Model-Driven Engineering of a General Policy Modeling Language

Nima Kaviani\textsuperscript{1}, Dragan Gašević\textsuperscript{2}, Milan Milanović\textsuperscript{3}, Marek Hatala\textsuperscript{4}
\textsuperscript{1}University of British Columbia, Canada
\textsuperscript{2}Athabasca University, Canada
\textsuperscript{3}FON-School of Business Administration, University of Belgrade, Serbia
\textsuperscript{4}Simon Fraser University Surrey, Canada

nimak@ece.ubc.ca, milan@milanovic.org, dgasevic@acm.org, mhatala@sfu.ca

Abstract

Despite many efforts in developing policy languages which work in different logical domains and with various reasoning engines, there has been limited attention paid to bringing policy definition, design, and integration into the realm of the mainstream software development process. There seems to be a lack of appropriate software development tooling that can allow for easy representation and integration of policies with other pieces of software at the design time. This paper presents a General Policy Modeling Language (GPML), following the rationale of Model Driven Engineering (MDE), as a means to design policies and integrate them to the software development process. We describe the logical foundation and the modeling rationale behind GPML and show how it is adjustable to the existing policy languages.

1 Introduction

Inventing and offering more and more business services through the Web have created highly-dynamic environments in which today’s software systems have to work. To address the challenges of such environments, software engineers need to use novel software development and maintenance techniques. This includes techniques that address increased needs for software changes coming from either software’s internal business logic updates or external factors such as involved parties. We believe policies are promising solutions to address these problems from both internal and external perspectives.

Although policies have brought a lot of benefits into the realm of software development, the main challenge is how to integrate policies into the development process of software systems. This means that policies should be considered starting from the very beginning of software lifecycle stages, e.g. requirement engineering, to the last stages of software deployment and maintenance. Throughout the development process, various software languages (e.g., UML and WSDL) are used and policy languages need to be used in combination with all of them. The best way to do so is to provide an appropriate software development tooling [1] that facilitates the specification and integration process. That is, policies should complement existing languages, so that software engineers can easier comprehend a system under study, as they already have to deal with the burden of the complexity of systems.

In this paper, we propose the use of Model-Driven Engineering (MDE) [1] to address this problem. The goal of MDE is to focus on the problem under study, while hiding the complexities coming from the implementation details of different deployment platforms. Levering the MDE features, we define the general policy modeling language (GPML) by extracting common policy concepts from several policy languages and grounding them on the sound theoretical foundation of deontic logic, while keeping it away from the paradoxes known for deontic logic [6].

2 Model-Driven Engineering

The main goal of MDE is to switch the focus from low-level implementation details to problem-specific concepts [10]. The core activity is then to define languages for particular problem domains (in our case policies). Metamodeling is an approach used in MDE for defining languages. A metamodel is a model of a modeling language, that is, a metamodel defines a set of sentences that can be expressed in a modeling language [5]. As such, a metamodel can be regarded as an abstract syntax of a language.

Usually, a metamodeling architecture (e.g., OMG’s Model-Driven Architecture) is organized in a layered fashion, where typically there are three layers. The top most layer is called metamodel (and tagged with M3 or L3), and on this layer a metamodeling language is defined. Meta-Object Facility [7] and Ecore [2] are instances of this layer. The Definition of modeling languages (i.e., metamodels) is done on the metamodel layer (i.e., M2). To define a metamodel of a modeling
language (e.g., GPML), by using Meta-Object Facility [7] or Ecore [2], the UML graphical concrete syntax is used. Finally, concrete models are defined on the model layer (i.e., M1 or L1).

Model transformations are also an intrinsic part of MDE, as the idea is to transform M1 models to different platforms or to allow translation from one type of model (e.g., UML) to another (e.g., ODM). MOF Query/View/Transformation (QVT) is OMG’s official standard for model transformations [8]. While a metamodel is an abstract syntax, software developers need a concrete syntax of a language to be able to use it on the M1 layer as well. So, GPML may have more than one concrete syntax (graphical and textual), but the abstract syntax (metamodel) is always the same and its definition is fully decoupled from abstract syntax definitions.

### 3 GPML Requirements

Having introduced the basic concepts of MDE and specifying the needs for integrating policies into software development process, here we represent the main parts of GPML: i) General Policy Metamodel which is a MOF-based definition of GPML (i.e., an abstract syntax defined at the M2 layer). ii) General Policy UML Profile as a graphical concrete syntax of GPML; iii) XML-based textual concrete syntax for interchanging GPML policies. iv) A set of QVT transformations among the GPML metamodel and metamodels of other policy (e.g., XACML) and rule (e.g., Drools) languages to allow for deploying to concrete policy engines and integrating with business rules, respectively.

As policies are typically defined over business vocabularies and as they are also a part of a system’s business logic, they should be combined with business rules. We aim to have a general rule language that can allow for an easy deployment on, and sharing among, different rule engines. W3C’s Rule Interchange Format (RIF) initiative seems to be a sound approach to address the above requirements for GPML [3]. However, the initiative is still in its early stages of development. This is why we propose using REWERSE II Rule Markup Language (R2ML), a language that is designed to enable rule interchange as well as RIF. Hence, GPML is an extension of R2ML which specializes R2ML’s logical foundation, metamodel, and both graphical and XML-based textual concrete syntax.

### 4 R2ML: Policy Related Concepts

The full description of R2ML in the form of UML class diagrams is given in [9], while more details about the language can be found in [11]. R2ML consists of four main types of rules, i.e., derivation rules, production rules, integrity rules, and reaction rules; of which we describe derivation rules as one of the major assets in defining the policy rules.

A derivation rule has a set of conditions and a conclusion (see Figure 1) with the typical meaning that the conclusion can be derived whenever the conditions hold. While 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 (Negation as Failure, i.e., weak negation).

![R2ML’s derivation rule metamodel](image)

**Figure 1: R2ML’s derivation rule metamodel**

Conditions and conclusions are both defined by the use of Atoms that are the basic constituents of a formula in R2ML. A ReferencePropertyAtom associates object terms as “subjects” with other object terms as “objects”.

Terms are the basic constituents of atoms. Similar to atoms, the R2ML language distinguishes between object terms, data terms and generic terms. An ObjectTerm is an ObjectVariable, ObjectName, ReferencePropertyFunctionTerm, or ObjectOperationTerm.

R2ML also comes with a vocabulary that can be defined as a combination of Basic Content Vocabulary, Relational Content Vocabulary, and Functional Content Vocabulary to represent objects and classes, relations between them, and also corresponding functions.

### 5 General Policy Modeling Language

In this section, we present the theoretical foundation, the metamodel, the UML profile, and the model transformations of the GPML.

#### 5.1 Theoretical Foundation

To be able to fully cover the existing policy languages and their concepts by a general policy modeling language, we defined a layered approach for exchanging policies (see Figure 3). Looking back into the components of GPML, defined in Section 3, the interchange framework we represented in [4] can be used in parts (iii) and (iv) of GPML. As for part (iii) of GPML, the interchange framework helps with identifying the suitable concepts of R2ML to logically capture the semantics of various concepts in each policy language according to the underlying logic of the policy language (see Section 4) and by relying on R2ML’s rich set of elements to cover elements in different logical domains. In a similar way, for part (iv)
of GPML, the interchange framework helps with defining the appropriate QVT transformations from one policy language to another by recognizing the concepts in one policy language that either convey the same semantics or can be combined to carry equivalent semantics to the concepts of another policy language.

On the very bottom of the architecture from Figure 3, we identify the concepts for the underlying logic of each policy language which can be either rule-based logic programs (RL) or description logic (DL). RL and DL are the largest sets of logics, based on which different policy languages are defined. However, due to the differences in the constructs of these logics, the policies defined in one logic need to be regulated in accordance to the concepts of the other logic in order to make one policy rule operate in a different logical domain. This requires the concepts from DL to be mapped to RL and vice versa, in order to guarantee the proper exchange of policies across various logical domains. Relying on R2ML with its rich set of constructs to capture the concepts of either of the two logics, as the underlying rule markup language to define the policy rules, we can guarantee that our generated GPML model will preserve the logical intentions behind each policy rule.

The next layer in the layered framework is about policy concepts shared across multiple policy languages. These concepts are usually inspired by the modes of deontic logic and are synonymous to its modal concepts of permission, prohibition, obligation, and dispensation [6]. These four concepts are the major concepts that deontic logic is concerned with. For parts (iii) and (iv) of GPML, we have engineered a similar idea, by identifying the equivalent concepts or a series of concepts in one language that can convey the correct meanings of terms and elements from one policy language to another.

Finally, on the topmost layer, we handle the language specific concepts for each policy language.

5.2 GPML and its UML Profile

The abstract syntax for our GPML is defined by extending the MOF-based metamodel of R2ML. Figure 2 shows the GPML metamodel, which is an extension of the R2ML metamodel explained in Section 3. In the figure, we also show the icons used to graphically represent GPML constructs in the General Policy UML profile. As shown in the figure, a policy rule is defined as an R2ML derivation rule with its condition part composed of a conjunction of logical formulas and its conclusion part containing an R2ML ObjectDescriptionAtom. For each policy rule, this element holds the description for an instance of one of the modal concepts in the deontic logic, i.e., permission, prohibition, obligation, or dispensation. The policy instance represented by an ObjectDescriptionAtom conveys the derived policy decision upon satisfaction of the logical formulas in the condition. The condition part of each policy element holds the information about the actions, on which the deontic modal concepts operate, the actors for those actions, and the context of the action. By using R2ML’s ObjectClassificationAtom, we represent the class that each object belongs to and by using R2ML’s ReferencePropertyAtom we represent association relations in GPML. The metamodel has the following associations: performedBy, hasAction, hasContext, hasEffect, obliges, triggeredBy, location, time, and target. The performedBy association relates an action as the sub-
ject to an actor as the object. The \textit{hasAction} association relates a policy as the subject to an action as the object. Similarly, the \textit{hasContext} association relates the policy element as the subject to the context to which the policy is applied. It can also represent the context in which an action is performed. The \textit{hasEffect} element represents effects of applying a policy, for example to impose a penalty action. The \textit{triggeredBy} association represents the action upon its occurrence the policy is fired. \textit{Location} and \textit{time} respectively represent the location and the time where a policy is applied. Finally, the \textit{obliges} association is specific to the obligation policy rules and represents the obligatory task that an actor ought to perform upon execution of the policy.

We have developed a graphical concrete syntax for GPML. To do so, we have defined a UML stereotype for each GPML concepts represented in Section 4.2. As an example of the use of General Policy UML Profile, let us consider a policy in which “Only the doctor from the emergency section is allowed to access the medical test results of a patient and a notification email is required to be sent to the patient”. Figure 4 shows how the UML Profile of GPML represents the above policy.

5.3 GPML Transformations

Having the policy of Figure 4 defined in GPML, the next step would be to use the components in parts (iii) and (iv) of GPML to convert the graphical concrete syntax for the policy to its equivalent R2ML syntax and then to transform it to the target policy language. In Section 5.2 we explained the one to one mappings between the elements of GPML and their corresponding R2ML elements. The QVT graphical transformations from GPML to Rei, KAoS, and Ponder, as well as the sample transformation of the GPML policy of Figure 4 to R2ML and then to Rei can be found in http://www.ece.ubc.ca/~nimak/GPML/.

6 Conclusions

The proposed GPML allows for integration of policies into the software development lifecycle. Looking from a broader software engineering perspective, it directly fits the analysis and design phases of Unified Process, where UML class models are created. In our case, the graphical concrete syntax of GPML is defined over UML class models. However, our approach is fully based on the MDE concepts, which also enables for leveraging model transformations as well. This implies that our GPML models can directly be deployed to specific policy engines that can be used in the software operation phase. We demonstrated that by providing transformations between GPML and the KAoS, Rei, and Ponder2 policy languages (Sect. 5.4). We are also planning to extend GPML to support other types of policies such as delegation, request, and command. In the future, we will report on our lessons learned in using GPML with different policy languages. Finally, we are also going to investigate relations of GPML with other relevant MDE standards. In that respect, we will investigate how SBVR can be leveraged in requirement engineering to translate policies from structured English into GPML. Furthermore, as ODM has already been supported by mappings between OWL and R2ML vocabularies, we will leverage this to use GPML-based approach in designing Semantic Web-based (service-oriented) systems.

7 References