Security@Runtime: A Flexible MDE approach to Enforce Fine-grained Security Policies

Yehia Elrakaiby, Moussa Amrani and Yves Le Traon

University of Luxembourg, 4 Alphonse Weicker L-2721, Luxembourg,
{yehia.elrakaiby, moussa.amrani, yves.letraon}@uni.lu

Abstract. In this paper, we present a policy-based approach for automating the integration of security mechanisms into Java-based business applications. In particular, we introduce an expressive Domain Specific modeling Language (DSL), called *Security@Runtime*, for the specification of *security configurations* of targeted systems. The *Security@Runtime* DSL supports the expression of *authorization*, *obligation* and *reaction* policies, covering many of the security requirements of modern applications. Security requirements specified in security configurations are enforced using an *application-independent* Policy Enforcement Point (PEP)-Policy Decision Point (PDP) architecture, which enables the *runtime update* of security requirements. Our work is evaluated using two systems and its advantages and limitations are discussed.

Keywords: Java Security, Security Policies, Security Domain Specific Language, Access Control, Obligations

1 Introduction

Integrating security mechanisms into applications is necessary to ensure data confidentiality, data integrity and users' privacy preservation. Security is a crosscutting concern affecting most parts of an application and, therefore, decoupling security requirements from the code implementing system functionalities is desirable to achieve code modularity and simplify the correct development of systems and their maintenance. Previous works primarily focus on the separate specification of *access control requirements* and integration of access control enforcement mechanisms into applications using either *Aspect Oriented Programming* (AOP) [9] [4–8] or using a model-based approach [11–13]. In the former approach, access control enforcement mechanisms are automatically *weaved* into the application at compilation time, whereas in the latter approach, the system and its access control requirements are abstractly specified using *models*, from which implementation code is generated. Neither of these approaches allows for the *runtime* updating of security requirements.

The dynamic nature of modern applications and their sophistication requires however more than just *static* access control, typically the only security requirement covered in existing approaches (see Section 6 for a detailed discussion of

current approaches and their features). In particular, security requirements typically reflect regulatory and internal mandates, which are naturally dynamic and could change with time. Also, many systems today have requirements that go beyond access control such as usage control [1], which extends traditional access control by enabling specification of obligations that users must fulfill before, while or after access, and privacy obligations [3, 2], which dictate duties and expectations on how users' personal data should be handled.

In this paper, we propose a Domain Specific modeling Language (DSL) and an architecture for securing Java-based business applications to address the aforementioned issues. The DSL supports the expression of fine-grained contextual authorization, obligation, sanction and reaction policies, thus covering the expression of many of the sophisticated security requirements of modern applications. Security policies specified using the DSL are enforced into target applications using an application-independent architecture, which follows the Policy Enforcement Point (PEP) / Policy Decision Point (PDP) paradigm. The proposed architecture enforces security requirements into target applications in a non-intrusive manner using Aspect-Oriented Programming (AOP) [9], enabling a clean separation between the application's functional and non-functional requirements. Furthermore, the architecture supports the update of security requirements at runtime.

The remainder of the paper is organized as follows. Section 2 describes *S@R* (for *Security@Runtime*), our DSL for dealing with the identified challenges and its enforcement architecture. Section 3 illustrates our approach by presenting a complete application example. Section 4 describes the implementation of our tool prototype. Section 5 shows performance results of the prototype for two real-life systems. Section 6 presents related work; and Section 7 concludes the paper and discusses future work.

2 The Security@Runtime Approach

At the center of our approach is the *Security@Runtime* (*S@R*) DSL, used for the *configuration* of the security enforcement mechanisms. This Section starts by presenting our PDP/PEP architecture for enforcing *S@R* security *configurations*, then describes each *S@R* component in detail.

2.1 Architecture Overview

Figure 1 shows the main components of the security enforcement architecture, namely the (i) PAP, (ii) PEP and (iii) PDP. The Policy Administration Point (PAP) allows the specification of a *S@R configuration* for the PEP and the PDP. The PEP monitors the application, using AOP [9] (in our case, AspectJ), and filters out information that is irrelevant to policy enforcement based on the configuration. Three events are monitored by the PEP: *instance creation*, *instance field updates* and *method calls*. If an event is relevant to policy management or

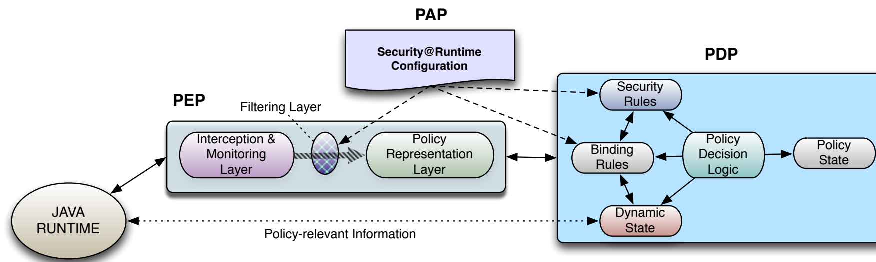

Fig. 1: Architecture Overview

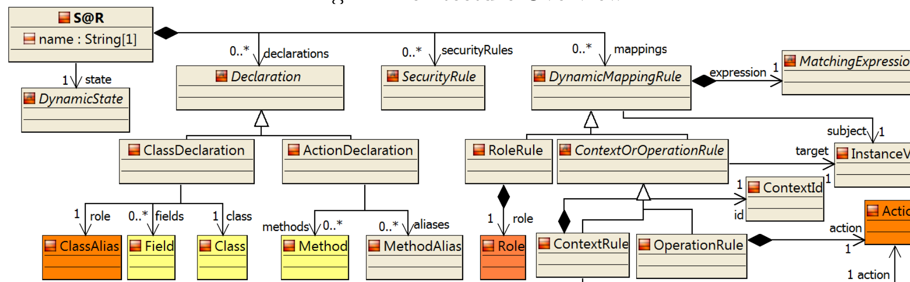

Fig. 2: The *S@R* Metamodel

enforcement, then the PEP notifies the PDP to update the *effective* security policy accordingly, e.g. activate a new obligation. When an event corresponds to a method call, the PDP computes an access decision. If access is granted, then execution proceeds; otherwise, different actions are possible: (1) a runtime security exception is raised with an appropriate message, (2) the system is stopped, or (3) the method execution is skipped. In our current prototype's implementation, a security exception is raised after access denial.

Figure 2 shows the four building blocks of *S@R*: (1) DynamicState, (2) Declarations, (3) SecurityRules and (4) DynamicMappingRules. The *Dynamic State* is a partial representation of the runtime state of the application and is automatically maintained and managed by the PDP. The other blocks define the *configuration* of the security mechanisms and are presented successively in the following: SecurityRules are introduced in Section 2.2; then Declarations and DynamicMappingRules are described in Section 2.3. A comprehensive example is given in Section 3 to illustrate the specification of security configurations.

2.2 Security Rules (SR)

A security policy is a set of security rules specifying what *subjects*, i.e. active entities in the system, are permitted, prohibited and obliged to do in the system. A security rule is contextual, i.e. it may apply only under certain conditions. A security rule includes the following elements:

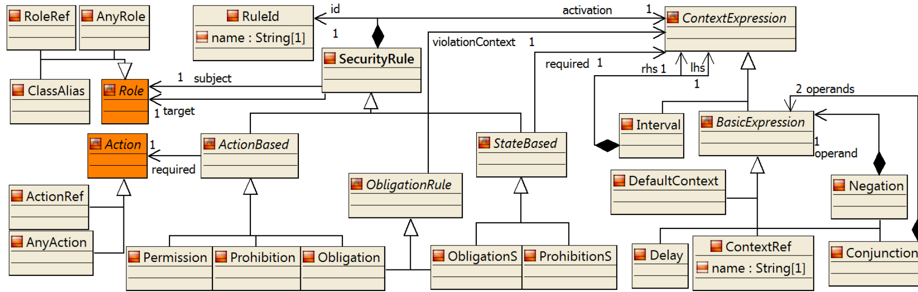

Fig. 3: Security Rules

- An *Identifier* of the security rule.
- A *Role* representing a set of system users or resources (we choose to abstract resources using roles, similarly to subjects/users, to minimize the number of basic policy entities).
- An *Action* representing an interaction between users and resources.
- A *Context* denoting a set of system state conditions.

Figure 3 shows the metamodel for security rules. Each `SecurityRule` has a unique identifier RuleId, a subject and a target role, and an activation context. The activation context defines the rule's applicability condition: a rule is active only if the evaluation of the boolean ContextExpression is true. A security rule is either a Permission, a Prohibition, or an Obligation and may be either ActionBased or StateBased. An ActionBased rule specifies that its subject role is permitted, prohibited or obliged to execute an `Action` on its `target` role. A `StateBased` rule specifies that its subject role is obliged or prohibited to maintain a required `Context`. An Obligation defines a violationContext that specifies under which conditions the obligation, after its activation, should be considered violated.

A `ContextExpression` is a boolean expression language constituted of BasicExpressions that can be composed with the usual boolean connectives. A BasicExpression is either a DefaultContext, a ContextRef, or a Delay. A DefaultContext is a special context that is always true. A Delay is a context that is true after the elapse of the time period specified in it. Finally, a ContextExpression can be an `Interval`, denoted [lhr, rhs]. An interval context holds since the left-hand side `lhs` holds until the right-hand side `rhs` holds.

2.3 Declarations & Dynamic Security Rules

Security policies are defined on the abstract level on roles, actions and contexts, allowing the use of the same policies in different systems. Declarations and DynamicMappingRules link elements of security rules to target applications by defining a mapping between these elements and the application classes, instances and method calls.

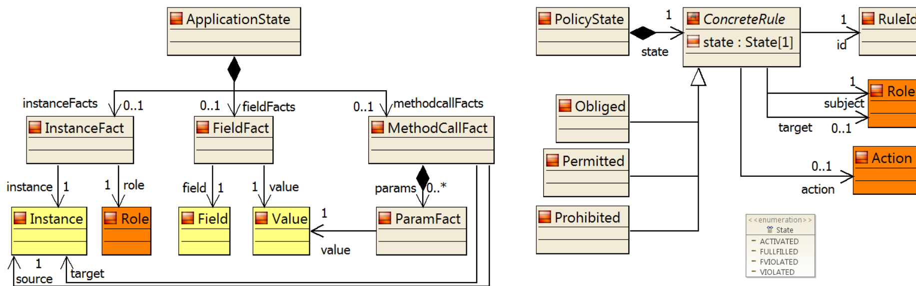

Fig. 4: Dynamic State

Declarations define aliases for the application classes and methods to simplify referring to them in security rules instead of using fully qualified names. A `Declaration` (cf. Fig. 2) is either a `ClassDeclaration` or an `ActionDeclaration`. A `ClassDeclaration` provides an alias for one application class and may optionally specify a list of the `Field`s of the class that are relevant to the enforcement of the security policy. This list improves system efficiency: only the updates of the relevant instance fields will be notified to the PDP, as opposed to notifying the PDP about changes of the value of every field of declared classes (see implementation details in Section 4). An `ActionDeclaration` provides an alias for one of the application methods or for every method in a sequence of (nested) methods. Declarations indicate which parts of the application are *relevant* to the security policy, therefore they are used by the PEP to filter information about changes in the application state that are being notified to the PDP.

Dynamic Mapping Rules describe the mapping between the *policy* entities (roles, actions and contexts) and the *application* entities (instances, fields, methods and their parameters). A RoleRule specifies which instances in the application are assigned to which role in the policy. An OperationRule specifies a correspondence between method calls and policy actions. A ContextRule defines a policy context as a condition on the application state: the context holds if its MatchingExpression holds on the application state. In the following, we describe the representation of the *application* and *policy* states (which compose together the PDP state) in *S@R*. Then, we explain how elements of the application state are mapped to elements of the policy state using dynamic mapping rules.

Application State At runtime, the state of an application consists of the set of active objects (or instances), the field instance values, and the stack of method calls. To correctly manage the security policy, e.g. activate contextual obligations, changes in the application state that are relevant to the enforcement of the policy need to be monitored. In our architecture, shown in Figure 1, these changes are monitored by the aspect layer within the PEP, which notifies the PDP when a relevant change is detected. Using the PEP's notifications, the PDP maintains a partial representation of the application state. Concretely, this state takes the form of a set of First-Order Logic (FOL) facts, allowing the specifica-

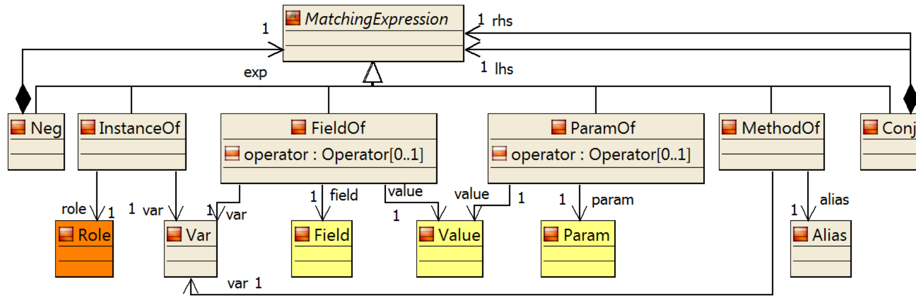

Fig. 5: Rule Condition Language

tion of security policies in FOL. This state is metamodelled in Figure 4 (left): an InstanceFact represents a class instance; a FieldFact represents an instance field; a MethodCallFact, together with ParamFacts, represents a method call. In *S@R*'s concrete syntax, these facts are written as follows:

- `instance_of(i,r)`: i is an instance of class r.
- `field_of(i,f,v)`: v is the value of the field f of the instance i.
- `call_of(c,m)`: c is a call of the method m.
- `param_of(c,p,v)`: v is the value of parameter p in the method call c.

Two special parameters, this and target, are systematically added to the normal parameter list of a method call to denote the calling and called instance respectively.

Policy State is managed by the PDP based on the application state. A PolicyState contains security rules that are *applicable*, or *effective* at a given time. Effective security rules are the set of ACTIVE permissions and the set of ACTIVE, FULFILLED or VIOLATED obligations (all values of the State enumeration in Fig. 4). In *S@R*'s concrete syntax, a policy state is represented using facts having one of the following forms:

- `permitted(r,s,a,o)`: rule r authorizes subject s to take action a on o.
- `prohibited(r,s,a,o)`: rule r prohibits s to take a on o.
- `obliged(r,s,a,o,t)`: rule r obliges s to take a on o.
- `obliged(r,s,c,o,t)`: rule r obliges s to maintain c on o.

Dynamic Mapping Rules If the MatchingExpression of a DynamicMappingRule holds on the application and policy states for some instantiation, i.e. a variable substitution making the MatchingExpression true, then the DynamicMappingRule holds for this instantiation. For example, a RoleRule of the form `role_of(I, role_name) <- instance_of(I,class_name)` assigns an instance x to the role `role_name` when `instance_of(x,class_name)` holds in the current state. Figure 5 shows the MatchingExpression's metamodel: it includes a matching expression for matching a class instance, a field value, a method call, or any logical

R1 A doctor can read the medical files of the patients he's treating [Access Control]

R2 A doctor should fill and submit a case evaluation report for each of his patients within one week [Non-persistent Obligation]

R3 A doctor should fill and submit a check-up report when he is assigned a patient within two days [Persistent Obligation]

R4 If the doctor does not submit the report, then he has to fill and submit a "violation of duty" report within one week [Sanctions]

R5 Meanwhile, his access to all files are suspended [Reactions]

R6 A doctor has to delete his patients' files that are private and not vital within two years [Privacy]

R7 A medical file has to be stored encrypted at most one minute after its creation by its creator [Confidentiality]

Table 1: Security Requirements for Hospital X

combination of these elements. Note that data operators can be used for the `FieldOf` and `ParamOf` expressions. For example, `field_of(D,age,`$\leq$`,18)` means that D is any instance whose value for the field age is less or equal than 18.

3 Example: The Medical System (MS)

Consider as an example the information system of Hospital X. The hospital needs to comply with some internal and regulatory mandates governing the activities of its personnel in order to protect the privacy of patients and guarantee the confidentiality and integrity of its information system. In Hospital X, a security policy should be specified to govern interactions between Doctors, Patients, Reports and Files, each of these roles being implemented into a simple class in the application. Table 1 describes this policy informally. The policy specifies one access control requirement (R1), a *non-persistent obligations* (R2), i.e. an obligation that may be cancelled after it is activated, *persistent obligations* (R3), i.e. an obligation that cannot be cancelled, *sanctions* (R4) and *reactions* (R5) to compensate the violation of obligations, and the obligations R6 and R7 to satisfy some privacy and confidentiality requirements respectively.

To enforce these requirements, the security officer of Hospital X should define a security configuration for the information system of Hospital X as follows: (i) *declare* the monitored aspects of the information system using Declarations; (ii) specify how application entities are *mapped* to policy entities using DynamicMappingRules; and (iii) define the SecurityRules formalizing the regulatory mandates.

Declarations are simply specified by defining aliases for classes and methods that need to be referenced in other parts of the configuration (dynamic mapping rules and security rules). The following example creates an alias "doctor" for the class *.Person.Doctor, declaring its field patients as the only policy relevant field (when the **attribute** clause is absent, all fields are considered relevant). Note that class aliases can then be used directly as role names (cf. Fig. 3). Similarly,

an alias "read" is created for the method `*.Server.readFile(String)`. Finally, the actions "readServer" and "readFile" are used as aliases for the method calls appearing in the sequence (denoted by `->`) of the method calls `readFileServer` followed by `readFile`.

class_alias	doctor	**method_id**	read
class	`*.Person.Doctor`	**method_sig**	`*.Server.readFile(String)`
attributes	`ArrayList<Patient>: patients`	**method_id**	readServer,readFile
		method_sig	`*.Server.readFileServer(String) ->*.File.readFile()`

Security Rules are defined according to the Security Requirements (cf. Tab. 1). Here, each requirement is expressed using one security rule. Rule `r3` has `doctor`, `submit` and report as its subject, action and `target` respectively. The rule's activation context is an Interval, thus r3 is activated when the context `assigned_doctor` becomes true, and is never cancelled (because the `Interval`'s rhs is `false`). Rule `r2` is non-persistent because its activation context is a BasicExpression: in this case, the obligation is activated when `assigned_doctor` holds, i.e. when a doctor is assigned to some patient, and it is cancelled when the activation context no longer holds, i.e. if this patient is no longer treated by this doctor.

```
permission(r1,doctor,read,file, assigned_doctor)
action_obl(r2,doctor,submit,report,assigned_doctor,delay<1:w>)
action_obl(r3,doctor,submit,report,[assigned_doctor,false],delay<2:d>)
action_obl(r4,doctor,submit,viol_report,violation_r2,delay<1:w>)
prohibition(r5,doctor,read,file,violation_r2)
action_obl(r6,doctor,delete,file,private & !vital, delay<2:y>)
state_obl(r7,doctor,file_encrypted,file,file_created,delay<1:m>)
```

Dynamic Mapping Rules are defined as follows. The first mapping rule is a RoleRule: it says that any instance of the class report whose field `type` has the value of `'violation report'` is assigned to the role of `viol_report`. The second mapping rule is an OperationRule: it says that if an instance (denoted here by `S`) calls the method `read_method` on a file (denoted by `F`), then the policy action "read" has subject `S` and target `F` (note the use of the special parameters `this` and target for the calling/callee instances). The fourth rule is a ContextRule that specifies that the context `private` holds for any file `F` whose field `classification` has the value of `''private''`. The ContextRule `violation_r2` is different, because it depends on the policy state: `violation_r2` holds for a doctor `D` for which `r2` is in the VIOLATED state.

```
role_of(R,viol_report)       ← instance_of(R,report) & field_of(R,type,'violation report')

operation(S,read,F)          ← call_of(read_method) & param_of(read_method,this,S) &
                               param_of(read_method,target,F)

operation(S,delete,F)        ← call_of(delete_method) & param_of(delete_method,this,S) &
                               param_of(delete_method,target,F)

hold(_,_,F,private)          ← instance_of(F,file) & field_of(F,classification,'private')

hold(D,_,_,violation_r2)     ← violated(r2,D,submit,report) field_of(F,type,'vital')

hold(D,_,_,file_created)     ← instance_of(F,file) & field_of(F,creator,D) & instance_of(D,doctor)

hold(D,_,P,assigned_doctor) ← instance_of(D,doctor) &
                               field_of(D,patients,contains,P) & instance_of(P,patient)

hold(_,_,F,file_encrypted)   ← instance_of(F,file) & field_of(F,encrypted,true)
```

One could also define the action `read` differently: an instance `S` `read`s `F` whenever `readServer` and `ReadFile` are called sequentially; then `S` is matched to the caller instance of the method aliased to `readServer` (precisely, `*.Server.readFile-Server(String)`, as declared before) and `F` to the target instance for the method aliased to `readFile` (declared previously as being `*.File.readFile()`).

```
operation(S,read,F) ← call_of(read_server) & call_of(read_file) &
                      param_of(read_server,this,S) & param_of(read_file,target,F)
```

4 Implementation

The architecture described in Fig. 1 is implemented using AspectJ for monitoring the target application, XSB Prolog [34] for computing access control decisions and policy management and Java/interProlog [35] for the communication between the Pep and the Pdp. EmfText [33] is used for parsing *S@R*'s concrete syntax and creating models.

4.1 Application Monitoring Layer

Each activity affecting the application state (instance creation, field update or method call) is monitored using an aspect. When an instance is created, if its class type is part of the Declarations within the *S@R* configuration, the aspect `RelevantClassObserver` detects the object using a pointcut of the form `execution(*.new (..))`, and passes it to the representation layer.

Field value update detection is more sophisticated: pointcuts of the form `set(* *)` only works for Java primitive types (strings, integers, booleans and so on). To monitor changes within the other supported data structures (like `ArrayList`s), a specific pointcut is defined to detect the execution of all methods altering the contents of the data structure (for example, `ArrayList.clear` or `ArrayList.set`). The pointcut below is specifically defined for the class

ArrayList. Currently, our implementation supports fields whose type is a primitive type and unidimensional structures (Vectors, HashSets and ArrayList). It is however straightforward to support more data structures.

Finally, method calls are intercepted using a pointcut of the form (call(public * *(..))). This aspect is of type *around*, i.e. the call is not executed until the aspect code is executed. This allows verification of the policy state at the PDP before allowing the execution of the method.

```
protected pointcut arrayListUpdate():
    (call(* java.util.ArrayList.add*(..)) ||
            call(* java.util.ArrayList.clear(..)) ||
            call(* java.util.ArrayList.remove*(..)) ||
            call(* java.util.ArrayList.retain*(..)) ||
            call(* java.util.ArrayList.set*(..))
            )
            && within(*.*) && !within(sr.*);
```

4.2 Policy Representation Layer

The policy representation layer consists of a recursive algorithm that processes Java objects and method calls in order to represent them using the facts described in Figure 4. For example, consider an instance of the declared class doctor with a field age of type Integer and another field patients of type ArrayList of Patient. Class Patient has a single field name of type String. Suppose there are two patients P_1 and P_2 in the treated patients list of doctor X. These objects would be represented as follows:

- instance_of(X,doctor) representing the instance,
- field_of(X,age,18) if the value of age of X is 18,
- field_of(X,patients,Y) where Y is an identifier of the ArrayList of patients,
- field_of(Y,e,P_1) where P_1 is an identifier of the first Patient, this fact denotes that P_1 is an element of the ArrayList of patients,
- field_of(Y,e,P_2) where P_2 is an identifier of the second Patient,
- field_of(P_1,name,'John') if the name of the first Patient is John,
- field_of(P_2,name,'Ben') if the name of the second Patient is Ben.

A method call is represented using a fact of the form call_of(method_id) where method_id is the method alias. Methods are processed similarly to class instances. However, param_of facts use the parameter position instead of attribute names: for example, a fact param_of(delete_method,1,X) means that X is the first parameter of the method call delete_method. Method calls have two additional parameters, namely the this and target parameters denoting the calling and called instances of the method call respectively. AspectJ enables an easy identification of these instances for intercepted method calls.

4.3 The Policy Decision Point (PDP)

The PDP is a policy engine implemented in Prolog. It computes a decision for access control requests and manages obligations in the policy according to notifications received from the PEP as follows: for an instance creation notification,

	Action	Conditions
1	**assert**(obliged(I,S,A,T,[Ca,Cd],Cv,active))	action_obligation(I,R,A,Rt,[Ca,Cd],Cv) instance_of(S,R) instance_of(T,Rt) hold(S,A,T,Ca)
2	**retract**(obliged(I,S,A,T,[Ca,Cd],Cv,active)) **assert**(obliged(I,S,A,T,[Ca,Cd],Cv,violated))	obliged(I,S,A,T,[Ca,Cd],Cv,active) hold(S,A,T,Cv)
3	**retract**(obliged(I,S,A,T,[Ca,Cd],Cv,active)) **assert**(obliged(I,S,A,T,[Ca,Cd],Cv,fulfilled))	obliged(I,S,A,T,[Ca,Cd],Cv,active) operation(S,A,T)
4	**retract**(obliged(I,S,A,T,[Ca,Cd],Cv,_))	obliged(I,S,A,T,[Ca,Cd],Cv,_) hold(S,A,T,Cd)
5	**retract**(obliged(I,S,Cf,T,[Ca,Cd],Cv,active)) **assert**(obliged(I,S,Cf,T,[Ca,Cd],Cv,fulfilled))	obliged(I,S,Cf,T,[Ca,Cd],Cv,active) hold(S,_,T,Cf)

Table 2: Obligation Management Rules

new Prolog facts representing the instance are inserted into the engine's knowledge base; for a field update notification, old facts specifying the field's value are retracted and replaced by new facts specifying the new value; for a method call notification, new facts corresponding to the call and its parameters' value (which works just as for instance fields) are inserted and the authorization policy is checked. An access control decision is then returned to the PEP and the facts corresponding to the call are retracted from the engine. After each notification, the PDP also updates the state of obligations.

Access Control After a method call is attempted, the call is interpreted by the PEP. An access is granted if it is permitted and not prohibited, i.e. the prohibitions are given an implicit priority over permissions for the resolution of potential conflicts between them. This access control policy evaluation strategy is specified using Prolog rules as follows:

```
allow_operation(S,A,T) ← operation(S,A,T), permitted(_,S,A,T), ¬ prohibited(_,S,A,T).

permitted(I,S,A,T)     ← permission(I,Rs,A,Rt,C), instance_of(S,Rs), action(A),
                         instance_of(T,Rt), hold(S,A,T,C).

prohibited(I,S,A,T)    ← prohibition(I,Rs,A,Rt,C), instance_of(S,Rs), action(A),
                         instance_of(T,Rt), hold(S,A,T,C).
```

Obligations The PDP manages the state of obligations by detecting their activation, cancellation, fulfillment and violation. We unify the representation of obligation activation contexts using Intervals: every activation context c is represented using an interval [c, !c].

An obligation is managed as follows: it is instantiated when its activation context holds. It is then fulfilled when its required action (context) is detected,

otherwise the obligation is violated if its violation context holds. An obligation is canceled at any time when its deactivation context holds. Table 2 shows the conditions for the detection of the activation, cancellation, fulfillment and violation of obligations and the update actions taken by the PDP when these conditions are detected. For example, Line 3 specifies that when the state of an obligation is active, its state is updated to violated when its violation context becomes true. State obligations (Line 5) are managed similarly, however their fulfillment is detected when their required context holds.

Support of Quaternary Predicates One advantage of using predicate logic to represent the application state is that it simplifies the definition of predicates for the expression of sophisticated state conditions. For example, the rules below specify the *contains* operator for data structures like *ArrayList*s, the less than or equals operator for numbers and the derivation of *violated* facts from obligation facts respectively.

```
field_of(Id,Name,<=,Val)       ← field_of(Id,Name,V), number(V), V =< Val.

field_of(Id,Name,includes,Val) ← field_of(Id,Name,V), field_of(V,e,Val).

violated(Id,S,A,O)             ← obliged(Id,S,A,O,[C_a,C_d],C_v,violated).
```

4.4 Policy Update

A runtime update of Security Rules is managed by simply adding (or retracting) the Security Rules from the policy engine (and their associated facts), i.e. if an obligation rule is removed then all its activated instances are also removed. When a new class is added (or removed) to the declarations part of the configuration, then the instances of this class are added to (or removed from) the knowledge base of the PDP as discussed in Section 4.2. Similarly, when an attribute is added (or removed), then the facts representing it are added (or retracted). A method call that is declared (undeclared) starts (ceases) to be monitored by the PEP. To handle these updates to declared classes and attributes, we need to keep a map of the class instances of the application at the PEP level.

5 Validation

To validate our approach, we considered two use case applications, namely our MS running example of Section 3, and an Auction Sales Management System (ASMS). The ASMS consists of 122 classes, 797 methods and about 11 kLoC. The ASMS implements an Auction system, where users can buy or sell products online, after joining an auction and placing bids. Users can also post, or read, comments from the Auction session. In the evaluation, we specifically targeted performance-related research questions:

- does the tool perform sufficiently well to be used in practice?
- what are the main factors impacting the tool performance?

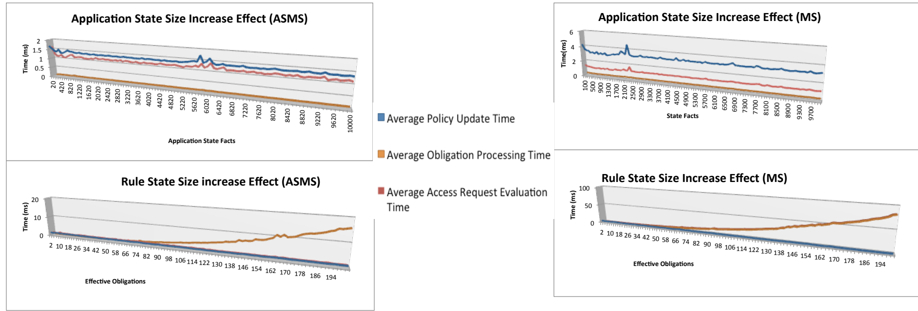

Fig. 6: Performance Results

To answer these questions, we defined a security policy for each example application and ran a scenario covering the different policy management operations, i.e. obligation activation, violation, access control, etc. We evaluated three factors: the time necessary to perform policy management operations, to evaluate an access request, and to update the (obligation) policy state, for different sizes of the application and policy states (number of activated obligations).

Figure 6 shows the results: policy management operations and access request evaluations are performed in a few milliseconds, and represent an almost constant overhead. On the other hand, obligation processing time increases with the number of (activated) obligations in the system: the activated obligations' contexts have to be verified individually to check whether they are canceled, violated or fulfilled, after each state update. We are currently investigating ways to improve this processing time.

6 Discussion and Related Work

Since the seminal contributions of Lodderstedt and Basin with SecureUml [10], and Jürgens with UmlSec [20] back in 2002, model-based development of secure systems has been an active research area. In Table 3, we compare several contributions with respect to several dimensions: system (DM) and security modeling (SM); contextual security (CS); security concerns (SC) (i.e. what kind of security properties can be expressed); and code generation (CG).

Domain & Security Modeling. Uml is the most common way to define the target application domain, as shown in the first column of Table 3. UML-based approaches annotate the business Uml model with their security requirements. Conceptually, our approach is different since we introduce the *S@R* DSL to specify security requirements and their mapping to target systems. One advantage of our approach is that it cleanly separates security from system specification making a true separation of concerns, as opposed to the use of OCL constraints to specify contextual policies when Uml profiles are used. Note that Xacml does not assume any specific domain modeling language and, therefore, it does

	DM	SM	CS	SC	CG
SecureUML[19]	UML	Profiles	OCL	AC	✓
UMLsec[20]	UML	Profiles	×	C,IF,AC	×
secureMDD[21]	UML	Profiles	×	AC	✓
ModelSec[31]	UML	SecML	○	MC	✓
SECTET[32]	SE-UML	Profiles	SE-PL	AC	○
XACML[36]	×	XACML	XACML	AC,OB	×
S@R	UML	S@R	S@R	AC,OB	✓

AC: Access Control, C: Confidentiality,
IF: Information Flow, OB: Obligations, MC: Multiple Concerns

DM: System Modeling, **SM**: Security Modeling, **CL**: Contextual Security
SC: Security Concerns, **CG**: Code Generation

Table 3: Comparison of *S@R* with Existing MDS approaches

not provide means to systematically integrate security mechanisms for enforcing Xacml policies in targeted systems.

Expressivity of Security Languages is a major challenge since it is necessary to cover the specification of many practical security concerns. Many systems today have security requirements that go beyond access control, as recognized by Basin, Clavel and Egea in [13] where they pointed out the need to add support for obligations. To the best of our knowledge, *S@R* is the first Dsl that supports management and enforcement of both authorizations and obligations. The specification of obligations is supported in Xacml [36]. However, obligations in Xacml are syntactic elements without formal semantics. Furthermore, Xacml does not provide management and enforcement support for obligations.

Violation Monitoring & Policy Runtime Updating. Runtime policy updating and security rule violation monitoring are not, to the best of our knowledge, present in any of the current approaches.

Security Infrastructure. Despite the use of automated transformations in Mds, it is still difficult to enable full code generation from high-level requirements. SecureUml [19] supports the generation of secure systems for two target architectures (Enterprise JavaBeans and Microsoft DotNet), but the generating mechanism relies on pre-existing security mechanisms. In Sectet, the information relevant for authorizations are specified using the tool's language, and transformed into Xacml specifications. In [31], security and business are composed into a model from which Java code is generated. Our approach generates Prolog code from security policies defined within *S@R*, whereas the rest (i.e., aspects monitoring and policy interpretation) is application-independent.

7 Conclusion

This paper proposes an approach for securing Java-based business applications using security policies. Our approach cleanly separates between security and business concerns, allowing the separate development and specification of business and security aspects. It also enables the specification of fine-grained contextual permissions and obligations and supports their management, enforcement and their update at runtime. We have demonstrated the expressiveness of our security policy language using a comprehensive example and validated our approach by using it to secure two different systems. We have identified some limitations of our framework, namely its scalability when the number of activated obligations in the system increases. Therefore, we plan to study optimization techniques to improve the tool's performance. We also intend to provide support for more advanced usage controls and more Java data structures.

References

1. Sandhu, R., Park, J.: The UCON ABC usage control model. In: ACM Transactions on Information and System Security (TISSEC) **7**(1) (2004) 128–174
2. Ni, Q., Bertino, E., Lobo, J.: An obligation model bridging access control policies and privacy policies. In: SACMAT'08 (2008) 133
3. Mont, M.: Dealing with privacy obligations in enterprises. In: ISSE 2004 Securing Electronic Business Processes (2004) 28–30
4. Erlingsson, U., Schneider, F.B.: SASI enforcement of security policies. In: NSPW (2000) 87–95
5. Bauer, L., Ligatti, J., Walker, D.: Composing security policies with polymer. In: ACM SIGPLAN Notices **40**(6) (June 2005) 305
6. de Oliveira, A.S., Wang, E.K., Kirchner, C., Kirchner, H.: Weaving rewrite-based access control policies. In: FMSE (2007) 71–80
7. Hamlen, K.W., Jones, M.: Aspect-oriented in-lined reference monitors. In: PLAS (2008) 11
8. Hussein, S., Meredith, P., Rolu, G.: Security-policy monitoring and enforcement with JavaMOP. In: PLAS (2012) 1–11
9. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-m.: J. Irwin, and C. Lopes, Aspect-Oriented Programming. In: ECOOP (1997)
10. Lodderstedt, T., Basin, D.: SecureUML : A UML-Based Modeling Language for Model-Driven Security. In: Proceedings of the 5th International Conference on The Unified Modeling Language (2002) 426–441
11. Mouelhi, T., Fleurey, F., Baudry, B., Traon, Y.L.: A model-based framework for security policy specification, deployment and testing. In: Model Driven Engineering Languages and Systems (1) (2008) 537–552
12. Morin, B., Mouelhi, T., Fleurey, F., Le Traon, Y., Barais, O., Jézéquel, J.M.: Security-driven model-based dynamic adaptation. In: ASE '10 (2010)
13. Basin, D., Clavel, M., Egea, M.: A decade of model-driven security. In: SACMAT'11, (2011) 1–10
14. Basin, D., Clavel, M., Egea, M.: A Metamodel-Based Approach for Analyzing Security-Design Models. In: MODELS (2007) 420–435

15. May, M., Gunter, C., Lee, I.: Privacy APIs: Access control techniques to analyze and verify legal privacy policies. In: 19th IEEE Computer Security Foundations Workshop (CSFW'06). (2006)
16. Barth, a., Datta, a., Mitchell, J., Nissenbaum, H.: Privacy and contextual integrity: framework and applications. In: IEEE Symposium on Security and Privacy (2006)
17. Barth, A., Mitchell, J., Datta, A., Sundaram, S.: Privacy and Utility in Business Processes. In: 20th IEEE Computer Security Foundations Symposium (2007) 279–294
18. Lam, P., Mitchell, J.: A formalization of HIPAA for a medical messaging system. In: Trust, Privacy and Security in Digital Business (2009) 73–85
19. Basin, D., Doser, J., Lodderstedt, T.: Model driven security: From UML models to access control infrastructures. In: ACM Transactions on Software Engineering and Methodology (TOSEM) **15**(1) (2006) 39–91
20. Jürjens, J.: UMLsec: Extending UML for secure systems development. In: UML '02 Proceedings of the 5th International Conference on The Unified Modeling Language (2002) 412–425
21. Moebius, N., Stenzel, K., Grandy, H., Reif, W.: SecureMDD: a model-driven development method for secure smart card applications. In: International Conference on Availability, Reliability and Security, 2009. ARES '09. (March 2009) 841–846
22. Cuppens, F., Miège, A.: Modelling contexts in the Or-BAC model. In: ACSAC (2003) 416–425
23. Elrakaiby, Y., Cuppens, F., Cuppens-Boulahia, N.: Formal enforcement and management of obligation policies. In: Data & Knowledge Engineering (2011) 1–21
24. Jajodia, S., Samarati, P., Subrahmanian, V.: A logical language for expressing authorizations. Proceedings. 1997 IEEE Symposium on Security and Privacy (1997) 31–42
25. Kagal, L., Finin, T.: A policy language for a pervasive computing environment. In: IEEE 4th International Workshop on Policies for Distributed Systems and Networks (2003) 63–74
26. Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A.: The Java Language Specification. Addison-Wesley Longman (2013)
27. Ben-Ghorbel-Talbi, M., Cuppens, F., Cuppens-Boulahia, N., Bouhoula, A.: A delegation model for extended RBAC. In: International Journal of Information Security **9**(3) (2010) 209–236
28. Cuppens, F., Cuppens-Boulahia, N., Ghorbel, M.B.: High Level Conflict Management Strategies in Advanced Access Control Models. In: Electronic Notes in Theoretical Computer Science **186** (July 2007) 3–26
29. Autrel, F., Cuppens, F., Cuppens-Boulahia, N., Coma, C.: Motorbac 2: a security policy tool. In: 3rd Conference on Security in Network Architectures and Information Systems (SAR-SSI 2008), Loctudy, France. (2008) 273–288
30. Kateb, D.E., Mouelhi, T., Traon, Y.L., Hwang, J., Xie, T.: Refactoring access control policies for performance improvement. In: ICPE. (2012) 323–334
31. Molina, F., Toval, A., Sánchez, O., Garca-Molina, J.: ModelSec: A Generative Architecture for Model-Driven Security. In: Journal of Universal Computer Science **15**(15) (2009) 2957–2980
32. Breu, R., Popp, G., Alam, M.: Model based development of access policies. In: International Journal on Software Tools for Technology Transfer **9**(5-6) (2007) 457–470
33. emfText. http://www.emftext.org/index.php/EMFText
34. XSB Porlog. http://xsb.sourceforge.net

35. interProlog. http://www.declarativa.com/interprolog
36. Extensible Access Control Markup Language (XACML) version 3.0. http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-cs-01-en.pdf