Domain-specific transformation of the REA enterprise ontology

Zdenek Melis  
University of Ostrava  
Czech Republic  
e-mail: Zdenek.Melis@Osu.cz

Jaroslav Zacek  
University of Ostrava  
Czech Republic  
e-mail: Jaroslav.Zacek@Osu.cz

Frantisek Hunka  
University of Ostrava  
Czech Republic  
e-mail: Frantisek.Hunka@Osu.cz

Abstract—The paper deals with the general description of the methodology of the transformation of basic concepts of the REA enterprise ontology (Resource, Event, Agent) from the initial visual modeling interface to model source codes. The paper describes the structure of the script of the domain-specific language (DSL) and its subsequent transformation into the executable source code using abstract classes for defining the structure of the template code. The aim is the creation of the basic structure of the concept of templates and layout the code generated from the created model. The model is due to excessive complexity limited to the use of three basic concepts of REA enterprise ontology.

Keywords—REA ontology; DSM; DSL; transformation

I. INTRODUCTION

Due to the increasing complexity of information systems, demands for the control clarity and the simplicity are increasing. Technologies that use a visual environment come to the fore. One of the leading technologies based on a visual programming is a domain-specific modeling (DSM) [1]. It focuses on one particular domain, which provides the syntax and the semantic of the visual language and the ability to transform created models into the executable source code [1]. Business process modeling is one of complex problems that use the visual modeling technology. With regard to the structure of the DSM and code generation possibilities, the most appropriate technology of the description of business processes seems to be the REA ontology [2]. The object-oriented structure of the REA ontology allows transformation of models into other structures, such as source code or database schema [8, 9, 10].

II. DOMAIN-SPECIFIC MODELING

Domain-specific modeling is a software engineering methodology for software development [1]. The primary focus of DSM is automated development of applications in a specific domain based on principles of visual programming. Unlike traditional modeling principles that have a universal focus, DSM is based on the entirely specific modeled domain.

The DSM architecture is three layered, although individual layers overlay to each other. The highest and for the user the most visible layer is the language. The narrow focus on a specific domain allows language to correspond by its whole structure to domain terms. Syntactic aspects of the language are not limited to textual form, but they can take any visual form representing concepts of particular domain. Unlike generally focused traditional principles, semantic aspects of the language are contained. The second layer is the generator, which transforms created model into language of the target platform. The last layer is the domain framework. It creates a supportive environment for the generator, which includes many features, such as defining the interface for the generator, integration of the generated code into the system, removing duplicate parts, and many others.

The main advantage of DSM is the automated transformation without mapping that is a frequent source of errors. Using of the domain language brings simplicity and easy manipulation with tool even for non-technical users who can intuitively use the tool through the knowledge of the problem domain. The toughest stage in the development process is creating of the modeling tool. Once the tool is developed, using this tool for creating software is very simple and fast. It can increase development productivity up to ten times [5, 6].

III. THE REA ONTOLOGY

The REA enterprise ontology is the concept for designing and model creation of enterprise infrastructures. It is based on resource ownership and its exchange. The aim of most businesses is the profit generation and therefore they must ensure the effectiveness, the efficiency and the adaptability of their business processes and it cannot be done without their modeling and subsequent analysis [4]. There are many of business processes modeling tools, but due to inadequate level of abstraction and the use of general concepts they are not usable enough for the business process modeling. Rather than general modeling techniques companies use expensive software created directly to the specific requirements of a particular enterprise.

REA ontology does not use general concepts but specific. They increase the amount of represented data, while maintaining the simplicity of the model. The REA ontology model offers 4 levels of abstraction [2, 7]. The highest level is Value System Level, which represents view of the flow of
resources between the company and its business partners. The second level is Value Chain Level describing links between business processes within a company. The third level, REA model level, describes a specific business process and represents the change in the value of resources. The various concepts of this level can be divided into two groups – the operational level and the policy level. The operational level includes the basic concepts describing the specific facts and events that have already happened. Here are included concepts forming the name of this ontology - economic resource, event and agent. The policy level is an extension of the operational level by concepts specifying rules or allowing planning. The lowest level of an abstraction is the Task level, which describes the model instance level, making it an implementation-dependent.

This paper deals only with the transformation of the operational level of the REA model level. It has three basic concepts: economic resource, event and agent. The resource represents an economic material that the business wants to plan, manage and monitor. It can include products, money, services or for example workmen. The economic agent is a person, group of people or the whole company that has control over resources. The economic event describes incremental or decrement change of resource. From the model perspective the economic event is key for preserving information, because it determine who, when, why and how the resource was changed [3].

One of reasons for choosing the REA ontology is object-oriented structure support that is necessary for successful transformation. The REA ontology model also includes internal rules for verifying the consistency of the model, ensuring correctness of created links. At the same time models are simple and understandable for ordinary users who will work with it, but sufficiently precise for its automation [2].

IV. MODEL TRANSFORMATION
The transformation itself consists of several steps. At the beginning the user creates a model of a business process in the visual interface. This interface contains methods for ensuring the basic model validation by preventing incorrect links creation alternatively prevent execution of the second phase - generation.

During the second phase the DSL script is generated from a visual model, containing basic elements of the structure of the model and on this basis the source code is generated.

A. Visual interface
The user creates a business process models in the visual environment that provides the basic user interface and performs the validation and the partial verification of the model. It provides basic semantic correctness of the model and prevents the generation of incomplete structures. The visual interface contains the common label of an entity type and its name and basic data attributes defining its properties.

For simplicity and clarity only basic entities representing resources, events and agents are used.

For simplicity and clarity only basic entities representing resources, events and agents are used.

Figure 1: Visual representation of the model

The header of each element contains the name and the type of entity. Under the header there are attributes with a predefined structure specified by the type of entity. If necessary the user can expand, or completely change the structure by adding new attributes or changing existing ones. The entity Car can serve as an example. In addition to basic attributes Name and Amount the entity is expand by an attribute representing a specific serial number of the car and Value that represents the value that the resource has for the company (for example total production costs).

The value on the link between the resource and the event indicates the amount of resources that is changed within the event.

B. DSL script
Once the model is completed and validation criteria are met, a domain-specific language (DSL) script constituting the regulation for the generator is created. The script is basically the export of the created model into XML format with the omission of data related to the visual interface of the model (such as the placement of elements, their size, color ...).

The following code fragment represents the DSL script showing the transfer of the resource entity from the model shown at Figure 1:

```
<resource title="Car">
    <id type="int">1</id>
    <name type="String">Audi A4</name>
    <sn type="String">35A76C38</sn>
    <amount type="int">1</amount>
    <value type="int" currency="USD">7200</value>
</resource>

<resource title="Money">
    <id type="int">2</id>
    <name type="String">Money</name>
```

Figure 1: Visual representation of the model
The code contains new items added by user - value that represents the value of the economic resource and currency for defining the exchange of funds handled by business.

Individual links are merged with the appropriate entities. An example may be the Stackflow link (link between the Resource and the Event) containing information on the amount of increase / decrease of resources within a single event. This attribute is converted into an Event entity. As mentioned before, the main carrier of information is an Event entity, which has a significant role in obtaining data from the model. For this reason, most of links of this model moves into this entity. Each entity has a unique ID, which is used for the unique identification of the element and for replacing individual links by creating reference to that ID.

The following code fragment shows one of event entities:

```xml
<event title="Sale">
  <id type="int">1</id>
  <name type="String">Sale</name>
  <eventType type="EventType">decrement</eventType>
  <agentReceiveID type="int">8</agentReceiveID>
  <agentProvideID type="int">2</agentProvideID>
  <resourceID type="int">1</resourceID>
  <date type="String">8.12.2002</date>
  <amount type="int">1</amount>
</event>
```

**EventType** attribute is determined by the type of link provide/receive, if the event is from the business point of view incremental or decrement and according to these links agents are determined too.

Exchange duality saves all references to events connected with duality.

```xml
<exchangeDuality>
  <id type="int">1</id>
  <eventId type="int">1</eventId>
  <eventId type="int">2</eventId>
</exchangeDuality>
```

C. Source code generator

The last phase of the transformation itself is generating the source code from the created DSL script. Creating a complete general generator would be inefficient and implementation difficult due to the unchanging structure of REA ontology elements. Fixed domain structure ensures durability and stability of domain terms, which allows predefining the general structure of basic elements. Due to their limited number it is possible to predefine a common part of the code that is same in all cases of the use of the entity as a template. In case of the above model, which is restricted for using of only three basic entities, two types of templates are used - an abstract class and a class template.

An abstract class contains the basic code structure, layout of methods and predefined basic attributes that the given entity must contain. In the case of transformation of the model restricted to using only three entities the same number of abstract classes are fully enough. When extending the model by other semantic abstractions, the number of abstract classes is not linear to the number of used entities because some entities may have more abstract classes based on specific uses of that entity in the model. Basic abstract classes are therefore AbstractAgent, AbstractEvent and AbstractResource.

The abstract class defining the agent entity contains the basic attributes *id, name* and *company* and their access methods, as shown in Figure 2.

![Abstract class for the agent entity](image)

These attributes are common to all agents and can be individually extended by additional parameters specifying a particular agent. Using default parameters is not mandatory, but it is recommended. The creator of the model can ignore these parameters and creates new one. The only restriction is that variable names defined in the parent class cannot be used again with different data type.

The abstract class for a resource entity is defined in a similar way (Figure 3). *Amount* attribute indicates the amount of resources from the business perspective. The other attributes (*id* and *name*) are the same as attributes in AbstractAgent class.
AbstractResource
- amount: int
- id: int
- name: String
+ getAmount(): int
+ getId(): int
+ getName(): String
+ setAmount(int): void
+ setId(int): void
+ setName(String): void
+ toString(): String

AbstractEvent
- agentProvideID: int
- agentReciveID: int
- amount: int
- date: java.util.Calendar
- eventType: EventType
- id: int
- name: String
- resourceID: int
+ getAgentProvideID(): int
+ getAgentReciveID(): int
+ getAmount(): int
+ getDate(): String
+ getEventType(): EventType
+ getId(): int
+ getName(): String
+ getResourceID(): int
+ setAgentProvideID(int): void
+ setAgentReciveID(int): void
+ setAmount(int): void
+ setDate(String): void
+ setEventType(EventType): void
+ setId(int): void
+ setName(String): void
+ setResourceID(int): void
+ toString(): String

Figure 3: Abstract class for the resource entity

The last abstract class is AbstractEvent that creates a draft for the event entity (see Figure 4). Unlike the other two abstract classes it contains the most methods because the event is carrier of basic properties of the model. Attributes agentProvideID, agentReciveID and resourceID store links with individual agents and a resource.

AbstractEvent
- agentProvideID: int
- agentReciveID: int
- amount: int
- date: java.util.Calendar
- eventType: EventType
- id: int
- name: String
- resourceID: int
+ getAgentProvideID(): int
+ getAgentReciveID(): int
+ getAmount(): int
+ getDate(): String
+ getEventType(): EventType
+ getId(): int
+ getName(): String
+ getResourceID(): int
+ setAgentProvideID(int): void
+ setAgentReciveID(int): void
+ setAmount(int): void
+ setDate(String): void
+ setEventType(EventType): void
+ setId(int): void
+ setName(String): void
+ setResourceID(int): void
+ toString(): String

Figure 4: Abstract class for the event entity

In addition to basic attributes such as id and name the attribute amount also figures here. It indicates the amount of resources that is changed within the event. Another necessary attribute is date, recording the date of the past event. Calendar class is intended for its preservation, but the output of the visual interface returns the date as a string and therefore it is necessary to convert the date inside access methods. The last parameter is eventType, that specifies the type of an event, whether it is from the business perspective incremental or decrement. It is determined by a constant of enumerator EventType, which is part of this class.

Abstract classes are used to define basic parameters for the generated class. They are generated using class templates. They operate on a simple principle: instead of generating the whole structure of the code, such as headers of classes, methods, etc., the appropriate template that contains all of these structures will apply and on the basis of the script missing data will be add. Part of data can be added by just replacing non-terminal symbol for the specific name from the script, other data provides a simple automaton according to the specified grammar. Non-terminal symbols are written in a template as non-terminal name between two percent signs. The following code shows a template for a resource entity:

```java
public class %className% extends
AbstractResource{

%attributesDeclaration%

/*
 * Default constructor
 */
public %className%(int id, String
name, int amount){
    setId(id);
    setName(name);
    setAmount(amount)
}

/*
 * Empty constructor
 */
public %className%(){

%fullConstructor%

%attributesGetterSetter%

public String toString(){
    return super.toString()%toString%;
}
}
```

Generator processes the template by its division by the `%` character to array strings - tokens. Each token is compared with the list of keywords and if the comparison did not find any results, the token is written to the file.

If the token is recognized as a keyword, the output of automaton corresponding to that keyword is written to the file. In above template there are many non-terminals. The first of them is an attribute `%className%`, which is
The attribute \%attributesDeclaration% is responsible for the declaration of new variables. This must be performed by automaton using the following grammar:

\%attributesDeclaration%:
  \%attributesDeclaration% -> private
  \%attribDataType% \%attribName%\%;\n
\%attributesDeclaration% -> %e%

The grammar has 2 rules. Until there are other undefined variables, the first rule is used, but once all new variables from the script are processed, the second rule is used. Known data types defined at an abstract class are skipped by the generator at this stage. Non-terminal \%attribDataType% is replaced by the appropriate data type corresponding to the attribute type in the DSL script in the section behind the element name. The element name itself is used to replace \%attribName% attribute. Non-terminal %n% is used by automaton as the command new line and %e% is empty non-terminal, in this case the automaton ends and the generator continues processing other parameters.

Another non-terminal symbol in the template is \%fullConstructor%, that is used to generate the constructor containing all used variables. Its structure is defined by following grammar:

\%fullConstructor%:
  \%fullConstructor% -> public \%name%(%int
  id, String name, String
  company\%attributes%){
    this(id, name, company);
    \%setAttributes%}
  \%fullConstructor% -> %e%

\%attributes%:
  \%attributes% -> ,\%attribDataType%
  \%attribName% \%attributes%
  \%attributes% -> %e%

\%setAttributes%:
  \%setAttributes% ->
  this.set\%attribName%(%attribName%);
  \%setAttributes%}
  \%setAttributes% -> %e%

The processing of this non-terminal occurs only when the DSL script contains new attributes, otherwise the output would be identical to the default constructor. That is why this structure is implemented by using the non-terminal instead of fixed placement in the template. The first part of the template generates the general structure of the constructor with fixed set of variables of the abstract class.

In the header of the method there is non-terminal \%attributes% which generates a list of all newly added variables including their data types separated by commas.

The body of method starts with calling the default constructor. Then \%setAttributes% non-terminal is used for generation of the code to ensure the assignment of method’s input values to particular variables.

The last non-terminal in the template is \%toString%, which is used for completing toString() method. This non-terminal is performed only if the element contains new attributes.

%toString%:
  %toString% -> +"\n\%attribName%:
      +get\%attribName%()"
  %toString% -> %e%

After the template processing is complete, the output is the class containing the complete source code corresponding to the particular element in the visual interface:

```java
public class Money extends
AbstractResource{
  private String currency;

  /**
  * Default constructor
  */
  public Money(int id, String name, int amount){
    setId(id);
    setName(name);
    setAmount(amount);
  }

  /**
  * Empty constructor
  */
  public Money(){}

  public Money(int id, String name, int
  amount, String currency) {
    this(id, name, amount);
    setCurrency(currency);
  }

  public void setCurrency(String
  currency){this.currency = currency;}
  public String getCurrency(){return
  this.currency;}

  public String toString(){
```
In a similar way templates for agents and events are created. In addition to these classes the new class is generated, which creates instances of created classes and fills them with data. By extending this class there is a space for the instance level of visual modeling or to extension the model by the simulation ability.

V. DISCUSSION ABOUT USING OF ABSTRACT CLASSES

Using abstract classes can greatly simplify the process of the source code generating, because it is not necessary to generate repeatedly general and often used structures. The disadvantage of this solution is the loss of generality of generated models. Globally, it is not possible to determine the exact structure of abstract classes of the REA ontology, because it depends on the specific modeled business process. It is only possible to determine the estimated parameters, but not mandatory parameters. For example, the agent has estimated parameter Name. An abstract class defines it as a String, but the model creator may require an instance of some object. Although it is possible to add a new attribute, it is necessary to choose a different label of the attribute. The question is, when it is appropriate to use an abstract class? If the usage of complex data structures in attributes is not expected, the application of abstract classes of will significantly facilitate the creation of the generator. On the other hand, if generality of models and their greater modeling expressiveness is required, then the application of abstract classes is not recommended.

ACKNOWLEDGMENT

The paper is supported by the grant reference no. 6141 provide by IGA Faculty of Science University of Ostrava.

REFERENCES