Towards simplified specification of policies in different domains

Kris Verlaenen, Bart De Win, Wouter Joosen
DistriNet, Department of Computer Science
Katholieke Universiteit Leuven
Celestijnenlaan 200A, 3001 Leuven, Belgium
Email: \{kris.verlaenen, bart.dewin, wouter.joosen\}@cs.kuleuven.be

Abstract—Policies can be used for the configuration and management of existing services, possibly at runtime, without having to change their implementation. They declaratively specify the behavior of these services in certain circumstances and can be used in areas like security or quality of service. In order to make the policy specification as effective as possible, specific policy languages have been defined for many different areas. We present a policy ontology where specific policy languages are created as simple extensions of a generic policy model. In addition, by explicitly representing (and mapping) the domain the policies are applied to, domain concepts can be used directly inside policies. As a result, policies can be more targeted towards different areas and domains.

I. INTRODUCTION

Policies are increasingly used in modern information platforms for configuring complex systems and controlling the interaction between different actors and entities. Policies allow changing the behavior of a system without changing its implementation, creating adaptable systems whose behavior can be altered dynamically.

Security policies for access control are probably the best known example, declaratively specifying who is allowed to access which services. But policies are used in other areas as well, like quality of service (QoS) management and user preferences. There is a need for a common (general-purpose) policy language that allows us to configure services in these different areas using the same core language.

But, in order to make policy specification as effective as possible, policies should be targeted to specific areas and application domains. Domain-specific languages allow policy authors to use higher-level concepts closely related to the problem at hand and to the domain policies are applied to, making them easier to understand and simplifying their specification.

There is an apparent tension between using a general-purpose (policy) language (GPL), which covers a broad domain but is usually harder to specify, and domain-specific policy languages (DSLs), which are easier to specify but can only be used in a limited setting and are harder to enforce. A solution in this context should address the following requirements:

- Common policy language: There is a need for a common policy language that can be used for expressing policies in different areas (like security, QoS, etc.) and in various application domains (like telecom, healthcare, etc.).
- Ease of use: Different types of users, ranging from system administrators to end users, must be capable of specifying their preferences using these policies. Therefore, a policy solution should simplify policy specification by offering (1) concepts at a higher level of abstraction, closely related to the problem domain and (2) user-friendly formats for specifying policies.
- Extensibility: It should be straightforward to extend existing policy languages with new concepts, add new domain concepts, create new policy languages, add new policy representation formats, etc.

Our contribution is a step towards bridging this gap between a general-purpose policy language and domain-specific policy languages. In this paper, we present a policy ontology that contains a generic policy model, allowing us to express different types of policies using the same core language. Specific policy languages, like for example a security and a QoS policy language, are defined as simple extensions of this core language, introducing concepts related to that specific area. In addition, by explicitly representing the domain model policies are applied to, and mapping it to concepts defined in our policy ontology, policy authors can directly use high-level domain concepts in their policies. As a result, all our policies are based on the same core language but are nevertheless targeted to a particular area and domain.

We further simplify policy specification by using specialized, user-friendly policy editors, defined on top of this policy ontology. Policy editors can be generic, allowing the specification of different types of policies, or targeted to a specific area and/or domain. As a result, both system administrators and non-experts can use policies to define their preferences, expanding the area where policies can be applied.

The main focus of this paper is on representing policies in different areas and application domains using the same core language. We did however implement a prototype capable of enforcing these policies as well. The impact of our approach on the enforcement architecture is therefore discussed when evaluating our prototype.

This paper is structured as follows: In section II we present our case study, a telecom platform, where policies are used in five different areas. Section III presents the policy ontology...
and explains how policies can be targeted to a particular area and domain by explicitly representing the application domain concepts and extending the core policy model. Section IV describes how policy editors can ease the specification of policies even further by using user-friendly (graphical) notations. In section V we evaluate our approach and related work is discussed in section VI. We state our conclusions in section VII.

II. POLICIES IN TELECOM

To show how different types of policies could be used to configure services in a particular domain, we apply policies in the context of a telecom Service Enabling Platform (SEP) [1]. The telecom market is confronted with an increasing demand for advanced, integrated and highly-customizable telecom services, like triple play services integrating the internet, video and telephony. In order to respond to these new requirements, telecom providers have introduced the concept of a SEP, a platform that allows the creation, configuration and execution of new telecom services, by assembling existing functionality in the form of capabilities and/or other services. It is a service-oriented architecture (SOA) that uses an Enterprise Service Bus (ESB) to manage the communication between all actors and to orchestrate the composed services [1].

To allow maximal reuse of functionality, most capabilities and services and the platform itself can be configured to behave differently based on the specific circumstances the capability or service is used in. For example, based on the identity of the requester of some service or the time of day, the service might be required to behave differently. Manually managing the configuration of these services is a complex and time-consuming operation, because different services offer different configuration mechanisms and configurations are spread out across the entire platform.

The best known examples of policies are probably authorisation policies, which describe whether a subject (= the requester) is allowed to perform a certain operation on a resource. But policies in a broader sense can be used in other areas as well. In this paper we describe five types of policies we were confronted with when investigating the configuration of the SEP. Each type is defined in more detail below and example policies are shown for each policy type. Note that our solution can easily be extended to support further types of policies if necessary.

- Security: Can be used to specify whether a subject is allowed to perform an operation on a resource. Other security-related issues like confidentiality and verification can be supported as well.
  - Allow service provider A access to method M on service X between 9am and 5pm.
  - All invocations to the banking service should be secure.
- SLA/SLO: Service level agreements (SLAs) and service level objectives (SLOs) specify QoS preferences (like guaranteed or preferred number of requests per second or number of simultaneous sessions) between end users, service providers and/or the network operator.
  - Guarantee that service provider A can send 100 requests per second.
  - Restrict service X to receive 1000 events per minute.
- User privacy: Allows a user to specify who is allowed to see or even change his/her personal information. This information is not only limited to (quasi-)static information like address and phone number, but also dynamic information like location and presence status.
  - Do not allow end user U to see my presence status.
  - Ask me for confirmation before sending out my location status.
- User preferences: Allow end users to specify their preferences when using services. A typical example is redirection (of phone calls, SMS or mail), where the end user should specify under which circumstances redirection should occur and where to.
  - During office hours, redirect home phone to office.
  - When out of office, receive e-mail as SMS/MMS.
- Operational policies: To make services more evolvable, a service provider could introduce business logic inside a configurable service by specifying it using policies. Policies are easier to adapt and manage, thus allowing service providers to change the business logic without having to re-implement the service.
  - If recharge amount $\geq 50EUR$, activate promotion X.

A policy language for the configuration of (services in) the SEP should allow end users, service providers and system administrators to express their preferences in each of these different areas.

III. POLICY ONTOLOGY

In this section, we present our policy ontology, which is used to represent different types of policies based on the same core language, but nevertheless allows us to target policies to a specific area and domain as well. An ontology is a definition of the concepts and the relationships that can exist between these concepts. Figure 1 gives an high-level overview of our approach. Our ontology can be divided into three main parts:

- The **generic policy model** defines the common structure of policies and how they can be combined into policy sets. It contains generic concepts that are common to different types of policy languages.
- In order to target policies to a specific area, like security or QoS, **specific policy language extensions** are created on top of the generic policy model. They extend this model by introducing concepts that are important in that specific area. For example, the security policy language defines security related properties like roles and signatures.
- By explicitly representing the **domain model**, policies can be targeted to a specific domain as well. If the domain model concepts, in our case the telecom domain, are mapped to concepts in the generic policy models,
A. Generic policy model

Although policies can be used in a lot of different areas, some core concepts are reused across all these types of policies and policies themselves share a similar structure. We believe that a significant part of the policy language can be reused for all policy types, while extension are needed to introduce concepts that are application-specific. The generic policy model describes the concepts common to all types of policies and can be divided into three parts:

- the information that can be used inside policies
- the structure of the policies
- how policies can be combined into policy sets

All models are represented as UML diagrams. The semantics is only described informally in the text. However, since the semantics of generic concepts might be dependent on the application domain and the environment policies are applied to, it can only be fully defined in a specific application context. For example, several policy languages might define the semantics of an event differently (e.g. instantaneous vs. durable).

The generic models are based on a combination of concepts found in existing policy languages (XACML [14], PCIM [16] and Ponder [13]) and the rules format of ILOG JRules [6], a general-purpose rule language. The general structure of policies and policy sets is similar to the Policy Core Information Model (PCIM) [16], but extends this model by specifying in more detail how policy conditions are structured. By increasing the level of detail in the generic policy model, specific policy language extensions can be kept simple. We also support advanced action and policy combination, simplifying the specification of complex policies. Each of these models are now described in more detail.

Information

Before describing our policy language, we first define the information that can be referred to when defining policies. Policies are used to configure the behavior of services in specific circumstances. This usually implies describing how a service should respond to specific events. An event might for example be the invocation of an operation in the system or a notification of a state change. This generic information model defines the structure of an invocation event, as shown in Figure 2. An event has associated attributes describing the properties of that event. Note that this model does not define the full semantics of an event, as its meaning might be dependent on the context it is used in (e.g. instantaneous vs. durable). The semantics should be extended when applying these concepts it in a specific environment.

If an event describes an invocation in the system, three subsets of properties can be distinguished: those related to either the subject (invoker), resource (target) and operation of the invocation.

- The subject represents the entity responsible for the event, e.g. the requester of the operation. It can contain
attributes like its name and other identification information.
- The resource is the target of the invocation. Resources could be categorized into different types.
- The operation that is invoked has attributes that represent the parameters and possibly the result of the invocation.
- Information that is relevant during policy evaluation but not directly linked to the invocation can be defined as environment information. Typical examples are the current time and date.
- The event itself can have additional attributes as well, like for example a priority indication.

The attributes defined here are generic, meaning that they might be useful in different types of policies. Specific policy languages can extend this model by defining additional attributes, as explained in section III-B. For example, a security policy language might define additional security-related attributes that can then be used when specifying security policies as well.

Policy

Policies are used to configure the behavior of services. Policies are described in a declarative way, specifying what should happen in certain well-defined circumstances, but not how it should be done. Policies can also be useful at different levels of detail. For example, service level objectives (SLOs) are typically high-level, implementation-independent and only refer to well-known domain model concepts. Operatioal policies configuring one specific service usually are lower-level and refer to concepts specific for that service.

All types of policies can be described as condition-action rules, where the action describes what should happen, and the conditions describe under which circumstances. For example, if user U specifies he wants to redirect his home phone to the office during office hours, two conditions are used to specify that this policy is only applicable when (1) someone tries to phone this user at home and (2) during office hours. The action specifies that the phone call should be redirected to the office.

Figure 3 gives an overview of the policy model. A policy has zero or more conditions. Conditions are boolean expressions that express when the policy is applicable. Expressions can be quite complex and support the following features:

- Expressions can refer to (input) variables that represent information relevant to the policy. Their value differs at runtime and their value can be evaluated. Each specific policy language must define their set of allowed variables. For example, policies describing how to react to specific invocations can use a predefined invocation variable that refers to the actual invocation (including the subject, operation and resource of that invocation).
- The policy author can specify an explicit value inside an expression. This could for example be a number, a certain date, etc. This is often used to specify values to which the value of some property of a variable should be compared.
- Arithmetic expressions allow the use of (unary or binary) arithmetic operators and brackets.
- Functions can be used in expressions. Examples are getters that retrieve an attribute of an element, comparison functions, helper methods that can be used to deduce or manipulate information, etc. A function can have a target, which is the object on which the function is invoked and parameters. A parameter has a name and the value of the parameter is specified by using another expression. This means that a parameter could either be an explicit value, (the result of) a function, a variable or an arithmetic expression. Functions can be nested, meaning that the result of one method call can be used as target of another method. Functions are defined using a FunctionDefinition, which specifies the name, parameters and result type of the function. A lot of predefined functions can be used to compare and manipulate basic data types.

The action of a policy specifies what should happen when the conditions of the policy are satisfied. A policy can only have one action. There are two types of actions: elementary and composite actions:

- An elementary action represents one decision that should be enforced when the policy is applied. Since the set of elementary actions that are allowed is different for each of the specific policy languages, the generic policy
model does not define any actions itself, but specifies that each specific policy language must define their own set of allowed actions. Section III-B explains how to this can be achieved and what actions are defined for each of the policy languages. Elementary actions can use work items to represent atomic units of work that could possibly be executed when enforcing the action. These work items are policy-independent and are an abstract representation of some service that could be used when enforcing an action. Typical examples are logging some message, sending an email, etc. They declaratively specify what should be done and they are used to abstract away from implementation details of how it should be done. Work items can use parameters to customize their behavior.

- A composite action combines two or more actions, specifying the order in which they should be executed, whether they should be executed in sequence or parallel, what to do if one of the actions cannot be executed, etc.

Policy sets

Multiple policies can be combined using policy sets. Because different actors can define policies, policy sets must take into account that more than one policy in a policy set may be applicable simultaneously, and that the actions suggested by these policies may be conflicting. Therefore, a policy set must specify how the individual policies should be combined to reach a single result.

Figure 4 gives an overview of the policy set model. Policies can be combined to form policy sets. A policy set is a combination of policy elements, where a policy element is either a policy or another policy set. Each policy set uses a policy combination algorithm that describes how the (possibly conflicting) policies should be combined.

A policy combination algorithm’s primary goal is to prevent or solve possible conflicts within one policy set, which can be achieved by specifying which of the different policies has precedence. However, more advanced algorithms can also be used to describe the order in which policies should be evaluated, possibly allowing parallelism, etc. Examples of possible policy combination algorithms are priorities (all policies are given a priority number and the policy with the highest priority is executed first), recency (policies that were defined or activated more recently have precedence), etc. An example of a more advanced policy combination algorithm is a policy flow, where the order in which policies should be evaluated is specified as a flow diagram. New types of policy combination algorithms can easily be added. For example, section III-D shows how custom combination algorithms can be used to combine the results of several security policies.

B. Specific policy languages

Our generic policy model already contains most of the ingredients for specifying policies, but does not specify what concepts can be used in each area, like security or QoS. To accomplish this, specific policy languages can be constructed, extending the core language with concepts targeted to one specific area. The different policy languages can be seen as domain-specific languages (DSLs) enabling policy authors to specify their preferences in one specific area using constructs closely related to the problem at hand. By using high-level constructs and limiting the concepts used to only those that are relevant to that specific area, even non-programmers can use these specific policy languages themselves.

Because the generic policy model already defines the structure of the policies (and policy sets), specific policy language extensions only have to define additional concepts related to that specific area. For example, security policies typically refer to security-related properties of the subject and the resource, like credentials, electronic signatures, etc. This can be achieved by extending the generic policy model in a few, predefined ways:

- The information model describing the information that can be referred to inside policies can be extended with additional elements and/or attributes of existing concepts like subject, operation, resource and environment (e.g. using subclassing).
- Conditions in the generic policy model can be extended to include new variables, functions, etc. by creating the appropriate Variable- and FunctionDefinitions.
- Each specific policy language should define the (elementary) actions that can be used as part of the policy. This extension is usually necessary since the set of allowed actions is almost always area-specific. An action is defined by its unique id and its parameters.
- Policy sets can be extended with new policy combination algorithms.

Our approach tries to minimize the effort required for creating a specific policy language. Because the core policy model defines the policy structure in detail, the extensions required are limited. We present two sample specific policy languages – a security and a QoS policy language – in section III-D.

C. Mapping the domain model

The policy models presented so far are not directly targeted to a specific application domain. Each of the policy languages can be used in different application domains, like telecom or healthcare. However, policy specification and management is much easier if domain-specific concepts can be used directly inside policies. We allow targeting policies to a particular domain by explicitly representing the high-level concepts in
the application domain and by mapping these to concepts in our policy ontology.

Separating the domain model specification from the policy model itself allows the reuse of the policy languages in different application domains. For most application domains, a domain model already exists in some form. In this case, only the selection of the concepts relevant during policy specification and their mapping is required. Using a standardized domain model allows the use of one policy language or even policies themselves across different platforms and implementations.

For example, for applying policies in the context of a telecom Service Enabling Platform (SEP), a specification of the relevant concepts in the SEP is required, combined with a mapping of these concepts to concepts defined in the policy model. Figure 5 shows a simplified version of the telecom domain model used in the context of the SEP. A telecom provider is the owner of the SEP, offering a wide spectrum of services to its clients, called the end users. Services can be provided by the telecom provider itself, but external 3rd party service providers could also plug in custom services.

If we explicitly map these concepts to related policy concepts, we can start using these concepts directly inside policies, making them easier to understand. This mapping should relate concepts in the domain model with corresponding elements in the policy model. For example, the information model in section III-A defines that an invocation event is linked to its subject, operation and resource. A subject in this situation can either be an end user or a service provider whose service is reusing another service. A service corresponds to the resource being invoked and the methods that can be invoked on a service are the operations of that service. These mappings are currently created by linking related concepts, as conceptually shown in Figure 5, but these mappings could be extended to take the properties of these concepts into account as well. A merged model can be generated by combining the properties of related concepts [4]. Similarly, the same policy languages can easily be targeted to other domains as well, for example in a healthcare context by defining and mapping concepts like patient, physician, medical record, etc.

This mapping of domain concepts to our policy ontology also enables basic domain-level validation. For example, whenever the policy author refers to a subject, we know this subject must be a valid end user or service provider. Similarly, all information about which operations are allowed on a specific service can be looked up by retrieving all methods provided by that service.

D. Specific telecom policy languages

In section II we specified how policies could be used in the context of a SEP to configure services, and we identified five different types of policies, each one used for the configuration of one specific aspect of the platform (like security or QoS). A specific policy language can be defined for each of these areas, by extending the generic policy model, as explained in section III-B. We now describe how the security policy language and the SLA/SLO policy language can be defined, providing examples of how to extend the generic policy model to allow the specification of security and QoS policies respectively.

Security policies

Security policies are used to define the behavior of services related to security. The best known security policies are authorization statements that describe whether a subject is allowed to perform an operation on a resource, but other features like confidentiality could be included too.

The security policy language allows the definition of positive (allow) and negative (deny) authorization statements, in combination with obligations which specify additional work items (e.g. logging or sending a notification) that must be executed before granting or denying access.

The model that describes the security policy language extends the generic policy model, meaning that all generic concepts (and their attributes), as defined in the policy model and telecom domain model, can be used during policy specification. The following security-related attributes and concept refinements are added as well:

- Information: Additional attributes are defined for subject, operation, resource and environment:
  - A new subject attribute roles defines the role(s) a subject has. This allows role based access control (RBAC). By linking roles to subject, the scalability and manageability of the solution is greatly improved.
  - A new operation attribute can be used to specify that a certain operation is a read or write operation. This allows giving users read and/or write access to certain resources.
  - Resources are extended with an attribute signer that specifies the authority that verified the implementation of the service. This would allow subjects to specify that they only want to execute services that were signed by a trusted verifier.

- Policy actions:
  - An allow and deny action are used to specify positive and negative authorization statements respectively. Both can be combined with obligations (e.g. logging...
or sending a notification), which are specified as work items that must be executed before granting or denying access.

- Actions like secure or make verifiable offer support for confidentiality and non-repudiation.
- A filter action allows the request but filters the result before returning it. Different filters can be specified (and parameterized) to support different filtering techniques.

- Two policy combination algorithms specific for security policies are defined as well:
  - Deny-overrides: If one of the policies in the policy set denies access, access to the resource is denied, even if other policies in the policy set allow access.
  - Allow-overrides: If one of the policies in the policy set allows access, access to the resource is allowed, even if other policies in the policy set deny access.

**SLA/SLO policies**

A Service Level Agreement (SLA) is a contract between a provider and a consumer that guarantees certain QoS at a pre-determined cost. Operationally, guarantees in a SLA are defined as a set of Service Level Objectives (SLO). A SLO is a combination of component measurements to which constraints are applied. SLAs also contain a description of the associated penalties if the targets are not consistently achieved.

SLOs describe the requested QoS in terms of certain key performance indicators that can be measured. Examples of such properties are:

- Performance: requests or sessions per time unit
- Availability: response time, transaction volume
- Security: number of failed logins or violations
- Business metrics: number of transactions per time unit

SLAs and SLOs are used to limit, guarantee or request a certain level of QoS. This can be either resource- or user-centered. To allow the specification of these policies, the SLA/SLO policy language extends the generic policy model with the following actions:

- Performance:
  - Restrict / guarantee the number of requests or sessions per time unit.
  - Route certain requests to specific locations (like high-performance services).
  - Prioritize certain requests.

- Availability:
  - Guarantee a response time or transaction volume.

- Security:
  - Restrict the number of allowed failed logins or violations per time unit.

- Business Metrics:
  - Restrict the number of transactions per time unit.
  - Guarantee a number of transactions per time unit.

In theory, all these extensions could be defined in a single, implementation-independent mapping. In our prototype (see section V), however, this is currently achieved using a combination of three different specifications:

- Additional attributes for information classes must be defined by subclassing the information classes in Java.
- Actions are defined in an XML Schema file that describes the syntax of a specific policy language. Listing 2 in the next section gives an example of such a file.
- Functions, variables and combination algorithms are defined by instantiating and registering a corresponding Definition object.

### IV. POLICY EDITORS

Service providers and end users can use specific policy languages to configure the behavior of certain services. A policy editor tool assists the user in this process and stores the created policies in a policy repository. A policy enforcement component can then later retrieve these policies from the policy repository.

The next section explains how XML is used as the base language for expressing policies and how an XML Schema Definition (XSD) is used to define the concrete syntax. But specifying policies in XML directly is not straightforward. We believe that policy authors should use more graphical, user-friendly tools to specify their preferences, which are then automatically transformed to the corresponding XML policies.

**Specifying policies using XML**

XML was chosen as base language for expressing our policies due to the ease with which its syntax and semantics can be extended, because it allows the definition of a system-independent and vendor-independent language and a large set of tools to parse, generate, transform, validate and manipulate XML already exists.

Listing 1 shows an example security policy that states that everyone is allowed to access ServiceX. This policy contains one condition, which checks whether the resource id (lines 5-9) is equal to (line 3) "ServiceX" (lines 12-14). The action of the policy (lines 18-20) is an allow action. Listing 2 shows two excerpts from the XML Schema Definition (XSD) that formally defines the syntax of the policy languages.

The first type defines that a policy is a combination of zero or more conditions and one action (and that a policy also has a name and an id). Conditions are defined as expression. For each of the types of policies, this generic XSD is extended to include the concepts that were introduced for that specific type of policy. For example, the second type defines the set of allowed actions for the security policy language as allow or deny by restricting the range of allowed values for the generic action type.

**User-friendly, graphical policy editors**

XML is not suited to be edited directly by policy authors. They prefer to use representations they are more familiar with (like for example a more graphical notation), and that do not require them to know the exact syntax of the policies but guides them more during policy specification. Because the
Listing 1. Example security policy rule

1 <security-policy name="Allow everyone access to ServiceX">
  2 <condition>
  3   <function function-id="equals">
  4     <target>
  5       <function function-id="getResourceId">
  6         <target>
  7           <variable variable-id="resource"/>
  8       </target>
  9     </function>
 10   </target>
 11   <parameter parameter-id="object">
 12     <explicit-value type="java.lang.String">
 13       <value>ServiceX</value>
 14     </explicit-value>
 15   </parameter>
 16 </function>
 17 </condition>
 18 <action>
 19   <allow/>
 20 </action>
 21 </security-policy>

Listing 2. XML Schema Definition for policies

1 <xsd:complexType name="T-policy">
  2 <xsd:sequence>
  3   <xsd:element name="condition" type="T-expression" minOccurs="0" maxOccurs="unbounded"/>
  4   <xsd:element name="action" type="T-action"/>
  5 </xsd:sequence>
  6 <xsd:attribute name="name" type="xsd:string" use="required"/>
  7 <xsd:attribute name="id" type="xsd:string" use="required"/>
  8 </xsd:complexType>
  9 <xsd:complexType name="T-security-action">
 10   <xsd:complexContent>
 11     <xsd:restriction base="T-action">
 12       <xsd:sequence>
 13         <xsd:choice>
 14           <xsd:element name="allow" type="T-allow"/>
 15           <xsd:element name="deny" type="T-deny"/>
 16         </xsd:choice>
 17       </xsd:sequence>
 18     </xsd:restriction>
 19   </xsd:complexContent>
 20 </xsd:complexType>

Fig. 6. A more graphical policy editor allowing the creation of policies using natural language statements and drop-downs.

that allow the policy author to only choose between valid values at any time during policy specification. By using natural language statements, the complexity of the policy language is hidden for the end user. It is constructed using the ILOG JRules [6] graphical rule editor. It uses a template-based approach where you have to specify how each (parameterizable) natural language statement can be translated to XML.

- Decision tables allow the specification of several policies at once using a tabular format. Figure 7 shows a decision table for expressing security policies. It is constructed using Drools (also known as JBoss Rules) [5] decision tables. For each type of condition or action allowed inside the policies, a separate column is present in the decision table. Each line represents one policy. By entering the appropriate values in each of the columns, a policy can specify its conditions and actions. For example, the first line specifies that service provider A is allowed to access method M on service X. The policies are executed in sequence (top-down) until a matching one is found. Although decision tables offer a user-friendly way of specifying policies that most users are already familiar with, it can only be used to specify rather simple policies. Complex policies would result in an explosion of columns.

These two policy editors are only two examples of different types of editors that could be designed, and how they offer support to service providers and end users during policy specification. Because graphical policy editors generate XML policies in a standardized format, new editors can easily be included since all policies are represented using our core policy model.

V. EVALUATION

To validate our approach, we applied it in the context of the telecom Service Enabling Platform (SEP) and created a prototype that allows the specification and enforcement of the five types of policies identified in section II. We now evaluate our approach based on the requirements as defined in section I:

- Common policy language: We are capable of expressing policies of the SEP in the different areas specified in section II by creating specific policy languages as extensions of our core policy language. Our policy language also
simplifies the specification of complex policies by using an elaborate expression model and allowing advanced action and policy combination.

- Ease of use: Not only system administrators but also service providers and end users can use policies to define their preferences themselves. (1) Specific policy languages allow the use of high-level domain concepts directly when specifying policies. (2) Using one of the more user-friendly policy editors, policies do not have to be defined using XML directly, but can take advantage of advanced visualizations like drop-downs (assisting users when specifying policies by showing allowed values) and decision tables (providing a table-like overview of a policy set).

- Extensibility: Since the generic policy language already defines most of the underlying concept needed for policy specification, creating a new specific policy language (or extending an existing one) is as simple as defining the necessary concepts and mapping these to policy concepts. New user-friendly policy editors can also be added as long as they can translate their custom format into the standardized XML.

The prototype allows us to actually enforce policies specified using any of the specific policy languages automatically. To allow enforcement of the policies, is must be possible to automatically execute the actions and evaluate the conditions specified in those policies. Therefore, a handler must be defined for each of the different actions capable of automatically executing those actions (command pattern). Similarly, functions and combining algorithms must be linked to their actual implementation. Variables must also be bound to actual objects at runtime before evaluation, based on the context the policies are evaluated in. Finally, a Policy Information Point (PIP) capable of retrieving the values of the (possibly high-level) properties of specific elements must be implemented.

Our policy enforcement architecture – described in more detail in [7] – is used to intercept all communication between the actors and makes sure that policies related to this communication are enforced. Because the SEP uses an Enterprise Service Bus (ESB) to manage the communication between different actors, messages can easily be intercepted and redirected to a policy engine, responsible for actually enforcing the policies, as shown in figure 8.

The policy engine itself is based on a rules engine where the policies defined in XML are translated to rules and used in the rules engine to evaluate the requests. A rules engine is a good candidate for implementing a policy engine since it uses declarative rules which makes the translation of the policies straightforward and since rules engines are especially designed to efficiently evaluate conditions, even when numerous rules are present. We used both Drools (also known as JBoss Rules) [5], an open-source rules engine in Java, and ILOG JRules [6], a commercial rules engine, to implement our prototype. Tests show that the performance of our policy engine is acceptable (for both engines): it takes about 15ms to evaluate a request, dependent on the number of conditions that must be evaluated but not directly dependent of the number of policies in the system.

However, our approach and prototype still exhibit some limitations. Firstly, the current ontology focusses on event-driven policies. Further evaluation, preferably in another domain using a different enforcement architecture, should determine whether it can be applied to other types of policies, such as state-based policies. Secondly, explicit representation (and refinement) of the semantics of concepts is necessary to avoid misinterpretation of concepts. Thirdly, this paper does not address the problem of policy refinement, where higher-level policy concepts are mapped to implementation-specific classes. Although our policy ontology can be used to translate high-level concepts to domain concepts, the translation of these domain concepts to implementation classes is currently hidden in the implementation of our policy engine (e.g. services and their operations are defined using a WSDL interface). Finally, the prototype exhibits some limitations as well: the different ontologies and mappings as presented in this paper are currently specified using an implementation- or tool-dependent format (Java and JRules). It might be better to specify them in an independent format (e.g. OWL) and explicitly import them into the tools.
VI. RELATED WORK

A lot of research has already been performed in the area of policy specification. We briefly compare our approach with other policy languages and model mapping approaches.

In the last decade, several policy description languages have been developed already. They range from policy languages that can only be used in one specific area and/or domain to generic languages capable of expressing different types of policies.

- Most policy languages have been developed for configuring services in one specific area, like security or QoS. For example, XACML [14] is an XML-based language for expressing access control policies. Verma [9] describes a QoS language using a tabular format.

- A policy language can focus on one specific application domain, like for example telecom: The ACCENT project developed a policy language called APPEL [8] that can be used to specify preferences for call control. Lago [10] describes a high-level policy language for personalizing communication.

- More generic policy languages allow the specification of policies in more than one area: Ponder [13] does not only offer support for access control policies but for other management policies as well. The Flexible Authorization Framework (FAF) [15] allows the specification of different types of access control policies based on different authorization models.

Our approach extends these approaches by allowing the specification of several types of policies based on one core policy model in broad spectrum of different areas. The Internet Engineering Task Force (IETF) and the Distributed Management Task Force (DMTF) have also created a generic policy model that allows the specification of any type of policies. The Policy Core Information Model (PCIM) [16] is a generic object-oriented information model for representing policy information. Specific policy models, based on the PCIM, are defined for specific areas like QoS (QPM), security (ISPS), etc. We extend this model by specifying in more detail how policy conditions are structured, simplifying specific policy language extensions. We also support more advanced action and policy combination.

Other policy languages like K AoS [11] and Rei(n) [12] also embrace the use of ontologies and enrich their language specification with explicit semantics. We believe that semantic technologies are promising candidates for defining the different ontologies and their semantics explicitly, and for describing more advanced mappings. They might even be used for policy reasoning (like consistency checking and conflict detection). Therefore, they are subject to future research.

Domain-specific or model-driven language frameworks like openArchitectureWare’s xText tool also offer support for generating a DSL (and a corresponding editor). They could be considered as alternative to the JRules DSL framework as a basis for implementing the editors and translators. Other model-driven approaches support the definition of high-level domain concepts and map these to their actual implementation.

This information could be used for implementing the PIP, which must be capable of retrieving the actual values of the properties of high-level domain concepts.

VII. CONCLUSION

When using policies for the configuration of existing services, one is confronted with the choice between a general-purpose policy language allowing the specification of policies in a broad domain, and domain-specific policy languages which are targeted to one particular area and/or domain and therefore are easier to specify.

We have taken a significant step towards bridging this gap by using (1) an expressive core policy model as the basis for specific policy languages targeted to one specific area and (2) allowing the use of domain-level concepts in policies directly. In addition, custom policy editors simplify policy specification by offering a user-friendly format users are more familiar with.

By applying this approach in the context of a telecom platform, service providers and even end users can now use policies to configure services in different areas like security, QoS and user preferences.

ACKNOWLEDGMENT

The research in this paper has been conducted in the context of the Tcase project of the Interdisciplinary institute for BroadBand Technology (IBBT).

REFERENCES