Towards Model-Driven Development of Access Control Policies for Web Applications *

Marianne Busch
LMU München
Oettingenstr. 67
80538 Munich, Germany
busch@pst.ifi.lmu.de

Nora Koch
LMU München
Oettingenstr. 67
80538 Munich, Germany
kochn@pst.ifi.lmu.de

Massimiliano Masi
Tiani "Spirit" GmbH
Guglgasse, 6
1110 Vienna, Austria
massi@tiani-spirit.com

Rosario Pugliese
Università di Firenze
Viale Morgagni, 65
50134 Firenze, Italy
rosario.pugliese@unifi.it

Francesco Tiezzi
IMT Advanced Studies Lucca
Piazza S. Ponziano, 6
55100 Lucca, Italy
francesco.tiezzi@imtlucca.it

ABSTRACT

We introduce a UML-based notation for graphically modeling systems' security aspects in a simple and intuitive way and a model-driven process that transforms graphical specifications of access control policies in XACML. These XACML policies are then translated in FACPL, a policy language with a formal semantics, and the resulting policies are evaluated by means of a Java-based software tool.

Keywords

Security, Model-driven development, Web engineering

1. INTRODUCTION

Security is an important issue in software, in particular in software publicly available, as web applications. Software security requires protection of the system's resources against unauthorized access, ensuring as well accessibility by authorized users whenever needed. Due to the ever increasing relevance of security, the software development processes are being improved for comprising security aspects, such as integrity and confidentiality, from the beginning on, i.e. in early phases like requirements engineering and design instead of adding them at implementation level.

Access control is a fundamental means for restricting what operations (authenticated) users can perform on protected resources. Different access control systems have been developed to support security. The main components of these systems are the security policies, a security model and the implementation mechanisms [7]. The policies define the rules according to which access control must be regulated. The model provides a formal representation of the policies and how they work. The mechanisms define how the controls imposed by the policies and the model are implemented.

Several languages for access control have been proposed. Many of them are XML-based as, e.g., the OASIS standard eXtensible Access Markup Language (XACML) [13]. XACML is the de-facto industry standard for expressing and enforcing policy-based authorizations. However, the XML syntax of XACML can make the task of writing policies difficult and error-prone, besides it is not adequate for formally defining the semantics of the language and reasoning on it. Other languages rely on concepts and techniques from logic, which instead offer the advantage of formal foundation and possibilities of analysis, but have the drawback of not being easily usable for a wide spectrum of users.

Access control languages might thus be too low level and impractical for developers accustomed to work with abstract architectural models of systems. Our aim is to make the specification of access control policies accessible to people not necessarily familiar with such languages. The proposal we illustrate hereafter is to initially specify the security requirements using a high-level, graphical, modeling language called UWE [5, 6], which is based on the de-facto standard modeling language UML and thus provides a human understandable view of the access control policies in force at the system. UWE is already equipped with tools for policy editing. We then automate the policy development process towards a formally founded language (called FACPL [12]) by means of a suitable software toolchain, comprising the transformations UWE2XACML and XACML2FACPL. The former transformation enables the possibility of integrating into the toolchain other software tools, e.g. generation of test cases that take XACML as input language. The latter transformation gives to software engineers a powerful tool with solid mathematical foundations that enables formal reasoning on the policies of applications developed using UWE. The proposed model-driven process, shown in Figure 1, solves the problems mentioned before. It offers the advantages of an easy to learn and intuitive visual specification

*(Does NOT produce the permission block, copyright information nor page numbering). For use with ACM_PROC_ARTICLE-SP.CLS. Supported by ACM.

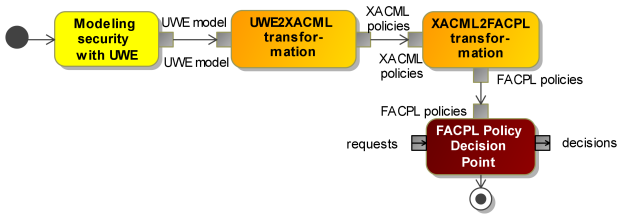

Figure 1: Toolchain for the model-driven approach

language for policies, which can be also translated automatically to a formal specification enabling evaluation of policies and access requests through the FACPL Policy Decision Point. Instructions and software for installing our toolchain can be found at `http://uwe.pst.ifi.lmu.de/uwe2facpl`. We illustrate our approach by means of a running example from the e-Health domain.

2. MODELING ACCESS CONTROL

In this section, we outline UML-based Web Engineering (UWE) [5, 6], an engineering approach for modeling web applications. We focus, in particular, on the access control aspects of UWE and use an example from the e-Health area.

2.1 UML-based Web Engineering (UWE)

UWE uses the extension mechanisms provided by UML via the definition of a UML profile, which provides a set of stereotypes, tag definitions and constraints. One of the cornerstones of the UWE language is the "separation of concerns" principle using separate models for views such as content, navigation, presentation, processes, etc. The most relevant UWE models for this work are: (1) The *Content Model*, representing the domain concepts that are relevant for the web application and the relationships between them. (2) The *Role Model*, defining a hierarchy of user groups with the purpose of authorization and access control. It is usually included in a *User Model*, which specifies basic structures as e.g. that a user can take on certain roles simultaneously. (3) The *Basic Rights Model*, expressing role based access control on the domain concepts specified in the content model, picking the roles from a user model. (4) The *Navigation Model*, providing a graphical representation of the path the user can navigate in the web system. This model also represents security features as, e.g., authentication, access control and secure connections [5].

For each view, an appropriate type of UML diagram is selected and a set of stereotypes, tag definitions and constraints is provided. Concepts of the content and role models and their relationships are shown as classes and associations in a UML class diagram. The basic rights model connects role instances with content classes or their attributes/methods using stereotyped dependencies. These dependencies specify create/read/update/delete/execution rights. For the navigation model, UWE provides two graphical representations: a structural visualization as UML stereotyped class diagrams and a behavioral form using UML state machines.

2.2 The HospInfo Case Study

Our case study, called *HospInfo*[1], is a prototype of a web-based Hospital Information System. The roles identified for

[1] *HospInfo*. A secure hospital information system. `http://uwe.pst.ifi.lmu.de/exampleHospInfo.html`

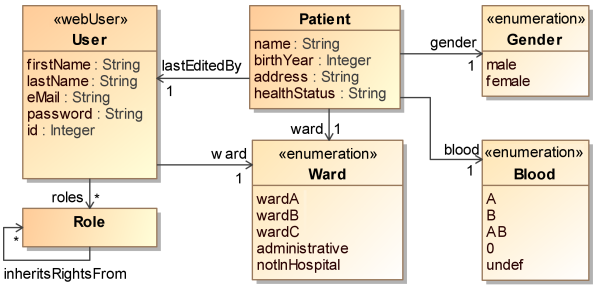

Figure 2: *HospInfo* content model

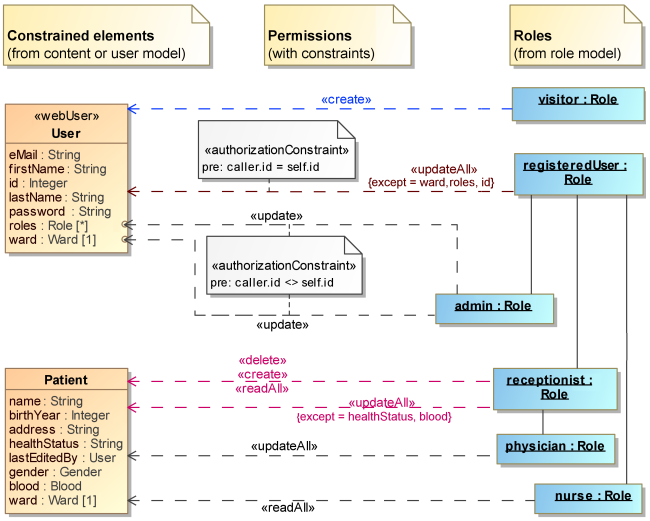

Figure 3: *HospInfo* basic rights

this web application are: visitor, registeredUser, nurse, receptionist, physician, and admin. Its main requirements are: (1) staff members should be able to register; (2) an administrator can set roles to staff members; (3) physicians need the permission to create new patient records or change information of patients; (4) nursing staff should be able to read the health records of the patients; (5) receptionists can read and update all information with exception of health related data, while only physicians can update the latter ones.

The focus of interest of the content model is on the `Patient` class with attributes as name, address, ward or gender (see Figure 2). The classes `User` and `Role` (from the user model) are included as well in Figure 2, for showing the associations to the content model elements. Figure 3 depicts the basic rights model with access specifications for the classes `User` and `Patient`. The rule that admins cannot change their own user account is depicted with the OCL [14] *authorizationConstraint* in the center of the UML diagram. Thereby, the variable `caller` stands for the operating user, i.e. the admin. The {except=healthStatus, blood} tag on the «updateAll» dependency between `Receptionist` and `Patient` specifies that the updates on all other attributes of `Patient` are permitted. Conversely, physicians can update all `Patient` attributes without any {except} restrictions.

Figure 4 shows an excerpt of the main navigation state diagram for *HospInfo*. The whole application *HospInfo* transmits all information in a ***c***onfidential way and cares for the ***i***ntegrity and the ***f***reshness of the data (denoted by «session»{transmissionType="cif"}). Basically, *HospInfo* consists of the two navigation areas depicted in Figure 4: a

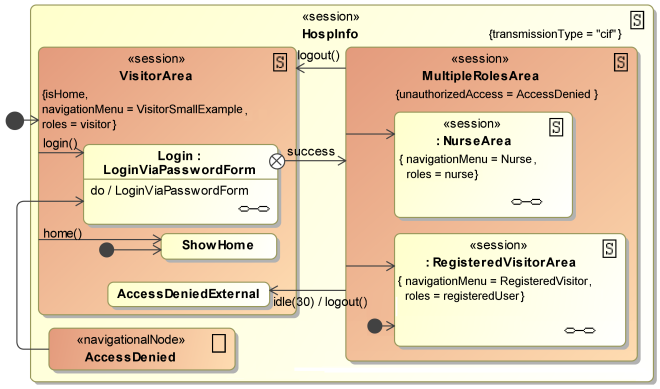

Figure 4: *HospInfo* navigational states (excerpt)

visitor area (on the left) and an internal area (on the right), which is guarded according to the existing roles.

2.3 The MagicUWE CASE Tool

UWE models can be built using any UML CASE tool that enables the use of profiles. The UWE profile can be downloaded from the UWE website[2]. We use the MagicUWE plugin [6] implemented for MagicDraw[3] that provides additional support to the developer.

With MagicUWE, repetitions can be avoided. Thus, instead of creating a basic element, as a class, and applying a stereotype to it, UWE's stereotyped elements can be inserted directly from the MagicDraw toolbar. Besides, transformations between UWE models can be performed semi-automatically. E.g., the modeler can start modeling the content class diagram; then the model can be transformed with MagicUWE to a navigation model. Additional stereotyped elements can be added to the navigation diagram from the toolbar as, e.g., an element to indicate the home node that represents the main entry of a web application.

3. POLICY TRANSFORMATION

In this section, we describe our approach for generating XACML and FACPL policies from UWE models.

3.1 The XACML Standard

XACML permits decoupling the access control from the application's flow. In its underlying access control model, the access to each resource is regulated by one or more policies, i.e. XML documents expressing the capabilities and credentials that a requestor must have for accessing the resource. A request to access a resource can be created by, e.g., a remote-access gateway, a Web server or an email user-agent.

Evaluation of XACML access requests is as follows. The authorization decision is made by the Policy Decision Point (PDP) by checking the *matching* between values of request's attributes and the corresponding values retrieved from the policies. The decision can be one among permit, deny, not-applicable and indeterminate: the first two values have an obvious meaning, 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.

[2]UWE website. `http://uwe.pst.ifi.lmu.de/`
[3]MagicDraw. `http://www.magicdraw.com`

Let us now consider the policy language provided by the standard. The basic element of this language is `<Policy>`. A `<Policy>` is composed of a `<Target>`, which identifies the set of capabilities and credentials that the requestor must expose, and some `<Rule>`s. Every `<Rule>` contains the logic for the access control decision and has an `Effect`, which can be either `Permit` or `Deny`. A `<Policy>` also specifies a combining algorithm that defines what is the final decision for a request when there are contradictory rule decisions (e.g. both permit and deny results are returned). The most relevant algorithms are: `deny-overrides`, if any rule in the considered policy evaluates to deny, then the result of the policy is deny; `permit-overrides`, it is like `deny-overrides`, but permit takes precedence over the other results.

A `<Target>` is composed of four sub-elements: `<Subjects>`, `<Actions>`, `<Resources>`, and `<Environments>`. Each category is composed of a set of target elements, each of which contains an attribute identifier, a value and a matching function. Such information is used to check whether the policy is applicable to a given request. Specifically, the matching function retrieves a value from the designed attribute in the request and matches it with the values specified in the target element, according to the function's semantics. If, for all four categories, at least a matching of a target element succeeds, then the policy is applicable to the request.

Besides the `Effect`, a `<Rule>` may specify a `<Target>`, which refines the applicability established by the target of the enclosing policy, and a `<Condition>`, i.e. a combination of functions that operate on values coming from the request. The `Effect` is propagated to the upper level policy if the `<Target>` of the rule matches and the `<Condition>` holds.

Policies can be combined into a `<PolicySet>`, which specifies a combining algorithm and a `<Target>`. The latter is evaluated before the targets of the included policies are.

3.2 Model Transformations to XACML

By means of our running example, we present here how to transform UWE basic rights models into XACML policies.

Transformation of UWE models into XACML policies. As the length of the resulting XACML file for our *HospInfo* example (cf. Figure 3) exceeds several hundred lines, we sketch the rough structure of the file below (detailed policies and the sources of our tool can be found at `http://uwe.pst.ifi.lmu.de/uwe2facpl`).

```
PolicySet ..... root element, permit-overrides
 ├─ PolicySet ..... for each role
 │   ├─ Target ..... role and all sub-roles
 │   └─ Policy ..... for each pair of role and target
 │       ├─ Target ..... constrained element
 │       └─ Rule ..... permission for action
 │           ├─ Resources   e.g. attributes
 │           ├─ Actions ... permitted action
 │           └─ Condition   transformed constraint
 └─ Policy ..... deny all
```

Intuitively, the transformation generates a `<PolicySet>` for each role, each of which contains one `<Policy>` for any class connected to the considered role (e.g. both `<PolicySet>` for `Receptionist` and `Physician` includes one policy for the

content class `Patient`). Furthermore, a single `<Policy>` is used to deny access to all resources not specified in the `<PolicySet>`, which is the default behavior of UWE's basic rights models.
To allow a sub-role of a given role to use the permission specified by the super-role, the target of the `<PolicySet>` corresponding to the super-role is extended to also match requests from the sub-role (e.g. the target of the `<PolicySet>` for `receptionist` specifies two subjects, with roles `receptionist` and `physician`, respectively).
Each `<Policy>` for a constrained class contains one `<Rule>` for each action between the role and the class. For example, the `receptionist` policy comprises rules for actions «delete», «create», «read» and «updateAll» {except = healthStatus, blood}. Attributes targeted by `*All` actions are divided into a set of `<Resources>`, omitting those from the {except} tag. OCL constraints inside UML comments with «authorizationConstraint» stereotype are transformed to a `<Condition>`. The condition is located within a `<Rule>` representing the appropriate action. For the time being, we implemented only a few basic OCL constraints.
Notably, XACML is more expressive than UWE, hence a transformation from XACML to UWE is not feasible.

Implementation. Technically, our UWE2XACML transformation from projects modeled with the UWE Profile v.2.2 to XACML 2.0 is implemented using the modeling framework of Eclipse Juno[4]. We also used Xpand 1.2.1[5], a language specialized on code generation based on models defined by the modeling component of Eclipse. Xpand is based on workflows, which apply templates in order to parse the model and to produce the desired code.
To be able to use the project files of MagicDraw 16.8 for the transformation with Xpand, they have to be exported as Eclipse UML2 (v3.x) XMI file. For complex tasks as the transformation to XACML, Java extensions are used from within the Xpand templates, because Java enables us, e.g., to group several dependencies with equal constraints to only one `<Rule>`. This is also needed for our *HospInfo* example regarding both update permissions (roles and ward) from the admins.
Our algorithm transforms not only the basic rights model, but also states with a {roles} tag from the navigation model (see Figure 4). The aim is to constrain whether or not a user is allowed to navigate to a certain area.

3.3 The FACPL Policy Language

The Formal Access Control Policy Language (FACPL) [12], which we describe in this section, provides a manageable alternative syntax to XACML through a BNF-like grammar. FACPL syntax is reported in Table 1. As usual, square brackets are used to indicate optional items.

To base an authorization decision on some characteristics of the request, like e.g. the subject's identity or the resource's identifier, FACPL provides *(structured) names*, ranged over by name. They permit to identify specific values (called *attribute* values) contained in the request. The language is also

[4]Eclipse. `http://www.eclipse.org/`
[5]Xpand. `http://wiki.eclipse.org/Xpand`

Policies	::=	$\{Alg\,;\mathsf{target}:\{\,[\,Targets\,]\,\}\,;Policies\}$
	|	$\langle Alg\,;\mathsf{target}:\{\,[\,Targets\,]\,\}\,;\mathsf{rules}:\{Rules\}\rangle$
	|	*Policies* *Policies*
Alg	::=	deny-overrides | permit-overrides | ...
Targets	::=	*MatchId*(value,name) | $Targets \vee Targets$
	|	$Targets \wedge Targets$ | $Targets \sqcap Targets$
MatchId	::=	string-equal | integer-equal | ...
Rules	::=	$(Effect\,[\,;\mathsf{target}:\{Targets\}\,][\,;\mathsf{condition}:\{\mathsf{expr}\}\,])$
	|	*Rules* *Rules*
Effect	::=	permit | deny

Table 1: FACPL syntax

equipped with *expressions* that permit to specify conditions.

FACPL policies can be simple policies of the form $\langle Alg\,;\mathsf{target}:\{\,[\,Targets\,]\,\}\,;\mathsf{rules}:\,\{Rules\}\rangle$ or, recursively, *policy sets* of the form $\{Alg\,;\mathsf{target}:\{\,[\,Targets\,]\,\}\,;\,Policies\}$. Both policies and policy sets specify the algorithm for combining the results of the evaluation of the contained elements and a target to which the policy/policy set applies. A *target* identifies the set of access requests that a rule, a policy or a policy set is intended to evaluate. Specifically, a target specifies the set of *subjects*, *resources*, *actions* and *environments* to which the corresponding rule/policy/policy set applies. In the XML-based syntax of XACML, the target element may contain four separate elements, one for each of the above categories. To obtain a more compact notation, FACPL represents a target as an expression built from *match elements*, i.e. terms of the form *MatchId*(value,name), by exploiting an operator for logical disjunction, $\vee$, and two operators for logical conjunction, $\wedge$ and $\sqcap$. Each match element spells out a specific value that the subject/resource/action/environment in the decision request (identified by the given name) must match, according to the matching function *MatchId*. A disciplined use of structured names and the three logical operators permits properly expressing XACML targets. For further details on this topic, the reader is referred to our previous work [12].

A single policy contains a (non-empty) set of rules such as $(Effect\;[\,;\mathsf{target}:\{Targets\}][\,;\mathsf{condition}:\{\mathsf{expr}\}\,])$, each specifying: *(i)* an *effect*, which indicates the rule-writer's intended consequence of a positive evaluation for the rule (the allowed values are permit and deny), *(ii)* a *rule target*, which refines the applicability established by the target of the enclosing policy, and *(iii)* a *condition*, which is a boolean expression that may further refine the applicability of the rule. In a rule, target and condition may be absent.

Transformation of XACML policies into FACPL. The transformation, performed by the XACML2FACPL component in Figure 1, is straightforward. Its flow loops over the policy sets creating the necessary data structures for the FACPL representation. The original XML document is read by using JAXB[6]. The loop over the elements is driven by the XACML schema definitions by traversing its data types. We show below the FACPL policies resulting from the transformation of the *HospInfo* basic rights model (only

[6]JAXB. `http://jaxb.java.net`

for roles `receptionist` and `physician`).

```
{permit-overrides;
 target :{ string-equal("physician", subject.role) };
 ⟨permit-overrides;
  target :{ string-equal("patient", resource.id) };
  rules :{(permit;
           target :{ string-equal("update", action.id)
                   ⊓ string-equal("name", resource.attr)
                     ∨ string-equal("birthYear", resource.attr)
                     ∨ ...
                     ∨ string-equal("ward", resource.attr) })
          (deny) } ⟩
 ⟨permit-overrides; target :{ }; rules :{(deny) } ⟩ }

{permit-overrides; target :{string-equal("receptionist", subject.role)
                          ∨ string-equal("physician", subject.role)};
 ...
 ⟨permit-overrides;
  target :{ string-equal("patient", resource.id) };
  rules :{  ...
           (permit; target :{ string-equal("delete", action.id) })
           ...
  } ⟩ }
```

4. POLICY EVALUATION

In this section, we sketch the formal semantics of FACPL, which is at the basis of the FACPL policy evaluation tool.

The semantics of FACPL policies is given in a denotational style, i.e. it is defined by a function $[\![\cdot]\!]_R$ that, given a policy/policySet and a set R of access requests, returns a *decision* tuple of the form

$$(\mathsf{permit}: R_p; \mathsf{deny}: R_d; \mathsf{not\text{-}applicable}: R_n; \mathsf{indeterminate}: R_i)$$

where R_p, R_d, R_n and R_i form a partition of R according to the results of the requests' evaluation. Notably, R can contain, e.g., all possible requests, only requests with a given structure or a single request. The definition of $[\![\cdot]\!]_R$ relies on an auxiliary function $(\!|\cdot|\!)_R$ that, given a target, returns a *matching* tuple of the form

$$(\mathsf{match}: R_m; \mathsf{no\text{-}match}: R_n; \mathsf{indeterminate}: R_i)$$

where R is partitioned into R_m, R_n and R_i according to the results of the target evaluation. We refer the interested reader to [12] for a full account of the FACPL semantics.

As an example, let us consider the following access requests:

```
request :{                          request :{
 (subject.lastName,"House")          (subject.lastName,"Cameron")
 (subject.role,"physician")          (subject.role,"receptionist")
 ...                                 ...
 (resource.id,"patient")             (resource.id,"patient")
 (resource.attr,"healthStatus")      (resource.attr,"healthStatus")
 (action.id,"update")                (action.id,"update")
}                                   }
```

The request on the left is made by the physician House for updating the health record of a patient, while the request on the right is made by the receptionist Cameron for performing the same action. Given the above requests and the *HospInfo* policies generated by the XACML2FACPL component of our toolchain, the FACPL semantics returns a decision tuple where the request on the left is in the permit set while the request on the right is in the deny set (indeed, receptionists have no permission to update patient health records).

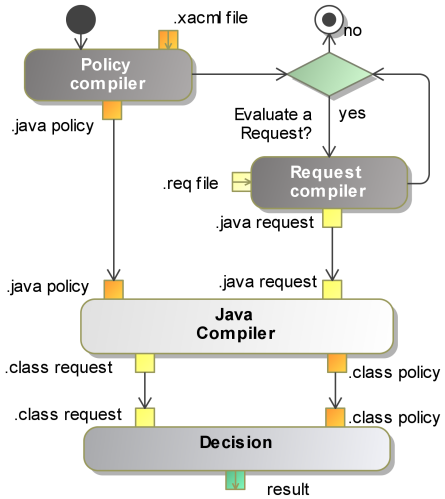

Figure 5: FACPL Policy Decision Point

The implementation[7] of the FACPL language is made in Java. The workflow of such a tool is graphically depicted in Figure 5. This tool "compiles" a policy written in the syntax presented in Sec. 3.3 into a Java class following the semantics rules defined in [12]. Thus, a repository storing some policies, and the related PDP, consists of a Java archive containing all the Java classes generated from the policies. Similarly, an access request is compiled into a Java class. A policy decision is then computed by executing the generated policy code with the request code passed as parameter to an entry method. The generated PDP can be then integrated as a module into the code of the main Web application, possibly obtained from other UWE models.

5. RELATED WORK

This work is related to engineering approaches, which address the specification of secure systems, and to works on XACML's formalization. UMLsec [9] is a UML extension emphasizing on secure protocols. Its UML profile includes stereotypes for security concepts like authenticity, freshness, and secure information flow. In particular, the use of constraints gives criteria to evaluate the security aspects of a system design. UMLsec models compared to UWE ones are very detailed, therefore less appropriate for modeling web applications at a high-level of abstraction. SecureUML [11] is a UML-based modeling language for secure, distributed systems. It provides modeling elements for role-based access control and the specification of authorization constraints. In UWE, we use dependencies instead of the SecureUML association classes, which avoids the use of method names with an access related return type. However, UWE's basic rights models can easily be transformed into a SecureUML representation. UACML [16] provides a UML-based meta-metamodel for access control, which can be specialized into various meta-models for, e.g., role-based access control or mandatory access control. Conversely to UWE, the resulting diagrams are overloaded as subjects like users are not modeled separately. ActionGUI [1] is a MDD approach that uses SecureUML and ComponentUML to model access control rules, and a GUI model enriched with OCL constraints.

[7]Source and binary code of the FACPL implementation are available from `http://rap.dsi.unifi.it/xacml_tools`

It provides a formal specification of functionalities and access control policies. Compared to UWE, ActionGUI restricts the web application to a smaller set of features (e.g. menus are not available).

Regarding works on XACML's formalization, a largely followed approach is based on 'transformational' semantics (see, e.g., [10, 4, 3]). The target formalisms have in their turn their own semantics. This makes it more difficult to understand the formal meaning of policies with respect to FACPL formal semantics, which directly associates mathematical objects (i.e. 4-tuples of request sets) to policies. These concepts are easier and more understandable than terms like, e.g., description logic expressions. In fact, FACPL semantics has been conveniently exploited to drive a formal-based XACML implementation (cf. Sec. 4). It differs from the many XACML implementations (see, e.g., the OASIS website) because it enables the development of reasoning tools. Besides, when policies do not change frequently, our implementation enables a faster decision because it does not need to parse the same XML tree at each request, but just to instantiate a Java object already in the classpath. In more dynamic scenarios, however, the generation of the PDP may add a constant time to policy evaluation. Finally, the use of a non-XML syntax for XACML is not new; e.g., a syntax similar to that of FACPL is proposed in [15], while a 'display' notation that combines a graphical interface with a natural language like format is introduced in [17]. But, again, such approaches do not rely on a formal semantics.

6. CONCLUSIONS AND FUTURE WORK

To support software engineers in the task of specifying and analyzing access control aspects of web applications, we put forward the use of graphical tools and of a policy language with mathematical foundations. Specifically, the main contribution of this work is a toolchain comprising a graphical editor for modeling security aspects, tools for transforming access control policies from the graphical notation to XACML and from this to FACPL, and a compiler for getting Java classes from FACPL policies.

We can currently transform a subset of OCL constraints in XACML, but we plan to exploit XACML *obligations* to deal with the whole set. Such constraints will be separately evaluated and, thus, the request will be authorized if the policy decision is positive and all obligations (i.e. the related OCL constraints) hold. We also intend to integrate tools for analyzing policies and generating test cases, such as Margrave [8] and X-CREATE [2], respectively; our intermediate transformation in XACML enables this possibility. We plan to compare the performance of our tool with those of other implementations based both on XACML, such as Axis2's XACMLight[8] and Sun's XACML[9], and on different approaches to access control, such as Spring Security[10]. Finally, our work relies XACML 2.0, but we plan to also support the upcoming version.

[8]XACMLight. `http://xacmllight.sourceforge.net`
[9]Sun's XACML. `http://sunxacml.sf.net`
[10]Spring Security project. `http://www.springsource.org/spring-security`

Acknowledgments. This work has been partially sponsored by the EU projects ASCENS, FP7 257414, and NESSoS, NoE 256980.

7. REFERENCES

[1] D. Basin, M. Clavel, and M. Egea. Automatic Generation of Smart, Security-Aware GUI Models. In *ESSoS*, LNCS 5965, pages 201–217. Springer, 2010.

[2] A. Bertolino, S. Daoudagh, F. Lonetti, and E. Marchetti. The X-CREATE Framework - A Comparison of XACML Policy Testing Strategies. In *WEBIST*, pages 155–160. SciTePress, 2012.

[3] J. Bryans. Reasoning about XACML Policies using CSP. In *SWS*, pages 28–35. ACM, 2005.

[4] J. Bryans and J. S. Fitzgerald. Formal Engineering of XACML Access Control Policies in VDM++. In *ICFEM*, LNCS 4789, pages 37–56. Springer, 2007.

[5] M. Busch, A. Knapp, and N. Koch. Modeling Secure Navigation in Web Information Systems. In *BIR*, LNBIP 90, pages 239–253. Springer, 2011.

[6] M. Busch and N. Koch. MagicUWE — A CASE Tool Plugin for Modeling Web Applications. In *ICWE*, LNCS 5648, pages 505–508. Springer, 2009.

[7] S. De Capitani di Vimercati, P. Samarati, and S. Jajodia. Policies, models, and languages for access control. In *DNIS*, LNCS 3433, pages 225–237. Springer, 2005.

[8] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. Verification and change-impact analysis of access-control policies. In *ICSE*, pages 196–205. ACM, 2005.

[9] J. Jürjens. *Secure Systems Development with UML*. Springer, 2005.

[10] V. Kolovski, J. A. Hendler, and B. Parsia. Analyzing Web Access Control Policies. In *WWW*, pages 677–686. ACM, 2007.

[11] T. Lodderstedt, D. Basin, and J. Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In *UML*, LNCS 2460, pages 426–441. Springer, 2002.

[12] M. Masi, R. Pugliese, and F. Tiezzi. Formalisation and Implementation of the XACML Access Control Mechanism. In *ESSoS*, LNCS 7159, pages 60–74. Springer, 2012.

[13] OASIS XACML TC. eXtensible Access Control Markup Language (XACML) version 2.0, 2005.

[14] OMG. Object Constraint Language (OCL) v2.3.1, 2012. `http://www.omg.org/spec/OCL/2.3.1/`.

[15] OpenLiberty. Easy XACML syntax with OpenAzPolicyReader, 2010. From OpenAz maillist. `http://lists.openliberty.org/pipermail/openaz/2010-July/000074.html`.

[16] N. Slimani, H. Khambhammettu, K. Adi, and L. Logrippo. UACML: Unified Access Control Modeling Language. In *NTMS 2011*, pages 1–8, 2011.

[17] B. Stepien, A. P. Felty, and S. Matwin. A Non-technical User-Oriented Display Notation for XACML Conditions. In *MCETECH*, LNBIP 26, pages 53–64. Springer, 2009.