` or `indeterminate`, with the corresponding obligations, while the intermediate nodes, i.e. policy and policy sets, combine the decisions and the obligations returned by the processing of their child nodes through the specified combining algorithm. The computation of an authorization decision terminates when the root is reached.

For example, the instantiation request shown in Section 3.1 is evaluated with respect to the load balancing policy to `permit`, in case of low hypervisors' load. Indeed, the `less-that-or-equal` matching function of the rule (enclosed in the `SLA_Type1` policy) for `HYPER_1` succeeds in checking the load status; then, the obligation for creating a new VM can be fulfilled, e.g., as follows

$$[\texttt{M create}(\text{“}\texttt{HYPER\_1}\text{”}, \text{“}\texttt{vma345b}\text{”}, \text{“}\texttt{TYPE\_1}\text{”})]$$

where the identifier of the new VM is retrieved at runtime by the context handler.

The formal semantics of FACPL is reported in [16] and is given in a denotational style, i.e. it is defined by a function $[\![\cdot]\!]_R$ that, given a FACPL specification and a set R of access requests, returns a four-elements *decision tuple* of the form

$$\begin{array}{l}\langle\ \texttt{permit} : \{< r_i, Obl_i >\}_{i\in I_p},\\ \quad\texttt{deny} : \{< r_j, Obl_j >\}_{j\in I_d},\\ \quad\texttt{not-applicable} : R_n,\\ \quad\texttt{indeterminate} : R_i\ \rangle\end{array}$$

where $\{r_i\}_{i\in I_p}$, $\{r_j\}_{j\in I_d}$, R_n and R_i form a partition of R according to the authorization results of the requests' evaluation computed. Requests in the `permit` and `deny` sets may have attached sequences Obl of obligations. A decision tuple represents the evaluation result computed by the PDP and passed as input to the PEP for its evaluation. The final result does not contain obligations, as they are discharged during the enforcement process by the PEP.

This formal semantics permits to precisely formalize each evaluation step of the authorization process and, hence, to avoid erroneous interpretations that a large specification can give rise. This semantics can be used by policy developers for inspecting the possible authorization decisions for the policies under design, and thus for preventing unexpected behaviour and security flaws. In fact, the FACPL semantics identifies the attributes constraints, characterizing the sets forming the requests partition, that lead to each of the four authorization decisions. For example, the policies for the load balancing strategies authorize a releasing request r, i.e. the evaluation returns a `permit` decision, if the following constrain holds

$$\begin{array}{l}(\texttt{action/ation-id}, \texttt{"RELEASE"}) \in r\\ \wedge\ (\,(\texttt{subject/profile-id}, \texttt{"P\_1"}) \in r\\ \qquad \vee (\texttt{subject/profile-id}, \texttt{"P\_2"}) \in r\,)\\ \wedge\ \exists\, Id\ :\ (\texttt{resource/vm-id}, Id) \in r\\ \qquad\qquad \wedge\ Id \in (Ids_1 \cup Ids_2)\end{array}$$

where attributes $(\texttt{system/hyper1.vm-ids}, Ids_1)$ and $(\texttt{system/hyper2.vm-ids}, Ids_2)$ are retrieved by the context handler at evaluation time. This definition of the `permit` set of the decision tuple results from the application of the formal semantics of the `permit-overrides` algorithm to the evaluation of the two rules enclosed in `Release_Policies`. Thus, given the following request

```
Request:{ Release_VM
  (resource/vm-id, "vma345b")
  (action/action-id, "RELEASE")
}
```

to establish if it is authorized or not we can simply check whether it satisfies the constraints above. Actually, the request cannot be evaluated to `permit`, because it does not contain the subject's profile identifier.

Besides providing a clear and rigorous meaning to policies, which paves the way for reasoning on them, the formal semantics has been also used to drive the

```
{
 pep: deny-biased
 pdp: permit-overrides
      PolicySet Create_Policies
      { permit-overrides
        target: equal("CREATE", action/action-id)
        policies:
          Policy SLA_Type1
          < deny-unless-permit
            target:
              ( equal("P_1", subject/profile-id) || equal("P_2",subject/profile-id) )
              && equal("TYPE_1", resource/vm-type)
            rules:
              Rule hyper_1
              ( permit
                target: less-than-or-equal(1, system/hyper1.availableResources)
                condition:
                  less-than-or-equal(system/hyper2.availableResources,system/hyper1.availableResources)
                obl: [permit M create("HYPER_1", system/vm-id, "TYPE_1")]
              )
              Rule hyper_2
              ( permit
                target: less-than-or-equal(1, system/hyper2.availableResources)
                condition:
                   less-than-or-equal(system/hyper1.availableResources, system/hyper2.availableResources)
                obl: [permit M create("HYPER_2", system/vm-id, "TYPE_1")]
              )
            obl: [deny 0 warning("Not enough available resources for TYPE_1 VMs")]
          >
          Policy SLA_Type2
          < deny-unless-permit
            target: equal("P_2", subject/profile-id) && equal("TYPE_2", resource/vm-type)
            rules:
              Rule hyper_1_create
              ( permit
                target: less-than-or-equal(2, system/hyper1.availableResources)
                condition:
                  less-than-or-equal(system/hyper2.availableResources, system/hyper1.availableResources)
                obl: [permit M create("HYPER_1", system/vm-id, "TYPE_2")]
              )
              Rule hyper_2_create
              ( permit
                target: less-than-or-equal(2, system/hyper2.availableResources)
                condition:
                  less-than-or-equal(system/hyper1.availableResources, system/hyper2.availableResources)
                obl: [permit M create("HYPER_2", system/vm-id, "TYPE_2")]
              )
              Rule hyper_1_freeze
              ( permit
                condition:
                  or(and(equal(0, system/hyper1.availableResources),
                         less-than-or-equal(2, system/hyper1.vm1-counter)),
                     and(equal(1, system/hyper1.availableResources),
                         less-than-or-equal(1, system/hyper1.vm1-counter)))
                obl:
                  [permit M freeze("HYPER_1", subtract(2, system/hyper1.availableResources), "TYPE_1")]
                  [permit M create("HYPER_1", system/vm-id, "TYPE_2")]
              )
              Rule hyper_2_freeze
              ( permit
                condition:
                  or(and(equal(0, system/hyper2.availableResources),
                         less-than-or-equal(2, system/hyper2.vm1-counter)),
                     and(equal(1, system/hyper2.availableResources),
                         less-than-or-equal(1, system/hyper2.vm1-counter)))
                obl:
                  [permit M freeze("HYPER_2", subtract(2, system/hyper2.availableResources), "TYPE_1")]
                  [permit M create("HYPER_2", system/vm-id, "TYPE_2")]
              )
              obl: [deny 0 warning("Not enough available resources for TYPE_2 VMs")]
          >
      }
      Policy Release_Policies
      < permit-overrides
        target:
          equal("RELEASE", action/action-id)
          && ( equal("P_1", subject/profile-id) || equal("P_2", subject/profile-id) )
        rules:
          Rule hyper_1_release
          ( permit
            condition: at-least-one-member-of(resource/vm-id, system/hyper1.vm-ids)
            obl: [permit M release("HYPER_1", resource/vm-id)]
          )
          Rule hyper_2_release
          ( permit
            condition: at-least-one-member-of(resource/vm-id, system/hyper2.vm-ids)
            obl: [permit M release("HYPER_2", resource/vm-id)]
          )
      >
}
```

Table 2 Load balancing policies

implementation of development and evaluation tools. Specifically, guided by the semantic rules, we have developed a conformance test-suite that verifies each evaluation unit of the FACPL implementation (e.g., matching functions, rule evaluation) according to the formal specification. The suite of unit tests is available from the FACPL web-site [10].

4 FACPL vs. XACML

In this section, we first provide a brief overview of XACML, the XML standard for defining access control policies in computing systems. Then, we compare it with FACPL, in order to show the benefits of our proposal.

4.1 The XACML standard

XACML is a wide-spread standard that is currently used as a basis for enforcing access control in many large scale projects (see, e.g., [29, 31]) and standards (see, e.g., [24, 23]).

Specifically, the XACML standard provides a language for expressing access control policies and access requests, and a framework to evaluate access requests with respect to policies and to enforce the authorization decision. The access to each resource is regulated by one or more policies. These are XML documents expressing the credentials that a requestor must have for accessing the resource. For the sake of readability, in the rest of this section XML elements are denoted in sans-serif font (e.g., element `<rule> ... </rule>` is written Rule), while elements' attributes are denoted in italics.

The basic elements of the XACML standard are the Rules. As in FACPL, a Rule specifies the logic for the access control decision by means of an *effect*, a Target, a Condition, and some ObligationExpressions and AdviceExpressions. An obligation differs from an advice because its successful discharge is mandatory for enforcing the authorization decision.

A Target consists of a (conjunctive) sequence of AnyOf elements. Each AnyOf element contains a (disjunctive) sequence of AllOf elements, each of which contains a (conjunctive) sequence of Match elements. A Match element specifies an attribute identifier, a typed value, and a matching function. Such information is used to check whether the parent element of the Target (e.g., a Rule) is applicable to a given request. Specifically, the matching function retrieves a set of values from the designed attribute in the request and matches them with the value specified in the Match element, according to the function's semantics. If, for all AnyOf elements, at least the evaluation of an AllOf element is positive (i.e. all matchings of its Match elements succeed), the parent element to the Target is applicable to the request.

A set of Rules can be combined into a Policy that, besides its own Target and obligations/advices, specifies a *combining algorithm* that, from the set of rules' decisions, computes what is the policy decision corresponding to a given request. In case of particular combining strategies, an additional CombinerParameter can be defined. Finally, a set of policies can be combined together into a PolicySet that, again, specifies a combining algorithm, a target and some obligations/advices. A PolicySet can be also nested within another one, thus creating an hierarchical structure.

The XACML standard also introduces additional features to, e.g., define the issuer of a policy or pointers to expressions, by means of PolicyIssuer and Variable elements, respectively. Anyway, they do not increase the expressiveness of the language.

The evaluation process of XACML requests, as well as that of FACPL, is based on the standard PCIM framework [21] and, hence, is performed by two different actors: PDP and PEP. Similraly, the resulting authorization decision is one among *permit*, *deny*, *not-applicable* and *indeterminate*.

4.2 Comparing FACPL with XACML

The FACPL policy language takes advantage from its high-level syntax for significantly simplifying the tasks of designing, writing, reading and maintaining policies with respect to XACML. To show this, in this section we compare XACML and FACPL by focussing on the main pros and cons.

The XML format of XACML policies brings the benefit of enabling cross-platform interoperability, but can make the development of policies difficult and error-prone and is not adequate for formally defining the semantics of the language and reasoning on it. The FACPL language, instead, relies on an EBNF syntax and is equipped with a rigorous semantics given in a denotational style, which permits to formalize the authorization process of access requests. Interoperability of FACPL policies is anyway fully ensured both in Java-based applications, through the FACPL enforcement library written in Java, and in non-Java ones, through a specific feature of the FACPL development tools providing the possibility of exporting FACPL polices as XACML ones. We refer to Section 5 for a description of these supporting tools.

Policy	Number of lines		Saved	Number of characters		Saved
	XACML	FACPL	lines	XACML	FACPL	characters
SLA_Type1	162	22	86,42%	5.607	663	88,17%
SLA_Type2	349	36	89,68%	12.715	1.364	89,27%
Release_Policies	113	15	86,72%	4.193	438	89,55%
Overall policies	648	101	84,41%	23.436	3.030	87,07%

Table 3 FACPL vs. XACML on the cloud case study (load balancing policies)

Another significant advantage of FACPL with respect to XACML is that the former provides a more compact notation for writing policies. Indeed, the markup style of XACML is more verbose, even for expressing elementary checks on request values. In fact, to conform to the XACML XML Schema, it is required to explicitly specify the whole XML structure enclosing the check. Moreover, each identifier should be prefixed by a (quite long) XACML namespace. Besides adversely affecting policies' length, the markup style significantly undermines their readability and, hence, their specification and maintenance. For example, the first rule condition for balancing machines load in the policy reported in Table 2 corresponds to the following XML code.

```
<Condition>
  <Apply FunctionId="urn:oasis:names:tc:xacml:1.0
      :function:integer-less-than-or-equal">
    <AttributeDesignator DataType="http://www.w3.org
        /2001/XMLSchema#anyURI"
    MustBePresent="false"
    Category="urn:oasis:names:tc:xacml:3.0:attribute-
        category:system"
    AttributeId="urn:oasis:names:tc:xacml:3.0
        :system:hyper2.availableResources" />
    <AttributeDesignator DataType="http://www.w3.org
        /2001/XMLSchema#anyURI"
    MustBePresent="false"
    Category="urn:oasis:names:tc:xacml:3.0:attribute-
        category:system"
    AttributeId="urn:oasis:names:tc:xacml:3.0
        :system:hyper1.availableResources" />
  </Apply>
</Condition>
```

It is 12 lines length in XACML, while it takes only 1 line in FACPL.

Notably, the improvement of the language usability is due not only to the adoption of a non-XML format, but also to the design choices. For example, a FACPL target is defined by using only logical conjunction and disjunction of matching functions, thus avoiding the plentiful structure AnyOf-AllOf-Match of XACML. Furthermore, commonly used XML elements of XACML, as e.g. `AttributeDesignator` for identifying attributes, correspond to more lightweight linguistic constructs. For instance, the second `AttributeDesignator` element in the code above is rendered in FACPL simply as `system/hyper1.availableResources`.

These differences become more evident when XACML elements contain expressions. E.g., the condition for freezing a `TYPE_1` machine takes 4 lines in FACPL (lines 51-54 in Table 2) and 42 lines in XACML. We report in Table 3 a comparison, in terms of code length, between the FACPL policies of the cloud case study and the XACML corresponding ones (both groups of policies can be downloaded from `http://rap.dsi.unifi.it/facpl/FCloud_Policies/`).

The data shows that the use of FACPL can bring a relevant advantage to policy developers as well as to all users that anyhow need to understand the meaning of policies (e.g., application developers, stakeholders, lawyers, etc.).

The correspondences between XACML elements and FACPL terms are summarized in Table 4. In most of the cases the correspondence is straightforward; thus, we comment only on the salient points. Besides `AttributeDesignators`, XACML also provides `AttributeSelectors` to retrieve, by means of XPath [6] expressions, attributes values. In FACPL, simple XPath expressions, without additional operators for, e.g., arithmetic or node comparisons, can be represented by structured names. For example, the following XACML selector

```
<AttributeSelector
 MustBePresent="false"
 Category="urn:oasis:names:tc:xacml:3.0:attribute-
     category:resource"
 Path="/hyper1/availableResources/text()"
 DataType="http://www.w3.org/2001/XMLSchema#string"/>
```

looking for the amount of the available resources in the first hypervisor, is rendered as `resource/hyper1.availableResources`. Anyway, supporting all XPath functions, and in general the XML data-types, is out of scope for the FACPL language.

It is worth noticing that FACPL rules out all those elements that do not significantly affect the expressiveness of the language and the evaluation process. For example, in FACPL we cannot use variable declarations or specify the policy's issuer. Anyway, to enhance code modularity, we can use the `include` operator supporting cross-name references to policies and policy sets.

To sum up, FACPL can express all the main elements of XACML, avoiding redundant and verbose XML notations and superfluous characteristics. Therefore, the formal semantics of FACPL permits to indirectly formalize the XACML standard.

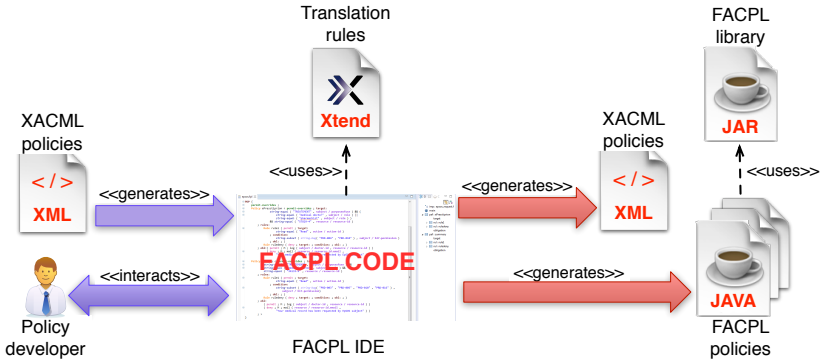

Fig. 2 FACPL toolchain

XACML	FACPL
PolicySet	`PolicySet { ... }`
Policy	`Policy < ... >`
PolicySetIdReferences PolicyIdReferences	`include` *ID*
Rule	`Rule ( ... )`
Request	`Request { ... }`
Target	`... && ... && ...`
AnyOf	`... || ... || ...`
AllOf	`... && ... && ...`
Match	*MatchId*(*Value*,*Name*)
Attribute AttributeDesignator AttributeSelector	*Identifier*/*Identifier*
ObligationExpression	`[` *Effect* `M ... ]`
AdviceExpression	`[` *Effect* `O ... ]`
Apply Condition	*Expression*
CombinerParameters	`custom-algorithm`
VariableDefinition VariableReference Function Description PolicyIssuer	Not-Available

Table 4 Correspondences between XACML and FACPL

5 Supporting tools

In Section 3, we have seen how the FACPL language can be used to define policies that are intuitive and easy to read, write and understand. In order for FACPL to be actually used in real systems, we provide policy developers with a software tool supporting policy development and a software architecture for policy enforcement. In this section, first we present the main features of these supporting tools, then we show them at work on the IaaS cloud case study.

The FACPL's supporting tools are developed by using Java-based technologies. Their source and binary files[6], as well as their documentation, can be found at the FACPL's website [10].

5.1 FACPL development environment and enforcement library

In Figure 2 we show the toolchain supporting the use of FACPL. The FACPL Integrated Development Environment (IDE) allows the policy developer to specify the system policies in FACPL. In addition to policies, the IDE permits also specifying user requests in order to test and validate the policies. The specification task is facilitated both by the high level of abstraction of FACPL and by the graphical interface provided by our IDE. Furthermore, the developer can automatically create FACPL code starting from XACML policies. Obviously, the tool just accepts XACML inputs that only contain the supported elements shown in Table 4. This feature is available both in the IDE and in the "Try FACPL in your Browser" application accessible from the FACPL's website. The latter is a web application allowing newcomer users to experiment with FACPL without installing any software.

By exploiting some translation rules, written using the Xtend language[7], which provides facilities for defining code generators, the IDE generates the corresponding low-level policies both in Java and in XML. The latter format obeys the XACML 3.0 syntax and can be used to connect our toolchain to external XACML tools (as, e.g., the test cases generator X-CREATE [2]). The former format relies on a Java library specifically designed for compile- and run-time supporting FACPL code. Once these Java classes are compiled, they can be used by the enclosing main application (i.e., the cloud manager in our case study) for evaluating client

[6] All FACPL software tools are freely available and open-source.

[7] Xtend - `http://www.eclipse.org/xtend/`

requests, simply by means of standard method invocation. Whenever new policies are introduced, new Java classes will be generated and compiled. The (run-time) integration of policy classes with the enclosing application is described in Section 5.2.

The FACPL IDE. The IDE is developed as an Eclipse plug-in by using the Xtext framework[8]. Xtext provides development instruments to design and implement domain-specific languages given their grammar. The plug-in is available on-line and can be installed by resorting to the standard procedure for installing new software into Eclipse.

The IDE provides a multi-page editor where the code writing activity is supported by syntactical controls, auto-completion and code suggestion. Besides these features, the IDE implements static checks for expression typing and for achieving uniqueness of identifiers. Moreover, for facilitating code organization, it is possible to split the code into multiple files.

The Java and XML code corresponding to a FACPL policy can be generated by means of specific commands available in the contextual menu. While the translations from and to XACML are straightforwardly derived from Table 4, the one from FACPL to Java is closely driven by the formal semantics of FACPL presented in [16]. In particular, to enable the automatic translation from FACPL to XACML, the boolean expression forming a FACPL target must be reorganized, by exploiting the distributive properties of the two boolean operators `&&` and `||`, so that to create a three-level expression of the form *conjunction-disjunction-conjunction* which directly corresponds to the shape Target-AnyOf-AllOf of XACML targets.

The policy editor is organized as shown by the screenshot in Figure 3: the view on the left shows the project structure which permits accessing to FACPL files, the multi-page editor in the center highlights the language keywords with different colors, and the view on the right shows the navigational outline. Toolbar and pop-up menus provide commands for generating Java and XACML code. Notably, to compile the generated Java code, the FACPL library is needed. The right dependencies between the generated code and the FACPL library, together with other supporting libraries (e.g., SLF4J[9] providing logging functionalities), are automatically initialized through the creation of a new Eclipse FACPL project.

The FACPL library. The FACPL code is executable through a Java library that implements all the semantic tasks of the evaluation process informally described in Section 3. The library is designed by exploiting Java *reflection* and best-practice software engineering techniques in order to achieve a flexible and extendible framework. In fact, the framework can be easily extended to incorporate custom matching functions and combining algorithms defined by the user to deal with, e.g., new value types or specific decisions combination.

To each language element corresponds an abstract class in the FACPL library, which provides a method for evaluating requests with respect to such element. Therefore, a FACPL policy is rendered as a Java class that extends the corresponding abstract class. The policy elements, i.e. combining algorithm, target, rules and obligations, are then added by the class constructor using specific methods, e.g. `addTarget`. Policy sets are translated similarly. For example, an excerpt of the Java code corresponding to the policy `SLA_Type1` is reported below; it is followed by some comments on the code.

```
public class Policy_SLA_Type1 extends Policy {
    public Policy_SLA_Type1() {
        addId("SLA_Type1");
        addCombiningAlg(it.unifi.facpl.lib.algorithm.
            DenyUnlessPermit.class);
        addTarget(new TargetTree(Connector.AND,
           new TargetTree(...), new TargetTree(...)))
                ;
        addRule(new hyper_1());
        addRule(new hyper_2());
        addObligation(new ObligationExpression("
            warning",Effect.DENY,
            TypeObl.O,"Not enough available resources
                 for TYPE_1 VMs"));
    }
    private class hyper_1 extends Rule {
        hyper_1 (){
          addId("hyper_1");
          addEffect(Effect.PERMIT);
          addTarget( ... );
          addConditionExpression(null);
          addObligation(new ObligationExpression ("
              create",Effect.PERMIT,
            TypeObl.M,"HYPER_1",new StructName("
                system","vm-id"), "TYPE_1"));
        }
    }
    private class hyper_2 extends Rule { ... }
}
```

Policy evaluation is coordinated by the class implementing the combining algorithm (i.e., `DenyUnlessPermit.class`). The expression corresponding to the policy target is structured as nested expressions organized according to the structure of the original FACPL target (possibly defined by brackets). Since rules are only used inside their enclosing policy, for each of them the policy class contains an inner class. In the code above, these are the classes `hyper_1` and `hyper_2`.

A FACPL request generates a class containing the attributes and a reference to the context handler. The translation from FACPL to Java indeed generates a stub for the context handler, which can be implemented

[8] Xtext - `http://www.eclipse.org/Xtext/`
[9] SLF4J - `http://www.slf4j.org`

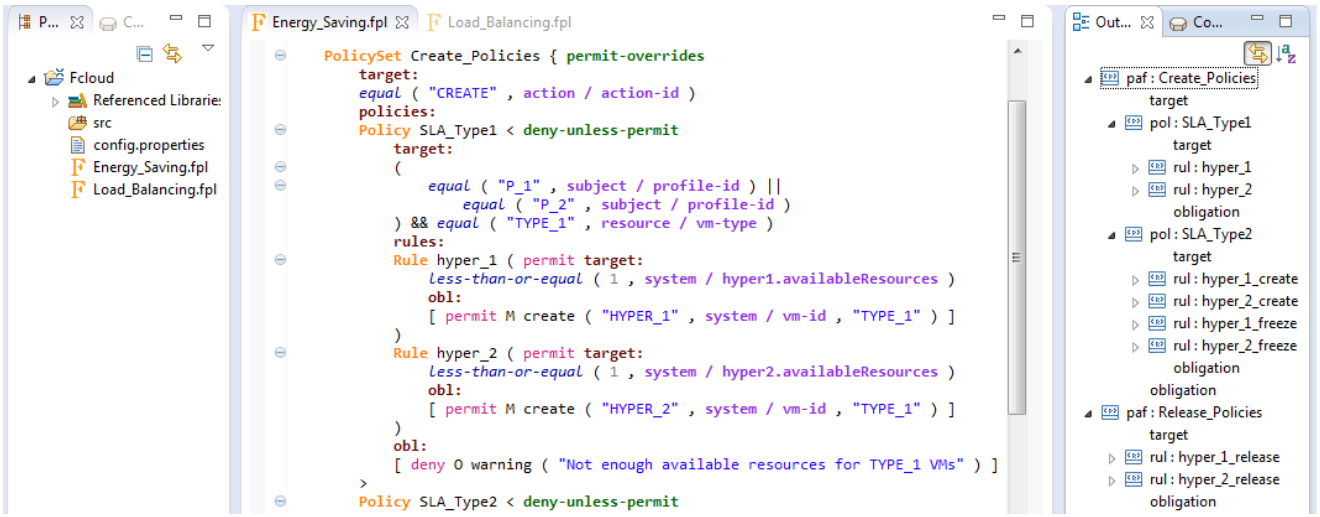

Fig. 3 The FACPL IDE

to enable the Java classes implementing the PDP to retrieve external information needed for evaluating requests. This is exploited in our cloud system implementation for accessing, e.g., the value of the attribute `system/hyper1.availableResources`.

Once all policies and requests have been translated into Java classes, and possibly the context handler stub has been implemented, the evaluation process can be started by invoking an entry method of the class corresponding to the PEP.

5.2 IaaS case study implementation

The IaaS case study described in Section 2 has been first implemented as a mock-up application[10] and then as a more realistic application managing XEN hypervisors[11]. Both applications provide a front-end for the administrator, from where he can manage the policies governing the hypervisors, and a front-end for the clients, from where they can submit requests for the creation or the shutdown of VMs. The two applications expose the same behavior to users and only differ in their back-end, as the cloud manager in the mock-up application interacts with a simple emulator of the hypervisors, while the manager in the XEN-based one interacts with real hypervisors running on server machines. Notably, the mock-up application allows different users to independently experiment with the cloud system by creating a dedicated session for each of them. Instead, users of the XEN-based application interacts with the same servers and, as it happens in real cloud platforms, they share the same servers loading.

[10] Available at `http://150.217.32.61:8080/FCloud-WebApp/`
[11] Available at `http://150.217.32.61:8080/FCloud-XEN/`

The two applications have been implemented by using the Java code automatically generated from the same FACPL policies. This gives a practical evidence of the interoperability and composability of the Java code generated from FACPL code. Indeed, such classes are decoupled from their working environment, hence no changes are needed to embed them in a real-world application rather than a mock-up one, apart from the implementation of the context handler.

For both applications, the server-side implementation is a Tomcat server that, by integrating FACPL and Xtext libraries, is able to accept FACPL polices, parse and compile these policies, and finally enforce the corresponding decisions for adapting hypervisors' state to client requests.

The administrator panel is shown in Figure 4. The policies chosen by the administrator can be either those previously outlined or other ones written by using the on-line editor. For analyzing the current state of the cloud system, a graphical representation of the load of each hypervisor is provided.

When the administrator submits a new FACPL policy, the cloud provider automatically translates it into Java classes by applying the Xtend parsing rules. Then, if all classes are successfully created, the provider compiles the classes by relying on the FACPL library. If no error occurs during any stage of this workflow, these policies became the new policies in force in the platform; otherwise, no policy update is made and the system goes on with the current, unchanged policies.

To submit a request, a client must provide, by choosing them on the front-end panel, the information needed for the wanted request: the profile identifier, i.e. `P_1` or `P_2`, and the VM type, for creating a VM, and the profile identifier and the VM's id, for shutting down a

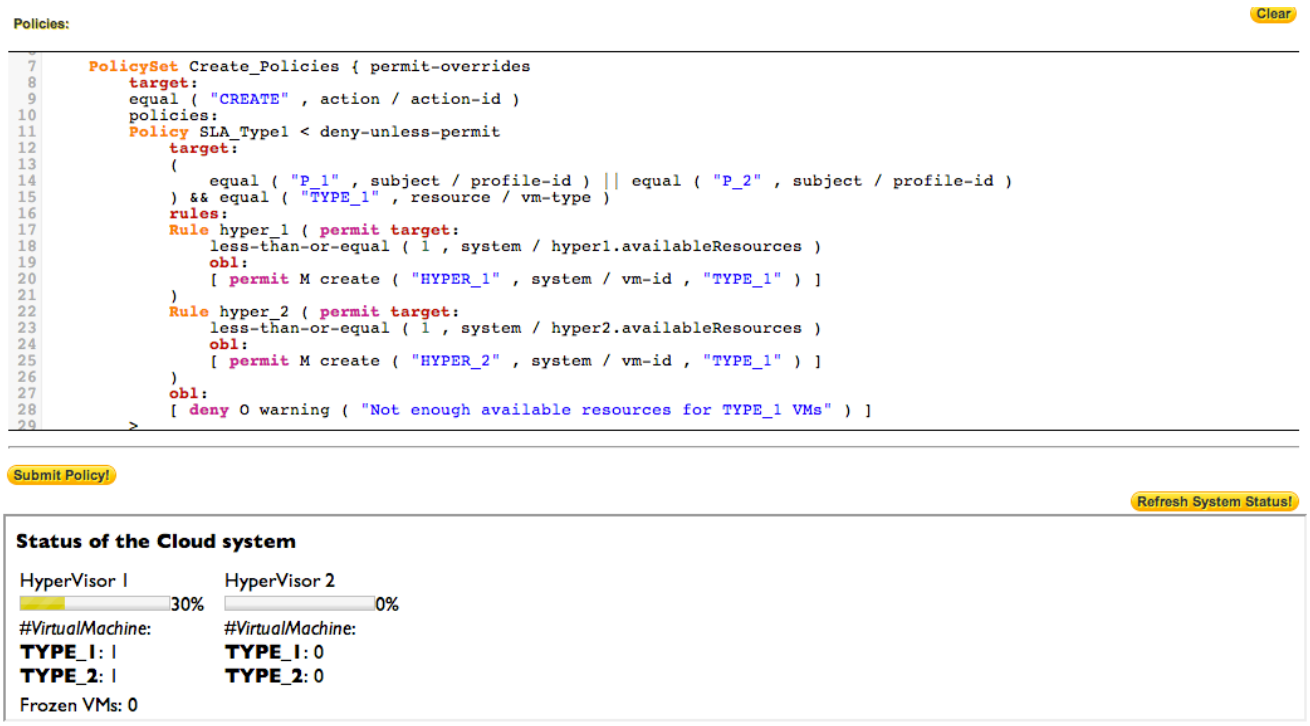

Fig. 4 The Administrator Panel

VM. While profile `P_1` only permits to create `TYPE_1` VMs, `P_2` permits to create both types of VMs. For the sake of simplicity, profile identifier and VM type are the only authentication information we use to determine if client requests should be granted or denied. Of course, in a real application, access control should rely on more trustful authentication information, which could be provided, e.g., through mechanisms such as SAML [24].

When the system load is high and it is needed to freeze some VMs, the owners of such VMs get some credits as reward. On the server side, the frozen VMs are added to a queue, which is periodically checked with the aim of trying to reactivate suspended VMs, according to the incoming order in the queue. This event is reported to the client through the front-end panel, where the possibly received credits and the status of the client's VMs are shown.

The deployment of the application interacting with XEN servers requires some additional tasks with respect to the mock-up one: *(i)* the configuration of the XEN servers; *(ii)* the creation of two different types of VMs; *(iii)* the invocation of remote methods for the management of VMs. All these tasks have been accomplished in a standard way, by applying the instructions reported on the official documentation of XEN.

Once the hypervisors configuration has been set up, we have assembled the cloud system by integrating the front-end web application (running on Tomcat) with the FACPL-based cloud manager, generated by means of the tools described in Section 5.1, and the two XEN hypervisors. The key role to achieve this integration is played by the context handler, whose implementation allows the PDP to appropriately access the status of the XEN servers during the evaluation. Notably, the cloud manager can be easily adapted to work with different hypervisor configurations by simply implementing different context handlers. Then, we have defined a PEP implementation enforcing the creation, shutdown and freezing of VMs on servers. PEP's code, invoked after the completion of each request evaluation, executes the remote invocations on the hypervisors returned as obligations by the evaluation.

6 Related work

The work closest to ours is [19], where a preliminary version of FACPL is introduced to formalize the semantics of XACML. However, the language considered here is more expressive as, e.g., it is equipped with features such as obligations and policy references. Most of all, the contribution of the two works are different: in [19] it is the formalization of XACML, here it is the development of supporting tools for FACPL and the illustration of their effectiveness by means of a significant case study where policies regulate different behavioural aspects. The preliminary version of FACPL is used in [4]

to formalize UML-based models of access control policies for Web applications, while [18] sketches an early description of the development methodology for FACPL (access control) policies.

Recently, many policy-based languages and models have been developed for managing different aspects of programs' behaviour as, e.g., adaptation and autonomic computing. For example, [13] introduces PobSAM, a policy-based formalism that combines an actor-based model, for specifying the computational aspects of system elements, and a configuration algebra, for expressing autonomic managers that, in response to changes, lead the adaptation of the system configuration according to given adaptation policies. This formalism relies on a predefined notion of policies expressed as Event-Condition-Action (ECA) rules. Adaptation policies are specific ECA rules that change the manager configurations. Our notion of policies, being defined for a broader application, is instead more flexible and expressive. To approach autonomic computing issues, IBM has developed a simplified policy language, named Autonomic Computing Policy Language (ACPL) [12]. Such language, however, comes without any precise syntax and semantics. It is worth noticing that, in our case study the cloud manager acts as an autonomic manager for the system's hypervisors.

Another policy language is Ponder [8], for which a number of toolkits have been developed and applied to various autonomous and pervasive systems. The language borrows the idea introduced in [27] of using two separate types of policies for authorization and obligation. Policies of the former type have the aim of establishing if an operation can be performed, while those of the latter type basically are ECA rules. Differently from Ponder, and similarly to more recent languages (e.g., XACML), obligations in FACPL are expressed as part of authorization policies, thus providing a more uniform specification approach.

As a result of the widespread use of policy-based languages in service-oriented systems and international projects, many attempts of formalization have been made. A largely followed approach is based on 'transformational' semantics that translates from a policy-based language, e.g. XACML, into terms of some formalism. For example, [14] uses description logic expressions as target formalism, [3] exploits the process algebra CSP, and [11] the (multi-terminal) binary decision diagrams. The main advantage of these approaches is the possibility of analyzing policies by means of off-the-shelf reasoning tools that may be already available for the target formalisms. From the semantics point of view, this approach provides some alternative high-level representations of policies, which have their own semantics, but not all of them are suitable to drive implementation.

Concerning policy evaluation tools, there are by now many un-official implementations of policy-based standards, especially of XACML. In software production, the most used tools are SUN XACML [26] and HERAS^{AF} [30], which manage XACML policies written in XML. To evaluate a request, they parse the XML policies and then visit parts of the generated DOM trees for calculating the authorization decision. This differs from our enforcement tool, where we evaluate (sets of) requests by executing Java classes implementing the semantic representations of policies, written in the more intuitive syntax of FACPL.

A tool close to our approach is the ALFA Eclipse plugin [1] developed by Axiomatics. It provides a domain specific language to represent and automatically build XACML policies. The Eclipse environment provides code completion and reuse of identifiers. However, differently from our IDE, this plugin does not freely provide a request evaluation feature, since the Axiomatics's evaluation engine is a proprietary software.

7 Concluding remarks

We have described a user-friendly, uniform, and comprehensive approach to the development and enforcement of behavioural policies which is based on FACPL and its software tools. The policy language FACPL can indeed be used to develop policies controlling various aspects of information systems, ranging from access control to resource usage and adaptation. It has a compact and intuitive syntax and is endowed with a formal semantics, which lays the basis for developing tools and methodologies for analyzing policies. Most of all, for what concerns this paper, FACPL is equipped with easy-to-use software tools, i.e. a powerful IDE and a Java library, supporting policy developers and system administrators in the policy development and enforcement tasks. We have illustrated potentialities and effectiveness of our approach through a significant case study from the cloud computing domain.

As a future work, we plan to continue the validation of FACPL and its tools. On the one hand, we will apply them to a real-world, open-source, IaaS cloud system (e.g., OpenNebula or OpenStack) and to case studies from different domains (e.g., smart grids). In the former case, we plan to adapt our FACPL-based cloud manager to be properly embedded in the cloud systems. On the other hand, we intend to experimentally evaluate the performance of our enforcement tool. We will also extend our IDE with new features as, e.g., a

dedicated view to summarize and analyze policy evaluation results. Another research line we intend to pursue is the development of methods and techniques for analyzing FACPL policies. In particular, they will be first theoretically defined and, then, integrated in our software tools in order to achieve a complete framework for developing trustworthy policies.

References

1. Axiomatics: Axiomatics Language for Authorization (ALFA) (2013). http://www.axiomatics.com/axiomatics-alfa-plugin-for-eclipse.html
2. Bertolino, A., Daoudagh, S., Lonetti, F., Marchetti, E.: The X-CREATE Framework - A Comparison of XACML Policy Testing Strategies. In: WEBIST, pp. 155–160. SciTePress (2012)
3. Bryans, J.: Reasoning about XACML policies using CSP. In: SWS, pp. 28–35. ACM (2005)
4. Busch, M., Koch, N., Masi, M., Pugliese, R., Tiezzi, F.: Towards model-driven development of access control policies for web applications. In: MDsec. ACM (2012)
5. Chen, Y., Paxson, V., Katz, R.H.: Whats new about cloud computing security? Tech. Rep. UCB/EECS-2010-5, EECS Department, University of California, Berkeley (2010). URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-5.html
6. Clark, J., DeRose, S.: XML Path Language (XPath) version 1.0 (1999). http://www.w3.org/TR/xpath
7. Clayman, S., Galis, A., Chapman, C., Toffetti, G., Rodero-Merino, L., Vaquero, L.M., Nagin, K., Rochwerger, B.: Monitoring service clouds in the future internet. In: Future Internet Assembly, pp. 115–126. IOS Press (2010)
8. Damianou, N., Dulay, N., Lupu, E., Sloman, M.: The Ponder Policy Specification Language. In: POLICY, LNCS 1995, pp. 18–38. Springer (2001)
9. Davide Lamanna, D., Skene, J., Emmerich, W.: Slang: a language for defining service level agreements. In: FTDCS, pp. 100–106. IEEE (2003)
10. FACPL Site: (2013). http://rap.dsi.unifi.it/facpl/
11. Fisler, K., Krishnamurthi, S., Meyerovich, L.A., Tschantz, M.C.: Verification and change-impact analysis of access-control policies. In: ICSE, pp. 196–205. ACM (2005)
12. IBM: Autonomic Computing Policy Language - ACPL (2006). http://www.ibm.com/developerworks/tivoli/tutorials/ac-spl/
13. Khakpour, N., Jalili, S., Talcott, C.L., Sirjani, M., Mousavi, M.R.: Formal modeling of evolving self-adaptive systems. Sci. Comput. Program. **78**(1), 3–26 (2012)
14. Kolovski, V., Hendler, J.A., Parsia, B.: Analyzing web access control policies. In: WWW, pp. 677–686. ACM (2007)
15. Ludwig, H., Keller, A., Dan, A., King, R.P., Franck, R.: Web Service Level Agreement (WSLA) Language Specification, v1.0 (2003). URL http://www.research.ibm.com/wsla/WSLASpecV1-20030128.pdf
16. Margheri, A., Masi, M., Pugliese, R., Tiezzi, F.: A Formal Software Engineering Approach to Policy-based Access Control. Tech. rep., DiSIA, Univ. Firenze (2013). Available at http://rap.dsi.unifi.it/facpl/research/Facpl-TR.pdf
17. Margheri, A., Masi, M., Pugliese, R., Tiezzi, F.: Developing and enforcing policies for access control, resource usage, and adaptation: A practical approach. In: WS-FM, LNCS. Springer (2013). To appear
18. Margheri, A., Masi, M., Pugliese, R., Tiezzi, F.: On a formal and user-friendly linguistic approach to access control of electronic health data. In: HEALTHINF, pp. 263–268. SciTePress (2013)
19. Masi, M., Pugliese, R., Tiezzi, F.: Formalisation and Implementation of the XACML Access Control Mechanism. In: ESSoS, LNCS 7159, pp. 60–74. Springer (2012)
20. Mell, P., Grance, T.: The NIST Definition of Cloud Computing (2011). NIST Special Publication 800-145
21. Moore, B., Ellesson, E., Strassner, J., Westerinen, A.: Policy Core Information Model – Version 1 Specification. RFC 3060 (Proposed Standard) (2001). URL http://www.ietf.org/rfc/rfc3060.txt. Updated by RFC 3460
22. NIST: A survey of access control models (2009). http://csrc.nist.gov/news_events/privilege-management-workshop/PvM-Model-Survey-Aug26-2009.pdf
23. OASIS: Cross-Enterprise Security and Privacy Authorization (XSPA) Profile of XACML v2.0 for Healthcare v1.0 (2009)
24. OASIS Security Services TC: Assertions and protocols for the OASIS security assertion markup language (SAML) v2.02 (2005)
25. OASIS XACML TC: eXtensible Access Control Markup Language (XACML) version 3.0 - Candidate OASIS Standard (2012)
26. Proctor, S.: SUN XACML (2011). http://sunxacml.sourceforge.net

27. Sloman, M.: Policy driven management for distributed systems. J. Network Syst. Manage. **2**(4), 333–360 (1994)
28. Subashini, S., Kavitha, V.: A survey on security issues in service delivery models of cloud computing. Journal of Network and Computer Applications **34**(1), 1 – 11 (2011)
29. The epSOS project: A european ehealth project (2008-2014). `http://www.epsos.eu`
30. The Herasaf consortium: HERAS^{AF} (2012). `http://www.herasaf.org`
31. The NHIN project: The Nationwide Health Information Network Project (2009). `http://healthit.hhs.gov/portal/server.pt`
32. Verma, D.: Supporting Service Level Agreements on IP Networks. Macmillan technology series. Macmillan Technical Pub. (1999)

Appendix

We report in Listing 1 the complete energy saving policies for the cloud IaaS case study. Specifically, these policies aim at concentrating the workload on hypervisor `HYPER_1`, considered as the primary hypervisor, and using hypervisor `HYPER_2` only when the other is fully loaded.

Listing 1 Energy saving policies

```
{ pep:
  deny-biased
  pdp:
  permit-overrides
  PolicySet Create_Policies { permit-overrides
   target:
      equal("CREATE", action/action-id)
   policies:
    Policy SLA_Type1 < deny-unless-permit
     target:
        ( equal("P_1", subject/profile-id)
          || equal("P_2", subject/profile-id) )
        && equal("TYPE_1", resource/vm-type)
     rules:
      Rule hyper_1 (permit
        target:
        less-than-or-equal(1, system/hyper1.
                           availableResources)
        obl:
        [permit M create("HYPER_1", system/vm-id,
                         "TYPE_1")]
      )
      Rule hyper_2 (permit
        target:
        less-than-or-equal(1, system/hyper2.
                           availableResources)
        obl:
        [permit M create("HYPER_2", system/vm-id,
                         "TYPE_1")]
      )
      obl:
          [deny 0 warning("Not enough available
            resources for TYPE_1 VMs")]
    >
    Policy SLA_Type2 < deny-unless-permit
     target:
      equal("P_2", subject/profile-id)
      && equal("TYPE_2", resource/vm-type)
     rules:
      Rule hyper_1_create (permit
       target:
        less-than-or-equal(2, system/hyper1.
                           availableResources)
       obl:
         [permit M create("HYPER_1", system/vm-id,
                          "TYPE_2")]
       )
      Rule hyper_2_create (permit
       target:
        less-than-or-equal(2, system/hyper2.
                           availableResources)
       obl:
         [permit M create("HYPER_2", system/vm-id,
                          "TYPE_2")]
       )
      Rule hyper_1_freeze (permit
       condition:
        or(and(equal(0, system/hyper1.
                        availableResources),
               less-than-or-equal(2, system/hyper1.
                                     vm1-counter)),
          and(equal(1, system/hyper1.
                       availableResources),
               less-than-or-equal(1, system/hyper1.
                                     vm1-counter)))
       obl:
         [permit M freeze("HYPER_1",
                          subtract(2, system/hyper1.
                                      availableResources),
                          "TYPE_1")]
         [permit M create("HYPER_1", system/vm-id,
                          "TYPE_2")]
      )
      Rule hyper_2_freeze (permit
       condition:
        or(and(equal(0, system/hyper2.
                        availableResources),
               less-than-or-equal(2, system/hyper2.
                                     vm1-counter)),
          and(equal(1, system/hyper2.
                       availableResources),
               less-than-or-equal(1, system/hyper2.
                                     vm1-counter)))
       obl:
         [permit M freeze("HYPER_2",
                          subtract(2, system/hyper2.
                                      availableResources),
                          "TYPE_1")]
         [permit M create("HYPER_2", system/vm-id,
                          "TYPE_2")]
      )
    obl:
        [deny 0 warning("Not enough available
                         resources for TYPE_2 VMs")]
      >
  }
  Policy Release_Policies < permit-overrides
    target:
      equal("RELEASE", action/action-id)
      && ( equal("P_1", subject/profile-id)
           || equal("P_2", subject/profile-id) )
    rules:
     Rule hyper_1_release (permit
       condition: at-least-one-member-of(
                             resource/vm-id,
                             system/hyper1.vm-ids)
       obl: [permit M release("HYPER_1",
                              resource/vm-id)]
     )
     Rule hyper_2_release (permit
       condition: at-least-one-member-of(
                             resource/vm-id,
                             system/hyper2.vm-ids)
       obl: [permit M release("HYPER_2",
                              resource/vm-id)]
     )
  >
}
```