ISSN 1392 – 124X INFORMATION TECHNOLOGY AND CONTROL, 2009, Vol. 38, No. 4

USING ATTRIBUTES AND MERGING ALGORITHMS FOR TRANSFORMING OCL EXPRESSIONS TO CODE

Andrius Armonas, Lina Nemuraitė

Kaunas University of Technology, Department of Information Systems
Studentų st. 50-308, LT-51368 Kaunas, Lithuania
e-mail: andrius@soften.ktu.lt, lina.nemuraite@ktu.lt

Abstract. In this paper, we examine OCL-to-code transformations, which are dedicated to modern model and metamodel repositories facing with the requirements to perform search, validation and transformation of models that are usually stored in external data storages, e.g. RDBMS. Diversity and fast changes of data storage technologies make the development of such transformations a real challenge. This paper presents a method developed by the authors enabling reuse of OCL transformations, their adaptation to various data storage environments and evolution by applying attributes and graph merging algorithms.

Keywords: UML, OCL, model repository, code generation, transformation, attribute, merging.

1. Introduction[1]

As modelling became one of the essential aspects of software development, the need for an efficient teamwork environment for modelling arose. Such an environment should be equipped with capabilities for merging, synchronizing, reusing and evolution of models and metamodels independently from any specific problem domain. Such requirements are partly met by Eclipse EMF, IBM Jazz and No Magic Cameo platforms. These platforms use the OCL language to implement queries and ensure well-formedness of models, metamodels and their relationships. Being faced with the requirements to perform search, validation and transformation, models in such repositories are usually stored in external data storages, e.g. RDBMS, therefore OCL expressions need to be transformed to the source code of the data storage. Practically, highly different ordinary and advanced modelling tasks (as e.g. [1–4]) are coping with necessity to have a solid support for OCL. This paper examines the problem of transformation of OCL to program code by reusing existing transformations and facilitating evolution of transformations by teams working with repositories of models and metamodels. Variability and commonality analysis principles [5, 6] were used during the development of the method.

The significance of the research presented in this paper is influenced by an increasing demand for metamodel and model repositories as well as increasing requirements for quality of data, stored in them, and services, provided by them: frequent and efficient searching, validation, transformation and synchronization of models. It is no longer sufficient only to provide these services – a need arises to adapt them to the existing user infrastructure. If there were no possibility to adapt and reuse *OCL* transformations to source code, the development of model repositories would slow down. Majority of modelling tool vendors are looking for faster ways to create new OCL transformations for generation of code in various languages by developing these transformations from existing ones.

Businesses would benefit from using this method as follows: it would improve design and support of OCL transformations to source code; it would enable reacting faster to changes in data storage software, decrease amount of required routine software coding work; provide an opportunity to reuse transformations and facilitate their collaborative development, making work of software analysts, designers and programmers easier.

The remainder of this paper is structured as follows: in sections 2 and 3 we review the related work in attribute grammars and graph merging algorithms; section 4 presents the created method; section 5 is devoted to method implementation and experiment; 6 – to its assessment. In section 7 we summarize the results of research and draw conclusions.

[1] The work is supported by Lithuanian State Science and Studies Foundation according to High Technology Development Program Project "VeTIS" (Reg.No. B-07042)

2. Attribute grammars

Attribute grammars simplify operation of interpreters and compilers for such tasks as, for example, semantic checking or transforming concrete syntax trees (CST) to abstract syntax trees (AST). They have been described several decades ago, however, quite a substantial amount of papers are being published refining them. There are attribute grammar modifications RAG [7–11], CAG [12–14], CRAG [15], ReCRAG [16], HAG [17], as well as composite [18] and conditional attribute grammars [19]. The method, described in this paper, uses the RAG attribute grammar. However, other modifications also are applicable.

Attribute grammars facilitate development of compilers and interpreters but they also decrease performance of these tools. For example, using ReCRAG grammars in Java compilers reduces their performance four times, even though language specification itself is two times smaller [16]. On the other hand, it is impossible to avoid using of these methods in large, continuous projects as they reduce amount of errors, make management of human resources more efficient, and cut down the time needed for analysis, implementation and support.

3. Difference and merging algorithms

For facilitating reuse of attributes, created in previous transformations, the following graph merging algorithms were analyzed: Alanen and Porres [20]; Zündorf, Wadsack and Rockel [21]; Sudarshan et al. [22]; Wang, DeWitt and Cai [23]; Cicchetti et al. [24–26]; Bartelt [27]; Kelter, Wehren and Niere [28]; Ohst, Welle and Kelter [29]; Eclipse EMF Compare tool. A more detailed analysis of similar algorithms is available in [30].

Analyzed algorithms can be classified into two categories by how they identify elements: algorithms that identify elements heuristically and algorithms that rely on uniquely identified elements. It was found in this research that none of these algorithms combine those two approaches and this can be considered as a deficiency, because in such languages as UML part of elements have to be compared by using unique identifiers and the other part should be compared heuristically.

It is important to note that the analyzed algorithms except one do not have dependency and conflict concepts. This means that the algorithms are only capable of bulk change copying to one of the contributor graphs, i.e. partial copying is impossible.

None of the analyzed algorithms evaluate previous merges. They calculate differences between two versions of a graph and try copying those changes to the graph that was derived from their common ancestor. Model merging difference from code merging here is that it is possible to get incomplete and incorrect models as there are many more dependencies and semantics when merging models instead of code. The aforementioned deficiencies are resolved in the method proposed in this paper. Our method uses merging algorithms to allow collaborative work over OCL-to-code transformations as well as reusability of those transformations.

4. Applying attributes and graph merging algorithms for transforming OCL expressions to code

This section describes a method for transforming OCL expressions to code using attributes and graph merging algorithms.

4.1. Method overview

A high-level use case diagram, depicted in Figure 1, shows how the created method is applied in two levels: in the overall software development process, which uses model repositories, and in the collaborative development of OCL-to-code transformations thus extending both software and transformation development domains.

Models are validated [31], transformed [32] or element search is performed in models throughout the whole *software development* process (see all use cases included into the *Model software* use case). Also, in the collaborative environment, models are exchanged between the local and remote repositories. The OCL language is used for validating, transforming, and searching data in models. When performing these activities in model repositories, OCL expressions are transformed to expressions of the underlying data storage environment and then are executed in it for returning the actual results to the software designer. The created method implements such a transformation that is represented by use case *Transform OCL to code using attributes* in Figure 1. OCL-to-code transformations are performed in the remote (central) model repository.

Software designer performance (in terms of his/her daily tasks) is directly influenced by the performance and quality of OCL-to-code transformations. In other words, software development activities depend on transformation development activities. The created method enables collaborative development of OCL-to-code transformations by allowing creation, modifying, and reusability of attribute sets (see the *Develop attribute sets for transformations* use case).

An overview activity diagram, representing the created method for generating code from OCL using attributes, is depicted in Figure 2. As stated earlier, software designers validate, transform, and search for elements in models by using the OCL language, whose expressions are specified in domain-specific models, created using domain-specific modelling languages (DSML) (e.g. [33]) . Inputs of the created method consist of the DSML model and OCL expressions specified for it. Metamodels and models for which OCL expressions are specified are both considered as DSML models. The method produces code, adapted for a specific data storage environment, performing data retrieval commands specified in the OCL expression for which the code is generated.

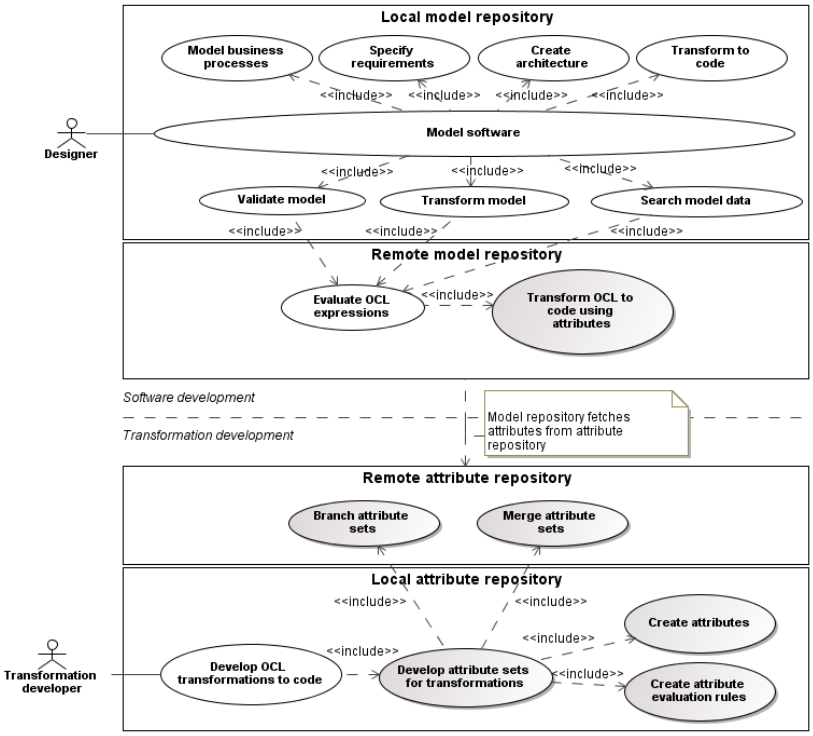

Figure 1. Method (represented by greyed use cases) usage in software and transformation development domains

Inputs of the method consist of the DSML model and OCL constraints specified for that model (see Figure 2). When transforming OCL to code, each OCL expression is transformed to the concrete syntax tree (CST), which is then transformed to abstract syntax tree (AST). Abstract syntax tree is augmented with references to elements of the DSML model. In this phase of the transformation, the method presented in this paper augments the AST tree with attributes and values for each attribute (attribute evaluation rules are employed for this purpose). Attribute evaluation is represented by the element *AST tree with evaluated attributes* in the diagram. Attribute specifications for the specific language for which the code is being generated are fetched from the remote (central) attribute repository. Attributes are created and adapted for specific data storage environments by transformation or modelling tool developers, not end-user system developers. The created method uses AST trees augmented with attributes to generate code that fully exploits capabilities of a specific data storage environment (this is represented by the element *Transform OCL to code using attributes* in Figure 2). The created method produces code running in the specific data storage environment.

As stated earlier, attributes are created and adapted for specific data storage environments by transformation and modelling tool developers. The created method specifies not only attribute usage for code generation but also describes principles for collaborative development (evolution) of attributes. Figure 3 depicts an overview of attribute creation and development process. When performing the actual OCL transformation, attributes are fetched from the remote (central) attribute repository. This repository stores committed attribute sets (i.e. attribute graphs). New attribute sets can be created by reusing parts of existing attribute sets. Transformation developers store intermediate versions of attribute sets in their local attribute repositories. Attribute sets between central and local repositories are synchronized by using branching and the created merging algorithm. This algorithm is also used for synchronizing changes between branches in the central repository.

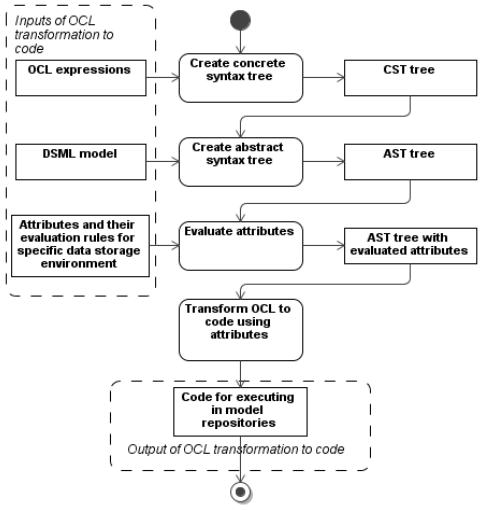

Figure 2. Overview of code generation using attributes

The structure of the central attribute repository is depicted in Figure 4. Attribute sets are described as graphs whose nodes are attributes and links between them – relationships between attributes. Every attribute graph is used for generating code for a specific

data storage environment. Central attribute repository stores sets of attribute graphs, tracks attribute graph changes, graph branching and merging actions by versioning them.

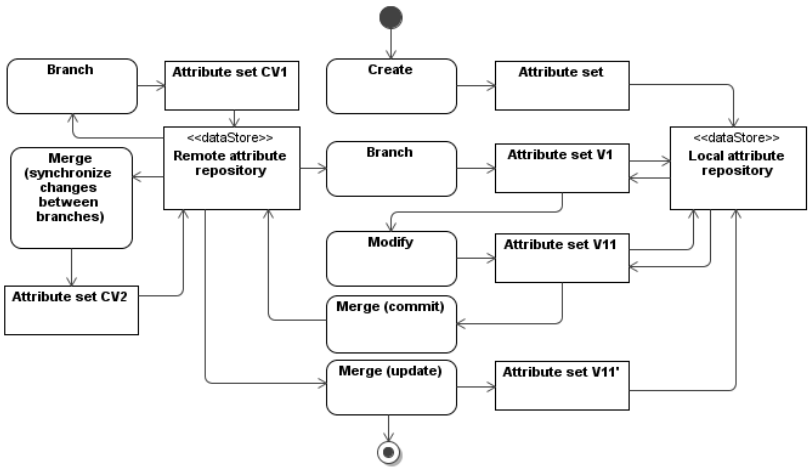

Figure 3. Overview of attribute creation and evolution process

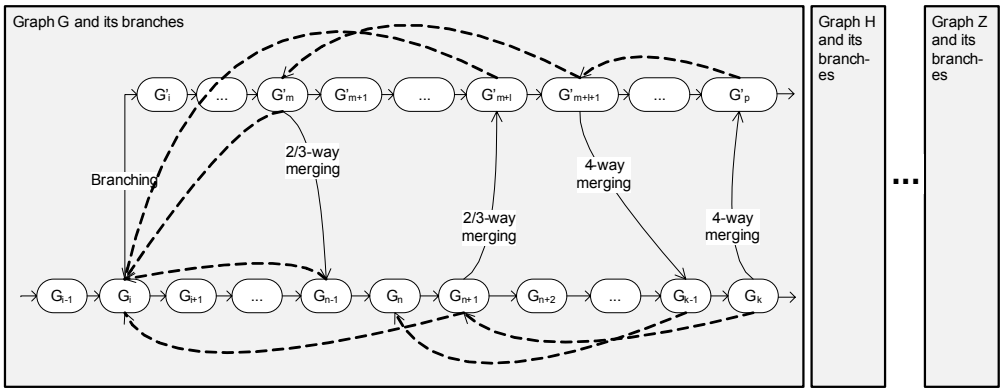

Figure 4. Overview of central attribute repository structure and related operations

Figure 4 shows attribute sets and their evolutions using elements *Graph G and its branches*, *Graph H and its branches*, and *Graph Z and its branches*. For example, element *Graph G and its branches* depicts attribute set *G* and its evolution (branching from version G_i of the graph *G* creates graph G'_i).

4.2. Using attributes in OCL AST trees

One of the best-known tools for generating code from OCL is Dresden OCL2 Toolkit. In this paper, we call transformations of this tool as *standard transformations*. The standard way of generating code from OCL expressions is insufficient, because it attempts to directly map single OCL constructs to target language constructs. This means that if source and target languages are not similar, i.e. if constructs from source language cannot be directly covered by constructs or groups of constructs from the target language, the standard way does not allow mapping from one language into other in all cases (or does this inefficiently). For being able to map complex constructs (subtrees) of the source language into target language constructs, it is necessary to have context information in OCL AST tree nodes enabling to generate code for the whole OCL AST sub-tree at once and ignore internal nodes in the following traversals.

In the created method, context information is expressed by attributes and generated code is adapted to concrete data storage platform by computing values of these attributes. Each OCL metamodel element can have one or more attributes defined. Values are assigned to attributes by computing attribute evaluation rules.

Attribute metamodel is depicted in Figure 5. For traversing AST trees, derivatives of the Visitor pattern are used. In the proposed method, an algorithm developed by Neff [34], called Bivisit, is used. This algorithm evaluates attributes on demand. It is presented in Figure 6 by an activity diagram where parameter of the algorithm is an OCL expression (activity parameter node *e*) whose AST tree is being traversed. This expression is of a certain type, e.g. it can be a comparison operation, expressed as an instance of a class OperationCallExp in OCL. Before generating the actual code for this particular node, its AST subtree is traversed: for each child node, the Bivisit algorithm is called. Only after all nodes have been visited, code is generated for each of them. If the node being visited has inherited attributes, their evaluation is performed upon entering the node.

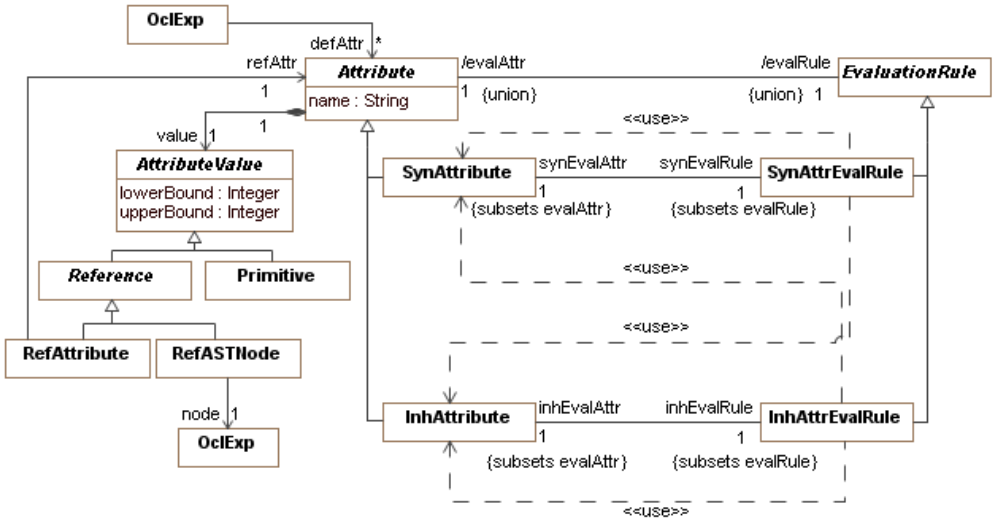

Figure 5. Attribute metamodel

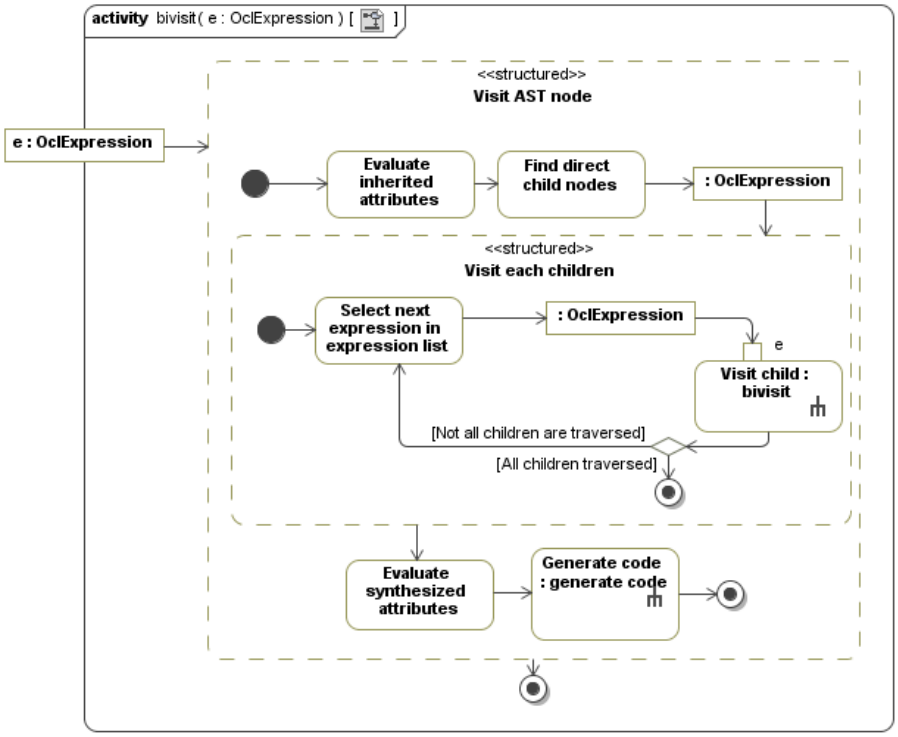

Figure 6. Algorithm for evaluating attributes in OCL AST tree

4.3. Generating code from OCL expressions using attributes

In previous sections it was described how attributes can be used to discover sub-trees of OCL AST trees. Having attribute set, identifying a certain OCL AST sub-tree, it is possible to generate code for this sub-tree. This allows adapting code to a specific data storage language and exploiting all its capabilities.

A metamodel for code generation from OCL expressions is depicted in Figure 7. AST tree (class ASTTree) consists of OCL metamodel elements (subclasses of the OclExpression class, which are instances of the OclExp class). For some of them attributes are defined (class Attribute, playing defAttr role). AST sub-trees (class ASTSubtree) consist of at least one OCL metamodel element (role node). A sub-tree can be identified by a group of attributes (class Attribute, playing identAttr role). Each sub-tree is associated with a template of a certain language (class Pattern). Each pattern has arguments. After filling-in arguments with values, the actual code is generated.

The algorithm for code generation from OCL AST trees is depicted in Figure 8. The algorithm does not generate the code instantly for each node if the code for that node has been already generated when generating code for the whole sub-tree.

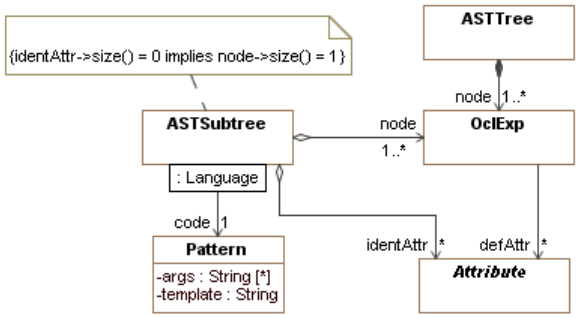

Figure 7. A metamodel for code generation from OCL

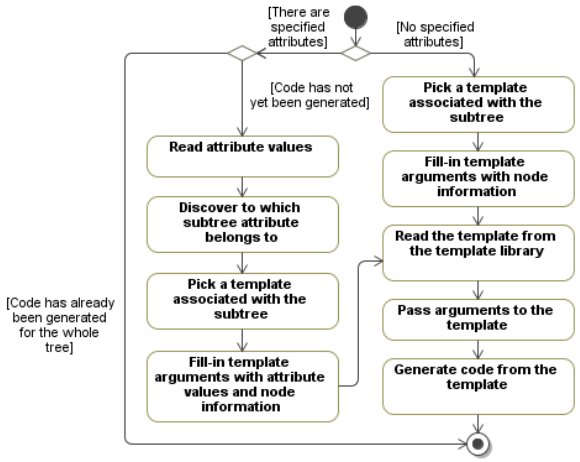

Figure 8. Algorithm for code generation from OCL AST using attributes

4.4. Using graph merging algorithms for transformation evolution and reuse

In this sub-section, principles of collaborative work over attribute sets and method for reusing transformations by reusing attributes is presented. Also, graph merging algorithm is presented upon which collaborative development of attributes is based.

4.4.1. Transformation development cycle

When working with attributes in a team, they are stored in a central repository (Figure 9). If a developer of a transformation decides to change attributes, he/she has to fetch them into the local private repository. This action is called *branching* as it creates a branch from an attribute set in a central repository. After modifying attributes, the developer commits changes to the central repository. This action requires attribute set *merging*, because two sets have to be merged: the one from the local repository with the one from the central repository. The developer may update its local repository with new changes from the central repository. This action also requires *merging*.

Attributes can be reused, i.e. attribute sets can use other attribute sets for context evaluation (Figure 10). The figure presents attribute sets A_1, A_2, A_3 and A_4. Reusable subsets of attribute sets are depicted as internal rectangles. Arrows show reused attribute sets, e.g. sets A_1 and A_3 use subset A_4' of attribute set A_4. This concept is elaborated in section 4.4.3.

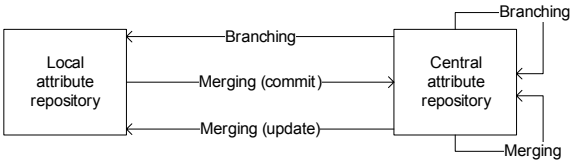

Figure 9. Attribute branching and merging schema

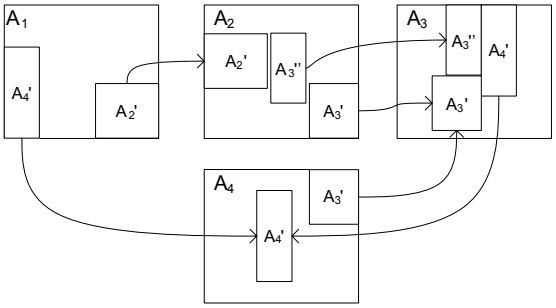

Figure 10. Attribute reusability

4.4.2. Graph merging algorithm

In this section, a method of the higher level of abstraction is presented, which does not only allow merging attribute sets, but also merges any models, that have elements and relationships between them. The presented method evaluates previous merges which is what existing methods do not do.

4.4.2.1. Merging algorithm types

We classify merging algorithms by the number of used data sources: 2-way, 3-way, and 4-way merging algorithms. Graphs are merged in two stages: changes (i.e. differences) between graphs are discovered and then they are copied into one of the compared graphs. Usually graphs residing in central and local repositories are compared.

The simplest way to merge graphs is to use 2-way merge. It uses two graphs: one from which changes are copied and one into which changes are copied. Changes are detected by comparing these two graphs. This method is unreliable because two graphs that have evolved in parallel are compared, thus it is impossible to say whether an element was deleted or created.

3-way merging algorithm resolves the unreliability problem found in the 2-way merging algorithm. In this algorithm, changes are detected between evolved graphs and their common ancestor, thus 3 data sources are used: two contributors and one common parent.

In this research, one of the biggest deficiencies of graph merging algorithms was identified that previous merges are not respected, and a 4-way merge algorithm was created, which evaluates previous merges, as shown in Figure 11. In this algorithm, branched graph is compared with a version of a branched graph, from which changes were copied last time to the same target contributor (graph G'_{m+1} is compared to G'_m).

Target graph is compared with the first target graph version that was created after the last merge into the same target from the same-branched graph. In other words, in our method it is discovered how graphs evolved from the last merge of the same direction.

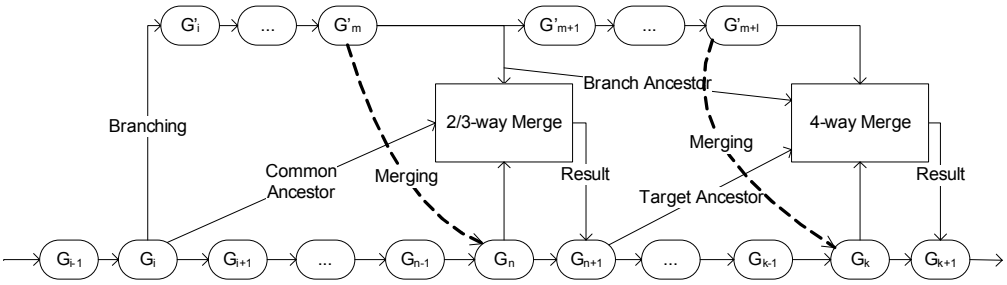

Figure 11. Schema of 4-way merge algorithm

4.4.2.2.Basic concepts of merging algorithms

The graph merging algorithm suits not only for attribute sets, but also for any models, that have elements and relationships. It can merge graphs whose metamodels conform to the one depicted in Figure 12. This metametamodel describes such metamodels, that have metaelements (class *Element*) having metaproperties (class *Property*). A metaproperty can be a primitive (class *PropertyTypedByPrimitives*) or a reference to other metaelements (class *PropertyTypedByElements*). Metaproperties can store many values whose order may be significant. If an element subclasses another element, the *general* property stores a reference to a more general element. The *isAbstract* attribute specifies whether metalement is abstract.

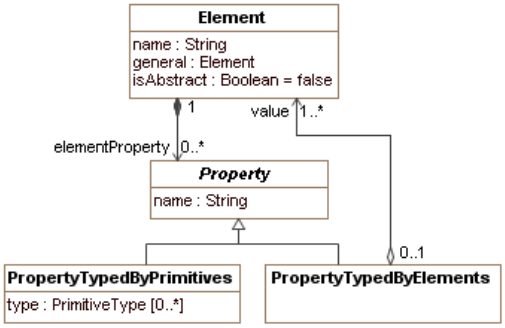

Figure 12. Graph merging metametamodel

4.4.2.3. Combined identification of elements

In existing merging algorithms elements are identified by using unique identification numbers or by analyzing relationships to other elements. In the created method, the two methods are combined into a single one. Let's define element identity function

$$\alpha(x, y),$$

where x is an element of the graph G; y – an element of the graph G'; α – function that returns values from *0* to *1*, where *1* means that elements are the same, and *0* – that they are not the same.

If compared elements x and y have unique identifiers, then function α returns *1* or *0*, because element identity can be discovered by comparing their identification numbers:

$$\text{if } \delta(x) = \delta(y), \text{ then } \alpha(x, y) = 1;$$
$$\text{if } \delta(x) \neq \delta(y), \text{ then } \alpha(x, y) = 0,$$

where δ is a function that returns a unique identifier of an element.

Elements may not have unique identifiers. If $\delta(x) = \emptyset$ or $\delta(y) = \emptyset$, then identify functions, returning values from *0* to *1,* have to be defined for these elements, i.e. $\exists f_x, f_y \in \Phi$, where f_x and f_y are identity functions for x and y elements; Φ is a set of identity functions. Then we can define element identity function $\alpha(x, y)$:

$$\text{if } f_x = f_y, \text{ then } \alpha(x, y) = f_x = f_y,$$
$$\text{if } f_x \neq f_y, \text{ then } \alpha(x, y) = 0.$$

Identity functions encode heuristic methods of identification, e.g. they compare elements by their names, properties, and relationships to other elements. If most of them are the same (e.g. *80%*), or elements are of the same type, we can state that these elements are probably identical. In such a case identity functions return equal values near to 1 (e.g. *0.8*) and $f_x = f_y$. Thus f_x and f_y can be used for identifying whether elements are identical or not.

4.4.2.4. Graph changes

A difference of two graphs is a set of elements describing how one graph differs from another graph. In the created method, elements of such a set are called *changes*. Sets of differences will be called graph differences. Graph difference is such a graph whose nodes are changes and relationships between nodes are their dependencies or conflicts.

Changes can be classified as additions, removals, attribute modifications, and attribute value order changes. Changes can depend on each other and conflict with each other. Each change can be accepted for copying into the graph or rejected. If to summarize, in the proposed method changes have the following properties:

- type (addition, removal, modification, order change);

- state (accepted or rejected);
- a reference to the changed element;
- references to changes, on which a change depends on and vice versa;
- references to conflicting changes.

A change can be selected to be copied into another graph or not. This is expressed as a change state which can be *Accepted* or *Rejected*.

When accepting one change for copying into the graph, it is sometimes needed to accept other changes too. In such a case one change depends on another. An example could be creation of a reference to a newly created element: element creation has to be accepted when accepting reference creation. Dependencies allow partial copying of changes to the graph, which is different from existing algorithms allowing copying all or no changes.

Conflicting changes are such changes that cannot be copied together into the graph. Conflicts occur in two different graph differences.

4.4.2.5. Rules for accepting and rejecting changes

In this section, we will use graphs *G'* and *G",* which evolved from their common ancestor graph *G*. Let's denote the difference of graphs *G', G* as *SG',* and the difference of graphs *G"* and *G* as *SG"*. The main idea behind the proposed merging algorithm is that when graphs *G'* and *G"* are merged into graph *MG*, it is built from graph G by using differences *SG'* and *SG"*. When the graph *MG* is created from the graph *G*, all changes that are in *Accepted* state, are copied into graph *MG*.

If change *A* depends on change *B*, then if *A* changes its state to *Accepted*, *B* also changes its state to *Accepted*. If change *A* depends on change *B* and change *B* changes its state to *Rejected*, change *A* also changes its state to *Rejected*. If change *A* conflicts with change *B* and change *A* changes its state to *Accepted*, change *B* changes its state to *Rejected*.

The merging method described in this paper differs from existing algorithms in that it does not modify change sets according to each other. Instead, it evaluates two difference sets and sets change states accordingly.

4.4.3. Reusing transformation parts

The created method introduces the concept of a module. A module is a reusable subset of an attribute set. A metamodel for modules is depicted in Figure 13.

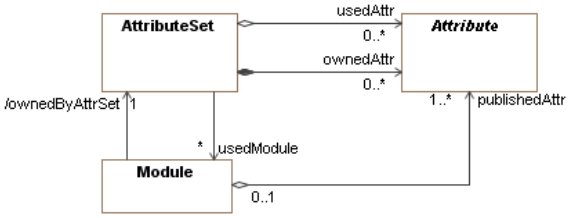

Figure 13. Module metamodel

A module makes one part of an attribute set (class *AttributeSet*) reusable (role *publishedAttr*). A module has at least one attribute. Attribute sets (class *AttributeSet*) may use other attribute sets, which form modules. If an attribute set uses a module (role *usedModule*), then all attributes from this module become accessible to the using attribute set (role *usedAttr*).

5. Method implementation and experiment

An experimental study consisting of two parts was carried out.

In the first part of the experiment it was verified whether the method is capable of adapting code generation from OCL expressions to code to a specific data storage platform. Eclipse EMF 3.4 with Dresden OCL2 Toolkit 2.0 plugin was chosen as an implementation platform, and Sun MySQL DBMS was chosen as a data storage platform.

For this purpose an attribute set was specified for transforming nested OCL conditional statements to non-nested SQL conditionals. A set of rules for such a mapping and a set of SQL templates for generating non-nested SQL conditionals were specified. The plugin for SQL code generation was developed on top of Eclipse platform and Dresden OCL2 Toolkit. A simplified UML state machine metamodel fragment with invariant containing nested OCL conditional statements was selected as a representative example and was transformed to relational schema of the chosen data storage platform (Sun MySQL DBMS). By comparing the code generated using attributes and without attributes, we have stated that the method works as expected allowing evaluation of the context and generation of a code adapted to a specific data storage platform. Also, the experiment has shown that the code, generated from nested conditionals without attributes (i.e. using standard transformation) cannot run on Oracle 10g and Microsoft SQL Server 2008 servers; it requires adaptation.

The objective of the second part of experiment was to ensure that the proposed graph merging algorithm can be applied for attribute set merging. The merging algorithm was implemented in one of the leading UML CASE tools MagicDraw 15.5 (later improved in versions 16.0 and 16.5). It allows merging UML and DSML models and is already used in commercial organizations. As a part of the experiment, the attribute metamodel was described as a domain-specific modelling language, attribute sets were expressed as DSML models. The experiment showed that changes can be copied between graphs, changes and conflicts are discovered properly, the merged graph is consistent and preserves all changes the user decided to accept.

6. Method assessment

Assessment of the generated code has shown that the concrete tested code was executed approximately 33% faster than the code generated without attributes. This assessment was done for a concrete situation and code. However, the code that is adapted to a specific platform will always run faster than code that was not adapted to it and attributes facilitate adapting the code to specific platforms and allow generating code that uses resources more efficiently.

Comparison of the created graph merging algorithm to other graph merging algorithms showed that it allows a better discovery of changes and a more flexible merging process. The algorithm discovers more change types, their dependencies and conflicts, and allows partial copying changes to the graph; it also allows easier and faster change analysis and merging by using 4-way merge. The created graph merging algorithm complies with 14 of the 17 criteria that are applied to merging algorithms that were analyzed in this research (Table 1). The rest 3 criteria are only partially satisfied as the algorithm is adopted for work in metamodel and model repository environment and has differences from code merging algorithms.

Table 1. Graph merging algorithm assessment

Algorithm / Criterion	The created algorithm (*MagicDraw UML*)	*SiDiff*	*EMF Compare*	*A. Cicchetti*	*M. Alanen* and *I. Porres*	*C. Bartelt*
Element identification by using unique identifiers	+	+/–	+/–	–	+	+
Element identification by its properties and relationships to other elements	+	+	+	+	–	–
Combined element identification	+	–	–	–	–	–
Model-based	+	–	+	+	+/–	+
Minimalistic	+	+/–	+/–	+	+	–
Self-contained	+/–	+/–	+/–	+	+	+
Transformative	+/–	+/–	+/–	+	+	+/–
Invertible	+/–	+/–	+/–	+	+	+/–
Metamodel independent	+	+/–	+	+	+	+
Attribute value order is significant	+	+	+	+	+	–
Attribute value order is insignificant	+	–	+	–	+	–
Attribute value order is significant or insignificant	+	–	+	–	+	–
Supports dependencies	+	–	–	–	+/–	–
Supports conflicts	+	–	–	+	+/–	+/–
Supports derived changes	+	–	–	–	–	–
Partial change copying	+	–	–	+/–	–	–
Evaluates previous merges	+	–	–	–	–	–
Complies fully with:	14	2	6	9	9	4
Complies fully or partially with:	17	8	12	10	12	7

7. Conclusions

As a result of creating and evaluating the method for transforming OCL to code using attributes and graph merging algorithms, we can draw the following conclusions:

1. Analysis of existing code generation from OCL methods has shown that they are insufficient for applying them in metamodel and model repositories in which OCL transformations to code have to be adapted to concrete data storage languages and developed by developer teams.
2. Analysis of attribute grammars has shown that attributes can be used in OCL-to-code transformations thus adapting such transformations to generate code for specific data storage languages and platforms. Attributes enable mapping of OCL construct groups to target language code thus exploiting all its capabilities.
3. The created graph merging algorithm and module system enables collaborative transformation development: attribute sets can be compared, reused, and merged. The created merging algorithm differs from the analyzed algorithms in that it allows combined element identification and partial copying of changes, discovers dependencies and conflicts, and evaluates previous merges. These features allow decomposing transformations into several parts, developing those parts separately and then merging results into a single transformation.
4. Method implementation in Eclipse platform and MagicDraw UML tool has shown that the method can be used in CASE tools. The merging algorithm implemented in MagicDraw UML is already used in commercial organizations.

5. An experiment was conducted during which SQL code was generated for one of the most popular open-source DBMS MySQL. The experiment has shown that attributes allow adapting code to the specific data storage platform and the code itself runs faster than the code generated without using attributes. The experiment has also shown that the code, generated from nested conditionals without attributes (i.e. code generated using standard transformation) cannot run on Oracle 10g and Microsoft SQL Server 2008 servers without adaptation.
6. The attribute set merging experiment, which was performed in MagicDraw UML tool by specifying attribute sets as domain-specific language models, has shown that the created merging algorithm works properly and can be used in teamwork for merging and reusing attribute sets.
7. The method can be developed further by creating a specialized attribute repository and evolving OCL-SQL transformations for model element searching, transformation, validation and other tasks that are relevant for collaborative model development using metamodel and model repositories.

References

[1] **S. Gudas, A. Lopata.** Meta-Model Based Development of Use Case Model for Business Function. *Information Technology and Control, Vol.* 36(3), 2007, 302–309.

[2] **O. Vasilecas, D. Bugaite.** Applying the Meta-Model Based Approach to the Transformation of Ontology Axioms into Rule Model. *Information Technology and Control, Vol.* 36(1A), 2007, 122–125.

[3] **L. Ceponiene, L. Nemuraite, G. Vedrickas.** Separation of Event and Constraint Rules in UML&OCL Models of Service Oriented Information Systems. *Information Technology and Control, Vol.* 38(1), 2009, 29–37.

[4] **S. Packevičius, A. Usaniov, E. Bareiša.** The Use of Model Constraints as Imprecise Software Test Oracles. *Information Technology and Control, Vol.* 36(2), 2007, 246–252.

[5] **V. Štuikys, R. Damaševičius.** Towards Knowledge-Based Generative Learning Objects. *Information Technology and Control, Vol.* 36(2), 2007, 202–212.

[6] **J. Coplien, D. Hoffman, D. Weiss.** Commonality and Variability in Software Engineering. *IEEE Software, Vol.* 15(6), November 1998, 37–45.

[7] **G. Hedin.** Reference Attributed Grammars. *Informatica (Slovenia), Vol.* 24(3), 2000, 301–317.

[8] **A. Poetzsch-Heffter.** Prototyping realistic programming languages based on formal specifications. *Acta Informatica, Vol.* 34(10), 1997, 737–772.

[9] **J. T. Boyland.** *Descriptional Composition of Compiler Components*: *Ph.D. thesis. University of California, Berkeley, California*, 1996.

[10] **G. Hedin.** An overview of Door Attribute Grammars. *In Proceedings of the* 5*th international Conference on Compiler Construction*, *April* 7-9, 1994, *Edinburgh, UK. London: Springer-Verlag*, 1994, 31–51.

[11] **J. T. Boyland.** Remote attribute grammars. *Journal of the ACM (JACM)*, *Vol.* 52(4), 2005, 627–687.

[12] **R. Farrow.** Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. *In Proceedings of the SIGPLAN´86 Symposium on Compiler Construction, June* 25-27, 1986, *Palo Alto, California. ACM,* 1986, 85–98.

[13] **L. G. Jones.** Efficient evaluation of circular attribute grammars. *ACM Transactions on Programming Languages and Systems*, *Vol.* 12(3), 1990, 429–462.

[14] **A. Sasaki, M. Sassa.** Circular Attribute Grammars with Remote Attribute References and their Evaluators. *New Generation Computing, Vol.* 22(1), 2004, 37–60.

[15] **E. Magnusson, G. Hedin.** Circular reference attributed grammars – their evaluation and applications. *Science of Computer Programming*, *Vol.* 68(1), 2007, 21–37.

[16] **T. Ekman, G. Hedin.** Rewritable Reference Attributed Grammars. *In Proceedings of 18th European Conference on Object-Oriented Programming* (*ECOOP* 2004), *June* 14-18, 2004, *Oslo, Norway. Springer Verlag,* 2004, 144–169.

[17] **H. H. Vogt, S. D. Swierstra, M. F. Kuiper.** Higher order attribute grammars. *In Proceedings of ACM SIGPLAN* 1989 *Conference on Programming language design and implementation*, 1989, *Portland, Oregon, United States. ACM,* 1989, 131–145.

[18] **R. Farrow, T. J. Marlowe, D. M. Yellin.** Composable attribute grammars: support for modularity in translator design and implementation. *In Proceedings of the* 19*th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1992, *Albuquerque, New Mexico, United States. ACM,* 1992, 223–234.

[19] **J. T. Boyland.** Conditional attribute grammars. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *Vol.* 18(1), 1996, 73–108.

[20] **M. Alanen, I. Porres.** Difference and Union of Models. *In Proceedings of the Unified Modeling Language, Modeling Languages and Applications,* 6*th International Conference*, *October* 20-24, 2003, *San Francisco, CA, USA. Springer,* 2003, 2–17.

[21] **A. Zündorf, J. P. Wadsack, I. Rockel.** Merging Graph-Like Object Structures. *In Proceedings of the Tenth International Workshop on Software Configuration Management*, *May* 12-19, 2001, *Toronto, Canada. IEEE Computer Society,* 2001.

[22] **S. Sudarshan, A. R. Chawathe, H. Garcia-Molina, J. Widom.** Change Detection in Hierarchically Structured Information. *In Proceedings of the* 1996 *ACM SIGMOD international Conference on Management of Data*, *June* 4-6, 1996, *Montreal, Quebec, Canada. New York: ACM,* 1996, 493–504.

[23] **Y. Wang, D. J. DeWitt, J. Cai.** X-Diff: An Effective Change Detection Algorithm for XML Documents. *In Proceedings of the* 19*th International Conference on Data Engineering (ICDE'03)*, *March* 5-8, 2003, *Bangalore, India. IEEE Computer Society,* 2003, 519–530.

[24] **A. Cicchetti, D. Di Ruscio, A. A. Pierantonio.** Metamodel Independent Approach to Difference Representation. *Journal of Object Technology, Vol.* 6(9), 2007, 165–185.

[25] **A. Cicchetti, D. Di Ruscio, R. Eramo, A. Pierantonio.** Automating Co-evolution in Model-Driven Engineering. *In Proceedings of the* 2008 12*th international IEEE Enterprise Distributed Object Computing Conference – Volume* 00, September 15-19, 2008, *Washington, DC, USA. IEEE Computer Society,* 2008, 222–231.

[26] **A. Cicchetti, D. Di Ruscio, A. Pierantonio.** Managing Model Conflicts in Distributed Development. *In Proceedings of the* 11*th international Conference on Model Driven Engineering Languages and Systems*, *September* 28 *– October* 03, 2008*, Toulouse, France. Springer-Verlag,* 2008, 311–325.

[27] **C. Bartelt.** Consistence preserving model merge in collaborative development processes. *In Proceedings of the* 2008 *international Workshop on Comparison and Versioning of Software Models*, *May* 17, 2008, *Leipzig, Germany. New York: ACM,* 2008, 13-18.

[28] **U. Kelter, J. Wehren, J. Niere.** A Generic Difference Algorithm for UML Models. *Software Engineering* 2005, *LNI Vol.* 64, 2005, 105–116.

[29] **D. Ohst, M. Welle, U. Kelter.** Differences between versions of UML diagrams. *In Proceedings of the* 9*th European Software Engineering Conference Held Jointly with* 11*th ACM SIGSOFT international Symposium on Foundations of Software Engineering*, *September* 1-5, 2003, *Helsinki, Finland. New York: ACM,* 2003, 227–236.

[30] **P. Selonen.** A Review of UML Model Comparison Approaches. *In Proceedings of the* 5*th Nordic Workshop on Model Driven Engineering, August* 27-29, 2007, *Ronneby, Sweden. Blekinge Tekniska Högskola,* 2007, 43–51.

[31] **E. Pakalnickiene, L.Nemuraite.** Checking of Conceptual Models with Integrity Constraints. *Information Technology and Control*, *Vol.* 36(3), 2007, 285–294.

[32] **K. Czarnecki, S. Helsen.** Classification of Model Transformation Approaches. *In J. Bettin, G. Van E. Boas, A. Agrawal, E. Willink, J. Bezivin (eds.) Proceedings of the* 2*nd OOPSLA workshop on Generative Techniques in the Context of Model-driven Architecture, October* 2003. *ACM Press,* 2003, 1–17.

[33] **D. Šilingas, R. Butleris.** Towards Implementing a Framework for Modeling Software Requirements in MagicDraw UML. *Information Technology and Control, Vol.* 38(2), 2009, 153–164.

[34] **N. Neff.** Attribute Based Compiler Implemented Using Visitor Pattern. *In Proceedings of the* 35*th SIGCSE technical symposium on Computer science education*, *March* 3-7, 2004, *Norfolk, Virginia, USA. ACM,* 2004, 130–134.

Received July 2009.