Execution Properties of a Visual Control Flow Language

László Lengyel, Tihamér Levendovszky, Gergely Mezei, Hassan Charaf

Budapest University of Technology and Economics
Goldmann György tér 3, H-1111 Budapest, Hungary
lengyel@aut.bme.hu, tihamer@aut.bme.hu, hassan@aut.bme.hu

Abstract: Graph rewriting-based model processing is a widely used technique for model transformation. Especially visual model transformations can be expressed by graph transformations, since graphs are well-suited to describe the underlying structures of graphical models. Model transformations often need to follow an algorithm that requires a strict control over the execution sequence of the transformation steps. Therefore, termination criteria for graph and model transformation systems have become a focused area. This work introduces a metamodel-based visual control flow language and discusses several consideration related to the termination properties of the presented language. Furthermore, it provides algorithms to compose metamodel-based model transformation steps in order to support the termination analysis of graph rewriting-based visual model processing.

Keywords: Control Flow, Metamodel-Based Model Transformation, OCL, Termination Properties, UML

1 Introduction

Visual Modeling and Transformation System (VMTS) [1] [2] is an n-layer metamodeling environment which supports editing models according to their metamodels, and allows specifying OCL constraints. Models are formalized as directed, labeled graphs. VMTS uses a simplified class diagram for its root metamodel ("visual vocabulary").

Also, VMTS is an UML-based [3] model transformation system, which transforms models using graph rewriting techniques. Moreover, the tool facilitates the verification of the constraints specified in the transformation step during the model transformation process.

Graph rewriting [4] is a powerful technique for graph transformation with a formal background. The atoms of the graph transformation are rewriting rules, each

rewriting rule consists of a left-hand side graph (LHS) and a 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 applied (host graph), and replacing this subgraph with RHS.

Model transformation means converting an input model available at the beginning of the transformation process to an output model. Several widely used approaches to model transformation use graph rewriting as the underlying transformation technique. Previous work [1] has introduced an approach – metamodel-based rewriting rules –, where the LHS and RHS of the transformation steps are built from metamodel elements. This means that an instantiation of LHS must be found in the host graph instead of the subgraph isomorphic to LHS. This metamodel-based approach facilitates to assign OCL constraints to pattern rule nodes (PRNs) – nodes of the rules.

The Object Constraint Language (OCL) [5] is a formal language for the analysis and design of software systems. It is a subset of the UML standard [3] that allows software developers to write constraints and queries over object models.

A *precondition* assigned to a transformation step is a Boolean expression that must hold at the moment when the step is fired, and a *postcondition* assigned to a step is a Boolean expression that must hold after the completion of a transformation step. If a *precondition* of a step is not true, then the step fails without being fired. If a *postcondition* of a transformation step is not true after the execution of the step, then the transformation step fails. A direct corollary of this is that an OCL expression in LHS is a precondition to the transformation step, and an OCL expression in RHS is a postcondition to the transformation step.

In general, the termination of graph rewriting is undecidable [6], but under certain conditions it is possible to prove termination for numerous model transformations. This paper gives results related to the examination of such conditions. Therefore, the motivation of the work presented in the current paper is to support making the decision related to termination of metamodel-based visual control flow languages. Section 2 introduces the VMTS Visual Control Flow Language (VCFL), which termination properties and transformation step composition possibilities are discussed in Section 3.

2 VMTS Visual Control Flow Language

One of the most important capabilities of a control flow language is the possibility to express a transformation as an ordered sequence of the transformation steps. Classical graph grammars apply any production that is feasible. This technique is appropriate for generating and matching languages but model-to-model transformations often need to follow an algorithm that requires a more strict

control over the execution sequence of the steps, with the additional benefit of making the implementation more efficient.

The VMTS approach is a visual approach, and it also uses graphical notation for control flow: stereotyped UML activity diagrams, which is a technique to describe procedural logic. UML activity diagrams are intended to describe business processes, and workflows.

In VMTS transformation steps, the *internal causality* is a relation between LHS and RHS elements, it makes possible to connect an LHS element to an RHS element and to assign an operation to this connection. An internal causality describes what we to do during applying a transformation step (element creation, element deletion, attribute modification). The *create* and *modify* operations are accomplished by XSL scripts. The XSL scripts can access the attributes of the objects matched to LHS elements, and they produce a set of attributes for RHS element to which the causality point.

As it is presented through examples in Section 3, the expressiveness of the multiplicities in the metamodel-based model transformation steps is high. In VMTS both sides of the transformation steps use formalism similar to that of the UML class diagram. The same instantiation rules apply to the LHS as to the UML class diagram, with two exceptions. (i) An association with the multiplicity * matches all the edges of the appropriate type in a given position. (ii) A type can appear more than once in the rules. These types are processed by the matching algorithm as if they were of different types. However, care is taken that the same model element cannot be matched to two different LHS objects.

The interpretation of the undefined * multiplicity is not evident on the right-hand side. In VMTS, this value is determined by the result of the attribute transformation. The advantage of this approach is the flexibility that an attribute can influence the structure. Therefore, a simple transformation step can have an effect on an arbitrary part of the input model.

Sequencing transformation steps results in a transformation which contains the steps in an ordered sequence ($S_{0,}$ $S_{1...}$ S_{n-1}). Assume the case that the input model of the step i (S_i) is the model M_i and the result of the S_i is the model M_{i+1} (where $0 \leq i \leq n-1$). In this case the input model of the step $i+1$ (S_{i+1}) is the model M_{i+1}. This means that during the execution of the step sequence, each step works on the result of the previous step.

The interface of the transformation steps allows the output of one step to be the input of another step, in a dataflow-like manner. This is used to sequence expression execution. In VCFL, this construct is referred to as an *external causality*. An external causality creates a linkage between a node contained by RHS of the step i and a node contained by LHS of the step $i+1$. This feature accelerates the matching and reduces the complexity, because the step i provides partial match to the step $i+1$.

Branching with OCL Constraints. Often, the transformation that we would like to apply depends on a condition. Therefore, a branching construct is required. In VCFL, OCL constraints assigned to the decision elements can choose between the paths of optional numbers, based on the properties of the actual input model and the success of the last transformation step (*SystemLastRuleSucceed*).

Hierarchical Steps. The VCFL supports hierarchical specification of the transformation steps. High-level steps can be created by composing a sequence of primitive steps and can be viewed as separate transformation modules. A high-level step can contain several simple steps, hiding the details which could be unimportant on a specific abstraction level and represents the contained steps as coherent units.

Iteration (Tail Recursion) and Parallel Executions of the Steps. The iteration is achieved with the help of the decision objects and the OCL constraints contained by them. A decision object evaluates the assigned constraints, and based on the results selects a flow edge which could be a follow-up or a backward edge as well. Recursion can be solved with the composition of the iteration and external causalities. A high-level step can call itself, where external causalities represent the actual parameters of the recursive call. The parallel execution of the independent transformation steps is supported by the *Fork* and *Join* elements.

In the VCFL, a transformation step has two specific attributes: *Exhaustive* and *MultipleMatch* [7]. Recall that applying a model transformation step means finding a match of LHS in the input model and replacing this subgraph with RHS. An *exhaustive* transformation step is executed continuously as long as LHS of the step can be matched to the input model. The *MultipleMatch* attribute of a step allows that the matching process finds not only one but all occurrence of LHS in the input model, and replacing is executed on all the found places.

In VCFL, if a transformation step fails and the next element in the control flow is a decision object, it provides the next branch based on the OCL statements and the value of the *SystemLastRuleSucceed* variable. If no decisions can be found, the control is transferred to the parent state, and if there is no parent state, the transformation terminates with error [8].

3 Termination Aspects

The termination properties of a transformation are really important for model transformation. We want to investigate that under which conditions an arbitrary VCFL transformation can satisfy termination criteria.

Definition. A *VCFL Transformation* is a stereotyped UML activity diagram. A *VCFL Transformation T* defines a strict order of the contained transformation

steps $S_0, S_1 ... S_{n-1} \in STEPS \in T$, where S_0 is the start step of *T*. Transformation *T* contains OCL constraints assigned to decision objects to choose between different control flow branches, and external causalities between transformation steps to support parameter passing.

Definition. A VCFL transformation *T* for a finite input model G_0 *terminates*, if there is no infinite derivation sequence from G_0 via transformation steps $STEPS \in T$, where starting from S_0 (start step of the *T*) steps $STEPS$ are applied as it is defined by the transformation *T*.

Termination of transformations is not always guaranteed. If a control flow model contains an exhaustive step that can be applied infinitely to the result models, such that the transformation does not terminate.

All derivation sequences over transformation steps $STEPS \in T$ are terminating if each transformation step $S \in STEPS$ terminate. Since the non-exhaustive termination steps terminate, we can state that a VCFL transformation *T* terminates if all exhaustive transformation step $S_E \in STEPS$ and loop $L \in T$ terminate.

In order to examine the termination properties of VCFL exhaustive transformation steps and VCFL loops, we need a mechanism to compose the transformation steps. This is a way to examine all the possible transformation execution without the actual input models. The composed step can equivalently replace the original steps, because it produces the same result and imposes the same input conditions as the sequence composed of individual rules [9]. Moreover, to examine the termination properties of the exhaustive transformation steps we also use the composition algorithm. Recursively composing an exhaustively applied transformation step S_E with itself *n* times results a transformation step S_C we can use once for an input model G_0 instead of applying the original step S_E *n* times.

3.1 Composing Metamodel-Based Model Transformation Steps

The VMTS approach uses metamodel-based transformation steps. This means that steps are built from metamodel elements, and the LHS and RHS of the steps are the metamodels of the matched and produced model parts.

The composition algorithm takes not only the structure of the steps into consideration but also the external- and internal causalities, the metatypes of the PRNs and edges and the constraints contained by PRNs.

The external causalities defined between the steps simplify the complexity of the step composition. They exactly define the mapping between the RHS elements of the step *i* and the LHS elements of the step *i+1*. Internal causalities connect the LHS and RHS elements within a transformation step. Therefore, taking both the

internal- and external causalities into account, we can follow the node mapping between the transformation steps and also within them. The metatypes of the PRNs and edges, compared to the general case, also simplify the computation complexity of the algorithm. The metatypes narrows the search space. The constraints propagated to the PRNs must be checked whether there is any contradiction between the constraints contained by the PRNs mapped to each other during the composition. Furthermore, multiplicities that are allowed to be specified in metamodel-based transformation steps, also need to be taken into account.

The algorithm works based on the double pushout (DPO) approach concurrency theorem [10]. The VMTS Transformation Steps Composing (VTSC) algorithm is as follows.

```
VTSComposing(TransformationStep[] TSs, bool exhaustive): TransformationStep[]
 1 compositionList = GetFirstStep(TSs)
 2 if exhaustive
 3 return VTSRecursiveComposing(GetFirstStep(TSs)
 4 else
 5   foreach Transformation Step stepNext  in TSs (except first step)
 6     foreach Transformation Step S in compositionList
 7       initialMapping = CreateMappingByExtCaus(S, stepNext)
 8       newCompositions = ExtendMappingByMetaTypes(
                                         initialMapping, S, stepNext)
 9       CalculateMultiplicities(newCompositions)
10       AddToList(newCompositionList, newCompositions)
11     end foreach
12     compositionList = newCompositionList
13   end foreach
14 end if
15 return compositionList
```

3.2 Self-Composing Metamodel-Based Model Transformation Steps with

The goal of this section is to provide considerations (algorithm) related to metamodel-based transformation step composition, namely how to compose transformation steps with themselves arbitrary times. This type of composition is required in order to examine the termination of the exhaustively applied transformation steps.

On the instance layer the rule composition is constructed in [9], also based on the double pushout (DPO) approach concurrency theorem [10].

Our aim is to work out an equivalent transformation step composition method on the meta-layer (on the layer of the metamodel-based transformation steps). The current section provides several examples and an algorithm for transformation step composition, furthermore, proofs that the result of the composed step created by the presented algorithm for an arbitrary input model on the instance level is the

same as the result of the original transformation steps. In order to state and prove the general case formally, we have to examine the composition properties of several typical transformation step structures. The examined structures are the *tree*, *inserting* and *deletion structure steps*. The following examples and considerations discuss the properties of these structures and the possibilities of their composition with themselves. This type of examination of the transformation step structures is chosen, because an optional transformation step is built from these structures. Therefore, if we show and prove the correctness of these structures one-by-one, then it will be easier to generalize the consideration for the composition of the general transformation step.

In the following example structure there is no external causality defined, the transformation step is executed exhaustively. Furthermore, the structures are so simple that there is only one possible composition between the RHS and LHS nodes.

Definition. The nodes generated and attached to the input model by a *tree structure transformation step* are connected only with one edge to the input model. A tree structure transformation step does not generate circles into the model.

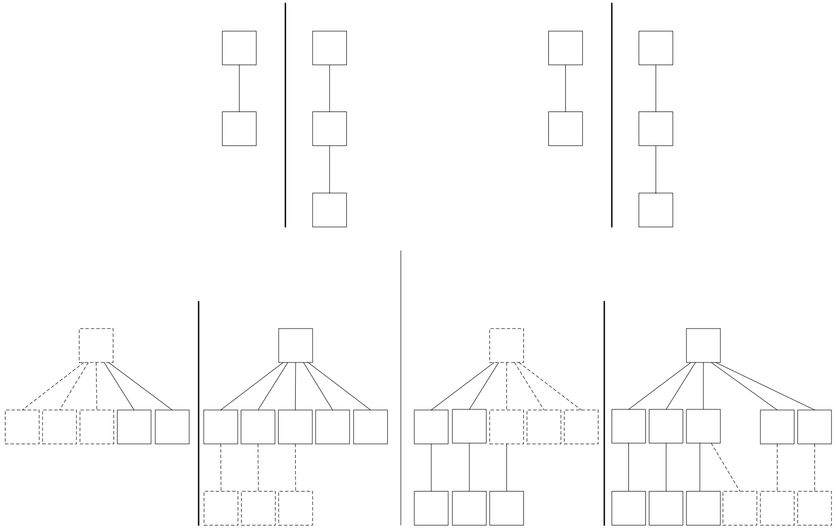

Figure 1

Composition of transformation steps with tree structure

An example transformation step with tree structure is depicted in Fig. 1a. The first application of the step on an input model is presented in Fig. 1b/1. It matches an *A*

(a) Tree Structure
Transformation Step

type node with three *B* type nodes and generates three *C* type nodes into the model. The second application of the transformation step (Fig. 1b/2) is executed on the result of the first execution. In Fig. 1b the dashed lines denote the matched part of the input model and the created part of the output model. In the second step, the same *A* and *B* type nodes can be matched again, but it is also possible that only a part of them with new ones or totally new nodes are found by the matcher algorithm. The algorithm the VMTS uses to compose metamodel-based model transformation steps with themselves is as follows.

```
VTSRecursiveComposing(TransformationStep originalStep, TransformationStep[]
inComposedSteps, int numOfComposition): TransformationStep[]
 1 if (numOfComposition == 0) return inComposedSteps
 2   foreach Transformation Step S in inComposedSteps
 3     initialMapping = CreateMappingByExtCaus(S, originalStep)
 4     newCompositions = ExtendMappingByMetaTypes(initialMapping, S, originalStep)
 5     CalculateMultiplicities(newCompositions)
 6     AddToList(newCompositionList, newCompositions)
 7   end foreach
 8 return VTSRecursiveComposing(originalStep, newCompositionList, numOfComposition-1)
```

The transformation step presented in Fig. 1a is composed with itself once and the result is depicted in Fig. 1c. In the current case the composed step differs from the original one only in the modified multiplicities. The multiplicities of the matched and the created nodes vary between well defined limits. Multiplicities must allow the cardinality values between *m* and $n*m$, where *m* denotes the cardinality of the original multiplicity, and *n* the number of the application of the step. The exact rules for calculating the multiplicities during transformation step composition are summarized in Table 1.

Table 1

Rules for calculating multiplicities during transformation step composition

Multiplicity in step S_1	**Multiplicity in step S_2**	**Multiplicity in composed step**
p	*r*	min(*p, r*)..*p+r*
p..s	*r*	min(*p, r*)..*s+r*
p1..s1	*p2..s2*	min(*p1, p2*)..*s1+s2*

Proposition. Let *S* be a tree structure transformation step and S^n be the transformation step resulted by the *n* times composition of the step *S* using VTSRecursiveComposing algorithm. Applying the transformation step *S n* times on an arbitrary finite input model G_0 is equivalent to a single execution of the step S^n.

Proof. First we examine the case when the transformation step *S* applied twice for the input model G_0. The composition means multiplicity modification. The match found during the second application of the step *S* can contain nodes that are also matched by the first execution of the step. Therefore, the generated nodes can be connected to the same nodes or also to different ones. The step S^2 expresses the same notion, because each multiplicities *r* and *p..s* from step *S* are replaced with

the multiplicities $r..2r$ and $p..2s$ in the step S^2. The maximum cardinality of the multiplicities is duplicated. In fact this means the following. The transformation step S is composed with itself. The lower bound of the allowed cardinality does not change, but the upper bound must be the sum of the allowed multiplicities in the original steps on the certain places. Consequently, applying the step S twice is equivalent to applying the step S^2 once. Assume the case that we have a transformation step S^k that is resulted by the k times composition of the step S, and also assume that applying the step S k times on the input model G_0 equivalent to a single application of the step S^k. We examine the next step, the composition of the transformation steps S^k and S. After finding the mapping between RHS nodes of the steps S^k and LHS nodes of the steps S, the multiplicities must be modified based on the rules presented in Table 1. Applying the rules for the edges between mapped nodes it results that the original multiplicities r and $p..s$ of step S in step S^{k+1} must be $r..(k+1)r$ and $p..(k+1)s$. This means that executing the step S $k+1$ times on the input model G_0 equivalent to a single application of the step S^{k+1}.

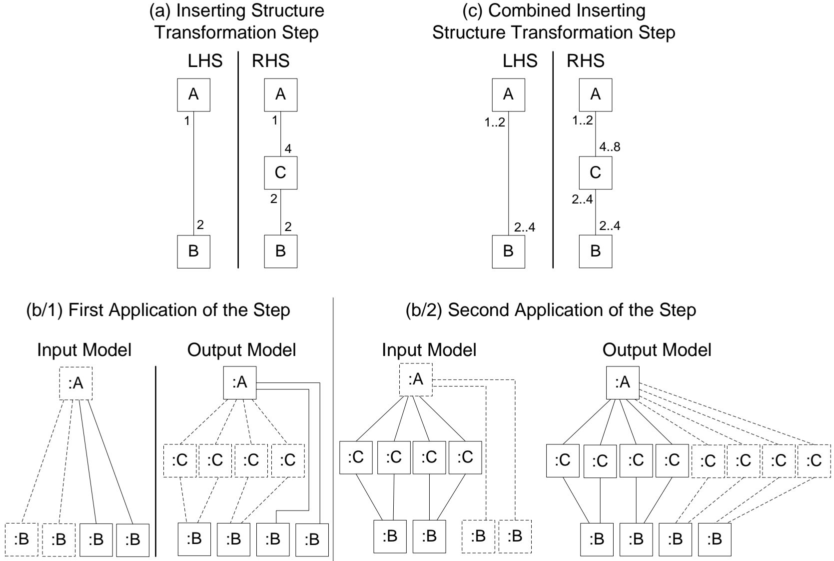

Figure 2

Composition of transformation steps with inserting structure

Definition. The nodes generated and attached to the input model by an *inserting structure transformation step* are connected at least with two generated edges to the input model. A inserting structure transformation step can generate circles into the model.

An example for the inserting structure transformation step is depicted in Fig. 2a. The first and second application on an input model of the step is presented in Fig

2b. The transformation step presented in Fig. 2a is composed with itself once, and the result is depicted in Fig. 2c.

We differentiate two cases. (i) The inserting structure transformation step deletes the edge between the matched nodes that are connected through the newly created node (Fig. 2a). In this case the match is destroyed and cannot be found again. (ii) In the second case the step does not delete the original edge that connects the nodes which are connected through the newly created node and edges (Fig. 3a).

Proposition. Let S be an inserting structure transformation step and S^n be the transformation step resulted by the n times composition of the step S using VTSRECURSIVECOMPOSING algorithm. Applying the transformation step S n times on an arbitrary finite input model G_0 is equivalent to a single execution of the step S^n.

Definition. A *deleting structure transformation step* deletes at least one node from the input model.

An example for a metamodel-based deleting structure transformation step is depicted in Fig. 3b/1, it is composed with itself, and the resulted step is presented in Fig. 3b/2.

Proposition. Let S be a deleting structure transformation step and S^n be the transformation step resulted by the n times composition of the step S using VTSRECURSIVECOMPOSING algorithm. Applying the transformation step S n times on an arbitrary finite input model G_0 is equivalent to a single execution of the step S^n.

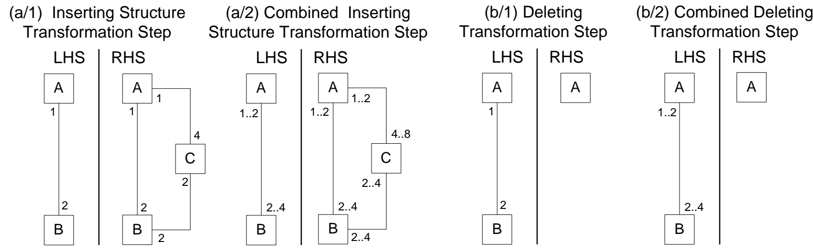

Figure 3

Composition of transformation steps with inserting and deleting structure

The following proposition states that there is no difference between applying a sequence of transformation steps or their composed step for an arbitrary input model. The two transformation result the same model.

Proposition. If the transformation steps $S_j, S_{j+1} \ldots S_k$ are applicable successfully for an input model G_0, then a transformation step S_C, created from

transformation steps $S_j, S_{j+1} \dots S_k$ using VTSC algorithm, has the same effect on the input model G_0 as the transformation steps $S_j, S_{j+1} \dots S_k$.

Proof. If the transformation steps $S_j, S_{j+1} \dots S_k$ are applicable successfully for an input model G_0, then it means that the step S_j can be executed on the model G_0 and produces the model G_j, the step S_{j+1} can be executed on the model G_j and produces the model G_{j+1}, and so on. Each step can be applied on the result of the previous step, and finally, the steps $S_j, S_{j+1} \dots S_k$ produce the model G_{j+k}. The step S_C produces the same result on the input model G_0 as the transformation steps $S_j, S_{j+1} \dots S_k$, because step S_C is created considering the causalities between the steps and all the modifications committed by steps $S_j, S_{j+1} \dots S_k$. Therefore, step S_C executes all the modifications of steps $S_j, S_{j+1} \dots S_k$ in a single step.

Conclusions

Using the presented approach, we can predict the behavior of a transformation. Not only its termination, as it is presented in this paper, but also other properties. For example, the method can be used to examine whether a transformation validate, guarantee or preserve an optional property of the input model, e.g. an attribute value of a input model node or a relation between model nodes.

The discussed method is independent from the input models, it makes its decision based on the transformation steps only, and the control flow model, therefore the result of a transformation examination holds for all input models.

The presented approach is an offline method; the termination in many cases depends not only on the VCFL transformation model and transformation steps but also on the actual input model. A simple constraint can be itself a significant difference between two steps or an attribute value between two models. The problem is not trivial. There are certain cases when the method can make a decision based on the VCFL transformation, and there are other cases when not.

In the cases when the algorithm cannot make decision we can use the design-by-contract-based VMTS approach [7]. We define transformation steps and specify them properly with constraints (preconditions and postconditions). If the transformation is executed successfully, then the generated output model is in accordance with the expected result described by the steps of the transformation refined with the constraints. Thus, it produces a valid result.

Termination is an important issue for model transformations. Since model transformations can become very complex, we consider not only the application of single transformation steps, but also transformations where step applications are restricted according to a strict control flow.

Acknowledgement

The fund of “Mobile Innovation Centre” has supported in part, the activities described in this paper.

References

[1] T. Levendovszky, L. Lengyel, G. Mezei, H. Charaf, A Systematic Approach to Metamodeling Environments and Model Transformation Systems in VMTS, ENTCS, International Workshop on Graph-Based Tools, GraBaTs, Rome, 2004

[2] The VMTS Homepage, http://avalon.aut.bme.hu/~tihamer/research/vmt

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

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

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

[6] D. Plump. Termination of graph rewriting is undecidable. Fundamenta Informaticae, 33(2):201–209, 1998

[7] L. Lengyel, T. Levendovszky, G. Mezei, B. Forstner, H. Charaf, Metamodel-Based Model Transformation with Aspect-Oriented Constraints, International Workshop on Graph and Model Transformation, GraMoT, Tallinn, Estonia, September 28, 2005

[8] L. Lengyel, T. Levendovszky, H. Charaf, A Visual Control Flow Language and Its Termination Properties, International Conference on Information Technology, ICIT 2005, Transactions on Enformatika Volume 8, ISBN 975-98458-7-3, Budapest, Hungary, October 26-28, 2005, pp. 163-168

[9] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer, Fundamentals of Algebraic Graph Transformation, EATCS Monographs in Theoretical Computer Science, Springer, 2005 to appear

[10] H. Ehrig, Introduction to the Algebraic Theory of Graph Grammars, In:Graph Grammars and Their Applications to Computer Science and Biology, Springer, Ed. V. Claus, H. Ehrig, G. Rozemberg, Berlin, 1979