World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:2, No:2, 2008

# Model Transformation with a Visual Control Flow Language

László Lengyel, Tihamér Levendovszky, Gergely Mezei and Hassan Charaf



***Abstract*—**Graph rewriting-based visual model processing is a widely used technique for model transformation. Visual model transformations often need to follow an algorithm that requires a strict control over the execution sequence of the transformation steps. Therefore, in Visual Model Processors (VMPs) the execution order of the transformation steps is crucial. This paper presents the visual control flow support of Visual Modeling and Transformation System (VMTS), which facilitates composing complex model transformations of simple transformation steps and executing them. The VMTS Visual Control Flow Language (VCFL) uses stereotyped activity diagrams to specify control flow structures and OCL constraints to choose between different control flow branches. This paper introduces VCFL, discusses its termination properties and provides an algorithm to support the termination analysis of VCFL transformations.

***Keywords*—**Control Flow, Metamodel-Based Visual Model Transformation, OCL, Termination Properties, UML.

## I. 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 and transformation steps 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. Replacing means removing the elements that are in LHS but not in RHS, and gluing the elements that are in RHS but not in LHS.

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 uses graph rewriting as the underlying transformation technique. Previous work [1] has introduced an approach – metamodel-based rewriting rules –, where the left-hand side (LHS) and right-hand side (RHS) graphs 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 rewriting 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.

The motivation of the work presented in this paper is to support the control flow in visual model transformation systems and to define the conditions exactly which guarantee that if a transformation fulfills them it terminates or not. An algorithm – VCFL Termination Algorithm (VTA) – is developed to support the termination analysis of VCFL transformations. The VTA is an offline algorithm, as an input it uses only the control flow model to make the decision. This means that the decision is independent from any host model.

## II. The 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 Activity Diagram, which is a technique to describe procedural logic, business process, and work flow. In many ways, it plays a role similar to flowcharts, but the principal difference between it

Manuscript received November 9, 2005. The fund of "Mobile Innovation Centre" has supported, in part, the activities described in this paper.

L. Lengyel, T. Levendovszky, G. Mezei and H. Charaf, are with Department of Automation and Applied Informatics in Budapest University of Technology and Economics, Goldmann Gyorgy ter 3, 1111 Budapest, Hungary; e-mails: {lengyel, tihamer, gmezei, hassan}@aut.bme.hu).

and flowchart notation is that activity diagrams support parallel behavior [6].

In Fig. 1 the control flow model of the breadth-first search (BFS) algorithm is depicted which is a tree search algorithm used for traversing or searching a tree, tree structure, or graph. Intuitively, one starts at the root (start) node and explore all the neighboring nodes. Then for each of those nearest nodes, explore their unexplored neighbor nodes, and so on until it visits all nodes or finds the goal. In the current case our aim is to visit all the nodes and sign them, this means that there is no searched node.

The pseudo code of the algorithm is as follows.

```
BREADTHFIRSTSEARCH (Graph G, Node startNode)
 1  SETVISITED (startNode)
 2  ENQUEUE (queue, startNode)
 3  while (queue in not empty)
 4     node = DEQUEUE (queue)
 5     foreach neighbor in GETNEIGHBORS (node)
 6        if NOTVISITED (neighbor)
 7           SETVISITED (neighbor)
 8           ENQUEUE (queue, neighbor)
 9        end if
10     end foreach
11  end while
```

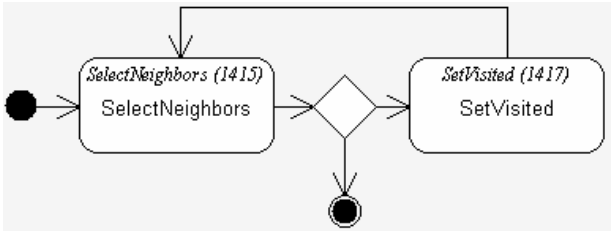


Fig. 1 The VCFL control flow model of the breadth-first search algorithm

In Fig. 2 the metamodel of the VMTS control flow is depicted, which describes that the root element is the *Transformation*. A *Transformation* can contain optional number of *FlowEdgeTarget* type object, this is denoted by stereotype <<*SystemContainment*>>. The *FlowEdgeTarget* is an abstract type which could be *Transaction*, *StartRule*, *Rule, HistoryRule*, *EndRule, FlowFinal, Decision, Merge, Fork* or *Join*. *FlowEdgeTargets* can be connected to each other using directed edges (*FlowEdge*). Types *Transaction* and *Rule* can contain another *FlowEdgeTargets*.

Moreover, the type *Rule* can contain *RuleNodes*. This is presented in the metamodel of the VMTS Rule Editor (Fig. 3). *RuleNode* is also an abstract type that can be *LHSNode* or *RHSNode*. A type *RuleNode* can contain or can be connected to another *RuleNodes*.

In the case study an arbitrary vertex from *G* to start the tree from is given as a pivot node (*startNode*). A pivot node is an input parameter of the control flow specified by the user. In the graph each vertex has a property (*IsVisited*) which determines if a vertex has already been visited by the algorithm. The transformation steps of the BFS (*SelectNeighbors* and *SetVisited*) are presented in Fig. 4.

The *Internal Causality* is a relation between LHS and RHS elements (Fig. 3), it makes possible to connect an LHS element to an RHS element and to assign an operation to this connection. In Fig. 4 internal causalities are denoted as dashed lines. An internal causality describes what we have to do during applying a transformation step (element creation, element deletion, attribute modification). The *create* and the *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.

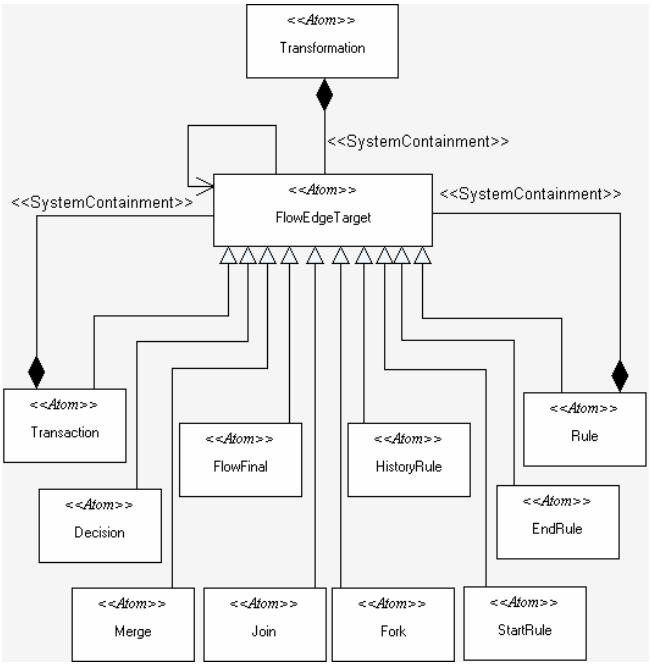


Fig. 2 The metamodel of the VMTS Visual Control Flow Language

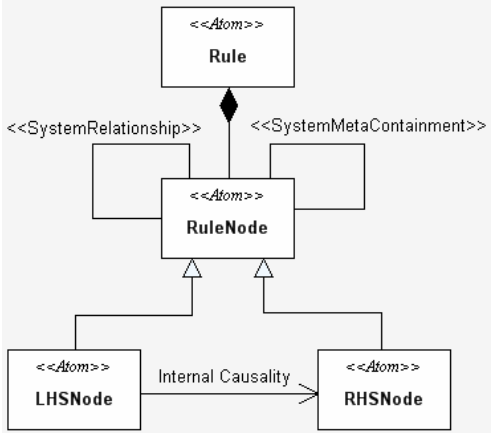


Fig. 3 The metamodel of the VMTS Rule Editor

The first transformation step selects the neighbors of a tree vertex which has at least one not visited neighbor (see 1..* multiplicity in LHS of the step). The not visited property is

validated by the constraint *Cosnt_Vis* propagated to LHS vertex *NeghborNode*.

```
context Node inv Const_Vis:
not IsVisited
```

While the graph contains at least one vertex which is not visited, then the step *SelectNeighbors* matches it and the step finishes successfully. The decision object based on the success of the first step selects the path to step *SetVisited*, otherwise to the rule end.

The second step is passed the selected vertices as external causalities and using an XSL script modifies their *IsVisited* property to true (other, more technical, transformation steps can also be defined [2] [7]). As result of the transformation all the graph nodes will be visited. The presented transformation does not modify the topology of the model but updates the attribute values.

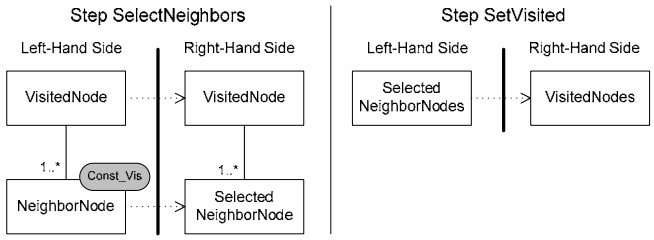


Fig. 4 Transformation steps *SelectNeighbors* and *SetVisited*

VCFL is a visual language for controlled graph rewriting and transformation, which supports the following constructs: sequencing transformation steps, branching with OCL constraints, hierarchical steps, parallel executions of the steps and iteration.

### A. Sequencing Transformation Steps

Sequencing transformation steps results a transformation which contains the steps in an ordered sequence ($S_0, S_1 \ldots S_{n-1}$). 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 $M_{i+1}$ (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. Obviously, except for the first step, which works on the input model. The result of the whole transformation is the result of the last step ($S_n$).

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 construction is referred to as 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 $i+1$. In our example we use external causalities to pass the selected vertices from step *SelectNeighbors* to step *SetVisited*.

### B. Branching with OCL Constraints

Often, the transformation 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 host model and the success of the last transformation step (*SystemLastRuleSucceed*). If the last transformation step fails, then VCFL could use the values of the system variables *SystemLHSFailure* and *SystemRHSFailure* for the decision. These variables represent whether a failure has occurred, because there was no proper match (LHS failure: structurally not suitable host model or there is at least one constraint not satisfied in LHS of the transformation step), or the transformation result was not sufficient (RHS failure: there was at least one constraint not satisfied in RHS of the transformation step).

In VCFL, each branch has an exact OCL guard condition which is evaluated by the execution engine during the execution.

When a step is connected to more than one follow-up steps, then at most one of the branch conditions is allowed to be true. This means that the conditions must not have any common part. This restriction ensures that the control flow execution of the VCFL is deterministic.

We applied VCFL in projects such as generating user interface from resource model and user interface handler code from statechart model for mobile platform [7]. These applications required control flow support, and all of them can be solved without non-determinism. However, VCFL provides an interface for non-deterministic control flow as well.

### C. 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 (Fig. 5).

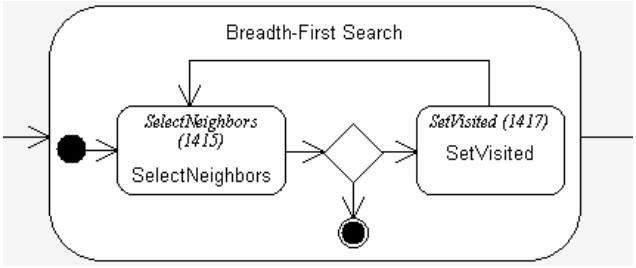


Fig. 5 A Hierarchical step

Often, the OCL constraints assigned to a decision object do not cover all possible cases. It could result that in certain cases

none of the branch paths is selected, in this case the parent step of the actual transformation handles the control flow: breaks the execution of the transformation on the actual level and continues the transformation on the parent level.

### *D. 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 (Fig. 1).

Recursion could be solved with the combination of the iteration and external causalities. A high-level step can call itself, where external causalities represent the actual parameters of the recursive call.

Flattening the state machine is an example when we have to apply a recursive algorithm that first calls flattening on its children before flattening itself.

The parallel execution of the independent transformation steps is supported by the *Fork* and *Join* elements.

In VCFL, if a transformation step fails and the next element in the control flow is a decision object then it could provide 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, if there is no parent state, the transformation terminates with error.

### *E. VCFL Algorithms*

VCFL provides algorithms to check whether a transformation contains isolated or illegal transformation steps and to validate that the OCL constraints assigned to a decision object are disjoint.

The *VCFL Isolated Transformation Steps* algorithm checks whether the user-specified control flow contains isolated transformation steps. This means that starting from the start step we can not reach these steps. The algorithm checks the constraints contained by the decision objects whether all of the branches related to the actual decision object could be selected by the constraints. If a branch is found that can never be selected, the flow edge related to this branch is not taken into consideration by the algorithm. This means that not only the structure of the control flow model but the constraints contained by the decision objects are also taken into account.

In first step, checking the decision objects, the algorithm signs the invalid flow edges, and in second identifies the isolated steps using a modified breadth-first search. Transformation steps which are not found by the search are the isolated steps. The pseudo code of the algorithm is as follows.

```
VCFLISOLATEDSTEPS (Transformation T) : NodeCollection
 1 foreach decision in T
 2   foreach constraint in decision
 3     if NOTSUITABLECONSTRAINT (constraint)
 4       SIGNFLOWEDGEBYCONSTRAINT (constraint)
 5     end if
 6   end foreach
 7 end foreach
 8 SETVISITED (startNode of T)
 9 ENQUEUE (queue, startNode of T)
10 while (queue in not empty)
11   node = DEQUEUE (queue)
12   foreach neighbor in GETNEIGHBORS (node)
13     if NOTVISITED (neighbor)
14       SETVISITED (neighbor)
15       ENQUEUE (queue, neighbor)
16     end if
17   end foreach
18 end while
19 return GETNOTSIGNEDNODES (T)
```

The *VCFL Illegal Transformation Steps* algorithm detects steps in control flow models from which *EndRules* and *FlowFinals* are unreachable. The algorithm is similar to the *VCFL Isolated Transformation Steps* algorithm with the following difference. The modified breadth-first search is started from *EndRules* and *FlowFinals*, and uses the edges in reverse direction as they are in the control flow model. Transformation steps which are not found by the algorithm are the steps from which end steps are unreachable.

The *VCFL Disjoint OCL Constraint* algorithm validates whether the OCL constraints assigned to a decision object are disjoint. This algorithm ensures that at the same time maximum one of the branch conditions of a decision is allowed to be true. Using this algorithm it is guaranteed that the control flow execution of the VCFL is deterministic. The algorithm utilizes that the OCL statements are boolean expressions. It does an *AND* operation for each couple of the OCL statements and if the result is false in each cases then only one of the OCL statement could be true at the same time. The pseudo code of the algorithm is as follows.

```
VCFLDISJOINTCONSTRAINT (VCFLDecision D) : ConstraintPairList
 1 foreach constraintA in D
 2   foreach constraintB in D
 3     if constraintA != constraintB and DOANDONCONSTRAINTS
(constraintA, constraintB)
 4       ADDTOLIST (constraintPairList, constraintA, constraintB)
 5     end if
 6   end foreach
 7 end foreach
 8 return constraintPairList
```

The most complex and maybe the most important VCFL algorithm is the VCFL Termination algorithm which is discussed in next section.

## III. TERMINATION PROPERTIES

The termination properties of a transformation are really important for model transformation. We want to investigate under which conditions an arbitrary VCFL transformation can satisfy termination criteria. The difference between a *transformation* and a *finite sequence of steps* is that a finite sequence of steps always terminates, but a transformation, can contain infinite number of steps. Our aim is that VCFL transformations terminate, therefore an algorithm (VCFL Termination algorithm) has been developed to support the early detection of the infinite loop and the validation of the

control flow that from each step can reach an end step.

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

*Definition (VCFL Transformation).* 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 the *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 (Termination of VCFL transformations).* 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*.

For non-exhaustive and also for exhaustive transformation steps, the *MultipleMatch* attribute of the steps does not modify the termination property of the VCFL control flows for optional finite input model $G_0$.

The termination checker algorithm has to differentiate between certain cases. It needs to take into account whether the VCFL transformation contains loops with decision object or exhaustive transformation steps.

### A. VCFL Control Flows with Non-Exhaustive Transformation Steps

*Proposition.* A VCFL transformation *T*, which contains only non-exhaustive transformation steps $(S_0, S_1 ... S_{n-1})$ and does not contain loops for an optional finite input model $G_0$ always terminates.

*Proof.* The transformation *T* contains finite number of transformation steps $(n = \#STEPS \wedge n < \infty)$. $\forall i \mid 0 \le i \le n-1$ $S_i \in STEPS$ is executed at the most once because it is a non-exhaustive step.

If the multiple match attribute of a step $S_i \in STEPS$ is true, all occurrence of the $S_i^{LHS}$ (LHS of the step $S_i$) is searched and the replacement is executed for all found matches, but step $S_i$ is executed only once. The number of the found matches ($m_i$) is also finite because of the finite input model $G_0$. $n < \infty \wedge m_i < \infty \mid 0 \le i \le n-1$, therefore $k = \sum_{i=0}^{n-1} m_i < \infty$. The number of the steps executed by transformation *T* is finite and *T* terminates.

### B. VCFL Control Flows with Exhaustive Transformation Steps

*Definition* $(\subseteq)$. $G_m \subseteq G_n$ if and only if $G_n$ has a structurally isomorphic subgraph $G_1$ to $G_m$, and in the $G_1$ and in the $G_m$ the corresponding nodes and edges have the same metatype, attributes, attribute values and OCL constraints.

An exhaustively applied step using external causalities gives itself input model and parameters. For an exhaustive step the termination algorithm has to take into consideration the attribute modifications and the generated and deleted elements. An exhaustive transformation step must contain either attribute modification or element deletion to prevent that the same match be found again and again by the matching process. A solution can be also if there is a create type causality and an OCL constraint which holds before the creation and become false afterwards, therefore it prevents to find the same match again on the same place. For example an OCL constraint can validate the existence of a neighbor node. In Fig. 6b the presented transformation step connects a married and unemployed man to a company. The unemployed property is checked by the *const_employer* constraint. After the execution of the step, the matching process does not match the same pattern again in the next iteration, because of the not satisfied constraint. Thus it forbids the repeated application of the same step on the same place again.

*Definition (Create Termination Step – CT step).* A *create termination step S* has only create type internal causalities, it contains an optional OCL constraint $C_1$ in $S^{LHS}$, which must to stand for the host models matched to the $S^{LHS}$ and as a result of the step execution the condition required by the constraint $C_1$ becomes false.

*Definition (Create Termination Step with constraint $C_2$ – CT step with $C_2$).* A *create termination step S* has only create type internal causalities, it contains the OCL constraint $C_2$ in $S^{LHS}$, which must to stand for the host models matched to the $S^{LHS}$ and as a result of the step execution the condition required by the constraint $C_2$ becomes false.

The difference between a *CT step* and a *CT step with $C_2$* is that in first case an arbitrary one of LHS constraints has to fulfill the condition, while in the second case the given constraint ($C_2$) has to comply it.

Obviously, this transformation step property is important only for exhaustive steps or steps which are in loops, because the creation can prevent to find the same match again on the same place and it helps to avoid infinite loops.

Following propositions contain statements about termination properties of the transformations with exhaustive transformation steps.

*Proposition.* Let the transformation step $S_i$ be an exhaustive step. If $S_i^{LHS} \subseteq S_i^{RHS}$ and the step $S_i$ has a match *M* on an optional input model $G_i$ the step $S_i$ never terminates for the

input model $G_i$.

*Proof.* The step $S_i$ has a match $M$ on the input model $G_i$ it generates its output ($G_i^1$) with the $S_i^{RHS}$. $S_i^{LHS} \subseteq S_i^{RHS}$, therefore the $S_i^{LHS}$ has match in $G_i^1$. The step $S_i$ is an exhaustive step and it always has match on the result model of the previous iteration, therefore the $S_i$ never terminates for the input model $G_i$.

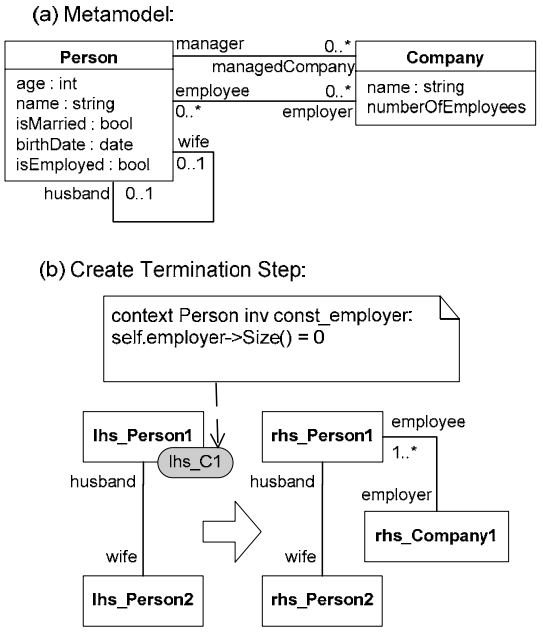


Fig. 6 An example metamodel and a Create Termination Step

*Proposition.* Let the transformation step $S_i$ be an exhaustive step which does not contain internal causalities of deletion and modification type, and $S_i$ is not a CT step. Assume that $T$ is a transformation and $S_i \in T$, the input model of the transformation $T$ is the model $G_0$, and the input model of the step $S_i$ is the model $G_i$. If the $S_i^{LHS}$ has a match $M$ on model $G_i$, the transformation $T$ never terminates for the input model $G_0$.

*Proof.* The step $S_i$ is an exhaustive transformation step, it is executed as long as the $S_i^{LHS}$ has match on model $G_i$. The $S_i$ has a match $M$, which is not modified by the step – there is no deletion, attribute modification, and $S_i$ is not a CT step –, therefore the matching process finds the match $M$ in each iteration. The step $S_i$ never terminates for the input model $G_i$, and $T$ never terminates for the input model $G_0$.

### C. Combining VCFL Transformation Steps

The intention of the transformation step combination is to create a single step $S_C$ from an optional number of transformation steps $S_j, S_{j+1} \ldots S_k$. The combined step can equivalently replace the original steps, because it produces the same result. In the termination analysis we can use the combined step instead of the original transformation steps. It facilitates to replace the steps contained by a VCFL loop with their combined transformation step. The result of the replacement is similar to an exhaustive transformation step, with the difference that a combined step may have a decision object.

The combination algorithm takes not only the structure of the steps into consideration but also their internal- and external causalities and the metatypes of the nodes and edges as well. The algorithm works based on the double pushout (DPO) approach concurrency theorem [8] [9].

An example for transformation step combination is depicted in Fig. 7.

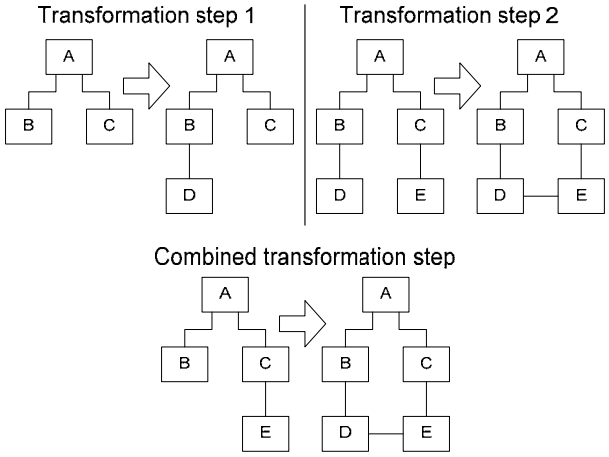


Fig. 7 An example for transformation step combination

### D. Termination Properties of VCFL Loops

A loop contains $n$ transformation steps (where $n>0$) and a decision object. A decision object evaluates the assigned constraints on the actual host model and based on the results selects a flow edge which could be a follow-up or a backward edge as well.

The main difference between a loop with only non-exhaustive steps and an exhaustive step is the exit condition. A transformation leaves an exhaustive step if there is no more match, while in the case of a loop the decision object determines about the exit. If a loop consists of non-exhaustive steps, the step combination algorithm combines them, and the decision about the termination is made based on the combined step and the OCL constraints of the decision object.

An exhaustive step is itself a specific loop. Therefore, if a loop contains exhaustive steps then it is a loop of loops. The algorithm examines separately the exhaustive steps and if each of them terminates then analyses the whole loop.

*Proposition.* Assume that the transformation $T$ contains a loop $L$, let $S_C$ be the combination of the non-exhaustive transformation steps $S_j, S_{j+1} \ldots S_k \in L$. The input model of the transformation $T$ is the model $G_0$, and the input model of the step $S_C$ is the model $G_C$. If $S_C^{LHS} \subseteq S_C^{RHS}$ and the step $S_C$ has a match $M$ on input model $G_C$ the transformation $T$ never terminates for the input model $G_0$.

*Proof.* The transformation step $S_C$ has a match $M$ on input model $G_C$ it generates its output model $G_C^1 \mid S_C^{RHS} \subseteq G_C^1$. $S_C^{LHS} \subseteq S_C^{RHS}$, therefore the $S_C{}^{LHS}$ has match on model $G_C^1$. The step $S_C$ represents a loop and it always has match on the result model of the previous iteration, therefore the $S_C$ never terminates for the input model $G_C$ and the transformation $T$ never terminates for the input model $G_0$.

### E. VCFL Termination Algorithm

For an optional VCFL transformation $T$ the termination algorithm validates the following.

1. If transformation $T$ does not contain loop or exhaustive transformation step then $T$ terminates.
2. If $S \in T$ is an exhaustive transformation step and $S^{LHS} \subseteq S^{RHS}$ the transformation $T$ does not terminate.
3. If $S \in T$ is an exhaustive transformation step, $S$ does not contain delete and modify type internal causalities and $S$ is not a CT step then the transformation $T$ does not terminate.
4. If $L \in T$ is a loop and $S_C$ is the combination of the transformation steps $S_h, S_{h+1} \ldots S_k \in L$ and $S_C^{LHS} \subseteq S_C^{RHS}$ the transformation $T$ does not terminate.

The pseudo code of the VCFL termination algorithm is the following.

```
VCFLTERMINATIONALGORITHM(Transformation T): retValue
 1 if T does not contain loop or exhaustive step then return retValue.true
 2 foreach Transformation Step S in T
 3   if S is exhaustive and RHS of the S contains LHS of the S then return
retValue.false
 4   if S is exhaustive and S does not contain modify or deletion and S is not
an ST step then return retValue.false
 5 end foreach
 6 foreach Loop L in T
 7   combinedStep = COMBINETRANSFORMATIONSTEPS(transformation steps
of the L)
 8   if RHS of the combinedStep contains LHS of the combinedStep then
return retValue.false
 9 end foreach
10 return retValue.undecided
```

If the transformation step contains *create* type internal causality, the algorithm checks whether the host model with the newly added elements contains new possible match places. The algorithm takes into consideration the structure of the pattern, metatypes of the nodes and edges, their attributes and attribute values and also the propagated OCL constraints.

During the combination of steps $S_1$ and $S_2$, the $S_1{}^{RHS}$ and the $S_2{}^{LHS}$ could have more than one matching variation. The algorithm checks all the possible variations in point of VCFL view (external causalities, metatypes).

In the case of loops the exit conditions (structure, attribute value by modify internal causalities and *SystemLastRuleSucceed*) are also checked by the algorithm.

VTA is an offline algorithm; the termination in many cases depends not only on the VCFL transformation model but also on the actual host model. A simple constraint could 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 algorithm can make a sure decision based on the VCFL transformation, and there are other cases when not.

### F. Summary of the Termination Properties

Termination of transformations is not always guaranteed. If a control flow model contains an exhaustive step that can be applied indefinitely to the result models, 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, therefore we can predicate the following proposition.

*Proposition*. A VCFL transformation $T$ terminates if all exhaustive transformation step $S_E \in STEPS$ and loop $L \in T$ terminate.

## IV. Related Work

Many approaches have been introduced in the field of graph grammars and transformations to capture graph domains; for instance, the GReAT [10] [11], the PROGRES [12], the FUJABA [13] [14], the VIATRA [15], the AToM$^3$ [16] and the Attributed Graph Grammar (AGG) [17]. These approaches are specific to the particular system, and each of them has some features that others do not offer.

The GReAT framework is a transformation system for domain specific languages (DSL) built on metamodeling and graph rewriting concepts. The control structure of the GReAT allows specifying an initial context for the matching to reduce the complexity of the general matching case. The pattern matcher returns all the possible matches to avoid the inherent non-determinism in the matching process. The attribute transformation is specified by a proprietary attribute mapping language, whose syntax is close to C. The LHS of the rules can contain OCL constraint to refine the pattern.

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 provides constructs for rule firing and for sequencing the rules to form a controllable transformation process. PROGRES offers refined control structures; both imperative and declarative approaches can be used in either a deterministic or a non-deterministic manner. ACID transactions are also allowed in the control specifications.

GReAT and the PROGRES have a test rules construction. A test rule is a special expression and it is used to change the control flow during execution. A test rule has only LHS. If a test rule is successful (the matching was successful), the rule

after the test node is executable.

In FUJABA the combination of activity diagrams and collaboration diagrams (story-diagrams) are used to express control structures. Story-diagrams are a visual programming language that facilitates the specification of complex application-specific object structures. Moreover, FUJABA extended story-diagrams by statecharts to story-charts. Story-charts use statecharts and activity diagrams to define complex control flows and collaboration diagrams to specify the entry, exit, do, and transition actions that deal with complex object-structures [14].

VIATRA (Visual Automated Transformations) is a model transformation framework developed mainly for the formal dependability analysis of UML models. In VIATRA, metamodeling is conceived specially: the instantiation is based on mathematical formalisms and called Visual Precise Metamodeling. The attribute transformation is performed by abstract state machine statements, and there is built-in support for attributes of basic Java types. The model constraints can be expressed by graph patterns with arbitrary levels of negation. The rule constraints are also specified by graph patterns. VIATRA uses abstract state machines (ASM) to define the control flow of the system.

The transformation and simulation tool AToM$^3$ uses model transformation to simulation traces in order to simulate the operations. The rule constraints can contain generalized negative application conditions and can be pre- and postconditions to events. Constraints can be both semantic and graphical constraints. Similarly to AGG, the control flow consists of layers; the rules are sequenced by priority numbers within the layers. A rule is executed only once, but in case of non-overlapping matches, the rules are applied to all the matches.

AGG is a visual tool environment consisting of editors, interpreter and debugger for attributed graph transformation; attribute computation by Java; supports a hybrid programming style based on graph transformation and Java. In AGG termination criteria are implemented for Layered Graph Transformation Systems (LGTS). The criteria they propose are based on assigning a layer to each rule, node and edge type. For termination, they define layered graph grammars with deletion and non-deletion layers. Termination criteria are expressed by deletion and non-deletion layer conditions. The layers fix the order how rules are applied. The interpretation process first has to apply all rules of layer 0 as long as possible, and then all rules of layer 1, etc. Rule layers allow specifying a simple control flow graph transformation. Once the highest layer has been finished the transformation stops, unless the option "loop over layers" is turned on.

Table 1 gives a comparison of control flow, constraint, and attribute transformation support of the presented approaches.

Contextual layered graph grammars (CLGGs) have been used in parsing, as they provide a natural way to steer the parsing process, thereby reducing its non-determinism and its complexity. A contextual layered graph grammar is a construct *CLGG = (S, T, P, cl, dl, rl),* where *S* is a labeled graph, called the *initial graph*, *T* is a set of node and edge *types* of labels and *P* is a set of rules. The *layering functions cl*, *dl*, and *rl* assign a *creation* and a *deletion* layer to elements of *T* and a unique layer to each rule $p \in P$, respectively. In [18], the following concrete termination criterion for CLGGs was discussed.

TABLE I
COMPARISON TABLE OF CONTROL FLOW, CONSTRAINTS AND ATTRIBUTE TRANSFORMATION SUPPORT FOR MODEL TRANSFORMATION TOOLS

| | Control Flow | Constraints in the rule | Attribute transformation |
|---|---|---|---|
| **VMTS** | Stereotyped activity diagrams | Instantiation + OCL | XSL |
| **GReAT** | Deterministic, non-deterministic, recursion | OCL | C-based attribute mapping language |
| **AGG** | Layers (exhaustive or once, loop) | JAVA, NAC | JAVA |
| **PROGRES** | Imperative and declarative, transactions | Attribute constraints, cardinality, negative edge | Built-in or host programming language (esp. C) |
| **VIATRA** | ASM | Graph pattern | ASM statements (built in support for basic JAVA types) |
| **AToM$^3$** | Layers with priorities, sequencing by priority. Parallel execution of non-overlapping matches. | Generalized NAC, application conditions. | Python |
| **FUJABA** | Story diagrams | Story diagrams, JAVA | Story diagrams |

The layering condition above guarantees the termination of the process, by producing a parsing derivation, or proving that the sentence cannot be parsed. The existence of the layering function *rl* allows the partitioning of the set *P* into a collection of sets $(P_1, P_2, \ldots P_k)$. Rules in a set $P_i$ can be used only after rules from the set $P_{i-1}$ have been used and are no longer applicable. Moreover, after using a rule from $P_j$ , $j \geq i$, no rule from $P_{i-1}$ can be applied any longer.

This also provides a generalization to the following approach. In [19] a contribution towards solving the termination problem for rewriting systems with external control mechanisms is given. It extends the concept of transformation unit to high-level replacement systems. For high-level replacement units, several abstract properties based on termination criteria are stated and proved. However, terminating rules do not always satisfy the measure function required by this approach, since attribute transformations and other constraint can also influence the termination properties of a rule.

The layering approach is not applicable for VCFL transformations. In CLGGs, there is no strict control flow, rules are created without any fixed order and assigned to different layers. In VCFL, we have a fixed control flow specified by stereotyped activity diagram. Therefore, we have

to examine the termination conditions of fixed control flows with given transformation steps and step structures (without any modification of the order of the steps).

## V. Conclusion

This paper has provided a control flow technique for model transformations based on graph rewriting. The transformations are represented in the form of explicitly sequenced transformation steps. We have shown the fundamental concepts of the VCFL approach.

As it was presented, a control structure language needs a sequence as well as a conditional branch mechanism, hierarchy, parallel executions and iteration constructs. VCFL has all these control structures in a deterministic implementation.

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.

In this work, we discussed the properties of the VMTS Visual Control Flow Language. We stated and proved several termination criteria for transformation steps, loops and transformations. An algorithm to validate the termination is also provided.

The introduced approach can be generalized to other control flow languages which facilitate to assign constraints to transformation steps and supports constraint evaluation. The presented concepts and algorithms can be reused with minor, approach related modifications.

VCFL has successfully been applied in industrial projects, like generating user interface from resource model and user interface handler code from statechart model for Symbian [7] and .NET Compact Framework mobile platform [20].

## 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/vmts
[3] OMG UML 2.0 Specification, http://www.omg.org/uml/
[4] G. Rozenberg (ed.), "Handbook on Graph Grammars and Computing by Graph Transformation: Foundations", Vol.1 World Scientific, Singapore, 1997.
[5] OMG Object Constraint Language Specification (OCL), www.omg.org
[6] M. Fowler, UML Distilled, "A Brief Guide to the Standard Object Modeling Language", 3rd edition, Addison-Wesley, ISBN: 0321193687, 2003.
[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] 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.
[9] H. Ehrig, M. Korff, M. Löwe, "Tutorial introduction to the algebraic approach of graph grammars based on double and single pushouts". In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, Proceedings of the 4th International Workshop on Graph-Grammars and Their Application to Computer Science, volume 532 of Lecture Notes in Computer Science, pages 24-37. Springer Verlag, 1991.
[10] G. Karsai, A. Agrawal, F. Shi, J. Sprinkle, "On the Use of Graph Transformation in the Formal Specification of Model Interpreters", Journal of Universal Computer Science, Special issue on Formal Specification of CBS, 2003.
[11] A. Agrawal, "A Formal Graph-Transformation Based Language for Model-to-Model Transformations", PhD Dissertation, Vanderbilt University, Dept of EECS, August, 2004.
[12] A. Schürr, A. Zündorf, "Nondeterministic Control Structures for Graph Rewriting Systems", in Proc. WG'91 Workshop in Graph- Theoretic Concepts in Computer Science, LNCS 570, Springer Verlag (1992), pp. 48-62, also: Technical Report AIB 91-17, RWTH Germany, 1991.
[13] FUJABA Homepage, http://wwwcs.upb.de/cs/fujaba/
[14] Hans J. Köhler, Ulrich A. Nickel, Jörg Niere, Albert Zündorf, "Integrating UML Diagrams for Production Control Systems", Proc. of the 22nd International Conf. on Software Engineering (ICSE) Limerick Ireland, ACM Press, 2000, pp. 241-251.
[15] D. Varró and A. Pataricza, "VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML", Journal of Software and Systems Modeling, 2003.
[16] J. Lara, H. Vangheluwe , M. Alfonseca, "Meta-modelling and graph grammars for multi-paradigm modelling in AToM", Software and Systems Modeling (SoSyM), 3(3):194-209, August 2004.
[17] G. Taentzer, "AGG: A Graph Transformation Environment for Modeling and Validation of Software", In J. Pfaltz, M. Nagl, and B. Boehlen (eds.), Application of Graph Transformations with Industrial Relevance (AGTIVE'03), vol. 3062. Springer LNCS, 2004.
[18] Hartmut Ehrig, Karsten Ehrig, Juan de Lara, Gabriele Taentzer, Dániel Varró and Szilvia Varró-Gyapay, "Termination Criteria for Model Transformation", LNCS, Vol. 3442: Fundamental Approaches to Software Engineering: 8th International Conference, FASE 2005, Edinburgh, UK, April 4-8, 2005, pages 49-63. Springer-Verlag, 2005.
[19] Paolo Bottoni, Manuel Koch, Francesco Parisi-Presicce, Gabriele Taentzer, "Termination of High-Level Replacement Units with Application to Model Transformation", Electr. Notes Theor. Comput. Sci. 127(4): 71-86, 2005.
[20] 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.