We are IntechOpen, the world's leading publisher of Open Access books Built by scientists, for scientists

4,800
Open access books available

122,000
International authors and editors

135M
Downloads

Our authors are among the

154
Countries delivered to

TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities

CLARIVATE ANALYTICS BOOK CITATION INDEX INDEXED

WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com

10

Application of Automata Based Approach for Specification of Model Transformation Strategies

Anna Derezińska and Jacek Zawłocki
Institute of Computer Science, Warsaw University of Technology
Poland

1. Introduction

Model transformation gains the increasing attention in software development and finds application in different areas. Model-to-model transformation can be specified in many different ways, see surveys in (Czarnecki & Halsen, 2006; Ehring at. al., 2005; Mens & van Gorp, 2006). Model transformation languages are developed within the MDA (Model Driven Architecture) initiatives, like QVT languages (MOF QVT Specification, 2008). Other solutions adopt often different, known methods for specification of transformation rules; like graph transformations, term rewriting, algebraic approaches. However, practical application of various types of model transformation, both corresponding to MDA and using other methodologies, is still not fully justified by the experiences gathered in practical projects.

We dealt with manipulation of models describing object-oriented projects. We were interested in the separation of different levels of transformation logic and in the support by a simple transformation engine. Therefore, we combined concepts of meta-modeling graph with extended automata and used for specification of a transformation strategy. The automaton-like graphical models are well understandable by engineers and computer programmers (Hopkroft at al,. 2000).

The main part of a transformation strategy consists of rules. Any rule will be specified by an extended automaton. Transitions between nodes are annotated with enabling conditions and actions. The set of rules is organized in a hierarchical structure. The structure is defined by a graph, conceptually similar to a meta-model graph. Association of the rules to vertices of the graph introduces priorities of the rules. A rule can be executed when its pre-condition is satisfied. We provide the basic definitions of the concepts of a strategy.

An automata-based transformation strategy was applied in the generic framework for traceability in object-oriented designs (Derezinska & Zawlocki, 2007). The framework is aimed at discovering relationships in a given object-oriented model according to a given traceability strategy. Requirements of the framework focus on the generic and highly flexible solutions. It accepts different modeling notations (including UML or its subsets) and

supports adaptation to different strategies according to user needs. The goals of the framework are realized by a chain of three transformations:
1) input transformation - from an object-oriented model to a model in the internal notation (so called Project notation),
2) traceability transformation - from a Project model with a given initial element to the resulting model (in this case so-called Dependency Area),
3) output transformation - from a Dependency Area to a resulting model in a desired notation.
The input and output transformations were realized using QVT approach and presented in (Derezinska & Zawlocki, 2008). This paper is devoted to the second transformation of the framework.
The general idea of Dependency Areas is based on exploration of possible direct and indirect relations among elements of a model. It can be viewed as a kind of traceability when impact relations between model elements are concerned. Identification of a dependency area can be realized as a transformation of an input model into another model.
The design of an executor of rules, used in the framework, is described with its meta-model and a draft of its algorithm. Exemplary strategies were specified for the subsets of UML meta-models. Few examples of the automata defining selected transformation rules will be presented. We show basic implementation of the strategy specification used in the framework and its application. We discuss also the advantages and disadvantages of the approach, its background and related work.

2. Related work

2.1 Model transformation and rule specifications

A model transformation is a mapping of a set of models onto another set of models or onto themselves, where a mapping defines correspondences between elements in the source and target models (Sendall at al., 2004). The description of a transformation is usually concentrated on a set of rules - basic units of the transformation. There can be static and dynamic rules, organized in different structural patterns and scheduled according to different schemata, as summarized in (Czarnecki & Halsen, 2006).
Within the OMG initiative, Query View Transformation (QVT) specification is developed that provides a set of formalisms to transform one object-oriented model into another one (MOF QVT Specification, 2008). QVT supports a relational as well as operational mappings language. The standardization effort triggered development of tools supporting the transformation process, but application of the standard is not yet common in industrial solutions. The QVT operational notation was applied for the input and output transformations of the framework considered in this paper (Derezinska, 2008).
There are many different approaches proposed for the specification of transformations; some of these include: relational/logic, functional, graph rewriting, generator/template-based, and imperative (Czarnecki & Halsen, 2006; Ehring at. al., 2005; Mens & van Gorp, 2006). Transformation rules were also described in specification languages like Object Constraint Language (OCL), Object-Z, B, Maude, etc. Transformations can be also implemented using general-purpose languages such as Java. A developer can access a model directly using any API and is not limited, but also not supported, by any rules. Decomposition of a transformation into a chain of basic units is discussed in (Vanhoof at al.,

2007a). The distinction between specification, implementation and execution of a transformation process is logical and similar to our approach.
Similarly to QVT some other approaches combine also different formalisms, like operational, declarative and hybrid transformation rules in ATL (Jouault & Kurtev, 2005). A transformation mechanism can be general, dealing with different kinds of models, or specialized to a given notation, often a subset of UML. In the USE environment (Buttner, & Bauerdic, 2006) transformations objects are added to the meta-level representations of UML models. The operational semantics is given in transformation meta-classes. Epsilon Wizard Language (EWL) supports update transformation for diverse modeling languages (Kolovos at al., 2007).
One category of transformations is based on graph transformations (Ehring at. al., 2005). A graph transformation rule is specified by two graph patterns - Left Hand Side and Right Hand Side. Graph transformation rules that can be scheduled according to a given state machine specification are, for example, supported in Viatra2 tool (Balogh & Varro, 2006). More examples of graph transformations can be found in (Ehring & Giese 2007).
In our solution we do not propose a new transformation language, but combine different notations on various abstraction levels. Information from a meta-model is stored in a hierarchy and used for prioritization of rules - therefore influences their scheduling. A rule is given in a graph notation, but it is not a rewrite rule. The details of the logic are specified in conditions and actions that can be expressed in a commonly known language, as for example Java in our implementation.
In different domains, rule specifications are also needed, like in knowledge management or business process description. Similarly, as in the case of model transformations, such rules can be specified by any known formalism, e.g. Prolog, Process Algebras, automata, decision tables or graphs, but also model-based notations. Proposal of *Production Rules Representation* dealing with business rules modeling as a part of modeling process is currently under development (W3C Recommendations, 2008). The UML notation, or its variants, like e.g. URML (Wagner at al., 2006), supported by constraints in OCL can be also used for specification of business rules. However, in this paper we do not discuss the problem of usage of the UML notation for rule description, but of automata-based rules used in object-oriented models and these models are illustrated by UML model examples.

2.2 Applications of extended automata

An automaton (Hopkroft at al,. 2000) is a mathematical model of a finite state machine, which definies traversals through a series of states accorging to a transition function. A name of extended automata is applied in many various contexts. It can describe different kinds of extensions of simple automata. These extensions refer usually to data labeling of transitions and mappings of these data. Extended automata can model finite state machines which are enriched with the ability to apply a string operation on a part of the input that has not been consumed yet (Bensh at. al., 2008).
In the context of the UML language, UML statecharts can be described using different kinds of extended automata approaches. For example, in MDA approache presented in (Dayan et al., 2008) standard UML statecharts are transformed into corresponding extended automata called Hierarchical Statechart Automaton (HSA). In this case the main extension refers to the substitution of the hierarchical composite states of statecharts by a combination of simple automata, and parallel execution of orthogonal states using so-called Parallel HSA. The idea

of transformation of UML statecharts into an equivalent format is widely used for the verification methods based on model checking. The models can be, for example, transformed into Extended Hierarchical Automata (Lakhnech, 1997). Further a model is translated into a language accepted by one of model checkers - CadenceSMV (McMillan, 2008).
It should be stressed that transformations of statechart and/or usage of different automata to description of statechart semantics are other application areas than discussed in this paper.

2.3 Traceability strategies

One of the challenges of complex software development is maintaining traceability links among model elements to support model evolution and roundtrip engineering (France & Rumpe, 2007). Automatic recognition of different relations among project elements can be especially important for industrial projects that are usual incomplete and/or inconsistent at different stages of software development. Those relations can be interpreted as a special kind of traceability concepts.
Traceability deals in general with different impact relations between various artifacts created in the process of software development. Traceability can assist in improving the quality of a project and in the project maintenance (Maeder, 2006). It can be used for tracing the requirements and the changes in a design (Spanoudakis, 2004). Traceability links are examined for the UML models at different levels of model abstraction (Letelier, 2002) and in the model refinement process (Egyed, 2004). Different relations are specified directly within the development and evolution process or are automatic (or quasi-automatic) derived basing on the interpretation of information hidden in the software artifacts. In (Walderhaug et al., 2006) the set of services: trace model management, trace creation, trace use and trace monitoring are discussed. The services could support any kinds of artifacts and relations in a heterogeneous MDD (Model Driven Development) environment.
Several other papers described also solutions about traceability within MDD, like in (Vanhooff et. al., 2007b) were generated traces provided information that could be further used in a transformation between models. This problem is therefore opposite to the one presented in this paper and also in (Derezinska, 2008) because we shown an application of MDD in a framework for traceability.

3. Automata-based description of transformation strategy

In this section the general ideas of the model transformation strategy will be presented. The application of these concepts for the traceability strategy in object-oriented designs is shown in the next section.

3.1 Basic concepts of strategy

The core of a strategy is defined by a set of rules. Any rule will be specified as an extended automaton. The rules have their pre-conditions and priorities. The set of rules is organized in a hierarchical structure. The rules are associated to vertices of the hierarchy. An algorithm of a rule execution takes into account a position in the hierarchy. Now, a transformation strategy will be described in more details.

Definition 1. An *extended automaton* is a tuple EA = <S, Σ, DC, C, DO, Γ, δ, iN >, where
S is a finite set of *states* (*nodes* of the automaton graph),
iN∈ S is the *initial state (initial node),*
Σ is a set of *input* symbols,
DC is an n-dimensional linear space,
C is a set of *enabling functions* called *conditions* C_i: DC→{False, True},
DO is an m-dimensional linear space,
Γ is a set of *update functions* called *actions*, γ_i : DO→DO ,
δ : S x Σ x DC x DO → S x DO is a *transition* relation.

We will consider as a set of input symbols Σ a subset of non-negative integer numbers {0, 1, 2, ... k}. Input symbols will be interpreted as priorities of the transitions. The extended automaton can be also viewed as a kind of a labeled transitions system with a finite number of states and a finite number of transitions, or as an extended finite state machine.

Definition 2. A rule is a tuple r = <EA, DC, C_r, p>, where
EA is an extended automaton,
DC is an n-dimentional linear space,
C_r: DC→{False, True} is an *enabling function* called *precondition* of the rule,
p is a non-negative integer number, a priority of the rule.

Precondition of a rule and conditions of transitions are predicates given in any notation. They evaluates to the Boolean values. Domains DC and DO can be arbitrary chosen for different applications of extended automata. An action can be performed during a given transition only if the appropriate condition is satisfied.
Below, we describe a structure that help organizing the set of rules and identify their priorities.

Definition 3. A *simple, finite graph* is a tuple G = <V, E, α>, where
V is a finite set of vertices,
E is a finite set of edges,
α : V x V → E is a function assigning an edge to a pair of verticies

Definition 4. A *model* is a tuple X = <G, Ψ, φ>, where
G = <V_G, E_G, α_G> is a *simple graph,*
Ψ is itself a *model* (so-called reference model of X) with its graph <V_Ψ, E_Ψ, α_Ψ >,
φ : $V_G \cup E_G \rightarrow V_\Psi$ is a function assigning elements (vertices and edges) of G to vertices of Ψ (meta-elements).

Model Ψ is a meta-model of the model X. The above definitions are general, as the concepts of meta-modeling. In fact, we would be interpreting the meta-modeling in the similar way as in the specifications of OMG, like UML (Unified Modeling Language, 2008). We assume that dependency graphs between conceptual classes are formed by unidirectional generalisation relationships. Therefore, for example, vertices V of graph G can be interpreted as classes and edges E between nodes represent relations between classes.
Any meta-model is also a model and can have its own meta-model. In special cases, graphs of a model and a meta-model can have the same sets of nodes, like in the MOF (Meta Object Facility, 2006).

Further in the strategy, we will take into account a model, which is a meta-model consisting of classes as graph vertices and a generalization relation as only relation between vertices. In general, we can specify such a model extending its definition with the statement that a graph G is a *directed acyclic graph* (Deo, 1974). Such a model will be called *a hierarchical model.* In strategies, a hierarchical model has usually a tree structure, but it is not limited to such structures.

Definition 5. A *hierarchical model* is a tuple $X = \langle G, \Psi, \varphi \rangle$, where
$G = \langle V_G, E_G, \alpha_G \rangle$ is a *directed acyclic graph*,
Ψ is itself a *model* (so-called reference model of X) with its graph $\langle V_\Psi, E_\Psi, \alpha_\Psi \rangle$,
$\varphi : V_G \cup E_G \rightarrow V_\Psi$ is a function assigning elements (vertices and edges) of G to vertices of Ψ (meta-elements).

Combining a set of rules (Def. 2) with a hierarchical model we can define a strategy.

Definition 6. A strategy is a tuple $\Xi = \langle X, R, \mu \rangle$, where
$X = \langle\langle V_G, E_G, \alpha_G \rangle, \Psi, \varphi \rangle$, is *a hierarchical model* - representing a meta-model
R is a finite, nonempty set of *rules*,
$\mu : V_G \rightarrow R$ is a surjective function, assigning a rule to a vertex of the meta-model.

If a meta-model vertex has assigned a priority, it is also a priority of the corresponding rule. Different vertices may have priorities of the same value.

3.2 Interpretation of rules in transformation process

A transformation process can be understood as the realization of appropriate subsets of rules from a strategy in a desired order. A transformation can be realized by a simple rule executor, because the entire logic is stored in the rules.
Main concepts of a rule executor are shown in Figure 1. The rule executor operates on a collection of rules. Each rule is defined by an automaton. It has a priority specified according to a mapping to the hierarchical model. Any rule has also its precondition. An automaton consists of a set of nodes. For any node a set of outgoing transitions can be specified. A transition has its priority. There is a condition and an action associated with a transition.
The rule executor is responsible for execution of rules. Appropriate operations in the conceptual model represent the execution of rules, the execution of its nodes, evaluation of conditions, and performing of actions. All rules are ordered according to their priorities and executed in the defined order. If two or more rules have the same priority the order of their execution is random.
Before executing a single rule, its precondition is checked. If it is satisfied, the initial node of the rule is considered. In case a node has more than one outgoing transitions, they are ordered according to their priorities. If a condition of a selected transition is satisfied, the transition is followed and its action performed. All nodes of the rule accessible from its initial node can be visited during the rule execution.
The application of the rule executor to the model transformation and its algorithms used in the framework are presented in the next Section.

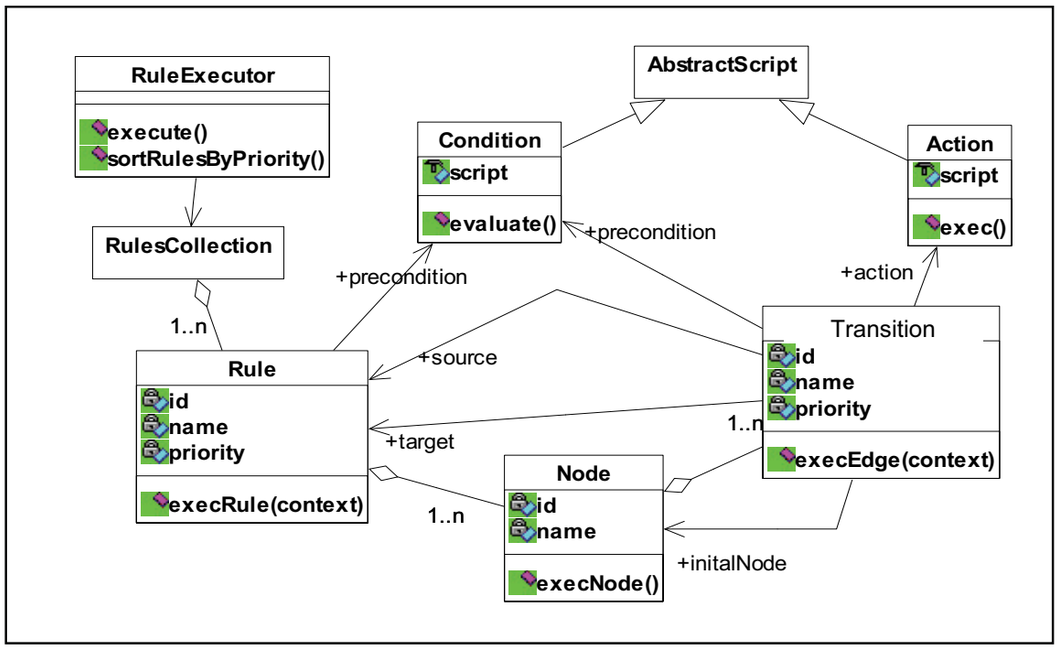

Fig. 1. Main concepts of the rule executor

4. Model transformation in the generic framework for traceability in object oriented projects

4.1 Background

Dependency areas (formerly named dependency regions) were introduced to deal with dependency relationships between subsets of related elements in UML models (Derezinska, 2004). The idea was aimed at the imperfect UML models; because real-world UML projects are often incomplete or inconsistent on different stages of software development. Therefore dependency relationships can be identified on the basis of different explicit relations present in a UML model, as well as reasoning based on supposed developer intentions derived from the model.

The idea of dependency areas was applied in a preliminary system supporting traceability in object-oriented designs (Derezinska, 2004). The system integrated CASE tools for requirements analysis (Rational Requisite Pro), and UML modeling (Rational Rose). Requirements could be associated with corresponding use cases. Dependency areas of a use case could be identified. Elements belonging to a considered dependency area were marked with stereotypes and could be further instrumented during code generation, if necessary. Rules used for identification of dependency areas were specified using structures similar to decision tables. In the tables, possible combinations of UML elements and their relations were taken into account. This solution was precise, but not very flexible, because it was strictly dependent on a given version of the UML specification.

The prototype system was implemented supporting a subset of UML - elements of use case models, class models, components and packages. The subset was limited, but it covered all elements, which were further used in the model to code transformation. The system was used in experiments tracing changes of requirements or model elements to the code of a

target application. Such impact analysis was, for example, used for the selection of related code extracts. Next, fault injection experiments were performed only on these selected parts of the code. The impact of changes on fault susceptibility of the application was therefore effectively measured.

The concepts of dependency areas were further generalized (Derezinska & Bluemke, 2005; Derezinska, 2006). Dependency areas were identified using the sets of propagation rules and strategies. Selection strategies controlled application of many, possible rules. Bounding strategies limited the number of elements assigned to the areas. The framework for dependency area identification was specified as a meta-model that extended the UML meta-model (Unified Modeling Language, 2008). Propagation rules were specified using the Object Constraint Language (OCL) (Kleppe & Warmer, 2003).

Based on the framework a new system for traceability in object-oriented projects was developed (Derezinska & Zawlocki, 2008). The system was intended to be a generic one, dealing with different project input and output notations, as well as different strategies for identification of dependency areas among project elements. The high level of flexibility was achieved using different model transformations. The general process realized within the framework consists of three main steps performed as the following model transformations:

1. input transformation,
2. traceability transformation,
3. output transformation.

The transformation engines were not limited to the UML approaches. Input and output transformations were realized using standard QVT (Query/View/Transformation) (MOF QVT, OMG Spec., 2008) developed as a part of MDA approach (Frankel, 2003). The details of input and output transformations were presented in (Derezinska & Zawlocki, 2008). The main model transformation of the system was based on an automata-based approach and its principles are presented in this paper.

4.2 Subjects of transformation

The general idea of Dependency Areas is based on exploration of possible direct and indirect relations among elements of a model. Dependency Area is a subset of all possible elements accessible from an initial element through relations available in the design and selected according to a given strategy. It can be viewed as a kind of traceability when impact relations between model elements are concerned.

Traceability analysis is defined as a transformation that maps a project P with a given initial element IE and traceability strategy Ξ into a resulting dependency area DA

$$\{P, IE, \Xi\} \rightarrow \{DA\} \tag{1}$$

where: P – *Project*
IE – *InitialElement*, $IE \in P$
Ξ – *TraceabilityStrategy* is a strategy $\Xi = <X, R, \mu>$ (see Def. 6),
DA – *DependencyArea*

A project is any object-oriented model. It reflects a general model in order to make the idea independent of any specific version of a modeling notation. Any model in other notation can be easily transformed into Project notation. Therefore this assumption does not limit the generality of the concepts.

An initial element is an element of the project that is a starting point of the traceability analysis. A traceability strategy is a strategy comprising a set of transformation rules associated with a given graph of hierarchy. The set of rules can be ordered according to the hierarchy of their priorities.
Dependency area is an output model consisting of elements assigned to a given dependency area. A dependency area can be easily transformed to any notation, which is convenient to a user.

4.3 Meta-model of Project

Any model analyzed in a framework is converted to the internal *Project* notation. This process is realized by the input transformation (Derezinska & Zawlocki, 2008). *Project* notation is a general notation covering a wide range of possible modeling concepts. The basic ideas of the Project notation are shown in its meta-model (Figure 2).

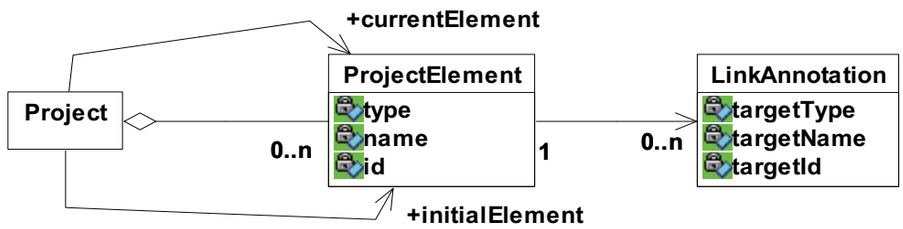

Fig. 2. The core of the Project meta-model

Any project is an aggregation of many elements (class *ProjectElement*). An element has its name, identifier and a type. Using different values of the *type* attribute a variety of different elements can be introduced in Project notation. One of the elements is distinguished as an initial element. An element of a Project can have any number of annotations (class *LinkAnnotation*) referring to a transformation process. Annotations define a set of other elements related in the project, specifying their identifiers, types or names. In a Project model, typical UML relations or properties, like generalization or association between classes, a class attribute or operation, will be converted to objects of class *ProjectElement* with appropriate types.
Transformation strategies are designed to operate on any model consistent with the Project meta-model. Such project can be understood by the executor of traceability rules.

4.4 Dependency Area meta-model

Transformation realized in the traceability framework is a kind of a model-to-model rephrasing. In the traceabilty process a dependency area is created for a model given in *Project* notation and an initial *ProjectElement*. According to Model Driven Engineering paradigma, a transformation is specified by rules dealing with meta-models of a source and target model. Any instance of meta-model *Project* should be transformed to a dependency model. The model is a resulting model of the traceability process and it is an instance of the meta-model of Dependency Area (Figure 3).

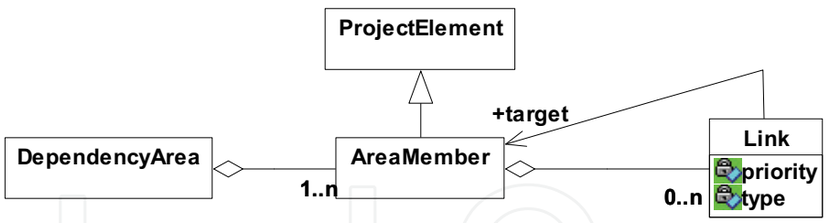

Fig. 3. The core of the Dependency Area meta-model

A dependency area consists of a set of area members. Each area member is a specialization of a *ProjectElement* of the Project meta-model. An area member can have a number of links (class *Link*) to other elements of the dependency area. Links can have their priorities and types.

Definition 7. Two dependency areas DA_1 and DA_2 are *equivalent* if:
1) their initial elements are identical,
2) they consists of the same elements,

$$x \in DA_1 \Leftrightarrow x \in DA_2$$

3) any pair of elements associated with a *Link* in one area is also associated with a *Link* in the second area

$$x \in DA_1 \ \wedge \ y \in DA_1 \wedge \exists_L \text{ (L=Link(x,y) or L= Link(y,x))} \Leftrightarrow$$
$$x \in DA_2 \ \wedge \ y \in DA_2 \wedge \exists_K \text{ (K=Link(x,y) or K= Link(y,x))}$$

In *identical* dependency areas linking elements are of the same direction in both areas (i.e. L=Link(x,y) and K=Link(x,y)) and have the same types and priorities.

4.5 Execution of transformation rules

A strategy specified with an automata-based approach can describe a variety of processes. The general idea of the rule executor was adapted for the traceability process. The goal of the process is identification of a dependency area. An algorithm of the rule execution is shown below in a pseudo-code notation. At the beginning of the model transformation, we should have an input project *P* with its initial element $IE \in P$ and a strategy $\Xi = \langle X, R, \mu \rangle$. Using mapping μ, any rule is assigned to a vertex of model *X* and has its priority. Therefore, the nonempty set of rules *R* can be ordered according to their priorities. Any rule *r* from set *R* is described by its automaton. A rule can be performed in the context of the current *Project* element, it means its conditions and actions of the automaton are evaluated in this context.

```
1  Map (P, IE , R) {
2      create empty DA
3      add IE to DA
4      WHILE (exist x in DA and x was not considered) {
5            get_first x not considered element from DA
6            FOR (from first r from R, to last r from R) {
7                  execRule (r, x)}
8            mark x as  considered
9            }
10  }
```

Where:

DA - a created dependency area,
x - an element (*AreaMember*) of a given dependency area, and also an element of *Project* (see generalization in the meta-model Figure 3),
r - a rule from the set of rules *R*
get_first from DA - returns the first element from a dependency area according to the order of adding elements to the area,
loop *FOR* in 6th line - visits the set of rules *R* according to the priority order,
execRule(r, x) - calls execution of rule *r* in the context of element *x.*

```
11 execRule(r, x) {
12     IF (precondition of rule r is TRUE)
13          executeNode(r, iN, x)
14 }
15
16 execNode(r, N, x) {
17     sortTransitions (r, N)
18     FOR (from first transition y outgoing node N
19           to last transition y of N) {
20     IF (precondition of transition y is TRUE) {
21          perform_action_of_transition y
22          select node M targeted by transition y
23          execNode(r, M, x)
24     }
25 }
```

Where:

iN - the initial node of the automaton of rule *r,*
execNode(r, N, x) - calls execution of the automaton of rule *r* at node *N* in the context of element *x,*
sortTransitions (r, N) - sorts transitions outgoing node *N* in the automaton of rule *r* according to their priorities,
loop *FOR* in 18th line - visits the set of transitions outgoing the node *N* in the automaton of rule *r* according to the priority order,
y - a transition outgoing the node *N* in the automaton of rule *r,*
M - the node targeted by transition *y*

While performing an action of a transition, a *Project* element can be added to the dependency area. The properties of the identified traceability relation are specified in a *Link* object. The type and priority attributes of the link are defined by the target node and a priority of the transition, accordingly.

Elements assigned to a dependency area are considered by the executor in the order of their addition. The first one is the initial element indicated in the input project. In result of the algorithm, the whole set of rules is executed for any project element assigned to the dependency area. Any element is added only once to the resulting dependency area. If an action specifies assignment of an already existing element, only additional references between elements are added to the output dependency model.

In this solution, the transformation logic is divided between a hierarchy of rules, a structure of automata, predicates of conditions, and actions of transitions. The strategy defines in this way the criteria when and which project elements should be added to the output dependency area. The element added to the area is not removed from it, unless another area is created (e.g. for a different initial element or a different set of rules).
Another way of a strategy definition could be also considered - when some elements once added to the area would be excluded from it in the next step. Such approach was presented in (Derezinska & Bluemke, 2005), where different strategies were discussed. Rules of selection strategies were responsible for selecting and adding elements to a dependency area, whereas rules from bounding strategies decided about removing some, already added elements from the area. This approach was used in our previous framework for traceability.
According to our experiences, we decided to use only selection strategies which decide about adding only those elements to an area that should be certainly placed in it. Therefore the algorithm of rule execution can be quite simple. There is a tradeoff between the simplicity of the rule executor realization and a homogenous set of rules on the one hand, and a potential complexity of conditions and actions comprised in the rules on the other hand.
It should be noted, that there are different sources of nondeterminism in the transformation process. Many rules with satisfied preconditions can have the same priorities; different transitions outgoing the same node can be labeled with the same priorities. In such cases the order of rules and operations is random. It means that there could be different transformation processes for the same strategy. In general, we are interested in obtaining unique results. Good strategies should give identical or at least equivalent resulting dependency areas obtained in any transformation processes.
A strategy is *well defined* in respect to the transformation process if for any input project P with an initial element $IE \in P$ any pair of dependency areas DA obtained as a result of two transformation processes are always equivalent (Def. 7).

4.6 Example of rule execution

A model transformation will be illustrated on a simple example. It explains the rule execution process. There is a set of rules defined in an exemplary traceability strategy that are applied for any element of a project. Let assume, that according to these rules two elements alphabetically consecutive to a considered element are assigned to the current Dependency Area.
There are four rules defined in the current strategy *alphabeticRules* (Figure 4), named from *rule1* to *rule4*. The priorities have been already assigned to the rules; therefore no additional hierarchical model is necessary. The rules can be ordered according to their priorities. If two or more rules have the same priority the order of rule execution is not determined. Therefore two orders of the rules are possible {*rule1, rule2, rule3, rule4*} and {*rule1, rule3, rule2, rule4*}.

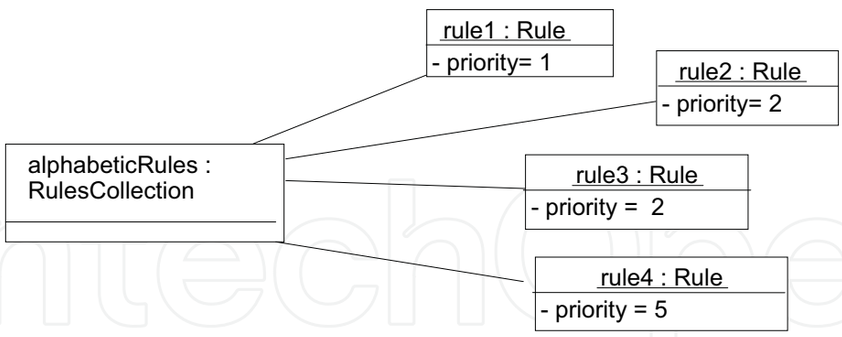

Fig. 4. Exemplary collection of rules.

An input project P1 is shown in an object diagram (Figure 5). The project consists of five elements {a, b, c, d, e}. Element *a* is the initial element and is assigned as a first element to the resulting Dependency Area *DA(a)*.

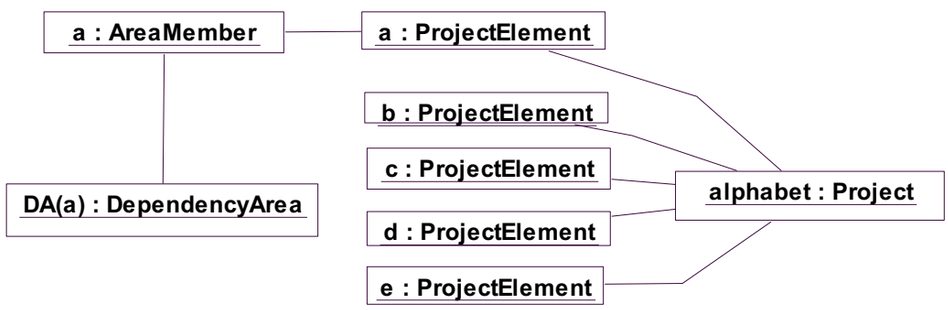

Fig. 5. A model of an input project - example P1

Next, after considering all rules for element *a,* two elements *b* and *c* will be assigned to the created dependency area *DA(a)* and link to element *a* (Figure 6). For the simplicity, the objects of the corresponding *AreaMembers* are directly connected on the diagram and the objects of *Links* between those elements are omitted. Then, element *b* is considered, because it was assigned first to *DA(a)*. Its two consecutive elements are *c* and *d*. Element *c* is already assigned, therefore only *Link* between *b* and *c* is created. Element *d* and a *Link* between *b* and *d* are added to *DA(a)*. For element *c*, element *e* and *Links* to *d* and *e* are assigned. Finally, a Link between elements *d* and *e* is added to *DA(a)*.
In this case, the given set of rules in respect to the transformation is well defined. An identical dependency area would be obtained if, for example, element *c* were considered before element *b*.

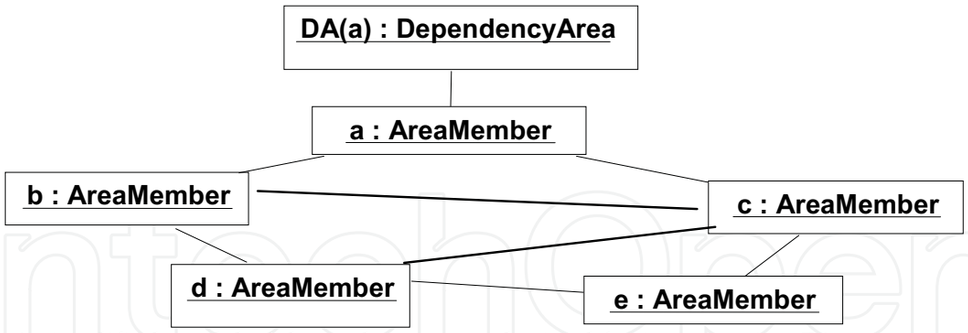

Fig. 6. A simplified model of dependency area obtained for input project P1.

5. Application of strategy specification to traceability of object-oriented projects

The concepts of transformation strategy were applied in the realization of the framework for traceability in object-oriented projects. The principles will be illustrated by examples concerning a subset of UML. The general architecture of the framework was presented in (Derezinska & Zawlocki, 2007). Input and output transformations were realized using QVT approach and existing, supporting it tools (West Team, MDA - Transf User Guide).

5.1 Traceability strategy for a subset of UML

The set of rules can be in general organized in different ways. The main goal of organizing a set of rules in a hierarchical form is identification of rule priorities and specification of their preconditions.

Considering a subset of UML, the basic concepts are described using class models comprising meta-models elements. The rule hierarchy corresponds to a hierarchy of meta-models. It should be noted, that we do not operate directly on a UML meta-model (Unified Modeling Language, 2008) but on a *Project* meta-model. However the necessary elements from the UML meta-model have the corresponding *ProjectElements.* Therefore the nodes contain similar elements as in the UML meta-model. In the UML meta-model one element can be related via generalization to many base elements. Therefore generalizations in the UML meta-model do not create tree-like structures.

A small extract of the hierarchy used in experiments illustrates the idea (Figure 7). A root of the hierarchy is a *ModelElement* with 0 priority (precisely - any rule which is associated with this vertex has priority equal to 0). Two other elements of the graph are *Classifier* and *Package* with rule priorities equal to one. A *ModelElement* can be a *Package* or a *Classifier,* and therefore they constitute the children of this vertex in the graph. A *Classifier* can be, for example, an *Actor,* a *Use Case* or a *Class*. Therefore they are the children of the vertex that corresponds to the *Classifier* rule. The elements lower in the hierarchy have higher priorities. It reflects the idea that any class at the meta level is interpreted at the first place as an actor, a class or a use case. It is less important, that this class is also a classifier. Finally, any class can be also interpreted as an element of a model.

In general, creating such a hierarchy we assign rule priorities, starting from 0 in the root, and incrementing the priority for each hierarchy level. The rules are executed in the order of their priorities. Therefore, for an *Actor* element in a model, firstly its specialized rule will be

performed, than a rule characteristic to a *Classifier*, and finally a rule specified for any model element. The similar situation states for a *Class* element and *UseCase* element. A rule associated with a *ModelElement* can be applied to any element of a Project model. It has zero priority i.e. the lowest priority in the strategy. Therefore according to rule executor this rule will be performed as the last rule for any element of a model.

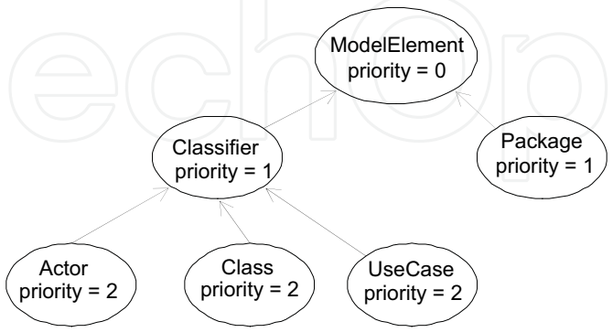

Fig. 7. Hierarchy of rules

The hierarchies of rules used in the traceability framework were created by hand, basing on the meta-model hierarchies given in the UML specification. It seems to be possible to support the construction of a strategy by a tool that analyses a given input meta-model. A graph with ordering of the rules, and description of selected rules could be derived from the meta-model in an automatic way, and further possibly modified by a user.
A traceability rule will be explained on two simple examples. Diagram in Figure 8 shows an automaton specifying a rule of the *Class* - the rule associated with the Class vertex from the hierarchy of rules (Figure 7). The rule has a priority equal to 2 and its precondition. The rule precondition is associated with the initial node of the rule, i.e., the *Class* node in this automaton. According to the precondition the rule is performed only if a current element is an *ProjectElement* of type *Class*.

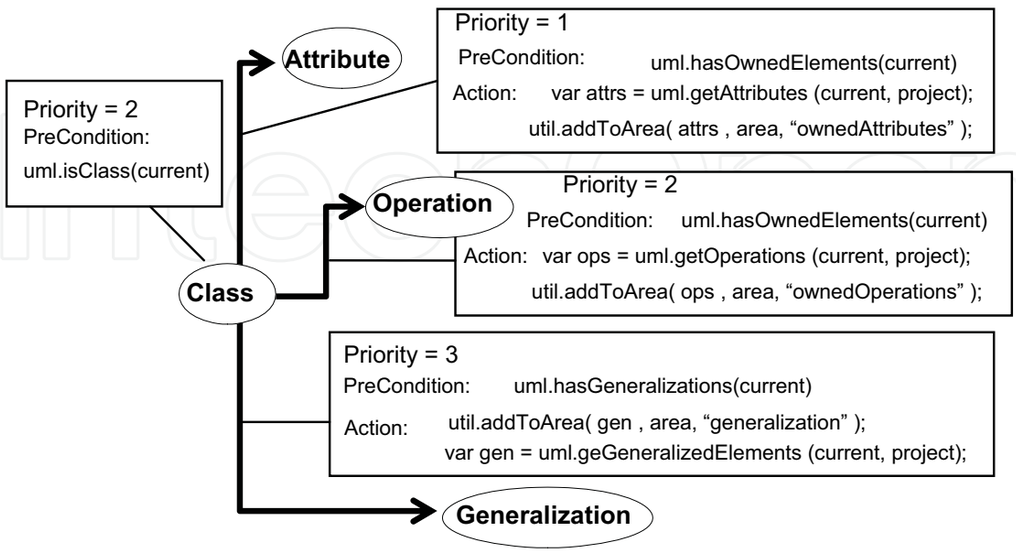

Fig. 8. An example of a traceability rule for metaclass *Class*

In the rule automaton there are three further nodes: *Attribute*, *Operation* and *Generalization*. The *Class* node is connected with transitions to these nodes. Any transition is labeled with a priority, a precondition and an action. For example, a transition to *Attribute* has priority equal 1, which is the lowest in the set of the edges outgoing from *Class* node. The precondition checks whether the current Project element is owned by the class that originates the transition. The action is responsible for adding the targeted attribute(s) to the dependency area of the considered class.
It should be noted, that there is no node corresponding to an association in the rule of the *Class*. A class is in UML specified as a classifier. Therefore in this simple strategy the node of an association will be handled in the rule of the *Classifier* (Figure 9). Such a rule will be performed later for any class, according to the rule priorities.
According to the precondition, the rule shown in Fig 9. can be applied for any initial element of the *Classifier* type. In this example Classifier node is connected to two nodes: *Association* and *ClassifierWithTheSameName*. The first transition with priority equal 1 is responsible for adding elements connected via association with a current project element. The second transition is labeled with an action that inserts to the dependency area an element having the same name as the currently considered classifier. The precondition of the transition is always satisfied (equal true), but the action is an empty operation if no elements with the same names exists. The automaton can be easily extended with other transitions and more complicated actions.

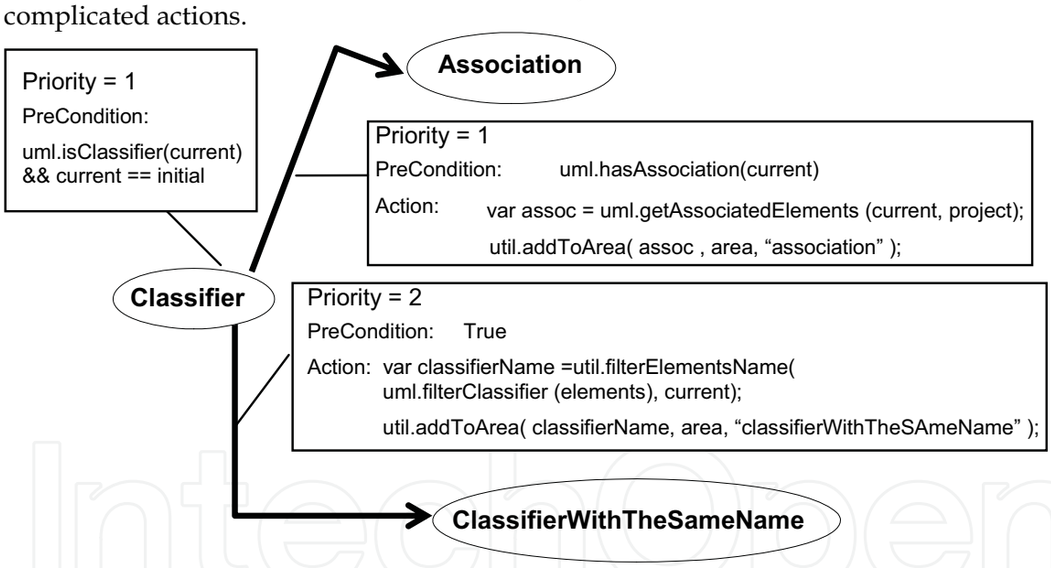

Fig. 9. An example of a traceability rule for metaclass *Classifier*

5.2 Realization of strategies

Any strategy can be specified in the XML format, comprising full descriptions of all automata of the strategy. For any automaton, all its features (identifier, priority, staring node, and the relation to an appropriate element from the meta-model hierarchy), as well as all nodes and transitions are specified with appropriate XML tags. A part of an exemplary specification is shown below. A rule starts with its condition - i.e. a condition of the initial node of the rule automaton. Next, all nodes of the rule are specified. For each node a list of outgoing edges is given.

```
<rules>
<rule id="rule-0" priority="0" initialNode="rule-0.node-1" name="modelElement" >
<condition>
                ...
</condition>
<node id="rule-0.node-1" name="modelElement" rule="rule-0">
        <edge   id="rule0.node-1.edge-1"    target="rule-0.node-2"    source="rule-0.node-1"
priority="1">
                <action>
                                ....
                </action>
                <condition>
                        ....
                /condition>
        </edge>
</node>
<node id="rule-0.node-2" name="alwaysTrue" rule="rule-0">
</node>
</rule>
<rule id="rule-1" priority="1" initialNode="rule-1.node-1" name="classifier">
<condition>
        ....
</condition>
<node id="rule-1.node-1" name="classifier" rule="rule-1">
<edge id="rule1.node-1.edge-1" target="rule-1.node-2" source="rule-1.node-1" priority="1">
        <action>
                ...
        </action>
....
```

Actions and conditions associated with nodes and transitions were written in a script language. In the implemented solution, instances of Project and Dependency Area classes are passed to the context of the script language. Therefore we can call directly those classes in the specification of actions and conditions. The methods of these classes can be executed; as well the values of public attributes can be got or updated. Functionality of the script language can be extended by other objects added to the context.
For this purpose JavaScript and Rhino library were used (Documentation of Rhino). Rhino library enables calling of scripts implemented in JavaScript directly from the Java code. For example, a pre-condition of a rule can be specified in the following way:

```
<condition>
  <script>
        <![CDATA[uml.isClassifier(current) && current == initial]]>
  </script>
</condition>
```

In this JavaScript code we can call the objects passed in the execution context: *uml* - a class supporting selected OCL operations, *current* - a current element of the considered project, *initial* - an initial element of the project.
In the similar way actions associated with transitions can be specified. For example, three actions of a rule are shown below. They used methods of object *log* - to log messages on a console, *uml* - to access model elements according to UML relations, and *util* - to manipulate on a dependency area.

```
<action>
  <script>
        <![CDATA[log.debug("rule1.node-1.edge-1 : start");
        var assoc = uml.getAssociatedElements(current, project);
        util.addToArea( assoc , area, "association");
        ]]>
  </script>
</action>
```

Actions and conditions written in JavaScript are included in XML file. They can be executed directly during a program run, as they are executed in the Rhino engine. Therefore the strategy can be easily changed by rewriting only those scripts. The executed code will be changed without modifying and recompiling the desired application.

5.3 Dependency Area Example

After specifying the input transformation to the model *Project* and the traceability transformation, we can use the framework. Dependency areas were created for a set of projects assuming different initial elements and different traceability strategies.
For the brevity reasons, we illustrate the idea using a simple plug-in for input the subset of UML and an exemplary traceability strategy dealing with this subset. Model elements not considered by the input transformation will be omitted during traceability analysis.
As an example, a simple *Cinema* system will be discussed. An actor of the system - *Client* can review the repertoire of the cinema and buy a ticket of a selected film (Figure 10).

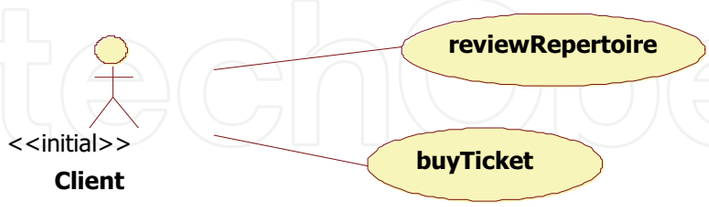

Fig. 10. Use cases of *Cinema* system

The system is divided into several packages. Classes *Client* and *Ticket* are included in *Client* package, and classes *Cinema*, *Repertoire* and *Movie* belong to *Cinema* package. Selected elements of the project used in this example are shown in Figure 11.

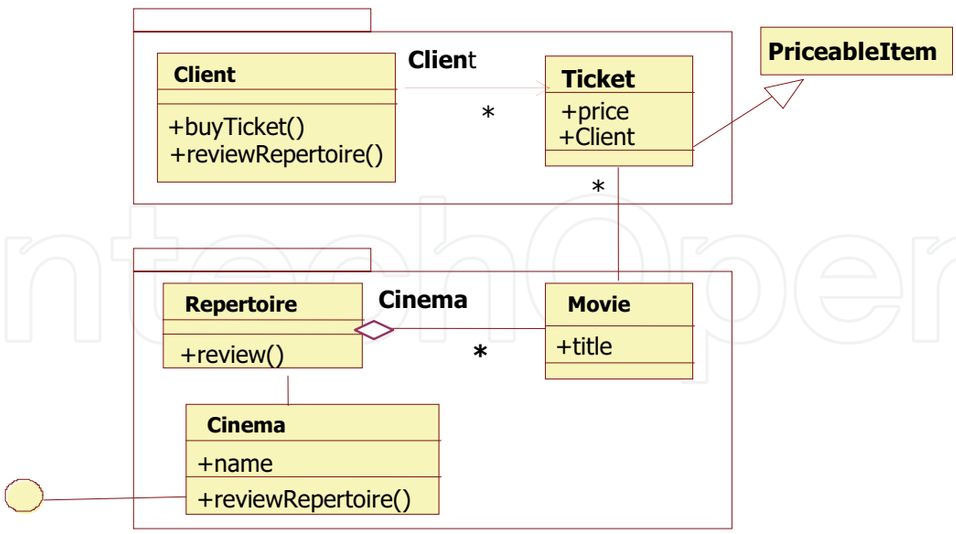

Fig. 11. Part of the *Cinema* class model

Let us assume that actor *Client* is an initial element. After application of a traceability strategy a dependency area is generated. The object diagram of the area is shown in Figure 13. For the simplicity, objects of class *Link* are omitted and the links between objects are labeled by the types of traceability relations. Precisely, the labels are values of the *type* attribute of *Link* objects (see meta-model in Figure 3). We can observe that several objects are related according to the similarity of names, for example, actor *Client* and class *Client*. If relations based on the name similarities were excluded, the resulting dependency area for the actor *Client* would include only its use cases linked via associations.

Another dependency area will be created if *Cinema* package is selected as the initial element (Figure 12). In this case, most of the relations reflect simple ownership relations of the UML meta-model elements. A dependency area is simpler and has a tree-like structure. All elements were included into dependency area according to one unique sequence of rules. In the case of Figure 13 there are several paths connecting the initial element with some elements, like class *Repertoire,* operations *reviewRepertoire, buyTicket,* attribute *Client.* It means that the same element was classified to this dependency area based on different sets of rules.

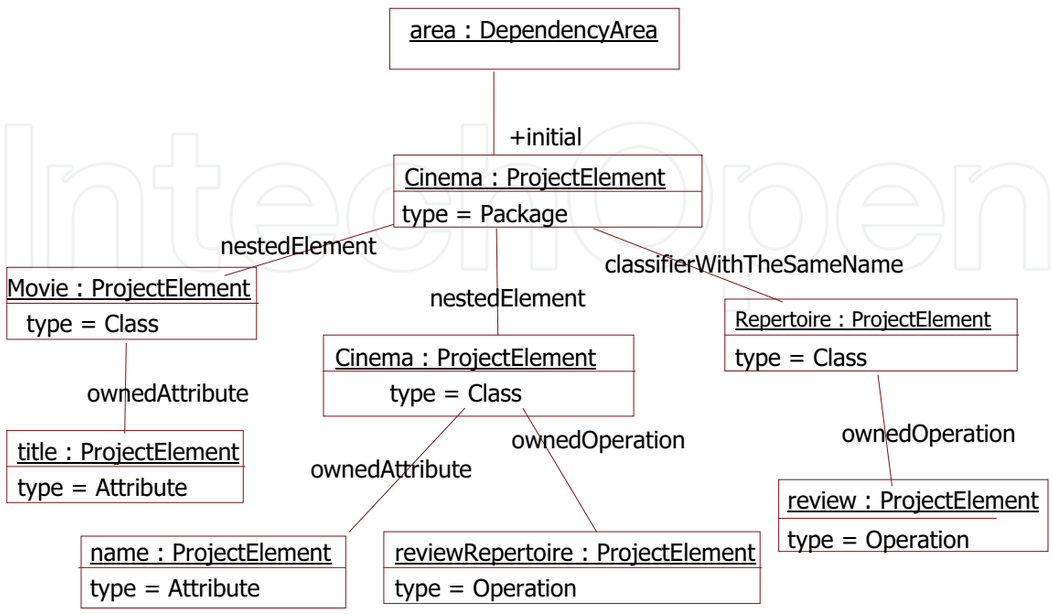

Fig. 12. Dependency relations for initial element associated with package Cinema

6. Conclusions

Automata-based approach for the specification of strategies was presented in this paper. The powerful and flexible mechanism for describing a wide scope of issues was introduced. The approach combines the hierarchy of basic concepts, similar to a meta-model graph, with sets of rules. An automaton defining a single rule is accompanied by conditions and actions that can be specified in a desired notation. In this way a strategy is separated into different abstraction levels, supporting the process of specification refinement and modification.

The automata-based strategy was applied in the generic framework for traceability of object-oriented project. An engine for rule interpretation was realized. The traceability logic can be described in the strategy by selecting appropriate priority levels of the rule hierarchy, creating the set of automata and finally specifying details in conditions and actions. An instance of the framework was implemented. It realized traceability rules based on the dependency area concepts. Exemplary strategies were specified for the subsets of UML models.

Flexibility of the methodology is very beneficial because the same rule execution engine can be used in many cases and strategies can be easily modified. On the other hand, the presented approach faced some problems. Preparation of a detailed specification of a non-trivial strategy requires a lot of work. It could be profitable if a strategy is used many times and the modified strategies are created as, for example, some variants of another, already specified strategy.

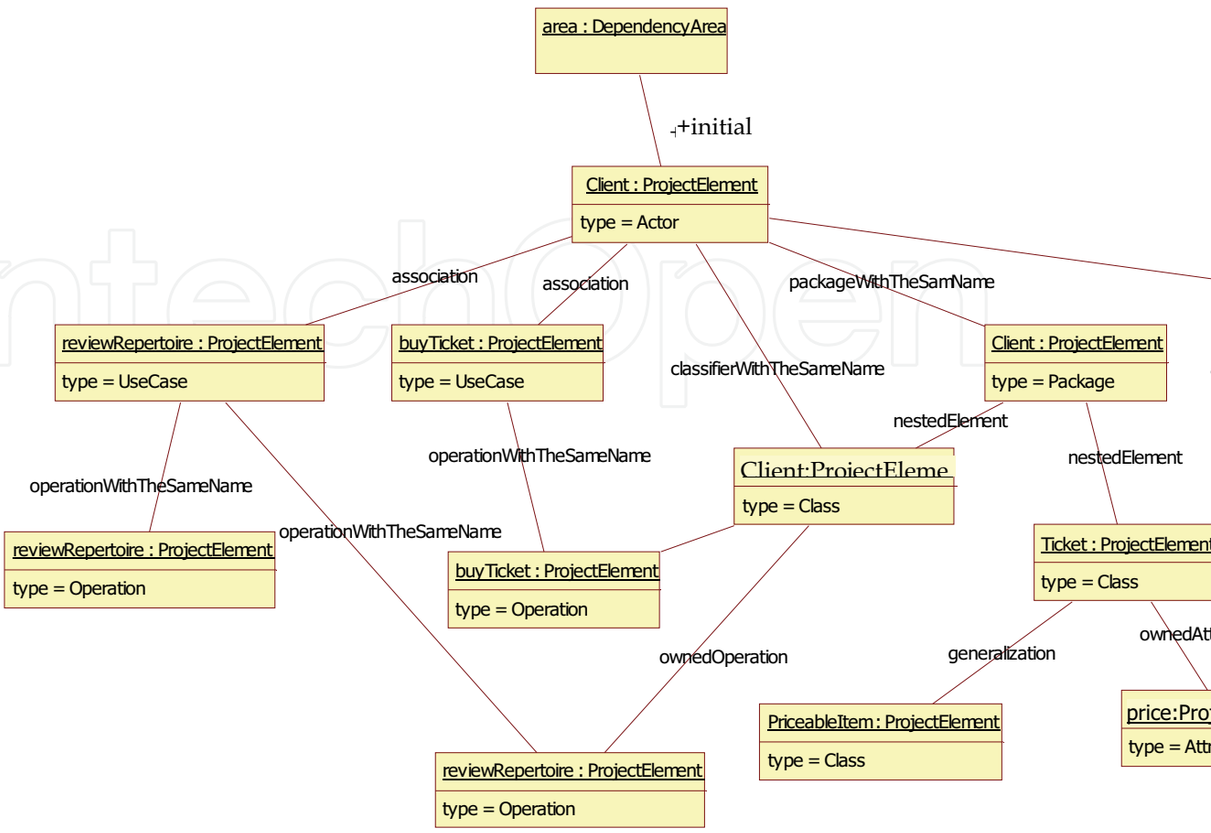

Fig. 13. Dependency relations for initial element associated with actor *Client*

Future work should be aimed at further verification of the approach in different experiments. Specification of various strategies, as well as input transformations for different sets of modeling elements can be applied in experiments using the framework. In the strategies, various intentions of developers can be taken into account, revealing relations between model elements that are not directly expressed in a model.
Another research direction is an application of the automata-based specification of a model transformation strategy in other kinds of model manipulations, like merging of models, weaving of model aspects, model refinement, etc.

7. References

Balogh, A. & Varro, D. (2006). Advanced model transformation language constructs in the VIATRA2 framework, *Proceedings of the ACM symposium on Applied Computing (SAC'06),* pp. 1280-1287, ISBN: 1-59593-108-2, Dijon France, April 2006, ACM Inc. New York, NY, USA

Bensh, S.; Bordhin, H.; Holzer, M. & Kutrib, M. (2008). Deterministic Input-Reversal and Input-Revolving Finite Automata, *Proceedings of International Conference on Language and Automata Theory and Applications (LATA 2008),* LNCS 5196, pp. 113-124, ISBN 978-3-540-88281-7, Tarragona Spain, March 2008, Springer, Berlin Heidelberg

Buttner, F. & Bauerdic, H. (2006). Realizing UML Model Transformations with USE, *Proceedings of Workshop on OCL for (Meta-)Models in Multiple Application Domains* (at Models 2006), pp. 96-110, ISSN 1430-211X, Genova, Italy, Oct. 2006, Technical Report of the Technische Universität Dresden

Czarnecki, K. & Helsen, S. (2006). Feature-based survey of model transformation approaches, *IBM System Journal*, Vol. 45, No 3 pp. 621-645, ISSN 0018-8670

Dayan D., et al. (2008). MDA PIM-to PIM transformation using extended automata, *Computer Modelling and New Technologies*, Vol. 12, No. 1, pp. 65-76, ISSN 1407-2742

Deo, N. (1974). *Graph Theory with Applications to Engineering and Computer Science*, Prentice-Hall Inc., ISBN 0133634736, Upper Saddle River, NJ, USA

Derezińska, A. (2004). Reasoning about Traceability in Imperfect UML Projects, *Foundations on Computing and Decision Sciences*, Vol. 29, No. 1-2, pp. 43-58, ISSN 0867-6356

Derezińska, A. & Bluemke, I. (2005). A Framework for Identification of Dependency Areas in UML Designs, *Proceedings of the 9th IASTED International Conference on Software Engineering and Applications SEA'05*, pp. 177-182, ISBN 0-88986-529-9, Phoenix, Arizona, USA, Nov. 2005, Acta Press, Clagary Canada

Derezińska, A. (2006). Specification of Dependency Areas in UML Designs, *Annales UMCS Informatica,* AI 4, Vol. 4, pp. 72-85, ISSN 1732-1360

Derezińska, A. & Zawłocki, J. (2007). Generic framework for automatic traceability in object-oriented designs, *Annals of Gdańsk University of Technology Faculty of ETI,* No 5, pp. 291-298 , ISBN 978-83-60779-01-9 (in polish)

Derezińska, A. & J. Zawłocki, J. (2008). Towards Model Transformation - A Case Study of Generic Framework for Traceability in Object-Oriented Designs, *IADIS Inter. Journal on Computer Science and Information Systems*, Vol 3, No 1 April 2008, pp. 29-43, ISSN: 1646-3692

Documentation of RHINO, http://www.mozilla.org/rhino/doc.html

Egyed, A. (2004). Consistent Adaptation and Evolution of Class Diagrams during Refinement, *Proceedings 7th Inter. Conf. on Fundamental Approaches to Software Engineering (FASE)*, LNCS 2984, pp. 37-53, ISBN 3-540-21305-8, Barcelona, Spain, March 2004, Springer, Berlin Heidelberg

Ehring, K. & Giese, H. (Eds.) (2007). Proc of the 6th Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT), Electronic Communications of the EASST, ISSN 1863-2122, vol. 10

Ehring K.; Guerra E. & de Lara, J., at al. (2005) Transformation by graph transformation: a comparative study, *Proccedings of the Model Transformation in Practice Workshop at MoDELS 05*, Montego Bay, Jamaica, October 2005

France, R. & Rumpe, B. (2007). Model-driven Development of Complex Software: A Research Roadmap, Proceedings of Future of Software Engineering (FOSE at ICSE'07), pp. 37-54, ISBN: 0-7695-2829-5, Minneaplois, Minnesota, May 2007, IEEE Comp. Soc. Washington, DC, USA

Frankel, D. S. (2003). *Model Driven Architecture: Appling MDA to enterprise computing*, Wiley Pub. Comp., ISBN 0-471-31920-1, Indianapolis, Indiana

Hopkroft J., Motwani R. & Ullman J. (2000) *Introduction to automata Theory, Languages, and Computation,* 2nd ed., Addison-Wesley, ISBN-10: 0201441241

Jouault F. & Kurtev I. (2005). Transforming Models with the ATL, *Proccedings of the Model Transformation in Practice Workshop at MoDELS 05*, LNCS 3844, pp. 128-138, ISBN 978-3-540-31780-7, Montego Bay, Jamaica, October 2005, Springer, Berlin Heidelberg

Kleppe, A. G. & Warmer, J. (2003). *The Object Constraint Language: Getting your models ready for MDA*, Addison-Wesley 2nd ed, ISBN 0321179366, Boston MA

Kolovos, D.S. at al. (2007). Update Transformations in the Small with the Epsilon Wizard Language, *Journal of Object Technology*, Vol. 6, No. 9, Special Issue TOOLS EUROPE Oct. 07, pp. 53-69, ISSN 1660-1769

Lakhnech, Y.; Mikk, E. & Siegel M. (1997). Hierarchical Automata as Model for Statecharts. *Advances in Computing Science - ASIAN '97, Proceedings of 3rd Asian Computing Science Conference*, LNCS 1345, pp. 181-196, ISBN 978-3-540-63875-9, Kathmandu, Nepal, Dec. 1997, Springer, Berlin Heidelberg

Letelier, P. (2002). A Framework for Requirements Traceability in UML-based Projects, *1st Int. Workshop on Traceability in Emerging Forms of Software Engineering* by IEEE Conf. on ASE, Sept. 2002, Edinburgh UK
http://www.soi.city.ac.uk/~zisman/WSProgramme.html

Maeder P. et al. (2006). Traceability for managing evolutionary change, *Proceedings of 15th Software Engeenierin Data Engeeniering SEDE* (ICSA), pp. 1-8, ISBN: 978-1-880843-59-5, Los Angeles USA, July 2006, ISCA, Cary, North Carolina USA

McMillan, K.: Cadence SMV. http://www.kenmcmil.com/ (visited 2009)

MDA Guide, Ver. 1.0.1, Object Management Group Document omg/2003-06-01 (2003)

Mens, T. & van Gorp P. (2006). Taxonomy of model transformation, *Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005),* Tallin, Estonia, Sept. 2005, *ENTCS,* Vol. 152 , March 2006, pp. 125-142, ISSN**:** 1571-0661, Elsevier

Meta Object Facility (MOF) , OMG Document, formal/2006-01-01, (2006)

MOF QVT Specification, OMG Document formal/2008-04-03 (2008)

Sendall S. at al. (2004). Understanding Model Transformation by Classification and Formalization, *Proceedings of Workshop on Software Transformation Systems* (part of 3rd International Conference on Generative Programming and Component Engineering GPCE), pp. 30-31, Vancouver, Canada, Oct. 2004, http://www.program-transformation.org/Sts/STS04

Spanoudakis, G. et al. (2004). Rule-based Generation of Requirements Traceability Relations, *Journal of Systems and Software*, Vol. 72, No. 22, pp. 105–127, ISSN: 0164-1212

Unified Modeling Language Superstructure v. 2.2, OMG Document formal/ptc/2008-05-05, (2008), http://www.uml.org

Vanhoof, B. at al. (2007). UniTI: A Unified Transformation Infrastructure, *Proccedings of of 10th International Conference Model Driven Engineering Languages and Systems (Models 2007),* LNCS 4735, pp. 31-45, ISBN 978-3-540-75208-0, Nashville, USA, Sept-Oct. 2007, Springer, Berlin Heidelberg

Vanhooff, B. et al. (2007). Traceability as Input for Model Transformations, *Proceedings of ECMDA Traceability Workshop,* pp. 37-46, ISBN 978-82-14-04056-2, Haifa, Israel, June 2007

Wagner, G.; Giurca, A. & Lukichev, S. (2006). Modeling Web Services with URML, *Proceedings of Workshop Semantics for Business Process Management*, Budva, Montenegro, June 2006, http://idefix.pms.ifi.lmu.de:8080/rewerse/index.html

Walderhaug S., et al. (2006) Towards a Generic Solution for Traceability in MDD, Proceedings *of 2nd ECMDA Traceability Workshop*. Bilbao, Spain, July 2006, ISBN 82-14-04030, SINTEF Report, http://modelbased.net/ecmda-traceability/index.php?option=com_content&task=view&id=23&Itemid=34

West Team, MDA - Transf User Guide, http://www.lifl.fr/west/modtransf/

W3C Recommendations for rules interchange http://www.w3.org/2005/rules/wiki/RIF_Working_Group (visited 2008)

Engineering the Computer Science and IT

Edited by Safeeullah Soomro

ISBN 978-953-307-012-4

Hard cover, 506 pages

Publisher InTech

Published online 01, October, 2009

Published in print edition October, 2009

It has been many decades, since Computer Science has been able to achieve tremendous recognition and has been applied in various fields, mainly computer programming and software engineering. Many efforts have been taken to improve knowledge of researchers, educationists and others in the field of computer science and engineering. This book provides a further insight in this direction. It provides innovative ideas in the field of computer science and engineering with a view to face new challenges of the current and future centuries. This book comprises of 25 chapters focusing on the basic and applied research in the field of computer science and information technology. It increases knowledge in the topics such as web programming, logic programming, software debugging, real-time systems, statistical modeling, networking, program analysis, mathematical models and natural language processing.

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen