Exchanging Policies between Web Service Entities using Rule Languages

Nima Kaviani[1], Dragan Gašević[2], Marek Hatala[1], Gerd Wagner[2], Ty Mey Eap[1]
[1]*Simon Fraser University Surrey, Canada*
[2]*Athabasca University, Canada*
[2]*Brandenburg University of Technology at Cottbus, Germany*
{nkaviani, mhatala, teap}@sfu.ca, dgasevic@acm.org, wagnerg@tu-cottbus.de

Abstract

Web rule languages with the ability to cover various types of rules have been recently emerged to make interactions between web resources and broker agents possible. The chance of describing resources and users of a domain through the use of vocabularies is another feature of Web rule languages. Combination of these two properties makes Web rule languages an appropriate medium to make a hybrid model of representing both contexts and rules of a policy-aware system, such as a web service. In this paper, we describe how REWERSE Rule Markup Language (R2ML) can be employed to bridge between different policy languages using its rich set of rules, vocabulary, and built-in constructs. We show how the concepts of the KAoS and Rei policy languages can be transformed to R2ML and then from R2ML to the other policy languages. Following these mappings, we have implemented transformers, which enable us not only to share policies between KAoS and Rei, but also to transform policies onto other rule languages (e.g., F-Logic) for which transformations from/to R2ML are already developed.

1. Introduction

Semantic Web services (SWS), as the augmentation of Web service descriptions through Semantic Web annotations, facilitate the higher automation of service discovery, composition, invocation, and monitoring on the Web. Semantic Web ontologies and ontology languages (OWL and RDF(S)) are recognized as the main means of knowledge representation for Semantic Web services [18]. Such ontology-enriched web service descriptions are later used during negotiation process between service clients and service providers, which is defined by a set of abstract protocols of Semantic Web Service Architecture (SWSA) [3].

However, the current proposed standards for describing Semantic Web services (i.e. OWL-S, WSDL-S, and Semantic Web Service Language – SWSL) demonstrate the importance of using a rule language in addition to ontologies. This allows run-time discovery, composition, and orchestration of Semantic Web services by defining preconditions or post-conditions for all Web service messages exchanged. For instance, OWL-S recommends using OWL ontologies along with different types of rule languages (SWRL, KIF, or DRS), while SAWSDL is fully agnostic about the use of a vocabulary (e.g., UML, ODM, OWL) or rule language (e.g., OCL, SWRL, RuleML). It is worth noting that there is no agreement upon which rule language to use for Semantic Web services or what type of reasoning (open or closed world assumption) to support.

Besides satisfying clients' goals when using Semantic Web services, trust is another important aspect that should be established between a client and a service. Addressing this problem, researchers proposed the use of policy languages. A policy is a rule that specifies under which conditions a resource (or another policy) might be disclosed to a requester [13]. To define polices on the Semantic Web, various policy languages have been proposed including KAoS [21], Rei [8], and PROTUNE [2]. As [13] reports, trust management policies are also defined as parts (most commonly preconditions) of Semantic Web service descriptions.

It is obvious that besides various Semantic Web service description languages, we have various policy languages and rule languages. All these languages are based on different syntactic representations and formalisms with no explicitly defined mapping between them. This hampers the use of Semantic Web services from two different perspectives. One is automatic negotiation between service client agents and service providers and automatic matchmaking, where agents and matchmakers should be able to "understand" various rule/policy/service web service description languages. Another perspective is that of a knowledge management worker who prefers to express the rules and policies in a single form rather than in a broad variety of forms. To attempt to represent the same rules and policies in many forms is cumbersome, time consuming and error prone but it is the only choice available if a broad base of interoperability is required.

In our approach, we propose the use of REWERSE Rule Markup Language (R2ML) [23], which addresses almost all use-cases and needs for a Rule Interchange Format (RIF) [4], along with a set of transformations between Semantic Web service description (e.g., WSDL-S), rules, and policy languages. We illustrate the benefits of our approach using a Semantic Web Service Architecture example where R2ML is used to share policies in the process of matchmaking, trust negotiation, and failure explanation. In the next section, we motivate our research by describing a trust negotiation scenario for using Web services.

2. Motivations

Web services in general and Semantic Web services in particular, are of the most important domains for applying Web policy rules. Policies can be regarded as constraints to be combined with Web services to identify explicitly conditions under which the use of a service is permitted or prohibited [7]. However, due to the diversity of existing policy languages, chances are that the policies used to protect the services or resources are not defined in the same language, which makes the process of communication impossible.

As an example, let us consider a scenario in which a Web service provider and a broker agent, with two different policy languages, need to negotiate their policies. In this scenario, we assume that the service provider has its policies defined in Rei and the broker agent has the policies defined in KAoS (cf. Section 3). Suppose that the broker agent has a policy which says: *"Client A can only communicate with service providers that support authentication with X509 certificates and have already been approved as trusted entities for Client A to communicate with."* Figure 1 represents the policy in KAoS defined by the broker agent. Now suppose that during the process of trust negotiation the client needs to ask the service provider whether it could provide any support for the X509 certificate authentication. If there is no constraint on releasing the policy, this can possibly happen by sending the policy of Figure 1 to the service provider. In case that there is no understanding about Rei by the client or no knowledge about KAoS by the service provider, the communication would fail. So, either the broker agent or the service provider must be able to exchange the policy to an equivalent Rei policy. The result of transformation can be similar to the policy defined in Figure 2.

```
<owl:Class rdf:ID="Policy_CommunicationCertificate_Action">
 <owl:intersectionOf>
  <owl:Class rdf:about="&KAoSAction;CommunicationAction"/>
   <owl:Class>
    <owl:Restriction>
     <owl:onProperty rdf:resource="&KAoSAction;performedBy"/>
      <owl:someValuesFrom>
       <owl:Class>
        <owl:oneOf rdf:parseType="Collection">
         <owl:Thing rdf:about="&InstanceElem;ClientA"/>
        </owl:oneOf>
       </owl:Class>
      </owl:someValuesFrom>
    </owl:Restriction>
   </owl:Class>
   <owl:Restriction>
    <owl:onProperty rdf:resource="&KAoSPolicy;hasPartner"/>
      <owl:allValuesFrom rdf:resource="&KAoSActor;
                #ServiceProviderSupportingX509Certificates"/>
   </owl:Restriction>
 </owl:intersectionOf>
</owl:Class>
<!--Checks on whether the entity that is receiving the request by Client
A is a trusted entity-->
<owl:Class rdf:about=" Policy_TrustedEntity ">
    <owl:intersectionOf rdf:parseType="Collection">
            <owl:Class rdf:about="&KAoSAction;ApproveAction"/>
            <owl:Restriction rdf:about="#checkOnCarryingMessages">
                    <!—Similar to the above restriction;
                         checks if the action is carrying a message -->
            </owl:Restriction>
            <owl:Restriction rdf:about="#TrustedEntity">
                    <!—Similar to the above restriction; checks on whether the
                        destination is a Trusted Entity -->
             </owl:Restriction>
            <owl:Restriction rdf:about="#actorRestriction">
                   <!—Similar restriction; checks on whether the actor ClientA -->
            </owl:Restriction>
    </owl:intersectionOf>
 </owl:Class>
<policy:PosAuthorizationPolicy
rdf:ID="Policy_CommunicationCertificate">
  <policy:requiresConditions rdf:resource="#Policy_ TrustedEntity "/>
  <policy:controls
rdf:resource="#Policy_CommunicationCertificate_Action"/>
  <policy:hasPriority>2</policy:hasPriority>
</ policy:PosAuthorizationPolicy>
```

Figure 1. A sample of a KAoS Policy

```
<entity:Variable rdf:ID="ActorVar"/>
<policy:Policy rdf:ID="policy_N100CE">
   <deontic:actor rdf:resource="#ActorVar" />
   <policy:grants rdf:resource="#Policy_CommunicationCertificate" />
   <policy:context rdf:resource="#CommunicationActionConstraint" />
</policy:Policy>
<deontic:Permission rdf:ID="Policy_CommunicationCertificate">
    <deontic:action rdf:resource="&KAoSAction;CommunicationAction"/>
    <deontic:actor rdf:resource="#ActorVar"/>
    <deontic:constriant rdf:resource="#ActionApprovalConstraint"/>
</deontic:Permission>
<constraint:And rdf:ID="ActionApptovalConstraint">
      <!--Conjunction of Constraints for the preconditionst plus the
CommunicationActor constraint-->
</constraint:And>
<!-- Constraints for the TrustedEntity and CarryMessage
 to control the behavior of the system -->
<constraint:SimpleConstriant rdf:ID="CommunicationActor">
          <constriant:subject rdf:resource="#ActorVar"/>
          <constriant:object rdf:resource="&rdfs;type"/>
          <constraint:predicate rdf:resource="&KAoSActor;ClientA"/>
</constraint:SimpleConstriant>
<constraint:SimpleConstriant rdf:ID=" CommunicationActionConstraint">
          <constriant:subject rdf:resource="#ActionVar"/>
          <constriant:object rdf:resource="&KAoSActor;
                          ServiceProviderSupportingX509Certificates"/>
          <constraint:predicate rdf:resource="&KAoSPolicy;hasPartner"/>
</constraint:SimpleConstriant>
```

Figure 2. An equivalent Rei policy of Figure 1

In a general case either the Web agent or the service provider must be able to negotiate to the other resources and agents with different languages for sharing their resources and policies as well. However, considering the number of available policy languages and services, it would be a lot of effort to develop one-to-one transformations between the policy languages. For *n* policy languages we would need *n*(n -1)* transformations. It gets even worse if we consider the constant changes that should be applied to the transformations due to the improvements and extensions of each policy language. On the contrary, using an intermediary language would considerably reduce the number of transformations. For *n* policy languages, we will need *n* transformations to the intermediary policy language, and then *n* transformations from this language back to all the policy languages. Thus the number of transformations will boil down to *2*n*.

Addressing the problem of negotiation in scenarios similar to the one explained above would significantly help to enable inter-enterprise interactions. It can also help with providing distributed trust-negotiation-failure explanations, as it has been discussed in [1]. Next we review the properties of KAoS and Rei and then explain our approach.

3. Rei and KAoS

Policies in the domain of autonomous computing are guiding plans that restrict the behavior of autonomous agents in accessing the resources [19]. The main advantage in using policies is their capability to dynamically regulate the behavior of the system without applying any change to the system's internal code.

Rei [7, 8] and KAoS [21, 22] are two semantically enriched Web policy languages that use Semantic Web ontologies to define the resources, the behavior, and the users of a domain. It enables these two languages to easily adjust themselves to the target system regardless of the number of resources and users in act. KAoS describes the entities and concepts of its world using OWL while Rei can understand and reason over a domain of concepts defined in either RDF or OWL.

In terms of available policy rules both KAoS and Rei have four main types. *Permission, Prohibition, Obligation,* and *Dispensation* in Rei are respectively equivalent to *PosAuthorizationPolicy, NegAuthorizationPolicy, PosObligationPolicy,* and *NegObligationPolicy* in KAoS. The defined policy rules in each of the languages are then sent to a reasoner that performs the process of conflict resolution and decision making for the rules that match the current state of the world. This task is done by using Stanford's Java Theorem Prover (JTP) in KAoS and a Prolog engine in Rei version 1.0. Rei version 2.0 has extended its reasoning engine to use F-OWL, an ontology inference engine for OWL, based on Flora and XSB Prolog [25].

Although these two policy languages have a lot in common there are many dissimilarities between them as well. The main difference between KAoS and Rei is the underlying formalism of the languages. KAoS follows description logic coded in the form of OWL expressions to define its elements and rules. On the other hand, Rei uses its own language that defines rules in terms of Prolog predicates expressed as RDF triples (see Figure 2). This way Rei follows Prolog's semantics which is itself built on top of the concepts of declarative logic.

The process of rule enforcement in KAoS is done by extending its enforcement engine depending on the domain it is going to be used in. In Rei, however, there is no rule enforcement engine. Yet, due to the deterministic properties of declarative logic, reasoning over dynamically determined values in Rei policies is more accurate than KAoS in which chances of dealing with unknown situations are likely to happen.

In order for processes and services to communicate remotely, Rei relies on a rich set of *Speech Acts*. In Rei, Speech Acts are used by a sender to express the request for performing one of the actions: *Delegation, Revocation, Request, Cancel, Command,* and *Promise* by the receiver. Conversely, in KAoS the remote communication procedure is done through the message passing of the underlying platform.

Defining KAoS policies as OWL expressions gives the language more flexibility to maneuver over the concepts of the world. Different quantifying expressions, inheritance relationships, cardinality restrictions, etc. can be explicitly expressed in KAoS thanks to the constructs of OWL. It also enables KAoS to perform static conflict resolution and policy disclosure. KAoS has its classes and properties already defined in OWL ontologies, referred to as KAoS Policy Ontologies (KPO) [21], which are accessible from [9].

A KAoS policy is defined as an instantiation of the policy class with all its properties have been allocated the values according to the entities (both users and resources) that play a role in firing the policy. Figure 1 shows a typical KAoS permission policy, named *Policy_CommunicationCertificate* with all its elements instantiated. The two elements *controls* and *requiresConditions* are the most crucial elements in a KAoS policy. *requiresCondition* is an OWL property with its range in the *Conditions* class of KPO. A condition element checks on the current situation of the world and upon the occurrence of an event that satisfies the defined conditions, the policy is fired. The *control*s element has the *Action* class as its range and defines the desired event to be enforced. It contains the user of an action and also a context to which the action should be applied. For example, in Figure 1, the *requiresConditions* element checks for the approval on passing messages to a service provider and *controls* checks on service provider's support for X509. The appropriate action, if the policy constraints meet, is to *carry* the *message* to the service provider by *client A*. There are also some other KAoS elements in a policy to define the priority of a rule, the site of enforcement, etc.

On the contrary, Rei policy rules are similar to the expressions in Prolog. This makes the rules easier to understand; but static conflict resolution and policy disclosure are not possible because the variables for the policies are instantiated during run-time and there is no possibility to process their values offline. Analogous to KAoS, a policy in Rei is instantiated from the policy class defined in Rei ontologies. The policy instance in Rei guides the behavior of entities in the policy domain [16]. It contains a list of one or more deontic rules (expressed as deontic objects) and a context for applying policies. Conditions defined for *context* element identify the suitable domain for the policy to be applied to. A deontic object consists of the action to be enforced, the related actor, and the set of conditions that must be true for the action to be performed. Figure 2 shows a sample Rei policy with one deontic rule.

Looking back to all the similarities and differences discussed, providing meaningful transformations between these two policy languages is not an easy goal to achieve. The transformation should care about the concepts that may miss in the procedure of transformation, the importance of the missed elements, the harms and threats that may happen to the resource due to the information loss, and etc. The intermediary language, to serve as the medium, should have enough elements and constructs to cover all the concepts of different policy languages. The loss of the concepts should not happen during the transformation from the source languages to the intermediary language, but the loss during transforming from the intermediary language to the target language might be inevitable.

4. Using Web Rule Languages for Policies

The new generation of policy languages goes beyond defining rules that control only the users. They also put constraints on the resources of a domain. The resources may evolve and expand during time, so the policy languages should be expandable as well. Since ontologies are easy to extend, Semantic Web has gained a lot of reputation in this area. Consequently, the intermediary language that is going to be used to transform the information should not only support the definition of the rules, but also be able to transfer the domain properties and features. In this paper, we argue that Semantic Web rules are appropriate solutions to this problem, as they can cover the policies through defining rules and ontologies. Further on this can be found in [11], where we have discussed and explained the logic of transforming between policy languages.

Most of the proposals on Web rule languages are trying to address the use cases and requirements defined by Rule Interchange Format Working Group [4].

Rule Interchange Format (RIF) [4] is an initiative to address the problem of interoperability between existing rule-based technologies. RIF is desired to play as an intermediary language between various rule languages and not as a semantic foundation for the purpose of reasoning on the Web. RIF Working Group has defined ten use cases which have to be covered by a language compliant to the RIF properties, three of which are dealing with policies and business rules, namely: *Collaborative Policy Development for Dynamic Spectrum Access, Access to Business Rules of Supply Chain Partners*, and *Managing Inter-Organizational Business Policies and Practices. SWRL* [6] and *RuleML* [5] are two of the ongoing efforts in this area trying to serve as rule languages to publish and share rule bases on the Web.

In this paper, we use *REWERSE Rule Markup language (R2ML)* as an attempt to address the use cases and requirements of RIF. In the next subsection, we briefly describe some concepts of R2ML before going through the description of the mappings between R2ML and KAoS and between R2ML and Rei.

4.1 R2ML Interchange Format

R2ML is a general rule interchange language that tries to address all RIF requirements. The abstract syntax of R2ML language is defined with a metamodel by using the OMG's Meta-Object Facility (MOF). This means that the whole language definition can be represented by using UML diagrams, as MOF uses UML's graphical notation. The full description of R2ML in the form of UML class diagrams is given in [15, 23]. The language also has an XML concrete syntax defined by an XML schema. There are also a number of transformations implemented between R2ML and rule-based languages (e.g., OCL, SWRL, and F-Logic).

R2ML considers four kinds of rules: *integrity rules, derivation rules, production rules,* and *reaction rules.* Our transformations of policy languages to R2ML use derivation rules which are the most suitable rules to demonstrate capabilities for inference and reasoning. A derivation rule has conditions and a conclusion (see Figure 3) with the ordinary meaning that the conclusion can be derived whenever the conditions hold. While the conditions of a derivation rule are instances of the *AndOrNafNegFormula* class, representing quantifier-free logical formulas with conjunction, disjunction and negation; conclusions are restricted to quantifier-free disjunctive normal forms without *NAF* (Negation as Failure, i.e. weak negation).

Conditions and conclusions are both defined by the use of *Atoms* which are the basic constituents of a formula in R2ML. For our transformation of policy languages, we use R2ML *ReferencePropertyAtoms* in the condition and R2ML *ObjectDescriptionAtom* in the conclusion part. A ReferencePropertyAtom associates object terms as "subjects" with other object terms as "objects". An ObjectDescriptionAtom refers to a class as a base type and to zero or more classes as categories, and consists of a number of property/term pairs (attribute data term pairs and reference property object term pairs). Any instance of such atom refers to one particular object that is referenced by an objectID, if it is not anonymous.

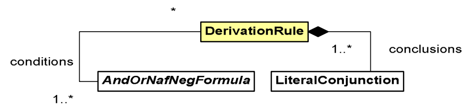


Figure 3. The UML representation of a derivation rule

4.2 Mapping between R2ML and Policy Languages

In this subsection, we discuss the mappings between different policy languages using R2ML derivation rules. In R2ML, we have both integrity rules and derivation rules defined; with the integrity rules divided into deontic rules and alethic rules. An integrity rule, also known as (integrity) constraint, consists of a constraint assertion, which is a sentence in a logical language such as first-order predicate logic. A derivation rule in R2ML is different from an integrity rule in the sense that there is no concrete proof for the correctness of a derivation rule. A derivation rule is a better construct to show the inference capabilities over existing facts to obtain new facts. This is exactly the same with policies as in most of the cases approval or denial of performing an action is based on an inference over the credentials provided by the user.

Transforming policies from KAoS to R2ML

In Section 2, we have mentioned that a KAoS policy is an object of the Policy class in KPO with its attributes instantiated to a set of users, events, and resources that make the policy fire. Considering the KAoS policy element (e.g. Figure 1) as a rule, the *controls* element is executed upon the occurrence of the events described in the *requiresConditions* element. Thus, to model the KAoS policy with a derivation rule, we place the content of the *controls* element in the conclusion part and the content of the *requiresConditions* element in the condition part of the rule. A *controls* element consists of the action to be performed, the actor of the action, and the context of performing the action. To model the actor, the action and the restrictions defined over the context of the to-be-executed action, we chose R2ML *ObjectDescriptionAtom*. This atom can neatly embed all of the mentioned concepts as arguments in its definition.

To model the condition part of a KAoS policy, we employed R2ML *ReferencePropertyAtoms*. Similar to the control part, conditions in KAoS are usually represented as a class defined over an occurred action or state with a set of properties that restrict the action. Although the conditions of a policy rule could also be modeled with *ObjectDescriptionAtom*, the main reason in choosing *ReferencePropertyAtom* was to be compliant with the definitions of Rei (defined in the form of triples) and also other R2ML transformations (e.g transformations between F-Logic and R2ML also have ReferencePropertyAtom in the condition part). It simplifies the later conversions of the policies to other rule languages for which we have R2ML transformations already defined (e.g F-Logic). Moreover, a ReferencePropertyAtom triple models a binary predicate. A set of ReferencePropertyAtoms with the same subject element can always be combined and converted to any element of higher arity (e.g. ObjectDescriptionAtom), and thus using ReferencePropertyAtom does not contradict with the use of ObjectDescriptionAtom. Furthermore, in our case, ReferencePropertyAtoms carry even a better semantic meaning for the transformations. Semantically they are equivalent to an OWL object property, and as KAoS is nothing but pure OWL, they model object properties of KAoS too.

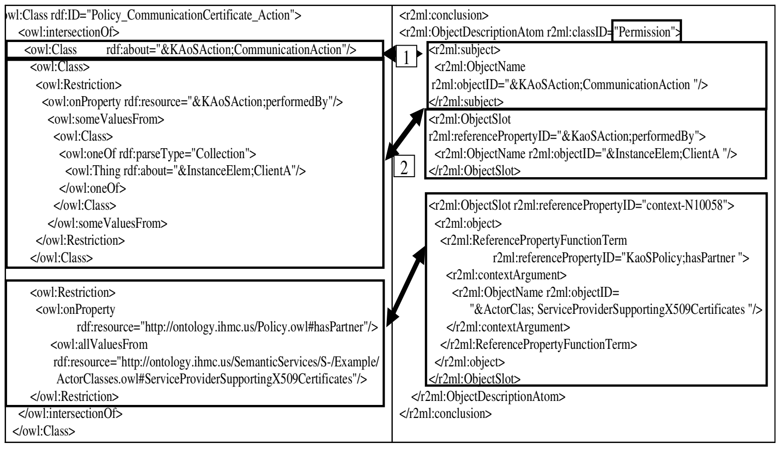
A KAoS policy might also have a *trigger* element. This element is only used with NegObligation- and PosObligation-Policies showing a set of events that trigger the occurrence of an action. In our transformations, we deal with those elements as ReferencePropertyAtoms in the condition part as well. However, to discriminate them from the preconditions, we annotate them as triggering elements.

We show an excerpt of transformation rules between KAoS and R2ML in Figure 4. The XSLT implementations of our transformations are available in [24], where further details can be found.

	KAoS Element	R2ML Element
Policy Element	<policy:**PosAuthorizationPolicy** rdf:ID="Policy_CommunicationCertificate"> <policy:requiresConditions rdf:resource="#Policy_ TrustedEntity "/> <policy:controls rdf:resource="#Policy_CommunicationCertificate_Action"/> </policy:PosAuthorizationPolicy>	<r2ml:DerivationRule> <r2ml:conditions> <!-- The mappings for the condition part goes here --> </r2ml:conditions> <r2ml:conclusion> <r2ml:ObjectDescriptionAtom r2ml:classID="**Permission**"> <!-- The mappings for the conlcusion part goes here --> </r2ml:ObjectDescriptionAtom> </r2ml:conclusion> </r2ml:DerivationRule>
Control Element	<owl:Class rdf:ID="Policy_CommunicationCertificate_Action"> <owl:intersectionOf> <!-- the set of constraints on the action in KAoS are defined as intersection of a series of restrictions in OWL --> </owl:intersectionOf> </owl:Class>	<r2ml:ObjectDescriptionAtom r2ml:classID="**Permission**"> <!-- The mappings for the conlcusion part goes here --> </r2ml:ObjectDescriptionAtom>
	<owl:Class> <owl:Restriction> <owl:onProperty rdf:resource="http://ontology.ihmc.us/ Action.owl#performedBy"/> <owl:allValuesFrom> <!-- Defines the set of actors that are responsible for the action--> </owl:allValuesFrom> </owl:Restriction> </owl:Class>	<r2ml:ObjectSlot r2ml:referencePropertyID=" http://ontology.ihmc.us/Action.owl#performedBy"> <!--We can use either r2ml:ObjectName or r2ml:ObjectVariable define the actors --> </r2ml:ObjectSlot>
	<owl:Restriction> <owl:onProperty rdf:resource="http://ontology.ihmc.us/Policy.owl#hasPartner"/> <owl:allValuesFrom rdf:resource="#TrustedServiceProvider"/> </owl:Restriction>	<r2ml:ObjectSlot r2ml:referencePropertyID="context-N10058"> <r2ml:object> <r2ml:ReferencePropertyFunctionTerm r2ml:referencePropertyID=" http://ontology.ihmc.us/Policy.owl#hasPartne"> <r2ml:contextArgument> <r2ml:ObjectName r2ml:name="#TrustedServiceProvider"/> </r2ml:contextArgument> </r2ml:ReferencePropertyFunctionTerm> </r2ml:object> </r2ml:ObjectSlot>

Figure 4. Mappings between KAoS and R2ML elements

KAoS uses role-value-map technique to deal with dynamic allocation of values to variables. However, in our implementation, we use a simpler model of defin-



```
wl:Class rdf:ID="Policy_CommunicationCertificate_Action">
  <owl:intersectionOf>
    <owl:Class        rdf:about="&KAoSAction;CommunicationAction"/>
      <owl:Class>
        <owl:Restriction>
          <owl:onProperty rdf:resource="&KAoSAction;performedBy"/>
            <owl:someValuesFrom>
              <owl:Class>
                <owl:oneOf rdf:parseType="Collection">
                  <owl:Thing rdf:about="&InstanceElem;ClientA"/>
                </owl:oneOf>
              </owl:Class>
            </owl:someValuesFrom>
        </owl:Restriction>
      </owl:Class>

      <owl:Restriction>
        <owl:onProperty
                rdf:resource="http://ontology.ihmc.us/Policy.owl#hasPartner"/>
           <owl:allValuesFrom
           rdf:resource="http://ontology.ihmc.us/SemanticServices/S-/Example/
            ActorClasses.owl#ServiceProviderSupportingX509Certificates"/>
      </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>
```

```
<r2ml:conclusion>
<r2ml:ObjectDescriptionAtom r2ml:classID="Permission">
  1  <r2ml:subject>
      <r2ml:ObjectName
     r2ml:objectID="&KAoSAction;CommunicationAction "/>
     </r2ml:subject>
     <r2ml:ObjectSlot
     r2ml:referencePropertyID="&KaoSAction;performedBy">
      <r2ml:ObjectName r2ml:objectID="&InstanceElem;ClientA "/>
  2  </r2ml:ObjectSlot>

     <r2ml:ObjectSlot r2ml:referencePropertyID="context-N10058">
      <r2ml:object>
       <r2ml:ReferencePropertyFunctionTerm
                  r2ml:referencePropertyID="KaoSPolicy;hasPartner ">
        <r2ml:contextArgument>
         <r2ml:ObjectName r2ml:objectID=
            "&ActorClas; ServiceProviderSupportingX509Certificates "/>
        </r2ml:contextArgument>
       </r2ml:ReferencePropertyFunctionTerm>
      </r2ml:object>
     </r2ml:ObjectSlot>
   </r2ml:ObjectDescriptionAtom>
</r2ml:conclusion>
```

Figure 5. Mapping of a KAoS *controls* element (left) to R2ML (right)

ing variables (similar to what Rei does) and convert role-value-mapped elements of KAoS to a variable-like definition using R2ML's *ObjectVariable*. This makes the process of converting R2ML variables to the variables of other languages easier. Figure 5 shows an excerpt of the policy that we described in Figure 1, converted to its R2ML equivalent based on the transformation rules explained in Figure 4. The conversion shows how a *controls* element of KAoS is represented in the conclusion part of an R2ML derivation rule. Due to the space limits, we do not give the explanations for transforming the other parts of the rule.

Transforming policies from Rei to R2ML

A Rei policy, similar to KAoS, is an instantiation of the Policy class, defined in the Rei ontology [16]. However, a policy element in Rei represents a list of policy rules (each defined as a deontic child element), while a policy construct in KAoS represents only one rule. Each R2ML derivation rule is also equivalent to one policy rule. Therefore, converting a policy from Rei to KAoS or R2ML may result in having more than one KAoS policy or R2ML rule. Furthermore, as Rei deals with variables similar to Prolog, and because variables can have different values during run-time, a single policy in Rei might be converted to multiple KAoS policies based on different combinations of values that the variables can take. Fortunately R2ML can accept a set of derivation rules by defining them as derivation rule set, in case more than one policy rule can be derived from a Rei policy.

Our R2ML rule structure is more similar to the structure of Rei than to KAoS and hence it is easier to convert a Rei policy rule to R2ML. Rei uses *SimpleConstraints* and *BinaryConstraints* to define the conditions of a deontic rule. All constraints for a deontic element are considered as preconditions of that deontic rule and treated the same way as *requiresConditions* element in KAoS. A deontic element in Rei has an action and an actor as its child elements as well. The actor is mapped as an object argument with a *performedBy* connector under the R2ML ObjectDescriptionAtom in the conclusion part of a rule, similar to what we did for KAoS. The actions in Rei are defined either by using Rei elements or OWL classes. For mapping the actions to R2ML, we can just refer to the already defined action or use R2ML vocabulary to redefine it. The action can then be placed in the subject element of our R2ML ObjectDescriptionAtom which is again similar to what we did for KAoS.

A policy element in Rei has also a child element, named context. The deontic set of rules operate on this context. So, in our R2ML transformation, we copy this same context for all derivation rules and store it as a ReferencePropertyFunctionTerm in the conclusion part of our derivation rule under the ObjectDescriptionAtom for the policy element. Thanks to the use of ReferencePropertyAtom in R2ML, we can easily convert each triple constraint of Rei to R2ML through a one to one mapping of the subject, object, and predicate elements. Figure 6 shows some of the mapping rules used to convert a Rei rule to an equivalent R2ML construct.

	Rei Element	R2ML Element
Deontic Element	`<deontic:`**`Permission`**` rdf:ID="PolicyName">` `<deontic:action rdf:resource="#actionToPerform"/>` `<deontic:actor rdf:resource="#ActorSet"/>` `<deontic:constriant rdf:resource="#ConstraintsSet"/>` `</deontic:Permission>`	`<r2ml:DerivationRule>` `<r2ml:conditions>` `<!-- The constraints are placed here -->` `</r2ml:conditions>` `<r2ml:conclusion>` `<r2ml:ObjectDescriptionAtom r2ml:classID="`**`Permission`**`">` `<!-- The mappings for the conclusion part goes here -->` `<!-- action and actor elements go here -->` `</r2ml:ObjectDescriptionAtom>` `</r2ml:conclusion>` `</r2ml:DerivationRule>`
Constraint	`<constraint:SimpleConstriant rdf:ID="ActorElem">` `<constriant:subject rdf:resource="#Actor"/>` `<constriant:object rdf:resource="#Student "/>` `<constraint:predicate rdf:resource="&rdfs;type "/>` `</constraint:SimpleConstriant>`	`<r2ml:ReferencePropertyAtom r2ml:propertyID="&rdfs;type">` `<r2ml:subject>` `<r2ml:ObjectVariable r2ml:name="#Actor"/>` `</r2ml:subject>` `<r2ml:object>` `<r2ml:ObjectName r2ml:objectID="#Student"/>` `</r2ml:object>` `</r2ml:PropertyAtom>`

Figure 6. Mappings between Rei and R2ML elements

Now that we have the transformations defined, let us apply them to the Rei policy of Figure 1. Figure 7 shows the result of applying the transformation rules. Comparing the R2ML snippet generated from the defined Rei policy in Figure 7 with the R2ML snippet of Figure 5 one would immediately realize that these two rules are identical although they have been generated from two completely different policy rules. It should be mentioned that the transformations are not always as straightforward as in this example. This might be a result of difference in the level of abstraction in the policy languages, the way they address variables, concepts, and entities, etc. The arrows in Figures 5 and 7 show the equivalent information pieces in the source and target languages.

Transforming from R2ML to Rei or KAoS

Transforming back to either of these policy languages is a much simpler task now that we have the transformation rules defined. Of course, the mappings are bidirectional. That is, the same way that, for example, a SimpleConstraint element of the Rei language would be converted to a ReferencePropertyAtom in R2ML, a ReferencePropertyAtom in the condition part of a rule can be converted to a SimpleConstraint in Rei. However, there are some concepts that are not simple to map. For example one may think how an OWL class will be created out of a set of ReferencePropertyAtoms. As we have already mentioned, the set of ReferencePropertyAtoms with the similar subject will create an OWL class with the properties modeled as restrictions on values in the object part of the ReferencePropertyAtom. In situations where the ReferencePropertyAtom carries a *trigger* tag, the obtained OWL class will be considered as a triggering class and otherwise it will be a precondition.

Another similar issue in our transformations happens when dealing with priorities. Priorities in KAoS are defined with numbered values, but in Rei we have meta-rules to give priority to one rule over the other. We have tried to solve this problem by converting the meta-rules of Rei to a numbered model in our R2ML transformation. During conversion from R2ML to Rei we convert the numbers to a form of meta-rules compliant with Rei syntax.

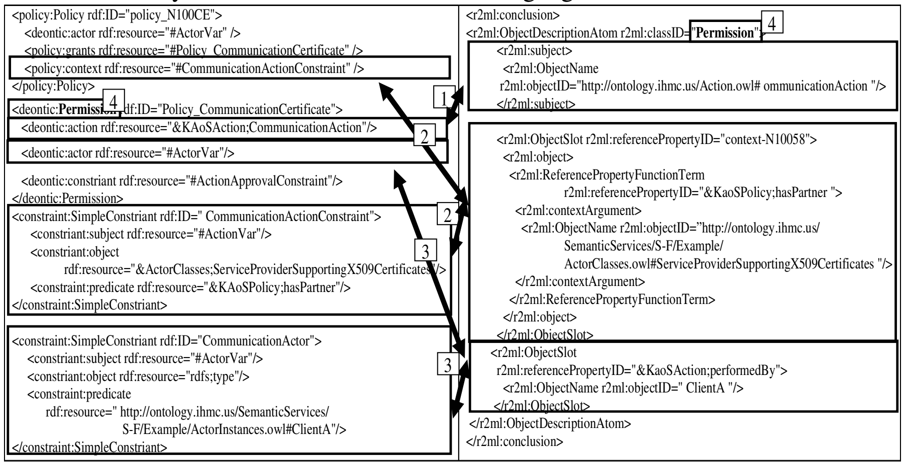
5. Discussion and Conclusion

We have so far described the transformations between policy languages and R2ML, thus proved that the transformations are possible. But it does not mean that we can fully transform between the two languages without any information loss. The characteristics of policy languages, especially KAoS and Rei (due to the differences in their underlying logic), makes an exact mapping between different elements difficult. KAoS follows description logic in which we have constructs for different quantifiers (both existential and universal), but Rei is similar to Prolog in which all the variables are universally quantified. This means while converting a KAoS rule to an equivalent Rei policy, we can not explicitly express that the defined variable in Rei should be existentially quantified.

In KAoS, we can define maxCardinality and minCardinality as restrictions on the number of events, but to the best of our knowledge there is no way to define such concepts in Rei. So, either these restrictions should be ignored or Rei should be expanded to cover them. It is worth mentioning that generality of R2ML helps in defining all the concepts above, because it has the required elements to cover the concepts in both domains (i.e. descriptive logic and declarative logic). We are now conducting research on the amount of harms and threats that may happen to the system due to the information loss during transformations.

As we have already mentioned, Rei has SpeechAct elements to describe the inter-process communications for remote policy control. In KAoS, there is no way to model these concepts and handling the communication between remote systems is supposed to be done by the underlying system. Therefore, although we can transfer SpeechActs from Rei to R2ML, they will be left out during the transformation to KAoS. Unfortunately, most of the time the constructs used in SpeechActs are inter-woven to the other policy elements of the language, such as Permission or Prohibition. In this case, a transformation to KAoS would not be helpful at all as the intended meaning of the policy will be lost.

We have also explained why derivation rules work better to define the policies. Now that we have the policies defined with derivation rules, we can use some other transformations that map a R2ML derivation rule with a similar structure to another rule language. One possible transformation is R2ML to F-Logic [17]. The existence of the transformation to F-Logic enables us to convert our policies into F-Logic and use them in systems compatible with F-Logic rules. According to the transformation rules provided


```
<policy:Policy rdf:ID="policy_N100CE">
   <deontic:actor rdf:resource="#ActorVar" />
   <policy:grants rdf:resource="#Policy_CommunicationCertificate" />
   <policy:context rdf:resource="#CommunicationActionConstraint" />
</policy:Policy>
<deontic:Permission rdf:ID="Policy_CommunicationCertificate">
   <deontic:action rdf:resource="&KAoSAction;CommunicationAction"/>
   <deontic:actor rdf:resource="#ActorVar"/>

   <deontic:constriant rdf:resource="#ActionApprovalConstraint"/>
</deontic:Permission>
<constraint:SimpleConstriant rdf:ID=" CommunicationActionConstraint">
     <constriant:subject rdf:resource="#ActionVar"/>
     <constriant:object
              rdf:resource="&ActorClasses;ServiceProviderSupportingX509Certificates"/>
     <constraint:predicate rdf:resource="&KAoSPolicy;hasPartner"/>
</constraint:SimpleConstriant>

<constraint:SimpleConstriant rdf:ID="CommunicationActor">
     <constriant:subject rdf:resource="#ActorVar"/>
     <constriant:object rdf:resource="rdfs;type"/>
     <constraint:predicate
          rdf:resource=" http://ontology.ihmc.us/SemanticServices/
                               S-F/Example/ActorInstances.owl#ClientA"/>
</constraint:SimpleConstriant>
```
```
<r2ml:conclusion>
<r2ml:ObjectDescriptionAtom r2ml:classID="Permission">
        <r2ml:subject>
         <r2ml:ObjectName
        r2ml:objectID="http://ontology.ihmc.us/Action.owl# ommunicationAction "/>
        </r2ml:subject>

        <r2ml:ObjectSlot r2ml:referencePropertyID="context-N10058">
         <r2ml:object>
          <r2ml:ReferencePropertyFunctionTerm
                    r2ml:referencePropertyID="&KaoSPolicy;hasPartner ">
           <r2ml:contextArgument>
            <r2ml:ObjectName r2ml:objectID="http://ontology.ihmc.us/
                    SemanticServices/S-F/Example/
                    ActorClasses.owl#ServiceProviderSupportingX509Certificates "/>
           </r2ml:contextArgument>
          </r2ml:ReferencePropertyFunctionTerm>
         </r2ml:object>
        </r2ml:ObjectSlot>
       <r2ml:ObjectSlot
        r2ml:referencePropertyID="&KaoSAction;performedBy">
         <r2ml:ObjectName r2ml:objectID=" ClientA "/>
        </r2ml:ObjectSlot>
 </r2ml:ObjectDescriptionAtom>
</r2ml:conclusion>
```

Figure 7. Generating the conclusion part of a R2ML rule (right) from a Rei policy (left)

in [17], the ObjectDescriptionAtom in our R2ML excerpt (Figure 5-right and Figure 7-right) for the policy language of Section 2 can be expressed as Figure 8. The given excerpt for the F-Logic transformation shows only the atom in the conclusion part of a rule, which illustrates the appropriate permission to be given. The ongoing efforts to provide transformations from R2ML to the other rule languages would add to the generality of R2ML and make it a suitable asset for the purpose of information exchange. For example, our transformations from Rei and KAoS to R2ML showed the need for having both existential and universal quantifiers defined for derivation rules, which currently only entail the universal quantifiers. The modifications to the language will appear in the next version of R2ML.

```
“Policy_CommunicationCertificatePermission” [
   hasAction → “CommunicationAction”;
   hasContext\_N10058 → hasPartner(
   “ServiceProviderSupportingX509Certificates”);
   performedBy → “ClientA”]
```

Figure 8. An R2ML element represented in F-Logic.

Policy-RuleML [14] is a similar attempt in the area of policy transformation. It aims at making RuleML a semantic interoperation vehicle for heterogeneous policies and protocols on the Web. However, to the best of our knowledge there is no proposed work done by this group, and thus our solution seems to be the first practical attempt in the area. The future goal of the research will be combining the semantic web service description languages with policy languages and trying to get different web services with different descriptions and policy languages to work together. The current proposals on combining semantic web services with policy languages have been proposed in [13] that combines WSMO and PeertTrust, [21] that uses KAoS to protect web services, and [7] that combines Rei and OWL-S. Our goal will be to let all of these services to communicate regardless of the languages they use for defining and describing their services and policies. Furthermore, we are working on developing transformations from OCL to R2ML [23]. Another goal will be to make all the transformations between policy languages and OCL consistent, so that we can eventually integrate Semantic Web policies for services with Model-Driven software development approaches. Developing a general policy language based on R2ML to cover the concepts of all abovementioned policy languages is also another future

References

1. Bonatti, P. et. al. (2006). “Semantic Web policies - a discussion of requirements & research issues,” *In* 3rd European Semantic Web Conference, Montenegro
2. Bonatti, P & Olmedilla, D (2005). “Driving & monitoring provisional trust negotiation with metapolicies”. *In IEEE 6th Intl. WSh on Policies for Dist. Sys. & Nets.*
3. Burstein, M., et. al. (2005) “A Semantic Web Services Architecture,” IEEE Internet Computing, 9(5), 72-81
4. Ginsberg, A. (2006). “RIF Use Cases and Requirements,” *W3C Wor. Draft*, http://www.w3.org/TR/rif-ucr/.
5. Hirtle, D., et al. (2006). “Schema Specification of RuleML,” http://www.ruleml.org/spec/
6. Horrocks, I., et al. (2004). “SWRL: A Semantic Web Rule Language Combining OWL and RuleML,” W3C Mem. Sub., http://www.w3.org/Submission/SWRL/.
7. Kagal, L., et al. (2004). “Authorization & Privacy for Semantic Web Services” *IEEE Intel. Sys.* 50-56,
8. Kagal, L., et al. (2003). “A policy language for a pervasive computing environment,” *In IEEE 4th In’l WSh of Policies for Dist. Sys & Nets*, pp. 6-74.
9. KAoS Policy Ontologies (KPO), online source : http://ontology.ihmc.us/ontology.html
10. Kaviani, N., et al. (2006). “Towards Unifying Rules and Policies for Semantic Web Services,” *In Proc. of the 3rd Annual LORNET Conf. on Intelligent, Interactive, Learning Object Repositories*, Montreal, QC, Canada.
11. Kaviani, N., et al. (2007). “Web Rule languages to Carry Policies”, 8th *IEEE Int WSh on Policies for Dist. Sys. & Nets*, Italy.
12. Nejdl, W. et al. (2004). “PeerTrust: automated trust negotiation for peers on the semantic web”. *In Proc. of the VLDB WSh on Secure Data Mang.*, Canada, 118-132.
13. Olmedilla, D. et al. (2004). “Trust negotiation for semantic web services,” *In Proc. of the 1st Int’l WSh on Semantic Web Services & Web Process Composition*, CA, USA.
14. (2004 Policy RuleML Tech. Grp, http://policy.ruleml.org
15. (2006). R2ML specification, http://oxygen.informatik.tu-cottbus.de/R2ML/,
16. (2006). Rei Ontology Specification, Version 2.0, http://www.cs.umbc.edu/~lkagal1/rei/
17. Giurca, A., Wagner, G., (2006). “Translating R2ML into F-Logic,” White Paper, http://oxygen.informatik.tu-cottbus.de/R2ML/0.3/R2ML-to-FLogic.pdf
18. Sheth, A. et al., “Semantics to energize the full services spectrum,” Comm. of the ACM, 49(7), 2006, pp. 55-61.
19. Toninelli A., et al. (2005). “Rule-based and Ontology-based Policies: Toward a Hybrid Approach to Control Agents in Pervasive Environments” *In Proc. of the Semantic Web & Policy WSh,* Galway, Ireland.
20. Tonti, G., et al. (2003). “Semantic Web Languages for Policy Representation and Reasoning: A Comparison of KAoS, Rei, and Ponder”. *In Proc. of the 2nd Int’l Semantic Web Conf.*, 419-437
21. Uszok, A., et al. (2003). “KAoS policy and domain services: toward a description-logic approach to policy representation, deconfliction, and enforcement,” *In Proc. of the 4th IEEE Int’l WSh on Policies for Dist. Sys & Nets.*
22. Uszok, A., et al. (2004). “KAoS: A Policy & Domain Services Framework for Grid Computing & Semantic Web Services”. *Proc. of 2nd Int’l Conf. on Trust Mgmt.*
23. Wagner, G. et al. (2006). “A Usable Interchange Format for Rich Syntax Rules Integrating OCL, RuleML and SWRL,” *In Proc. of WSh. Reas. on the Web*, UK.
24. XSLT transformations: http://cgi.sfu.ca/~nkaviani/cgi-bin/prjs.html
25. Youyong, Z., *et al.* (2004), "F-OWL: an Inference Engine for the Semantic Web ", *Book Formal Approaches to Agent-Based Sys.*