Transformation of graphical ECA policies into executable PonderTalk Code

Raphael Romeikat, Markus Sinsel, Bernhard Bauer

Angaben zur Veröffentlichung / Publication details:

Romeikat, Raphael, Markus Sinsel, and Bernhard Bauer. 2009. "Transformation of graphical ECA policies into executable PonderTalk Code." *Lecture Notes in Computer Science* 5858: 193–207. https://doi.org/10.1007/978-3-642-04985-9_19.

Transformation of Graphical ECA Policies into Executable PonderTalk Code

Raphael Romeikat, Markus Sinsel, and Bernhard Bauer

Programming Distributed Systems, University of Augsburg, Germany
{romeikat,sinsel,bauer}@ds-lab.org

Abstract. Rules are becoming more and more important in business modeling and systems engineering and are recognized as a high-level programming paradigma. For the effective development of rules it is desired to start at a high level, e.g. with graphical rules, and to refine them into code of a particular rule language for implementation purposes later. An model-driven approach is presented in this paper to transform graphical rules into executable code in a fully automated way. The focus is on event-condition-action policies as a special rule type. These are modeled graphically and translated into the PonderTalk language. The approach may be extended to integrate other rule types and languages as well.

1 Introduction

Increasing complexity of information systems complicates their development, maintenance, and usage. Due to this evolution, the Autonomic Computing Initiative by IBM [1] proposes self-manageable systems that reduce human intervention necessary for performing administrative tasks. For realizing autonomic capabilities within managed objects, policies are a promising technique. The idea behind policy-based management is allowing administrators to control and manage a system on a high level of automation and abstraction. According to [2], policies are an appropriate means for modifying the behavior of a complex system according to externally imposed constraints.

The focus of this paper is on a certain type of policy called Event-Condition-Action (ECA) policies. ECA policies are considered as reaction rules that allow for specifying which actions must be performed in a certain situation. They specify the reactive behavior of a system in response to events and consist of a triggering event, an optional condition, and an action term.

Policy-based management is also a layered approach where policies exist on different levels of abstraction. Wagner et al. consider three different abstraction levels [3]. The business domain level typically uses a natural or a visual language to define terms and constrain operations. The platform-independent level defines formal statements expressed in some formalism or computational paradigm, which can be directly mapped to executable statements of a software platform. The platform-specific level expresses statements in a specific executable language. Strassner defines a flexible number of abstraction layers as the Policy Continuum [4]. The idea is to define and manage policies on each level in a

domain-specific terminology, and to refine them e.g. from a business level down to a technical level.

An approach is presented that allows to graphically model ECA policies and transform those policy models into executable code. It uses techniques from model-driven engineering (MDE) to model policies in a language-independent way and to automatically generate code. Models are used to represent ECA policies based on common policy concepts that are represented in a generic metamodel. The policy language PonderTalk is also represented by a respective metamodel. Model transformations allow for generating executable PonderTalk code from an initial policy model. A full implementation of the approach exists as plugin for the software development platform Eclipse.

There have been other approaches for modeling information about policies and policy-based systems. The authors of [5] present a General Policy Modeling Language (GPML) as a means to design policies and map them to existing policy languages. This approach is also based on MDE concepts, but uses an UML profile for visualization and is based on the rule interchange language R2ML with a focus on logical concepts to map GPML policies onto existing policy languages. The Common Information Model (CIM) [6] by the Distributed Management Task Force (DMTF) represents a conceptual framework for describing a system architecture and the system entities to be managed. An extension to CIM to describe policies and to define policy control is provided by the CIM Policy Model [7]. Another type of information model is the Directory Enabled Networks next generation standard (DEN-ng) [8] by the TeleManagement Forum (TMF). DEN-ng is based on the Policy Continuum and considers different levels of abstraction. Policies are directly integrated into the models. Similar to the approach presented here, the CIM Policy Model and DEN-ng are independent of any policy language. They are as metamodels that enable the developer to describe a system and the enclosed policies in an implementation-independent way. Policies are specified in a declarative way while omitting technical details. However, only specification of policies is regarded in both approaches. They do not offer a possibility to transform a policy model to a particular language that can be executed by some engine. It remains an open issue to what extent PonderTalk and other policy languages are compatible with those policy models.

This paper is structured as follows. Section 2 gives an introduction to Model-driven Engineering and to the policy language PonderTalk. Section 3 describes the model-driven approach to transform graphical ECA policies into executable code. Section 4 describes the implementation of the approach. The paper concludes with related work and a summary in section 5.

2 Basics

This section presents a short introduction into model-driven engineering, which represents the foundation of the approach, and into the policy language PonderTalk, which is the target of the transformation.

2.1 Model-Driven Engineering

In software engineering one can observe a paradigm shift from object-orientation as a specific type of model towards generic model-driven approaches, which has important consequences on the way information systems are built and maintained. The model-driven engineering approach follows multiple objectives: apply models and model-based technologies to raise the level of abstraction, reduce complexity by separating concerns and aspects of a system under development, use models as primary artifacts from which implementations are generated, and use transformations to generate code with input from modelling and domain experts [9,10]. Model-driven solutions consist of an arbitrary number of automated transformations that refine abstract models to more concrete models (vertical model transformations) or simply describe mappings between models of the same level of abstraction (horizontal model transformations). Finally, code is generated from lower-level models. Models are more than abstract descriptions of systems as they are used for model and code generation. They are the key part of the definition of a system.

2.2 PonderTalk

Ponder2 [11] is a policy framework developed at Imperial College over a number of years. A set of tools and services were developed for the specification and enforcement of policies. Ponder2 offers a general-purpose object management system and includes components that are specific to policies.

Everything in Ponder2 is a managed object. Managed objects generate events and policies are triggered by those events to perform management actions on a subset of managed objects. This is also called local closed-loop adaptation of the system. There are managed objects that are available by default to interact with the basic Ponder2 system, i.e. factory objects to create events and policies. Besides that, user-defined managed objects are implemented as Java classes and used within Ponder2. Managed objects can send messages to other managed objects and new instances of managed objects can be created at runtime.

ECA policies are called obligation policies in Ponder2 and are specified with the language PonderTalk. PonderTalk has a high-level syntax that is based on the syntax of Smalltalk and is used to configure and control the Ponder2 system. Basically, everything in Ponder2 can be realized with PonderTalk, i.e. define and load managed objects, specify policies, or throw events that trigger policies. In order to realize a policy system in Ponder2, the respective PonderTalk code has to be implemented.

Example Scenario. Now, an example scenario is presented where ECA policies are used to manage the behavior of a communication system. Further sections will refer to this scenario when presenting examples.

The signal quality of wireless connections is subject to frequent fluctuations due to position changes of sender and receiver or to changing weather conditions. A possibility to react to those fluctuations is adjusting transmission power. A

good tradeoff between transmission power and signal quality is desired. Too high transmission power causes additional expenditures whereas signal quality suffers from too less transmission power.

In that scenario signal strength is managed autonomously by a policy system using ECA policies. A *Transmitter* adjusts transmission power with the actions *increase_power* and *decrease_power*, both of them expecting a value by which power should be increased and decreased. Whenever a change in signal quality is noticed, an *intensityChange* event is thrown that contains the *id* of the affected receiver and the signal quality's *oldValue* and *newValue*.

Two ECA policies *lowQuality* and *highQuality* are responsible for adjusting transmission power. They are triggered whenever an *intensityChange* events occurs and in their condition check the old and new signal quality enclosed in the event. If the transmission power falls below a value of `50`, the *lowQuality* policy executes a call of `increase_power(10)` to increase transmission power by 10 at the *Transmitter*. The other way round, the *highQuality* policy executes a call of `decrease_power(10)` at the *Transmitter* if transmission power goes beyond a value of 80.

The behavior of the transmission system can new be adjusted at runtime via the policies. The accepted signal quality is specified in the conditions of the two determining policies by means of the two boundaries 50 and 80. Changing those boundaries has immediate effect on the transmission power and signal quality.

3 Modeling and Transforming ECA Policies

In this section the overall approach to graphically model ECA policies and transform those policy models into executable code is presented. Various aspects have to be considered for the approach to be effective. Figure 1 illustrates how the various aspects of the approach are related to each other.

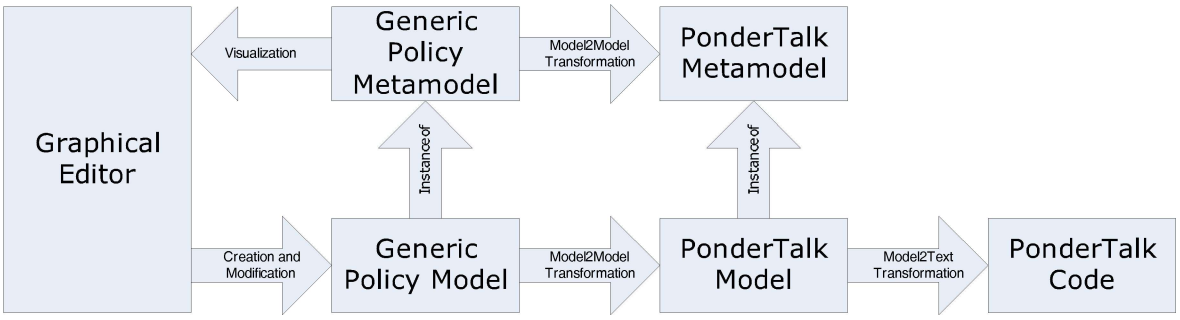

Fig. 1. From Graphical Policies to PonderTalk Code

First, a generic policy metamodel contains common concepts of ECA policies. It abstracts from special features and technical details that are specific to a certain policy language and thus allows to specify ECA policies independently of a particular language. Any ECA policy is initially represented as instance of that

metamodel to offer an abstract view onto the policy from a functional point-of-view. As only common concepts are contained in the generic policy metamodel, such a generic policy model can be transformed into executable code of a concrete policy language later. Next, the concepts of the generic policy metamodel have a graphical representation so the generic policy model is visualized as a diagram. A graphical editor offers functionality to create and modify models in a comfortable way.

Once an ECA policy has been modeled as a diagram, transformation into the target language can start. The starting point for defining that transformation is the generic policy metamodel, and a metamodel for the target language, namely the PonderTalk metamodel. As no formal metamodel was available for the PonderTalk, a metamodel was created from the language documentation [12]. A model-to-model transformation is defined on the metamodels and executed on the model. It takes the generic policy model as input and generates the respective PonderTalk model as output, which is an instance of the PonderTalk metamodel. Finally, a model-to-text transformation takes the PonderTalk model as input and generates a textual representation of that policy containing the respective PonderTalk code.

The following subsections present further details about the metamodels, the graphical visualization, and the model transformations. Various aspects will be illustrated by means of the example scenario presented in section 2.2.

3.1 Generic Policy Metamodel

The generic policy metamodel comprises common concepts of well-known policy languages such as PonderTalk [11], KAoS [**?**], and Rei [**?**]. It covers the essential aspects of those languages and contains classes that are needed to define the basic functionality of an ECA policy, i.e. events, conditions, and actions, amongst others as described in the following. The generic policy metamodel is specified as Essential MOF (EMOF) model. EMOF is a subset of the Meta Object Facility (MOF) [**?**] that allows simple metamodels to be defined using simple concepts. EMOF provides the minimal set of elements that are required to model object-oriented systems. Figure 2 shows the generic policy metamodel as UML class diagram.

The class *Entity* represents the components of the policy system. Each *Entity* has a *name* attribute and three more technical attributes. Those attributes may contain code fragments that are specific to the target language and that need to be included into the generated code so it is executable. In case of PonderTalk, *accordingClass* e.g. specifies the name of the respective Java class implementing that *Entity* as managed object in Ponder2. This is somehow contrary to the aspect of language independency, but on the other side it is a simple possibility to to generate code that is executable without further modification.

Entities can be organized in a *Domain* hierarchy, similar to the folders of a file system. A *Domain* is a collection of *Entities* that belong together with regards to content. Events, conditions, and actions can also be contained in a *Domain*. A *Domain* is an *Entity* itself as it can also be controlled by *Policies*.

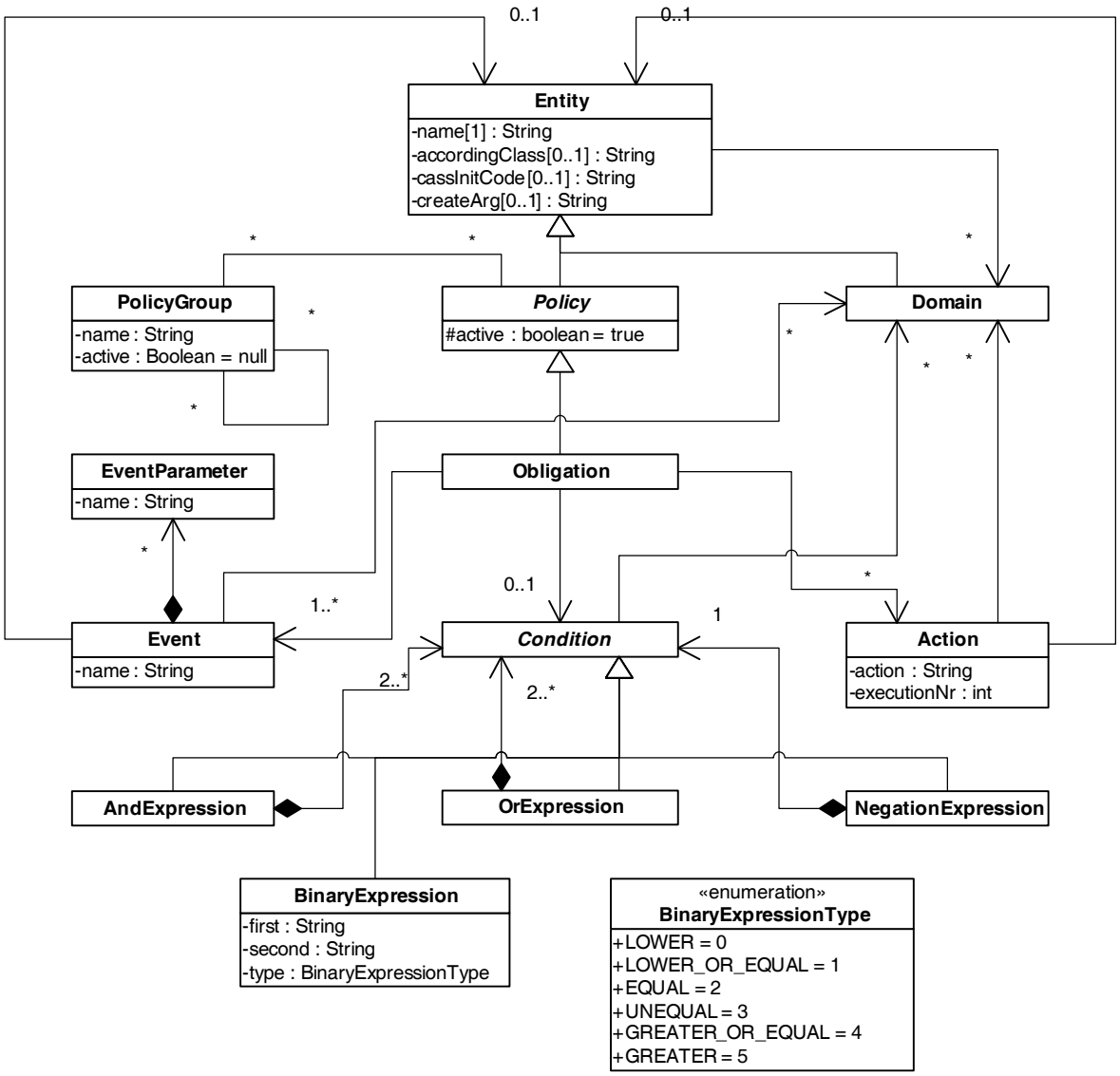

Fig. 2. Generic Policy Metamodel

The class *Policy* is the abstract superclass of all types of policy, whereas only one concrete type *Obligation* is included so far, representing ECA policies. The *active* attribute of a *Policy* describes its status and may be set to `true` or `false`. Only active *Policies* are triggered at runtime in a policy system.

Policies can be grouped in *PolicyGroups*. Groups are named and can contain other groups. In contrast to a *Domain*, a *PolicyGroup* has an administrative purpose and is used for activating and deactivating a set of *Policies* all at once using the *active* attribute of the group, which may be set to `true`, `false`, or `undefined`. The active status of a *Policy* is determined by the closest *PolicyGroup* in the group hierarchy with *active* set to `true` or `false` and where that *Policy* is contained. If at the respective level of the hierarchy some active and inactive *PolicyGroups* contain that *Policy*, the policy is regarded as being active. The *active* attribute of a *Policy* is only deciding if all *PolicyGroups* which contain this *Policy* have *active* set to `undefined`.

The classes *Event*, *Condition*, and *Action* represent the actual content of an *Obligation*. Each *Obligation* requires at least one *Event*, optionally has a *Condition*, and has an arbitrary number of *Actions* associated. An *Obligation* is triggered by at least one *Event* whereas at runtime the occurrence of one respective *Event* suffices to trigger that *Obligation*. An *Event* can be thrown by any *Entity*, has a *name*, and can contain a set of parameters represented by the class *EventParameter*. *EventParameters* are named and can be referred within the *Condition* that is associated with the respective *Obligation*.

A *Condition* is a boolean expression. A *BinaryExpression* is the simplest form of a *Condition* and compares two strings with each other. These strings represent the left land side (LHS) and right hand side (RHS) of the *Condition*, denoted by the attributes *first* and *second*. Those strings can contain the name of an *EventParameter*, which allows to analyze the *Event* that triggered the *Obligation*. Or, they can directly contain a simple value in the form of an enclosed string or numeric value. The comparison operator is defined by the attribute *type* and may be one of $<$, $\leq$, $=$, $\neq$, $\geq$, and $>$. An expression can additionally be negated using the class *NegationExpression*, or combined as conjunction or disjunction using the classes *AndExpression* and *OrExpression* respectively.

If the *Condition* of an obligation evaluates to true, the associated *Actions* are executed. Executing an *Action* means calling an *Operation*. The attribute *action* within the class *Action* specifies which *Operation* is called. The attribute *executionNr* must be used to denote the sequence of execution if two or more *Actions* are associated with an *Obligation*. Arbitrary numbers may be used as long as they are different from each other. They need not be consecutive, which provides some flexibility when associating multiple *Actions* to multiple *Obligations*.

3.2 PonderTalk Metamodel

As a next step, the PonderTalk metamodel is defined as the target of the model-to-model transformation. That metamodel is again specified as EMOF model; figure 3 shows it as UML class diagramm. It refers to the current version 2.840 of Ponder2 and contains only those concepts that are needed to represent an ECA policy in PonderTalk. Other functionalities of PonderTalk such as authorization policies are not adressed as they go beyond the expressiveness of the ECA policy metamodel. In the following, the PonderTalk metamodel is described with respect to its differences to the generic policy metamodel.

In PonderTalk an *Entity* is called *ManagedObject*. Apart from naming there is no difference between those two classes. The same applies to an *Obligation*, which is now called *ObligationPolicy*. The classes *Domain*, *Policy*, *Condition*, *BinaryExpression*, *NegationExpression*, *AndExpression*, *OrExpression*, and *BinaryExpressionType* do not differ from the generic policy metamodel.

PonderTalk does not know the concept of groups. Thus, a way has to be found to represent *PolicyGroups* when transforming into PonderTalk. This has an effect on the *active* attribute of a *Policy* and is described later.

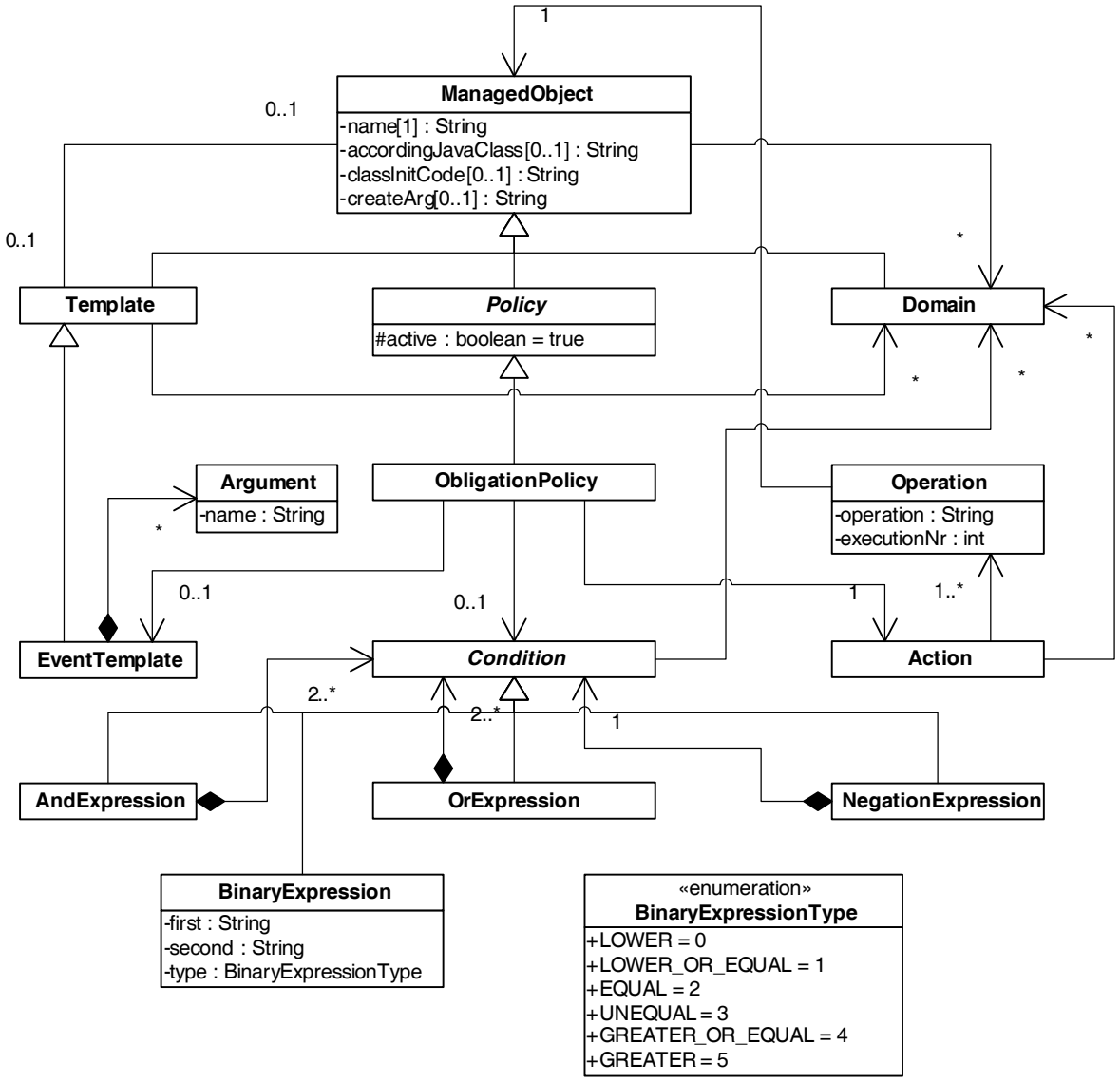

Fig. 3. PonderTalk Metamodel

On the other hand, PonderTalk introduces a new class *Template*. *Templates* are used to create new instances of *ManagedObjects*, *Policies*, or *Domains*. A *Template* itself is also a *ManagedObject*.

An *Event* in the generic policy metamodel is called *EventTemplate* in PonderTalk. An *EventTemplate* can contain an arbitrary number of named *Arguments*, which represent the respective *EventParameters*. A noticeable difference is that an *ObligationPolicy* in PonderTalk cannot be triggered by an arbitrary number of *EventTemplates*, but is triggered by at most one. This is taken into consideration by the transformation later. Additionally, an *EventTemplate* is an instance of *ManagedObject* in PonderTalk.

The condition part of an *ObligationPolicy* exactly corresponds to the generic policy metamodel, but there are important differences in the action part. An *ObligationPolicy* in PonderTalk does not execute an arbitrary number of *Actions*, but executes exactly one *Action*. An *Action* uses at least one *Operation*

to execute commands on a *ManagedObject*. The attribute *operation* is used to specify a particular PonderTalk command.

3.3 Graphical Visualization

Now, a graphical representation of a policy is created as a diagram. For this purpose, the classes of the generic policy metamodel that were instantiated when modeling the policy are visualized with all necessary information. Abstract classes in the metamodel do not have a graphical representation as no instances of them can be created. A visualization of the classes in the PonderTalk metamodel is not required either as that metamodel is only used as intermediate step in the transformation later and needs not be available as a diagram.

Figure 4 shows the graphical representation of the example scenario presented in section 2.2. Additionally to the scenario description, the two policies *lowQuality* and *highQuality* are put into a policy group named *quality*.

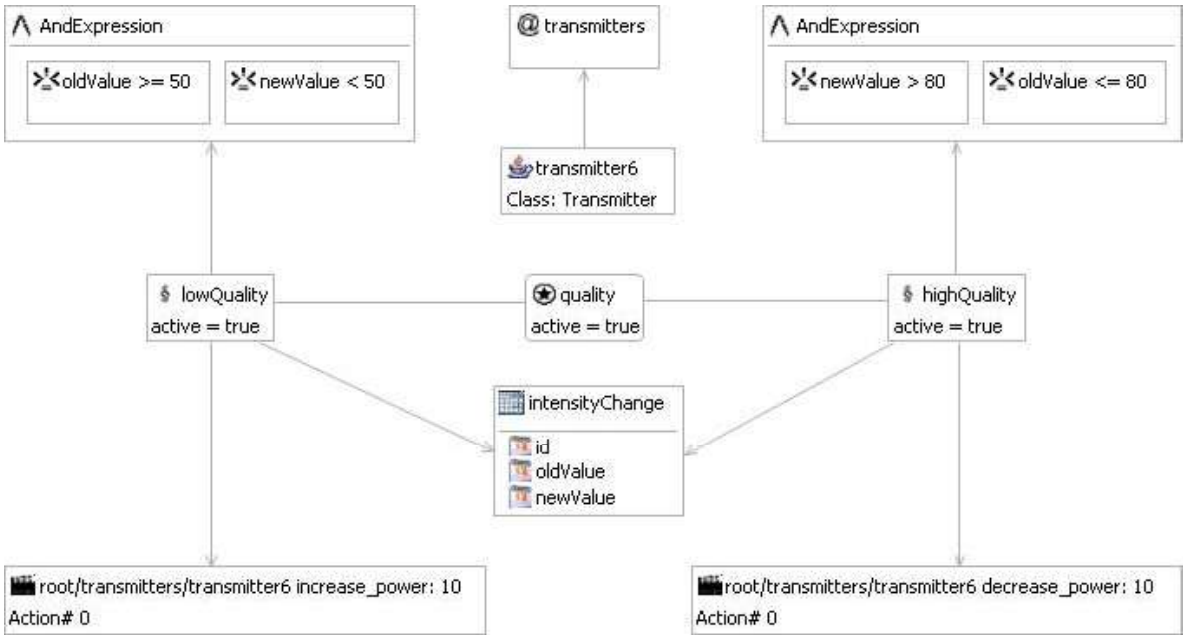

Fig. 4. Visualization of a Generic Policy Model

For visualizing the instantiated classes simple rectangular shapes were chosen that ressemble the way classes are visualized in UML. As header of each shape, a symbol and an identifying text are displayed to characterize it. That text contains the *name* attribute if existant in the respective class. For an action, the *action* attribute is used instead. For a binary expression, a textual representation of its attributes is used to visualize the LHS, RHS, and the operator, and for the other expression classes, the name of the class itself is used.

Further details of the classes are displayed in the shape body, which usually contains the attributes with their value. Event parameters are not visualized as rectangular shape, but they are visualized within the body of the enclosing

event, which can be seen with the event *intensityChange* and the contained event parameters *id*, *oldValue*, and *newVaule*. Conditions that are used within another condition are directly visualized within the body of the parent condition as shown with the two *AndCondition*s, which both contain two binary conditions. This way of integrating event parameters and nested conditions reduces the overall number of shapes in the policy diagram.

Finally, associations between classes are displayed as directed lines as known from UML class diagrams. The direction represents the visibility of the classes as defined by the respective associations in the metamodel.

The chosen way of visualizing classes and associations omits complex shapes and technical details, so it focuses on the essential information and developers should easily get familiar with it. In section 4 a graphical editor is presented that allows to create ECA policies based on the generic policy metamodel and the graphical visualization.

3.4 Model Transformations

An ECA policy is now specified as generic policy model using the generic policy metamodel and the graphical visualization. The next step is generating a representation of that policy as PonderTalk code as an implementation for Ponder2. For this purpose, model transformations take the generic policy model as input and generate the respective PonderTalk code. The overall transformation process is divided in two steps. First, the generic policy model is transformed into a PonderTalk model. That model is in a second step transformed into PonderTalk code. The necessary transformations are summarized in figure 5 and described in the following.

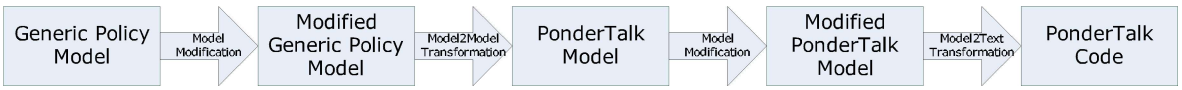

Fig. 5. Model Transformations

From a Generic Policy Model to a PonderTalk Policy Model. When transforming a generic policy model, a check is performed first whether the model fulfills the structural requirements of the metamodel with respect to the cardinalities of the associations. Furthermore, domains must not contain themselves nor contain two domains that are named equally. The same applies to policy groups. All entities of the model must have a name and names must be unique amongst obligation policies and amongst direct entities (without subtypes). Finally, any action must specify its action attribute. If all checks are passed, the model is well-formed and ready to be transformed.

Now, a model modification is executed to modify the source model. A modification does not create a new model as target, but the result of the modification is the modified source model itself. Model modifications are used in order to

enrich a model with additional information that was not modeled explicitly, or to modify details of a model to simplify further transformations. In the generic policy model, an *Obligation* can be triggered by various *Events* whereas in the PonderTalk model only one *EventTemplate* is allowed per *ObligationPolicy*. For this purpose, a model modification duplicates *Obligations* with two or more associated *Events* into several *Obligations* of with each one is associated with one of the original *Events*. The rest of the *Obligation* is duplicated without changes. This structural change allows the straightforward generation of the PonderTalk model from the modified generic policy model.

Then, a model-to-model transformation takes the modified generic policy model as input and generates the respective PonderTalk model as output. For this purpose, the transformation translates the concepts of the generic policy metamodel in a way so they are expressed by the concepts of the PonderTalk metamodel. The transformation is defined on the classes of the metamodel and is executed on the instances of those classes in the model. As a result, the PonderTalk model is generated as follows.

First, all instances of *Entity* are transformed into *ManagedObject*s one after another. The attribute values of an entity are copied to the respective managed object. Transforming the *Entities* includes transforming *Domain*s and *Obligation*s as they are *Entities* as well. When transforming an *Entity*, any associated *Entity* (i.e. the domain of an entity) is transformed immediately, and this is a recursive process.

It is important to notice that an *Entity* can be referenced multiple times by other *Entities* and whenever one reference is processed, the transformation of that *Entity* is called straightforward. However, a caching mechanism ensures that an *Entity* is actually transformed only once and with any further transformation call to the same *Entity*, the cached result is used instead. This ensures that any model element is created only once in the target model and the model elements need not be processed in a special sequence during the transformation.

*Event*s are transformed into *EventTemplate*s and *EventParameter*s into *Argument*s. In contrast to the generic policy metamodel, *EventTemplates* are subclasses of *ManagedObject* in PonderTalk, so the generated *EventTemplates* are internally marked to be *ManagedObjects* as well. Transforming the *Condition*s is performed by simply copying them as no differences exist between the two metamodels with respect to the condition part. *Action*s are transformed into *Operation*s. The *Action* objects in the PonderTalk model are created newly. For each *ObligationPolicy*, one *Action* object is created and associated with that *ObligationPolicy*.

When transforming *Obligation*s, the associated *PolicyGroups* are processed along the group hierarchy to determine the active status of the *Obligation* as described in section 3.1. *PolicyGroups* do not have a representation in PonderTalk and thus no more appear in the PonderTalk model. Their only purpose for the transformation was to determine the active status of policies whose *active* attribute was undefined.

From a PonderTalk Policy Model to PonderTalk Code. Executable PonderTalk code requires the standard *Domains* **root**, **policy**, and **event** to be

```
// Create Domains
root at: "transmitters" put: root/factory/domain create.

// Create event intensityChange
event := root/factory/event create: #( "id" "oldValue" "newValue" ).
root/event at: "intensityChange" put: event.

// Load the Transmitter class file
root/factory at: "transmitter" put: ( root load: "Transmitter" ).

// Create an instance named transmitter6 and put it in each associated domain
instance := root/factory/transmitter create.
instance intensityChangeEvent: root/event/intensityChange.
root/transmitters at: "transmitter6" put: instance.

// Create policy lowQuality
policy := root/factory/ecapolicy create.
policy event: root/event/intensityChange;
       condition: [ :id :oldValue :newValue | ((oldValue >= 50) & (newValue < 50)) ];
       action: [ :id :oldValue :newValue | root/transmitters/transmitter6 increase_power: 10 ].
root/policy at: "lowQuality" put: policy.
policy active: true.

// Create policy highQuality
policy := root/factory/ecapolicy create.
policy event: root/event/intensityChange;
       condition: [ :id :oldValue :newValue | ((newValue > 80) & (oldValue <= 80)) ];
       action: [ :id :oldValue :newValue | root/transmitters/transmitter6 decrease_power: 10 ].
root/policy at: "highQuality" put: policy.
policy active: true.
```

Listing 1.1. Generated PonderTalk Code

specified. A PonderTalk model might not explicitly contain those *Domains*. For this purpose, a model modification checks whether they are modeled and if not, inserts them into the model. Furthermore, that modification adds any *Obligation-Policy* that is not contained in the **policy** *Domain* to that *Domain* and it also ensures that any *EventTemplate* is contained in the **event** domain. Finally, it adds any *Domain* that is not contained in another *Domain* to the **root** *Domain*.

Now, a model-to-text transformation takes the modified PonderTalk model as input and generates the respective PonderTalk code as output. That transformation is also called code generation as it generates code for a programming language. The transformation defines for each class of the PonderTalk metamodel a respective textual representation as PonderTalk code. When the transformation is executed on the PonderTalk model, the respective code for the enclosed classes is generated step by step. Listing 1.1 shows the resulting PonderTalk code that corresponds to the policy diagram shown in figure 4.

In PonderTalk it is important to specify the statements in the correct sequence. *Domains* must first be declared before they can be referenced by other *ManagedObjects*. For this purpose, a sorting algorithm initially creates an ordering of the *Domains* along the hierarchy and ensures that code for the **root** *Domains* is generated before proceeding with the next level in the hierarchy, etc. Now, code for all domain declarations is generated with respect to that ordering. It is also worth to be mentioned that the transformation does not generate any code for the top-level *Domains root*, *policy*, and *event* as Ponder2 internally creates those *Domains* at startup before any PonderTalk code is executed at all.

Now, code for *EventTemplates* is generated. In PonderTalk a factory object is used to create an *EventTemplate* together with the enclosed *Arguments*. In the PonderTalk model, any *EventTemplate* is associated with the **event** *Domain*, which also results in a respective PonderTalk statement.

As next step, *ManagedObjects* (without subtypes) are transformed into code. For any *ManagedObject*, the respective Java class specified in the *accordingClass* attribute is loaded as factory object and put into the respective factory domain. For *ManagedObjects* that are associated with a *Domain* in the PonderTalk model, an instance is created additionally and added to that *Domain*. Arguments required for instantiation are specified in the *createArg* attribute of the *ManagedObject* and are added to the statement that creates the instance. *ManagedObjects* that are not associated with any *Domain* are only loaded as factory. This is be useful if instances of *ManagedObjects* should only be created at runtime.

Finally, code for *ObligationPolicies* is generated including the referenced *EventTemplate*, *Condition*, and *Action*. First, an *ObligationPolicy* is created with the policy factory. Next, the triggering *EventTemplate* is associated with that *ObligationPolicy*. If the *ObligationPolicy* contains a *Condition*, a textual representation of that *Condition* is generated for PonderTalk. Next, the *Action* is transformed into appropriate code including the referenced *Operations* in the sequence as defined by their attribute *executionNr*. Finally, the *ObligationPolicy* is put into all associated *Domains* and the status of the *ObligationPolicy* is set according to its *active* attribute.

4 Implementation

In order to demonstrate the approach, an implementation was developed as a set of plugins for the software development platform Eclipse. The implementation is called *PolicyModeler* and can be integrated into any Eclipse 3.5 (Galileo) installation via the update site `http://policymodeler.sf.net/updates`. Alternatively, a complete Eclipse installation including the PolicyModeler is available for instant usage at `http://policymodeler.sf.net/eclipse.zip`. This section presents important aspects about the implementation.

For specifying the metamodels, the Eclipse Modeling Framework (EMF) [16] is used in the PolicyModeler. EMF stores the specified metamodels in the Ecore format, which is an implementation of EMOF. With EMF a tree-like editor is generated to create and modify a metamodel as well as instances of that metamodel. However, that editor was not used for the creation of the generic policy metamodel and the PonderTalk metamodel; instead, annotated Java interfaces were used as they are a more effective way to specify metamodels in EMF.

For the graphical representation of the policies the Eclipse Graphical Modeling Framework (GMF) [17] is used. GMF offers the generation of a graphical editor that allows to create and modify a generic policy model as a diagram. For the generation of that graphical editor some input is required. First, the generic policy metamodel is referenced as instances of that metamodel are to be visualized. Second, a graphical representation is created for each model element to define its visualization in the diagram. Third, a toolbar is defined that offers means to create model elements and associations between them. Finally, all input is combined to define which element of the toolbar is used to create which model element and how that model element is visualized in the diagram.

The model transformations are developed with the Eclipse Modeling Framework Technology (EMFT) [18] and Model To Text (M2T) [19] projects, which support the implementation of various kinds of model transformations. They offer all functionality required for the transformations used in the approach, i.e. special languages to realize checks, model modifications, model-to-model transformations, and model-to-text transformations. The projects are available as Eclipse plugins themselves and thus offer good integration with the PolicyModeler.

An ECA policy is created in the PolicyModeler by composing the desired model elements into a diagram using the toolbar. If a generic policy model already exists as Ecore file without a graphical representation, that graphical representation can be generated automatically by the PolicyModeler. The transformation into corresponding PonderTalk code can be started directly from the diagram. The resulting code can then be executed within a Ponder2 installation.

5 Conclusion

In this paper an approach to graphically model ECA policies and generate executable code for the language PonderTalk from such models was presented. It is the first the approach innovative and transfers benefits of MDE to the development of policies such as reduction of development time. The generic policy metamodel allows to model ECA policies independently from a particular language and allows to generate code in an automated way. PonderTalk is used as target language, but the approach may be extended to target other policy languages by integrating the metamodel of the respective language and setting up the necessary model transformations. The metamodel covers important features of ECA policies. A developer might require expressiveness for his policies that is not covered by the metamodel. Further policy types and concepts may be integrated by extending the generic policy metamodel so it can express more than only ECA policies. However, a tradeoff must be made here. Full code generation is only possible if the target language can represent all concepts of the generic policy metamodel in an appropriate way. This is why the metamodel is basically limited to the important concepts of ECA policies. Adressing more policy languages and types are subject to future work. The same applies to reverse engineering of PonderTalk code into a graphical model, which is currently not possible. The approach is fully implemented as Eclipse plugin [20]. A small example was shown, but the approach was also applied to a larger case study that realizes the hospital scenario from the Ponder2 tutorial [21]. That case study regards all structural details of the two metamodels and is included in the Eclipse download mentioned in section 4.

References

1. Kephart, J.O., Chess, D.M.: The Vision of Autonomic Computing. Computer 36(1), 41–50 (2003)
2. Damianou, N., Dulay, N., Lupu, E., Sloman, M.: The Ponder Policy Specification Language. In: Sloman, M., Lobo, J., Lupu, E.C. (eds.) POLICY 2001. LNCS, vol. 1995, pp. 18–38. Springer, Heidelberg (2001)

3. Wagner, G., Antoniou, G., Tabet, S., Boley, H.: The Abstract Syntax of RuleML - Towards a General Web Rule Language Framework. In: IEEE/WIC/ACM International Conference on Web Intelligence, pp. 628–631. IEEE Computer Society Press, Los Alamitos (2004)
4. Strassner, J.C.: Policy-Based Network Management: Solutions for the Next Generation. Morgan Kaufmann Publishers, San Francisco (2003)
5. Kaviani, N., Gasevic, D., Milanovic, M., Hatala, M., Mohabbati, B.: Model-Driven Engineering of a General Policy Modeling Language. In: IEEE Workshop on Policies for Distributed Systems and Networks, pp. 101–104. IEEE Computer Society, Los Alamitos (2008)
6. Distributed Management Task Force: Common Information Model (CIM) Specification. DSP0004 (June 1999)
7. Distributed Management Task Force: CIM Policy Model White Paper. DSP0108 (June 2003)
8. Strassner, J.C.: DEN-ng: Achieving Business-Driven Network Management. In: IEEE/IFIP Network Operations and Management Symposium, pp. 753–766. IEEE Computer Society, Los Alamitos (2002)
9. Bézivin, J.: On the Unification Power of Models. Software and Systems Modeling 4(2), 171–188 (2005)
10. Flater, D.W.: Impact of Model-Driven Standards. In: Annual Hawaii International Conference on System Sciences, vol. 9, pp. 3706–3714. IEEE Computer Society, Los Alamitos (2002)
11. Twidle, K., Lupu, E., Dulay, N., Sloman, M.: Ponder2 - A Policy Environment for Autonomous Pervasive Systems. In: IEEE Workshop on Policies for Distributed Systems and Networks, pp. 245–246. IEEE Computer Society, Los Alamitos (2008)
12. Imperial College London: Ponder2. (June 2009), `http://ponder2.net`
13. Uszok, A., Bradshaw, J.M., Jeffers, R.: KAoS: A policy and domain services framework for grid computing and semantic web services. In: Jensen, C., Poslad, S., Dimitrakos, T. (eds.) iTrust 2004. LNCS, vol. 2995, pp. 16–26. Springer, Heidelberg (2004)
14. Kagal, L., Finin, T., Joshi, A.: A Policy Language for a Pervasive Computing Environment. In: IEEE International Workshop on Policies for Distributed Systems and Networks, June 2003, pp. 63–74 (2003)
15. Object Management Group: Meta Object Facility (MOF) Core Specification (January 2006), `http://www.omg.org/spec/MOF/2.0/PDF`
16. The Eclipse Foundation: Eclipse Modeling Framework (EMF). (June 2009), `http://www.eclipse.org/modeling/emf`
17. The Eclipse Foundation: Graphical Modeling Framework (GMF) (June 2009), `http://www.eclipse.org/modeling/gmf`
18. The Eclipse Foundation: Eclipse Modeling Framework Technology (EMFT) (June 2009), `http://www.eclipse.org/modeling/emft`
19. The Eclipse Foundation: Model To Text (M2T) (June 2009), `http://www.eclipse.org/modeling/m2t`
20. University of Augsburg: PolicyModeler (August 2009), `http://policymodeler.sf.net`
21. Imperial College London: Ponder2 Tutorial (May 2009), `http://www.ponder2.net/cgi-bin/moin.cgi/Ponder2Tutorial`