JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2006**

Vol. 5, No. 4, Mai-June 2006

Constraint Validation in Model Compilers

László Lengyel, Tihamér Levendovszky, Hassan Charaf, Budapest University of Technology and Economics, Hungary

Abstract

Model transformation has become one of the most focused research field, motivated by for instance the OMG's Model-Driven Architecture (MDA). Metamodeling is a central technique in the design of visual languages, and it reuses existing domains by extending the metamodel level. Metamodel-based software development requires the transformation of the models between various stages. These transformation steps must be formally and precisely specified, which can be accomplished along with constraints enlisted in transformation steps. Our metamodel-based approach uses graph rewriting techniques for model transformation. This paper summarizes our results related to the metamodel-based constraint validation during the model transformation. This work presents the Rule Constraint Validator (RCV) algorithm, the Invariant Analysis (IA) algorithm, the Persistent Analysis (PA) algorithm and the combination of the RCV and PA algorithms which results the Optimized Rule Constraint Validator (ORCV) algorithm. An illustrative case study for constraint validation in rewriting rules is also provided.

1 INTRODUCTION

OMG's Model Driven Architecture [OMG MDA] offers a standardized framework to separate the essential, platform independent information from the platform dependent constructs and assumptions. A complete MDA application consists of a definitive platform-independent model (PIM), and one or more platform-specific models (PSM) and complete implementations, one on each platform that the application developer decides to support. The platform-independent artifacts are mainly UML models containing enough specification to generate the platform dependent artifacts automatically by so-called model compilers. Hence software model transformation provides a basis for model compilers, which plays central role in the MDA architecture.

Model transformation means converting an input model that is available at the beginning of the transformation process to an output model, and it is a possible solution for model compiler realization. Model compilers can support properties to guarantee, preserve or validate them, and the presented approach is a practical application of these mechanisms.

Models can be considered special graphs; simply contain nodes and edges between them. This mathematical background makes possible to treat models as labeled graphs and to apply graph transformation algorithms to models using graph rewriting [Levendovszky04a] [Levendovszky04b]. The steps of graph transformation are rewriting rules, each rewriting rule consists of a left-hand side graph (LHS) and right-hand side graph (RHS). Previous work [Levendovszky04a] has introduced an approach, where LHS and RHS of the rules are built from metamodel elements It means that an instantiation of LHS must be found in the graph to which the rule is applied to (host graph) instead of the isomorphic subgraph of the LHS. Hence LHS and RHS graphs are the metamodels of the graphs which we search and replace in the host graph.

Often it is not enough to match graphs based on the topological information only, we want to restrict the desired match by other properties, e.g. we want to match a subgraph with a node which has a special property or there is a unique relation between the properties of the matched nodes. For example we want to match a node with a special integer type property which value is between 3 and 7. The metamodel-based specification of the rules allows assigning OCL [OMG OCL] constraints to the rules, using the guidelines of the UML standard [OMG UML]. Because these constraints are bound to the rules, they are able to express constraints local to the host graph area affected by the rules. This approach is inherently a local construct, because the elements not appearing in LHS or RHS cannot be directly included in the OCL statements. Although the specification has this local nature, it does not mean that validating them does not involve checking other model elements in the input model: constraint propagation needs to be taken into account by both the algorithms and the user of the transformation on specifying constraints. The OCL constraints which are enlisted in LHS and RHS graphs affect the matched instances of LHS and RHS graphs. A *transformation* and a *finite sequence of steps* consist of n number of rewriting rules (where $n > 0$) in an ordered sequence. This sequence defines the execution order of the containing rewriting rules. The difference between a *transformation* and a *finite sequence of steps* is that a finite sequence of steps always terminates. A transformation, however, can contain infinite number of steps.

Constraints (pre- and postconditions) facilitates to specify precisely the execution of the transformation steps and the whole transformation. Using constraints, we can specify the conditions of step firing and the required outputs.

If a transformation contains steps specified properly with the help of constraints, and the transformation has been executed successfully for the input model, then the generated output model is in accordance with the expected result, which is described by the steps of the transformation refined with the constraints. It means that the modeler’s task is to create proper transformation steps and fully specify them with constraints (pre- and postconditions) then if the execution of the transformation finishes successfully, it produces a valid result. A sample transformation step is presented in Fig. 1.

This work discusses new results, experiences and consequences evolved from the implementation and further conceptual development of constraint validation and application in a metamodel-based model transformation system. Our base algorithm is called Rule Constraint Validator (RCV) algorithm, which firstly finds the matches in the graph to which the rewriting rule is being applied, and having found the matches, it checks the constraints contained by the rewriting rule. If all of the preconditions are satisfied, the algorithm fires the transformation step. The aim of the work presented here was to analyze the possibilities of the acceleration and the optimization of the RCV algorithm. The main idea of the optimization is that the constraint validation is performed before and during the matching process and not after the matching. This early constraint validation accelerates the whole transformation process, since with the help of this method it is discovered earlier whether there is no proper match in a given position because of constraint contradiction. This work presents the Invariant Analysis (IA) algorithm which examines the constraints in the rewriting rule against the metamodel, and the Persistent Analysis (PA) algorithm which executes the constraint validation in the matching time. The paper analyses the computational complexity of the algorithms and the computational time that we can save using them.

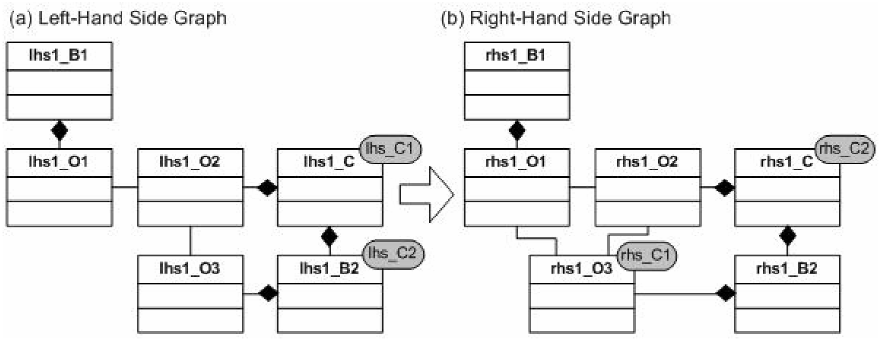


Fig. 1: Sample transformation step: (a) LHS of the transformation step, (b) RHS of the transformation step, and lhs_C1, lhs_C2, rhs_C1, rhs_C2 are the constraints (pre- and postconditions) assigned to the rule nodes

The rest of the paper is organized as follows: Section 2 describes the backgrounds and the related work, Section 3 (i) introduces the Visual Modeling and Transformation System (VMTS) [VMTS], which is our implemented metamodel-based transformation system, (ii) presents the RCV algorithm, (iii) provides the optimization possibilities (IA, PA, ORCV algorithms) and (iv) discusses the computational complexity of the presented algorithms. In Section 4 a case study illustrates the constraint checking facilities of the presented method, and in Section 5 conclusions are drawn and future work is presented.

2 BACKGROUNDS AND RELATED WORK

The purpose of contracts [Meyer88] is to help to build better software by organizing the communication between software elements through specifying the mutual obligations. Contracts are used to guarantee that these communications occur on the basis of precise specifications of what these services are going to be. For the software to be able to guarantee any kind of correctness and robustness properties, they must know the precise constraints over such communications. In a client/supplier relationship, where the client needs a certain service and the supplier provides that service, the client has to fulfill certain obligations before calls the supplier. These preconditions are obligations for the client. In the other direction, we are going to express the conditions that the supplier routine must guarantee to the client on completion of the supplier's task. That is the postcondition of the contract, specifically, the postcondition of that particular routine. The postcondition is also an obligation for the supplier. Besides pre- and postconditions the third fundamental elements of contracts are invariants. A class invariant is a condition that applies to an entire class. It describes a consistency property that every instance of the class must satisfy.

The Object Constraint Language [OMG OCL] is a formal language for analysis and design of software systems. It is a subset of the industry standard Unified Modeling Language [OMG UML] that allows software developers to write constraints and queries over object models. A constraint is a restriction on one or more values of an object-oriented model or system. There are four types of constraints: (i) An *invariant* is a constraint that states a condition that must always be met by all instances of the class, type, or interface. (ii) A *precondition* to an operation is a restriction that must be true at the moment that the operation is going to be executed. The obligations are specified by postconditions. (iii) A *postcondition* to an operation is a restriction that must be true at the moment that the operation has just ended its execution. (iv) A *guard* is a constraint that must be true before a state transition fires. Besides these applications, OCL can be used as a navigation language as well.

Graph rewriting [Rozenberg97] [Ehrig97] [Blostein95] is a powerful tool for graph transformation with strong mathematical background. The atoms of graph transformation are rewriting rules, each rewriting rule consists of a left-hand side graph (LHS) and right-hand side graph (RHS). Applying a graph rewriting rule means finding an isomorphic occurrence (match) of LHS in the graph to which the rule is being applied (host graph), and replacing this subgraph with RHS. Replacing means removing elements which are in LHS but not in RHS, and gluing elements which are in RHS but not in LHS. Replacing process consists of two steps: removing and gluing, this approach is the so-called double pushout (DPO) [Rozenberg97].

The metamodel-based constraint checking method presented later benefits from the results of the mathematical background of formal languages, graph rewriting and research related to the metamodel-based software model transformation. It also incorporates several ideas from other existing environments [Akehurst03], [Hamie98] [Loecher03], which implement the OCL, and enable constraints to be checked over models.

The GReAT framework [Karsai03] is a transformation system for domain specific languages (DSL), it is built on metamodeling and graph rewriting concepts; it uses a proprietary notation and interpretation instead of instantiation between the rules expressed with meta elements and the match.

PROGRES [PROGRES] is a visual programming language in the sense that it has a graph-oriented data model and a graphical syntax for its most important language constructs. PROGRES supports pre- and postconditions. The precondition of a transaction is a query, which should never fail, being applied to the input graph of the surrounding transaction. Similarly the postcondition of a transaction is a query, which should never fail, being applied to the output graph of the surrounding transaction.

3 CONSTRAINT VALIDATION

The Visual Modeling and Transformation System

The block diagram of the Visual Modeling and Transformation System (VMTS) [Levendovszky04a] [VMTS] is depicted in Fig. 2. VMTS is an n-layer multipurpose modeling and metamodel-based transformation system. The user interface (Adaptive Modeler) are functionally separated from the model storage unit (AGSI Core - Attributed Graph Architecture Supporting Inheritance), which uses an RDBMS to store the model information. The model transformation can be accomplished by Traversing Model Processors [Levendovszky04b], Rewriting Engine and other applications. The AGSI Core exposes its interface to any other applications, which may use a proprietary technique to process AGSI data.

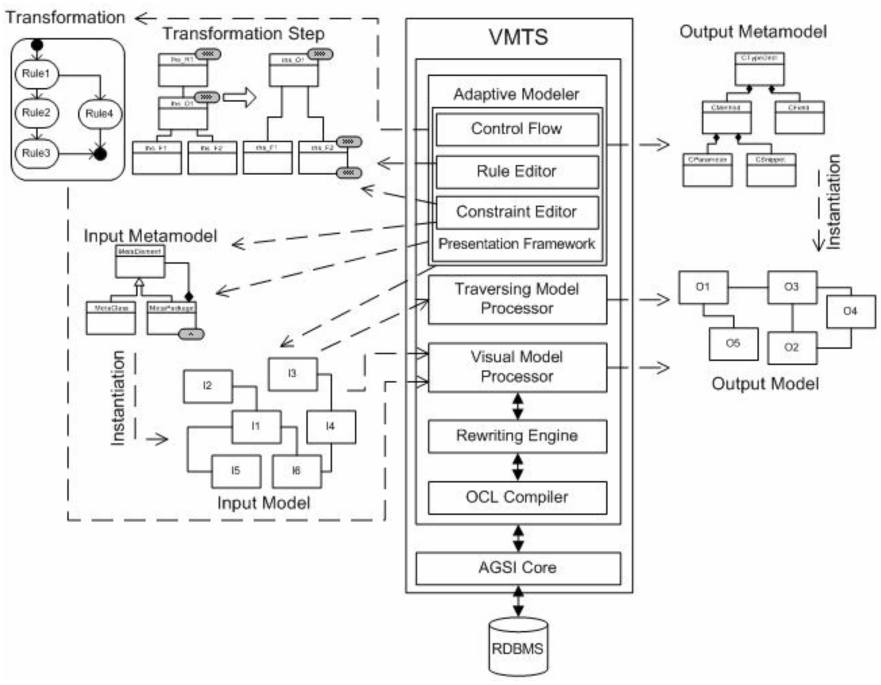


Fig. 2: Block diagram of VMTS

Using this environment it is easy to edit metamodels, design models according to their metamodels, transform models using graph rewriting [Levendovszky04a] [Levendovszky04c]. The Validation Module, which is a part of the AGSI Core facilitates to check constraints contained by metamodels during the metamodel instantiation, and to validate the constraints in the rewriting rules during the graph transformation process [Lengyel05a].

A *precondition* (*postcondition*) assigned to a rewriting rule is a boolean expression that must be true at the moment when the rewriting rule is fired (after the completion of a rewriting rule). If a *precondition* of a rewriting rule is not true then the rewriting rule fails without being fired. If a *postcondition* of a rewriting rule is not true after the execution of the rewriting rule, the rewriting rule fails. A direct corollary of this is that an OCL expression in LHS is a precondition to the rewriting rule, and an OCL expression in RHS is a postcondition to the rewriting rule. A rewriting rule can be fired if and only if all the conditions enlisted in LHS are true. Also, if a rewriting rule finished successfully, then all the conditions enlisted in RHS must be true. There are three properties: validation, preservation, and guarantee, which are checked during the rewriting process [Lengyel05a].

The Rule Constraint Validator (RCV) Algorithm

Fig. 3 presents a block diagram to illustrate how the RCV algorithm [Lengyel05a] of the VMTS checks the constraints contained by the rewriting rule during the rewriting process. Recall that LHS and RHS of the rewriting rules are built from the metamodel elements. It is possible in VMTS that LHS and RHS use different metamodels, but for the sake of simplicity in the block diagram they have common metamodel. The rewriting rule contains OCL constraints. VMTS does not interpret the constraints during the rewriting, but a binary is used, which is generated by an OCL Compiler [Lengyel05b]. The rewriting process uses matches found by the matching process and the compiled binary to validate the constraints on the matched parts of the host graph. If and only if a match satisfies the constraints (preconditions), then the rewriting process generates the rewriting result, and if and only if the rewriting result satisfies the postconditions, then the step was successful. In Fig. 3 the rewriting result is also an instance model of the metamodel, because LHS and RHS use the same metamodel.

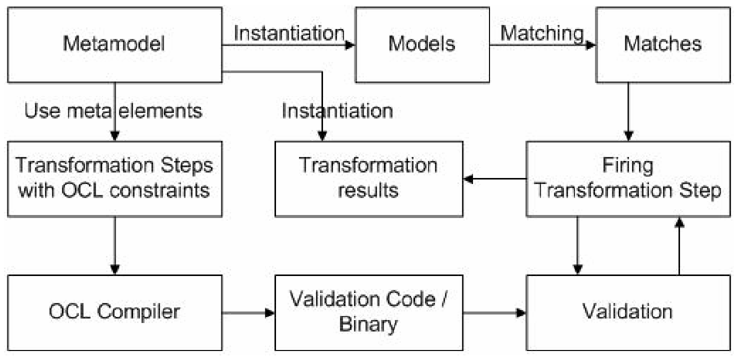


Fig. 3: The block diagram of the constraints checking during the rewriting process (RCV algorithm)

Proposition 1. The computational complexity of the RCV algorithm for constraint checking is the following.

a) $O(s) + O(\lg v_m)$ – once for a rewriting rule, and
b) $O(\lg v)$ – for each rule firing.

Where s is the number of the symbols in the OCL constraints, v_m is the number of the metamodel vertices and v is the number of the host graph (model) vertices.

The following pseudo codes illustrate the transformation execution (EXECUTE_TRANSFORMATION) and the constraint validation (CHECK_CONSTRAINTS). The EXECUTE_TRANSFORMATION method executes the rewriting rules contained by the given transformation on the passed match. If the rewriting rule has no generated validation binary, then it generates the validation code and binary based on the constraints contained by the rewriting rule. It means that for each rewriting rule this step has to be performed only for the first time. If a step has a precondition, the CHECK_CONSTRAINTS method is called, and if it returns false then the execution of the step and the whole finite sequence of steps fails, thus the algorithm returns false. Otherwise the EXECUTE_TRANSFORMATION method calls the FIRE_RULE function, and after that if the rule has a postcondition then the procedure is similar to that of the preconditions. The pseudo codes of the algorithm are as follows:

```
EXECUTE_TRANSFORMATION (VMTSTransformation transformation,
VMTSMatch match): bool
  foreach VMTSRule rule of
    transformation.getRulesInOrderedSequence()
     if not rule.HasGeneratedValidationCode and not
        GENERATE_VALIDATION_CODE(rule.Constraints)
        then return false
     if rule.hasPrecondition and not
        (CHECK_CONSTRAINTS(rule.PreConditions, match))
        then return false
     FIRE_RULE(rule, match)
     if rule.hasPostcondition and not
       (CHECK_CONSTRAINTS(rule.PostConditions, match))
        then return false
  end foreach
  return true

CHECK_CONSTRAINTS (VMTSConstraint[] constraints, Hashtable
match): bool
  foreach VMTSConstraint constraint of constraints
     destNode = NAVIGATE_TO_DESTNODE(constraint, match)
     if not CHECK(constraint, destNode) then return false
  end foreach
  return true
```

The Invariant Analysis (IA) Algorithm

We assume the case that all model conforms to its metamodel, every host graph satisfies its constraints defined in the metamodel. We can make that assumption without restricting the generality, because using metamodel-based tools the created models have to conform to their metamodels. Since a rewriting rule in VMTS is built from metamodel elements, we can apply a rewriting rule for a host graph if and only if the host graph and LHS of the rewriting rule have the same metamodel, or the metamodel of the host graph contains the metamodel of LHS graph. In other cases it is not possible to find match in host graph because LHS graph of the rewriting rule and the metamodel of the host graph contain different types.

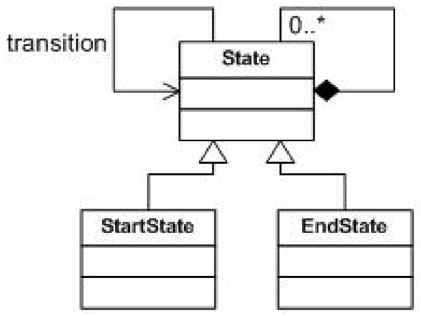


Fig. 4: Metamodel of the statechart diagram

Invariant Analysis means the comparison of the constraints contained by the metamodel and the rewriting rule. The goal of the Invariant Analysis is to find out immediately after the rewriting rule creation if a constraint in the rewriting rule contradicts to a constraint in the metamodel. This is an earlier phase where we can explore whether there is any contradiction between the constraints. Invariant Analysis can modify the *IsAlreadyChecked* property of a constraint in the rewriting rule to true if it is follows from a constraint contained by the metamodel that the examined rewriting rule constraint holds in every case. The following constraints provide an example. Constraint from metamodel (in Fig. 4 is depicted the metamodel of the statechart diagram):

```
context EndState inv InTransitions_metamodel:
   self.NumberOfInTransitions = 2
```

Constraint from rewriting rule:

```
context EndState inv InTransitions_rewriting_rule:
   self.NumberOfInTransitions >= 1 and
   self.NumberOfInTransitions <= 3
```

Invariant Analysis is an offline validation; it is not possible to validate pre- and postconditions offline, but we can check those constraints which use only type and not instance-specific properties.

After the creation of a rewriting rule the rule is modified only few times, but it is fired optional times. IA is independent from the rule firing; it runs only few times for a rewriting rule, immediately after the rule creation and modification, therefore the complexity of the IA does not increase the complexity of the transformation process.

To summarize the Invariant Analysis compares the constraints contained by the metamodel and the rewriting rules, and decides which constraints in the rewriting rule will be certainly proper and which not. The pseudo code of the algorithm is as follows:

```
INVARIANT_ANALYSIS (int ruleID, out
VMTSConstraintContradictionList contradictionList): bool
   VMTSRuleNode[] ruleNodes = getRuleNodesByRuleID(ruleID)
   foreach VMTSRuleNode ruleNode in ruleNodes
      VMTSNode typeNode = getMetaNodeByID(ruleNode.TypeID)
      VMTSConstraintContradictionList currentContradictionList
        = COMPARE_CONSTRAINTS(ruleNode.Constraints,
        typeNode.Constraints)
      if currentContradictionList != null then
        contradictionList.Add(currentContradictionList)
   end foreach
   if contradictionList.Count > 0 then return false
   else return true
```

The INVARIANT_ANALYSIS method queries the rule nodes contained by the rewriting rule based on the given *ruleID*. This method affects the rule nodes contained by both the LHS and the RHS graph. In the *foreach* loop the algorithm queries the metamodel node (*typeNode*) by the *typeID* of the actual rule node and calls the COMPARE_CONSTRAINTS method with the rule node and the constraints contained by the *typeNode*. The COMPARE_CONSTRAINTS method compares the passed constraints, and returns the contradictions if any. If there are returned contradictions between the constraints contained by the metamodel and the rewriting rule, then the algorithm places them into the *contradictionList*, which is an output parameter. The return value of the INVARIANT_ANALYSIS is false if there was at least one contradiction, true otherwise.

In AGSI Core the data is stored in datasets, which are an in-memory cache of the data retrieved from the database. There is a dataset for the actually used metamodel, one for the actually used model, and one for the actually used transformation. These datasets contain only two data tables: one for nodes and one for edges. We denote the number of the actual transformation (metamodel, model) nodes with v_r (v_m, v) and the edges with e_r (e_m, e). It means that the data tables containing the actual transformation (rewriting rules) nodes and edges have v_r and e_r data rows. The complexity of finding a row in a data table is O(lg n), where n is the number of the rows. In AGSI Core there are a Facade and a Manager layers which means O(4) steps to move the objects through these layers. Therefore the complexity to query a *VMTSRuleNode* or a *VMTSRuleEdge* from the dataset of the actual transformation is O(4) + O(lg v_r) = O(lg v_r) and O(lg e_r) steps. Similarly to obtain a metamodel (model) node or edge means O(lg v_m) and O(lg e_m) (O(lg v) and O(lg e)) steps.

The complexity of the *getRuleNodesByRuleID* method is O(lg v_r) and the complexity of the *getMetaNodeByID* method is O(lg v_m). The *foreach* loop runs for each rewriting rule node. The complexity of the COMPARE_CONSTRAINTS method depends on the number of the constraints contained by the passed rule node and the metamodel node. Let c_{ri} be the number of the constraints contained by the i^{th} passed rule node and c_{mi} the number of the constraints contained by the metamodel node of the same type. The algorithm needs $\mathrm{O}(c_{ri} * c_{mi})$ time to compare the constraints contained by the i^{th} rule node with the corresponding constraints in the metamodel node. The complexity of the complete *foreach* method is $\mathrm{O}\sum_{i=1}^{v_r} \lg v_m + (c_{ri} * c_{mi})$.

If we use the Invariant Analysis during the rewriting rule creation, and the algorithm detects that at least one of the constraints in the rewriting rule contradicts to a constraint in the metamodel, then the algorithm marks the rewriting rule which interferes the execution of the rule, because it would be unsuccessful and therefore the saved computational time is $SC_{IA} \geq n^k - C_{IA}$. SC_{IA} denotes the saved computational time using IA, n^k is the complexity of the matching process (where n is the number of the host graph nodes, and k is the number of nodes in the matched subgraph) and C_{IA} is the computational complexity of the IA.

Proposition 2.

a) The computational complexity of the Invariant Analysis algorithm is $\mathrm{O}(\lg v_r) + \mathrm{O}\sum_{i=1}^{v_r} \lg v_m + (c_{ri} * c_{mi})$.

b) The computational time which we can save using the Invariant Analysis algorithm is $SC_{IA} \geq n^k - C_{IA}$.

The saved computational time presented here stands only for the first time of rule firing, because we can assume that after the warning the rule creator will correct the wrong constraint. If the constraint is not fixed then the execution of the rule is interfered and the saved computational time is n^k at every try to fire the rule.

The Persistent Analysis (PA) Algorithm

The PA algorithm checks the constraints continuously during the matching process. As it has already been mentioned before, every constraint in the rewriting rule has a property *IsAlreadyChecked*, this property is false at the beginning of the matching (except if the IA algorithm changed it to true). If PA algorithm has all information to validate a constraint (precondition) then checks it, and if holds then sets the *IsAlreadyChecked* property of the constraint to true, otherwise if the constraint does not hold, the step fails. After the matching process starts the rewriting, the first step of this process is the validation of all the preconditions in the match, but in fact the algorithm has to check only the constraints with false *IsAlreadyChecked* value. This method accelerates the whole algorithm, because if it is revealed in the matching phase that one of the preconditions does not hold, then we can leave the current branch, backtrack and continue the matching from another position. This saves the computational time of finishing the matching at the given position and the time having elapsed between the start of the rewriting and finding the contradiction.

The computational complexity of the constraint validation during the matching process is at most $\mathrm{O}\sum_{i=1}^{k} c_{ri} + c_i$, where k is the number of nodes in the matched subgraph, c_{ri} is the number of the constraints contained by the rule node, which is matched to the i^{th} host graph node, and c_i is the number of constraints contained by the type node of the i^{th} matched host graph node. The expression 'at most' in the previous sentence refers to the worst case: the algorithm does not find a contradiction, and has to validate all constraints, otherwise if it finds an unsatisfied constraint then it does not have to continue the constraint validation, and the computational complexity is less. In this algorithm we can also assume the case that the host graph instantiates its metamodel, and in this case the computational complexity is $\mathrm{O}\sum_{i=1}^{k} c_{ri}$. While each rule node contains constant number of constraint, the computational complexity in fact is O(k). The saved computational time depends on the time, when the algorithm finds the first not satisfied constraint. If all preconditions are satisfied, the algorithm does not find a contradiction, and the saved computational time comes only from the fact that the constraint evaluation is faster during the matching because the host nodes are directly available. If there is at least one unsatisfied constraint then the saved computational complexity is the complexity of the unexecuted part of the matching algorithm. If we find a contradiction at the beginning, the saved computational complexity is near n^k.

Proposition 3.

a) The computational complexity of the constraint validation during the matching process (PA algorithm) is at most O(k), where k is the number of nodes in the matched subgraph.
b) The PA algorithm does not increase the total time of the rule firing, and using the PA algorithm the saved computational time is n^{k-r} – if the algorithm finds an unsatisfied constraint, while validates the constraints on the r^{th} matched host graph node.

The Optimized Rule Constraint Validator (ORCV) Algorithm

The RCV algorithm validates constraints after the matches are found, the optimization take advantage that it is possible to validate certain constraints during the matching process. The ORCV algorithm completes RCV algorithm with PA algorithm which results a more powerful constraint validation algorithm.

Proposition 4. The computational complexity of ORCV algorithm never exceeds the computational complexity of RCV algorithm (The computational complexity of ORCV algorithm only in the worst case reaches the computational complexity of RCV algorithm):

a) $C_{ORCV} = C_{RCV}$ (The computational complexity of ORCV algorithm equals with the computational complexity of RCV algorithm), if PA algorithm can not validate any constraint during the matching process.

b) The computational complexity of ORCV algorithm is less at least $p * \lg v$ than the computational complexity of RCV algorithm, if the PA algorithm can validate *p* constraints during the matching process and all of the constraints are satisfied. In this case the saved computational time is $SC_{ORCV} \geq p * \lg v$.

   The saved computational time comes from the fact that the nodes on which the algorithm has to validate the constraints are available during the matching process. (The formula contains the ≥ operator because if the constraints contain navigation steps and the host graph nodes, which are part of the navigation path, are already matched then they are directly available.)

c) The computational complexity of ORCV algorithm is less at least n^{k-r} than the computational complexity of RCV algorithm, if PA algorithm finds an unsatisfied constraint, while validates the constraints on the r^{th} matched host graph node.

   (The formula contains the ≥ operator, because if the algorithm finds a contradiction, the saved computational time also contains the complexity of the unexecuted part of the rewriting algorithm.)

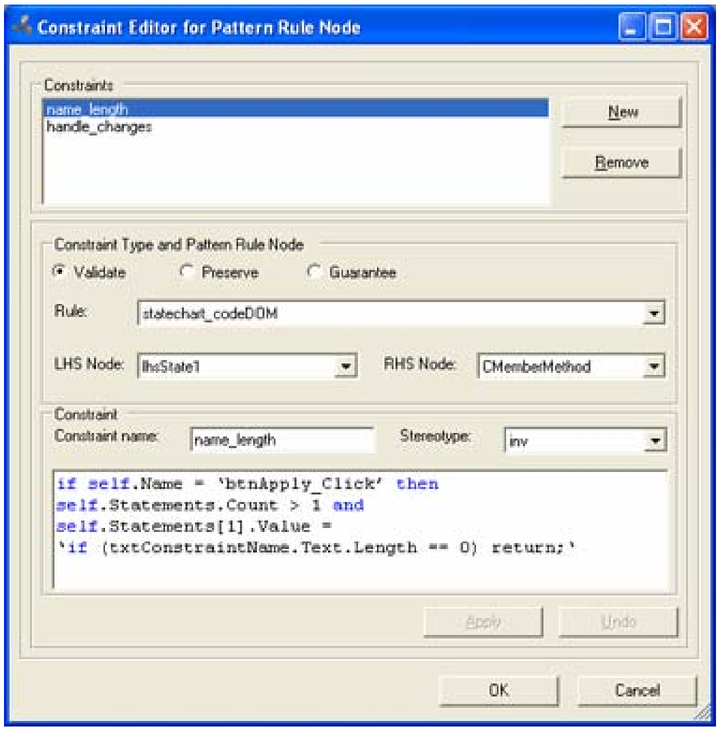
Where C_{ORCV} is the computational complexity of ORCV algorithm and C_{RCV} is the computational complexity of RCV algorithm. *p* denotes the number of the validated constraints by PA in the case when all the checked constraints are satisfied, *v* is the number of the nodes in the host graph (lg *v* is the complexity of querying a host graph node). SC_{ORCV} is the saved computational time using ORCV algorithm and n^{k-r} is the saved computational time if PA algorithm finds an unsatisfied constraint while validates the constraints on the r^{th} matched host graph node.

4 A CASE STUDY

Using a case study we introduce how VMTS generates the user interface handler code based on the statechart model for a form designed with a Form Editor; and the programmer needs to write the application specific parts only. The goal of this method is that if the statechart is specified in detail, then the generated code will handle the user interface of the system described by the statechart model.

In Fig. 5 a screenshot of VMTS Constraint Editor for Pattern Rule Node form is presented, its operation is modeled with a statechart diagram (Fig. 6). The user interface edition of the Constraint Editor for Pattern Rule Node form is accomplished with form designer of the Visual Studio.NET, but the handler code is automatically generated from the statechart model.

In VMTS it is possible to assign constraints to a selected rule node or to a transformation which consists of several transformation steps. In the prior case we use the Constraint Editor for Pattern Rule Node form for constraint specification. When the form appears with an empty list box *Constraints* (*lbConstraints*), buttons *New* (*btnNew*) and *Cancel* (*btnCancel*) are enabled and the rest of the controls are disabled. User can create new constraint with button *New* or exit with button *Cancel*. Clicking on button *New* the name of the newly created constraint appears in list box *Constraints*, the controls which contains the properties of the constraints become enabled, the constraint name appears in the text box *Constraint name* (*txtConstraintName*), and buttons *Remove* (*btnRemove*) and *OK* (*btnOK*) become enabled. In this state the user can specify the constraint properties: the transformation step (*cmbRule*), LHS and RHS rule nodes (*cmbLHSNode* and *cmbRHSNode*), the type of the constraint (*rdbValidate*, *rdbPreserve*, *rdbGuarantee*), the stereotype (*cmbStereotype*) and the text of the constraint (*txtActual*). If the user makes some modification, then buttons *Apply* (*btnApply*) and *Undo* (*btnUndo*) become enabled. Using *Apply* the user can accept, or using *Undo* reject the changes. If the user makes some modifications, does not apply them and tries to create new constraint or select another one from the list box *Constraints*, then the form shows a message box with the following question: "*Do you want to save your changes?*", on the message box there are buttons *Yes*, *No* and *Cancel*, and user can decide how to continue the constraint specification.


Constraint Editor for Pattern Rule Node

Constraints: name_length, handle_changes

Constraint Type and Pattern Rule Node: [✓ Validate] Preserve Guarantee

Rule: statechart_codeDOM

LHS Node: lhsState1 RHS Node: CMemberMethod

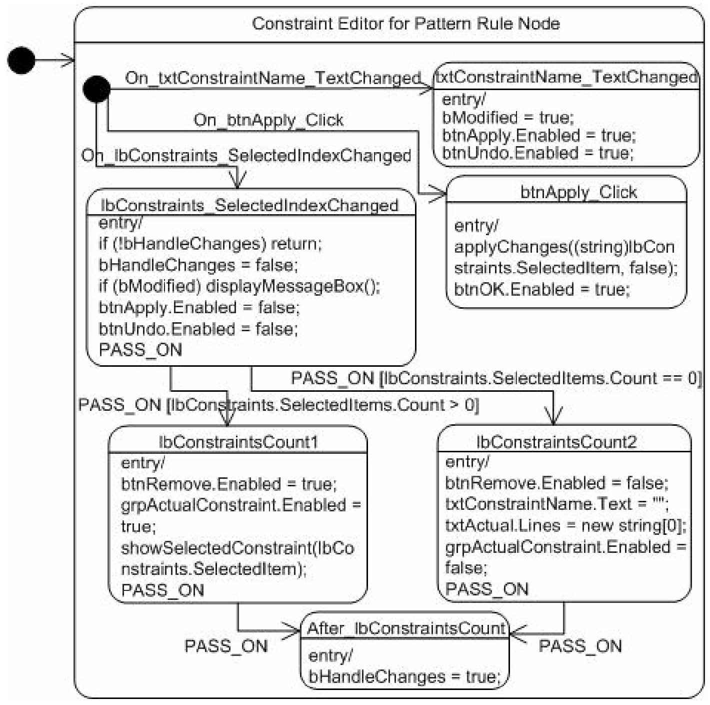
Constraint name: name_length Stereotype: inv

```
if self.Name = 'btnApply_Click' then
self.Statements.Count > 1 and
self.Statements[1].Value =
'if (txtConstraintName.Text.Length == 0) return;'
```

Fig. 5: Constraint Editor for Pattern Rule Node form of the VMTS

The incomplete statechart diagram of form Constraint Editor for Pattern Rule Node is presented in Fig. 6, where only three events are modeled: *txtConstraintName_TextChanged*, *btnApply_Click* and *lbConstraints_SelectedIndexChanged*. The complete statechart diagram is too large to present here but it can be accessed in [VMTS].

In Fig. 6 one can see that every event has at least one handler state. E.g. if the *On_btnApply_Click* event is fired then the *btnApply_Click* state handles it. The *On_lbConstraints_SelectedIndexChanged* event is managed by four states: *lbConstraints_SelectedIndexChanged*, *lbConstraintsCount1*, *lbConstraintsCount2*, and *After_lbConstraintsCount*. This event handler is decomposed because the handling code depends on the value of the *lbConstraints.SelectedItems.Count* property.


Constraint Editor for Pattern Rule Node
On_txtConstraintName_TextChanged
On_btnApply_Click
On_lbConstraints_SelectedIndexChanged
txtConstraintName_TextChanged
btnApply_Click
lbConstraints_SelectedIndexChanged
PASS_ON [lbConstraints.SelectedItems.Count == 0]
PASS_ON [lbConstraints.SelectedItems.Count > 0]
lbConstraintsCount1
lbConstraintsCount2
After_lbConstraintsCount
PASS_ON
PASS_ON

```
txtConstraintName_TextChanged
entry/
bModified = true;
btnApply.Enabled = true;
btnUndo.Enabled = true;

btnApply_Click
entry/
applyChanges((string)lbCon
straints.SelectedItem, false);
btnOK.Enabled = true;

lbConstraints_SelectedIndexChanged
entry/
if (!bHandleChanges) return;
bHandleChanges = false;
if (bModified) displayMessageBox();
btnApply.Enabled = false;
btnUndo.Enabled = false;
PASS_ON

lbConstraintsCount1
entry/
btnRemove.Enabled = true;
grpActualConstraint.Enabled =
true;
showSelectedConstraint(lbCo
nstraints.SelectedItem);
PASS_ON

lbConstraintsCount2
entry/
btnRemove.Enabled = false;
txtConstraintName.Text = "";
txtActual.Lines = new string[0];
grpActualConstraint.Enabled =
false;
PASS_ON

After_lbConstraintsCount
entry/
bHandleChanges = true;
```

Fig. 6: Statechart model of the Constraint Editor for Pattern Rule Node form

The case study uses the statechart model (Fig. 6) as input model and applies a rewriting rule (Fig. 7) for it. In the rewriting rule (Fig. 7) the metamodel of LHS is the Statechart metamodel [OMG UML] [VMTS] and the metamodel of RHS is the CodeDOM metamodel [.NET] [VMTS]. On LHS of the rewriting rule there are two states whose meta type is statechart state, and there is a transition between them with a 0..* multiplicity on the side of the target state. It means that exhaustively applying this rewriting rule for a statechart model, it will match all states with their target adjacent states. The rule has to match the accessible adjacent states, because we need them to generate the state-transitions into the source code. Of course it is possible that a state has no outgoing transition, and the reason why we enable the 0 in the multiplicity is that we want to match states having only incoming transitions to generate CodeDOM tree for them. On RHS of the rewriting rule the *CTypeDeclaration* represents a type declaration for a class, structure, interface or enumeration. *CMemberField* can be used to denote the declaration for a field of a type, and *CMemberMethod* to phrase the declaration for a method. *CParameter* represents a parameter declaration for a method, property, or constructor, and *CSnippetStatement* means a statement using a literal code fragment. The code generation is a syntax tree generation (CodeDOM tree) from which the framework generates the C# source code.

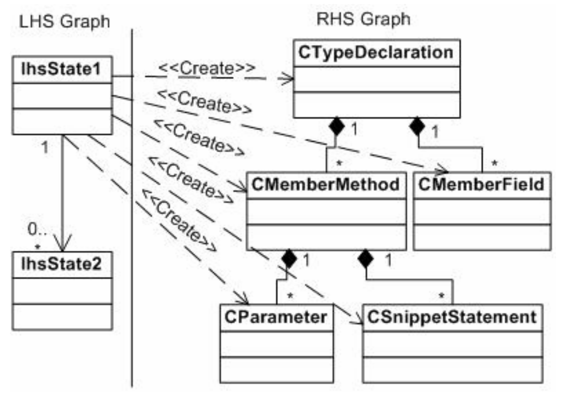


Fig. 7: Rewriting rule of the case study

In a rewriting rule we can connect LHS elements to RHS elements, this relation between LHS and RHS elements is called internal causality [Karsai03], which facilitates to assign an operation to this connection. Causalities can express modification or removal of an LHS element, and creation of an RHS element. In Fig. 7 the causalities are denoted as dashed lines. The create operation and attribute transformation, which is one of the most important part of the rewriting process, are accomplished by XSLT scripts. The XSLT scripts can access the attributes of the object matched to LHS elements, and produce a set of attributes for RHS element which the causality point to. VMTS stores models as labeled graphs, every node and edge has a property XML, which contains the attributes of the model element. In the case of current case study the VMTS rewriting engine concatenates the property XML of the matched states and transitions and uses the result as the input of the XSLT script.

Constraint Validation

To fully specify models and rewriting rules we assign constraints to model elements and to the steps accomplished by generators. With the help of these constraints we get precise and consistent models and transformation steps. In VMTS the principle of the constraint validation is the relation between the pre- and postconditions and the OCL constraints assigned to the rewriting rules.

In .NET when we initialize the controls e.g. change the *Text* value of a text box then it is passed a *TextChanged* event, or set the *SelectedIndex* property of a combo box then it gets a *SelectedIndexChanged* event. This behavior of the controls affects the operation of the form in an inappropriate way. An example for it in the case study is when the user selects an item in list box *Constraints*, the form has to show the properties of the selected constraint, hence it has to change the *Text* value of text box *Constraint name*, the

SelectedIndex value of the Stereotype combo box and so on. The result of these operations is that buttons *Apply* and *Undo* become enabled, but during the initialization we do not want it, because it is not a real property modification. We can eliminate this undesirable functioning with a constraint:

```
context CMemberMethod inv handle_changes:
if self.Type = ‘EventHandler’ then self.Statements.Count > 0
and self.Statements[0].Value = ‘if (!m_bHandleChanges)
return;‘
```

This invariant constraint describes that if the type of an *CMemberMethod* object is *EventHandler*, it should have more than zero *Statement*, and the value of the first statement should be ‘*if (!m_bHandleChanges) return;*’. A snippet statement is a code fragment, and this statement guarantees that event handler functions do not handle the events in that case if the value of *m_bHandleChanges* variable is false.

The C# source code of the *lbConstraints_SelectedIndexChanged* event handler method which is generated based on the statechart diagram (Fig. 6) is as follows:

```
private void lbConstraints_SelectedIndexChanged(object sender,
System.EventArgs e)
{
  if (!bHandleChanges) return;
  bHandleChanges = false;
  if (bModified) displayMessageBox();
  btnApply.Enabled = false;
  btnUndo.Enabled = false;
  if (lbConstraints.SelectedItems.Count > 0)
  {
    btnRemove.Enabled = true;
    grpActualConstraint.Enabled = true;
    showSelectedConstraint(lbConstraints.SelectedItem);
  }
  if (lbConstraints.SelectedItems.Count == 0)
  {
    btnRemove.Enabled = false;
    txtConstraintName.Text = "";
    txtActual.Lines = new string[0];
    grpActualConstraint.Enabled = false;
  }
  bHandleChanges = true;
}
```

In VMTS it is required that all constraint has a name, therefore if one would like to apply the changes, we have to validate that the length of the constraint name is more than 0. Therefore, if the value of the *txtConstraintName.Text.Length* property is 0, we have to prevent the save of the modifications until the user does not fill in the Constraint name text box. The constraint which describes this condition is the following:

```
context CMemberMethod inv name_length:
```

```
if self.Name = 'btnApply_Click' then self.Statements.Count > 1
and self.Statements[1].Value = 'if
(txtConstraintName.Text.Length == 0) return;'
```

The generated C# source code of the *btnApply_Click* method is as follows:

```
private void btnApply_Click(object sender, System.EventArgs e)
{
  if (!bHandleChanges) return;
  if (txtConstraintName.Text.Length == 0) return;
  applyChanges((string)lbConstraints.SelectedItem, false);
  btnOK.Enabled = true;
}
```

When we generate source code from a statechart model, in the generated source code usually there is a function for every state, which implements the behavior of the state (transitions and internal transitions as well). In the form-based, event-driven development the event handler methods of the controls provides the operation of the forms. Therefore the goal of the case study is to generate the skeleton of the user interface handler code; VMTS generates that part of the event handler methods for which it has enough information in the statechart diagram. E.g. based on the incoming and outgoing transitions and their conditions the generator can produce a complete event handler function from several model states. An example for it in the case study is the *lbConstraints_SelectedIndexChanged* event handler method which is generated from four states and it's *if* branches are generated from the transition conditions. Furthermore the transformation generates the code fragments recommended by the constraints; a part of this code can be assertion code. An assertion checks a condition and displays a message if the condition is false, assertions support the testing procedure and contribute to the correct operation.

Based on the presented principles, the whole process of the case study is the following: OCL Compiler generates the constraints validation binary, the matching process searches topological matches in the statechart model (host graph), the Validation Module uses the validation binary, and it checks LHS graph containing constraints (preconditions) continuously at matching time. If and only if a match satisfies the preconditions, the rewriting process with the help of a user defined XSLT script generates the rewriting result. The Validation Module checks RHS graph containing constraints (postconditions) on the rewriting result. If and only if the rewriting result satisfies the postconditions, the rewriting rule finished successfully.

5 CONCLUSIONS AND FUTURE WORK

Our metamodel-based specification of the rules allows assigning OCL constraints to the rules, and they are able to express local constraints. However, it does not mean, that validating them does not involve checking other model elements in the input model. OCL constraints which are enlisted in the rewriting rules affect the instances of these rewriting rules, to the matched and the replaced subgraphs.

We have shown how we can use the OCL constraints enlisted in the rewriting rules to check validation, preservation and guarantee properties, or simply how to check models with the help of metamodel-based graph rewriting during the rewriting process.

The main limitation of the constraint validation method is the local nature of the rules. If one wants to specify a constraint for an element, it must be included in a rewriting rule, or it must be referenced by the OCL traversal expressions assigned to the rule elements.

One of the most important parts of the constraint checking method is that our approach does not interpret the constraints; we generate source code and compile it to a binary which validates the constraints contained by the metamodel and the rewriting rule. This method facilitated to calculate precisely the complexity of the presented constraint validation methods. Comparing the algorithmic results to other approaches, PROGRES has language tools to support pre- and postconditions, but unfortunately its computational complexity has not been published.

In this paper algorithms are provided to validate the graph-rewriting-based model transformation with the help of the constraints contained by the rewriting rules. Invariant Analysis algorithm is applicable for both LHS and RHS graph containing constraints, while Persistent Analysis algorithm affects only the constraints in LHS graph (preconditions of the transformation step). This work has discussed the computational complexities of the presented algorithms and the computational time that we can save during the model transformation using these algorithms.

Future work includes the design and implementation of branch conditions in rule sequencing. With the help of branch conditions VMTS will support the branch during the transformation using the constraints contained by RHS graph. The result of an RHS graph containing constraint checking decides which transformation step is the following. Furthermore the aspect-oriented constraint specifications in rewriting rules are currently researched. This method will facilitate to create the rewriting rules and the constraints separately, and using pointcuts to assign constraints to the rule nodes in the rewriting rules. As a result the rewriting rules will not contain numerous constraints which make them more tangled, constraints will be reusable several times and their modification will be simpler.

ACKNOWLEDGEMENTS

The activities described in this paper supported, in part, by Information Technology Innovation and Knowledge Centre.

REFERENCES

[Akehurst03] D. Akehurst, O. Patrascoiu: "OCL 2.0 - Implementing the Standard for Multiple Metamodels", *Workshop Proceedings, 6th International Conference on the Unified Modeling Language and its Applications, <<UML>>2003*, Electronic Notes in Theoretical Computer Science (ENTCS), October 2003.

[Blostein95] D. Blostein, H. Fahmy, A. Grbavec: "Practical Use of Graph Rewriting", Technical Report No. 95-373, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, January, 1995.

[Czarnecki00] K. Czarnecki et al.: "Generative programming: methods, tools, and applications", Addison-Wesley, 2000.

[Ehrig97] H. Ehrig (ed.): "Handbook on Graph Grammars and Computing by Graph Transformation: Foundations", Vol. 2. World Scientific, Singapore, 1997.

[Hamie98] A. Hamie, J. Howse, S. Kent: "Interpreting the Object Constraint Language", *Proceedings 5th Asia Pacific Software Engineering Conference (APSEC '98)*, December 2-4, 1998, Taipei, Taiwan, 1998.

[Karsai03] G. Karsai, A. Agrawal, F. Shi, J. Sprinkle: "On the Use of Graph Transformations for the Formal Specification of Model Interpreters", *Journal of Universal Computer Science*, Special issue on Formal Specification of CBS, 2003.

[Lengyel05a] L. Lengyel, T. Levendovszky, P. Kozma, H. Charaf: "Compiling and validating OCL constraints in metamodeling environments and visual model compilers", *IASTED on SE*, Feb. 15-17, 2005, Innsbruck, Austria, pp. 48-54.

[Lengyel05b] L. Lengyel, T. Levendovszky, H. Charaf: "Implementing an OCL Compiler for .NET", In Proceedings of the 3rd International Conference on .NET Technologies, Pilsen, Czech Republic, May-June 2005, pp. 121-130.

[Levendovszky04a] T. Levendovszky, L. Lengyel, G. Mezei, H. Charaf: "A Systematic Approach to Metamodeling Environments and Model Transformation Systems in VMTS", *International Workshop on Graph-Based Tools (GraBaTs)* Electronic Notes in Theoretical Computer Science, Rome, 2004

[Levendovszky04b] T. Levendovszky, L. Lengyel, H. Charaf: “Implementing a Metamodel-Based Model Transformation System”, Buletinul Stiintific al Universitatii “Politehnica” din Timisoara, Romania Seria Automatica si Calculatoare Periodica Politechnica, Transactions on Automatic Control and Computer Science Vol.49 (63), 2004, ISSN 1224-600X.

[Levendovszky04c] T. Levendovszky, L. Lengyel, H. Charaf: “Software Composition with a Multipurpose Modeling and Model Transformation Framework”, *IASTED 2004*, Innsbruck, 2004, pp.590-594

[Levendovszky05] T. Levendovszky, H. Charaf: “Pattern Matching in Metamodel-Based Model Transformation Systems”, *submitted to Periodica Polytechnica Electrical Engineering*, ISSN 0324-6000

[Loecher03] S. Loecher, S. Ocke: “A Metamodel-Based OCL-Compiler for UML and MOF. In OCL 2.0 - Industry standard or scientific playground”, *Workshop Proceedings, 6th International Conference on the Unified Modeling Language and its Applications, <<UML>>2003*, ENTCS, October 2003

[Meyer88] B. Meyer: “Object-Oriented Software Construction”, Prentice Hall, New York, 1988

[.NET] Microsoft .NET Framework http://msdn.microsoft.com/netframework/

[OMG MDA] MDA Guide Version 1.0.1, OMG, doc. number: omg/2003-06-01, June 2003 www.omg.org/docs/omg/03-06-01.pdf

[OMG OCL] Object Constraint Language Specification (OCL), www.omg.org

[OMG UML] UML 2.0 Specifications, http://www.omg.org/uml/

[PROGRES] PROGRES system can be downloaded from http://www-i3.informatik.rwth-aachen.de/research/projects/progres/main.html

[Rozenberg97] G. Rozenberg (ed.): “Handbook on Graph Grammars and Computing by Graph Transformation: Foundations”, Vol.1 World Scientific, Singapore, 1997

[VMTS] VMTS Web Site, http://avalon.aut.bme.hu/~tihamer/research/vmts/

About the authors

László Lengyel is Ph.D student at Department of Automation and Applied Informatics of the Technical University of Budapest. His research field is Metamodeling, Graph rewriting-based software model transformation, Model-driven development, Constraint validation, Validated model transformation and Aspect-oriented techniques. E-Mail: lengyel@aut.bme.hu

Tihamér Levendovszky received his Ph.D degree in 2006. He is Professor Assistant at Department of Automation and Applied Informatics of the Technical University of Budapest. His research field is Metamodeling, Software modeling and design, Software model transformation, Generative and extreme programming. E-Mail: tihamer@aut.bme.hu

Hassan Charaf received his Ph.D degree in 1998. He is Associate Professor at Department of Automation and Applied Informatics of the Technical University of Budapest. His research field is Software engineering, Distributed systems, Identification and control of nonlinear systems using neural networks. E-Mail: hassan@aut.bme.hu