Termination Analysis of the Transformation UML to CSP

Márk Asztalos, László Lengyel, Tihamér Levendovszky, Hassan Charaf

Department of Automation and Applied Informatics
Budapest University of Technology and Economics
Goldmann György tér 3, H-1111 Budapest, Hungary
{asztalos, lengyel, tihamer, hasssan}@aut.bme.hu

Abstract: The transformation from UML activity diagrams to CSP models is a helpful model transformation, which can be used to analyze and verify some aspects of a UML activity diagrams. A working solution has been developed with our tool, the Visual Modeling and Transformation System, and in this work we formally prove that the transformation terminates for every valid input activity diagram model, and therefore it can be used in practice.

Keywords: model transformation, graph rewriting, termination, metamodeling, UML, CSP

1 Introduction

1.1 Motivation

Model transformation has become a focused area in model-driven research. Graph rewriting systems and the theories related to them are widespread applied in model transformations. Graph transformation provides a rule-based modification of graph models.

The analysis of formal properties of transformations is important. One essential property is the termination of a transformation. In general, it is undecidable whether a graph rewriting system is terminating [11]. In certain cases, we can guarantee the termination, for example if in the transformation, all rules are applied only once and no loop is available. We can also define constraints on the input models. Our aim is to guarantee that the transformation terminates after finite number of steps for every valid input model.

The Unified Modeling Language (UML) [1] is the de facto standard of software design. UML activity diagrams are used for example to describe the low-level behavior of software components and to model business processes.

Communicating Sequential Processes (CSP) [2] is a formal language for describing patterns of interaction in concurrent systems and it is supported by a mathematical theory.

To analyze the behavior of a UML activity diagram and to verify some aspects of it, we need a formal semantic [3], the CSP is appropriate for this function. This is why a transformation, which converts UML activity diagrams to a CSP models is very helpful.

In this work, we use our transformation, which generates CSP models from UML activity diagrams and prove that this transformation terminates, and therefore, this can be applied for any valid UML activity diagram input model.

1.2 Problem Statement

To use CSP as a semantic domain for activity diagrams we need a precisely defined mapping from diagrams to CSP models. In [4], the transformation method, and the metamodels of CSP models and UML activity diagrams are presented.

Our group in the Department of Automation and Applied Informatics has developed Visual Modeling and Transformation Systems (VMTS), a software tool that is able to design and run transformations. This environment is based on graph rewriting rules and metamodels. We published a solution for the UML to CSP transformation [5]. (To better understand the transformation itself, please read [5] or follow the steps in the tutorial [6].) The termination of this transformation has not yet been formally proved.

1.3 Structure of Paper

The rest of the paper is organized as follows: in Section 2, our UML to CSP transformation and the rules of the transformation are detailed. In Section 3, some important definitions and theorems are summarized, and finally termination analysis is presented in Section 4.

2 Transformation

The control flow of the UML to CSP transformation is shown in Figure 1. VMTS uses the notation of UML activity diagrams to describe transformations. In the

control flow, rules, directed edges and decision nodes can be found. There are two important properties, which cannot be seen in the figure:

1 The application of *Process Activity Edge* rule is exhaustive. This means that the rule is executed repeatedly while it is possible to apply. The transformation proceeds to the next rule only when it cannot be matched with the input model, or the execution was unsuccessful. *Process Final Node*, *Process Join Node* and *Process Merge Node* rules are also exhaustive rules.

2 Each decision node of the transformation control flow is left by two edges. The transformation proceeds to the edge with the label *success* if the application of the previous rule succeeded. If the previous rule could not be matched, or the application was unsuccessful, then the transformation proceeds to the edge with the label *failure*.

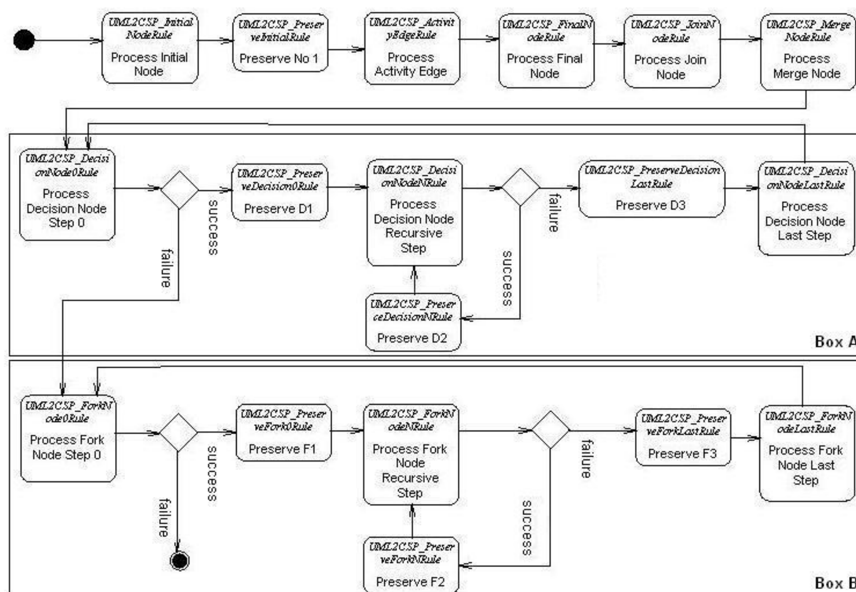

Figure 1
Transformation control flow

2.1 Metamodels

VMTS uses metamodels to validate the LHS and RHS patterns of the rules, and the input model as well. Metamodels of UML activity diagrams and CSP models are presented in [4].

2.2 Transformation Rules

In VMTS, model processing is based on graph transformations [9]. The elements of a transformation are graph rewriting rules. A rewriting rule consists of a left-hand side (LHS) and a right-hand side (RHS) pattern. LHS describes the part of the model we are searching for, RHS defines the replacement pattern.

In Figure 2, the *Process Initial Node* rule is shown. In RHS, a *CspContainer* element can be seen, which cannot be part of the activity diagram model because it is a CSP element. This is a newly created element, which will be placed into the output CSP model. In other rules, CSP model elements appear in LHS as well, but these are also elements of the output CSP model.

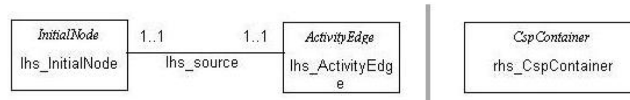

Figure 2
Rule *Process Initial Node*

Constraints can be defined for any elements of the LHS, or RHS using Object Constraint Language (OCL) [10]. Every activity diagram element has an attribute named *IsProcessed*. During the application of the rules we change the value of this attribute, and we define constraints in LHS to guarantee that certain elements cannot be processed twice with the same rule.

In Figure 3, the *Process Activity Edge* is shown. When the LHS of the rule is matched, a constraint is checked if the value of the *IsProcessed* attribute owned by the *Action* element is an empty string. If we apply the rule successfully, we change this value to 'processed' so this rule cannot be matched again with the same *Action* element.

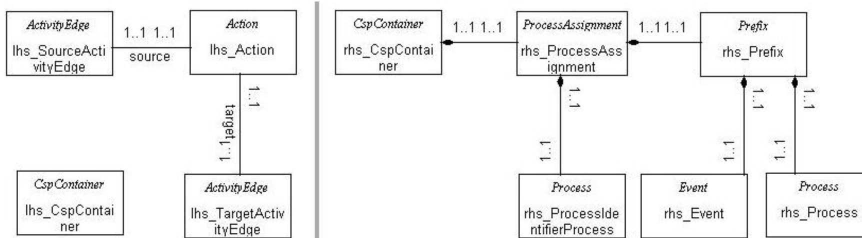

Figure 3
Rule *Process Activity Edge*

In Figure 4, the *Process Final Node* rule is shown. We use the *IsProcessed* attribute of the *FinalNode* element of the LHS, as described before.

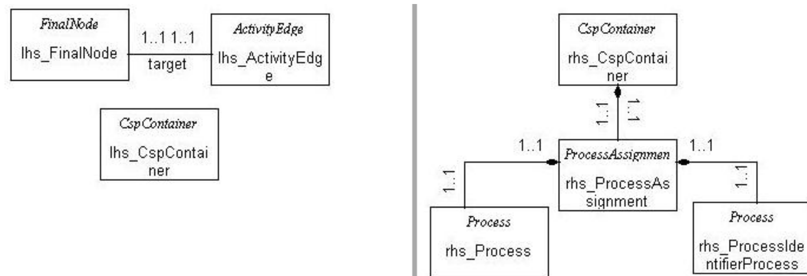

Figure 4
Rule *Process Final Node*

In Figure 5, the *Process Join Node* rule is presented. The value of *IsProcessed* attribute of the *JoinNode* of LHS is checked and modified during the application of the rule in the same way as in the two previous rules.

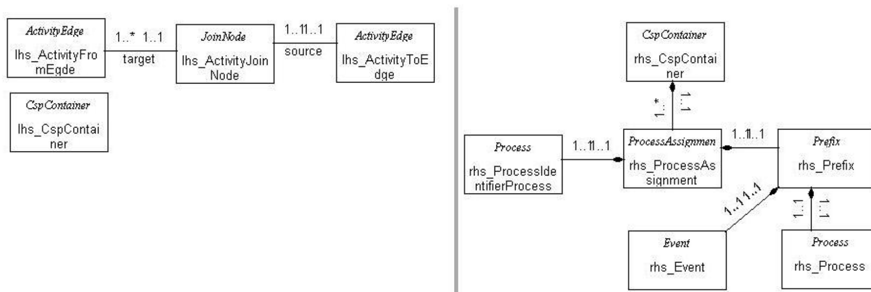

Figure 5
Rule *Process Join Node*

In Figure 6, the *Process Merge Node* rule is shown. We check and modify the *IsProcessed* attribute of the *MergeNode* element of LHS.

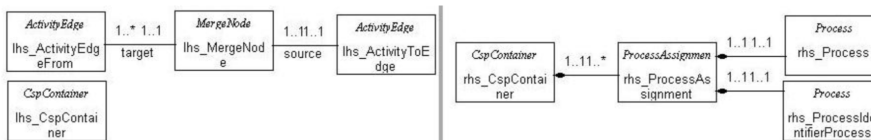

Figure 6
Rule *Process Merge Node*

The rules, the name of which starts with the word *Preserve* are needed because some characteristics of the VMTS control flow language, but they do not modify, create or delete any elements.

In Box A in Figure 1, the processing of the *decision nodes* of the input activity diagram is realized with a complex control flow, with more than one rule. The

Process Decision Node Step 0 (Figure 7/a) matches a *decision node* of the input model. When we apply this rule we check and modify the *IsProcessed* attribute of the *DecisionNode* element of LHS. The *Process Decision Node Recursive Step* (Figure 7/b) processes an edge which leaves the *decision node* in each step. We check and modify the *IsProcessed* attribute of the *ActivityEdge* of LHS. The *Process Decision Node Last Step* (Figure 7/c) processes the last edge, the guard condition of which is *else*.

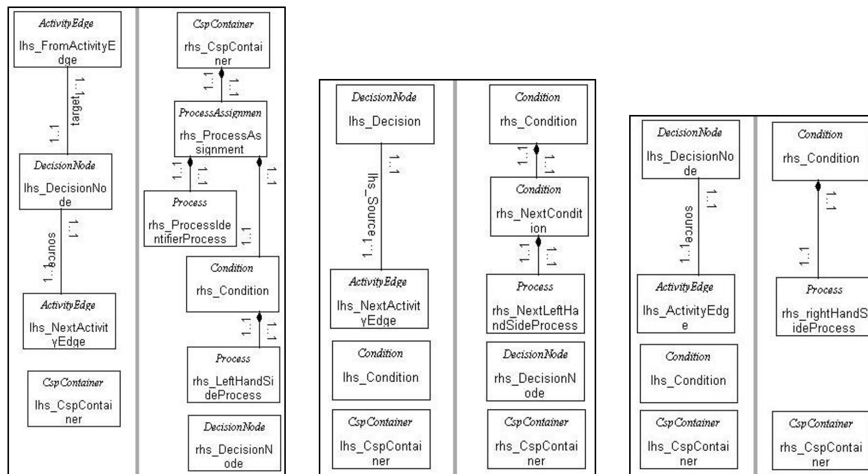

Figure 7
Rules *Process Decision Node Step 0* (a), *Process Decision Node Recursive Step* (b), *Process Decision Node Last Step* (c)

With external causalities [5], we guarantee that after the successful application of the *Process Decision Node Step 0* rule, the *Process Decision Node Recursive Step* rule and the *Process Decision Node Last Step* rule matches the same *DecisionNode* element in LHS that we matched in the first rule.

The processing of the *fork nodes* (Box B in Figure 1) operates in the same way as the processing of the *decision nodes*, the rules are shown in Figure 8.

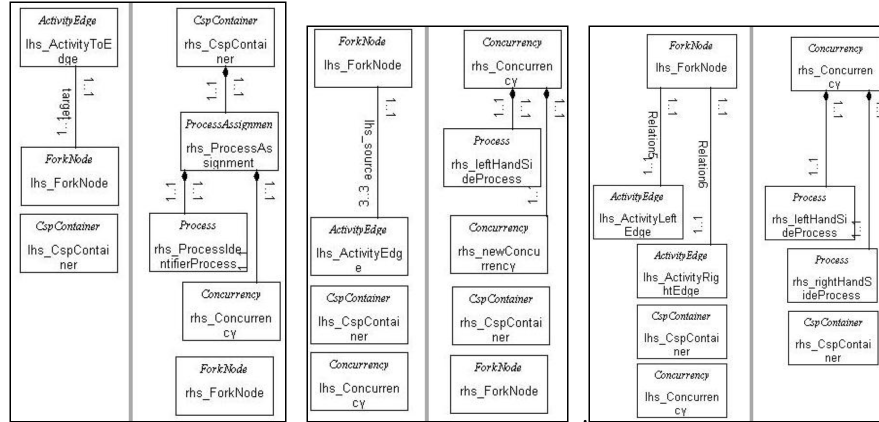

Figure 8

Rules *Process Fork Node Step 0* (a), *Process Fork Node Recursive Step* (b), *Process Fork Node Last Step* (c)

3 Backgrounds

To describe graph transformations and graph rewriting rules, we use the definitions and theorems presented in [8]. Note that the theorems in the referenced documents are proven to injective matches only. Here we also use the formalism of [7]. In the following we summarize some important definitions and a theorem.

Definition 3.1 Let $p_i (i = 1,2,...)$ be a sequence of graph productions and $((E_i, e_i*, e_{i+1}))$ a sequence of E-dependency relations.

Compose the sequence of E-based compositions $p_i* = (L_i* \leftarrow K_i* \rightarrow R_i*)$ where $p_1* = p_1$ and $p_n* = (p_1 +_{E_1} p_2) +_{E_2} ... +_{E_n} p_n$.

Definition 3.2 A cumulative LHS series of a production sequence is the graphs series L_i* consisting of the left hand side graphs of p_n*.

Definition 3.3 The cumulative LHS size series of a production sequence is the nonnegative integer series $| L_i*|$.

Theorem 3.4 A graph transformation terminates if for all infinite cumulative LHS sequence of the graph productions created from the members of the productions of the system, it holds that

$$\lim_{i \to \infty} | L_i* | = \infty .$$

(We assume finite input graphs and injective matches.)

4 Termination Analysis

In connection with the following analysis three notes have to be made:

1 VMTS uses finite input graphs and injective matches, so definitions and the theorem in Section 3 can be applied.

2 When we analyze the rules we only work with the model elements which come from the activity diagrams. In LHS and RHS the elements of the output CSP models also appear, but these parts only restrict the application of the rules.

3 During the whole transformation, we do not modify the structure of the input activity diagram except the value of the *IsProcessed* attribute of some elements.

To prove the termination of the transformation UML to CSP, we use the following lemmas.

Lemma 4.1 The exhaustive application of the rule *Process Activity Edge* terminates for every valid UML activity diagram input model.

Proof Our aim is to prove that we cannot apply the *Process Activity Edge* rule infinite number of times for any finite input UML activity diagrams.

In Figure 9, the rule itself is presented. Squares represent the graphs of LHS, RHS and the interface graph (*K*), and circles represent the *Action* elements. One circle in a square means, that there is exactly one *Action* element in the graph. Filled circle means, that the *IsProcessed* attribute of the element is set to *processed*, and empty circle means that the value of this attribute is an empty string.

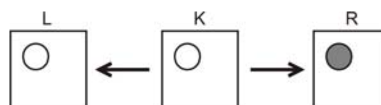

Figure 9
Rewriting pattern no. 1

In Figure 10, the E-based composition of two *Process Activity Edge* rules is shown. The cumulative LHS now contains two circles, because R1 and L2 contained exactly one-one *Action* nodes, but these elements cannot be identified with each other.

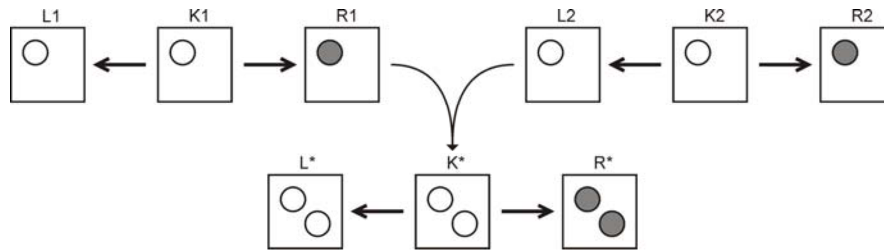

Figure 10
Composition of two patterns no. 1 rewriting rule

In Figure 11 the E-based composition is shown after *n* number of rule applications. The cumulative LHS contains only empty circles and the cumulative RHS contains only filled ones. The application of the next rule results that the cumulative left-hand side will contain one more empty circle (one more *Action* node).

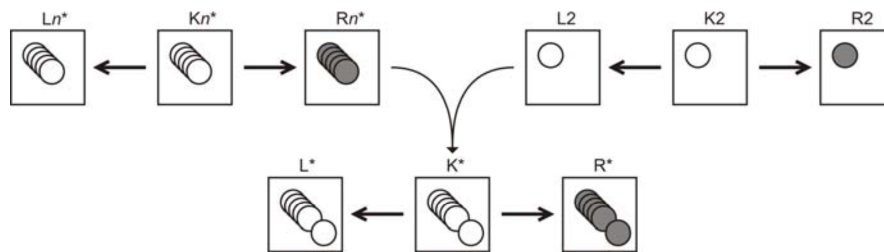

Figure 11
Composition of *n*+1 number of pattern no. 1 rules

Using the notation of Definition 3.2, let $nbr_{Action}(L_i*)$ be the number of *Action* elements in LHS. Every time when this rule is executed again, at least one new node appears in LHS of the composed rule, therefore $nbr_{Action}(L_i*)$ exceeds all limits. Since $0 \le nbr_{Action}(L_i*) \le |L_i*|$, $|L_i*|$ also exceeds all limits. Referencing the Theorem 3.4, this results that the exhaustive application of the rule *Process Activity Edge* always terminates. ■

Lemma 4.2 The exhaustive application of rules *Process Final Node*, *Process Join Node* and *Process Merge Node* separately terminates for every valid UML activity diagram.

Proof To prove Lemma 4.1, we used a pattern. The *Process Activity Edge* rule did not change the structure of the input activity diagram model in LHS, only the *IsProcessed* attribute of an element has been set to *processed*. Before we modified this attribute, to match LHS, and apply the rule, we checked that this value was an empty string. Nothing else specialty of the *Process Activity Egde* rule has not been exploited. Therefore the proof of Lemma 4.1 can be easily applied to the rules

Process Final Node, *Process Join Node* and *Process Merge Node* as well, because they behave similarly to rule *Process Activity Edge*, as presented in Section 2. ■

Lemma 4.3 The *decision node* processing part of the transformation (Box A, in Figure 1) terminates for every valid UML activity diagram input model.

Proof Each rule of Box A in Figure 1 is applied not exhaustively. By following the directed edges of the control flow, it is obvious that there are only two ways to produce a loop:

1. If the *Process Decision Node Recursive Step* can be successfully applied, then the next decision node of the control flow forwards the execution to the *Preserve D2* and then again to the *Process Decision Node Recursive Step*. Because *Preserve D2* rule does not modify anything in the model, the result of this loop is the same as we would apply the *Process Decision Node Recursive Step* exhaustively.
2. The other possible loop is the following: apply *Process Decision Node Step 0* successfully, and then apply the *Process Decision Node Recursive Step* as many times as we can. Finally use the rule *Decision Node Last Step*, and then start again with the first rule.

The first loop is equivalent to the exhaustive application of the rule *Process Decision Node Recursive Step*, therefore, we can apply the proving pattern that we used in Lemma 4.2, hence this loop is terminating.

To analyze the loop number 2, we use the same notation that we have presented in connection with Lemma 4.1. The only modification is that the circle means a *DecisionNode* element of the graph, instead of an *Action* element. If we want to describe the rewriting rule of *Process Decision Node Step 0*, we can do it with the pattern no. 1 rule in Figure 9.

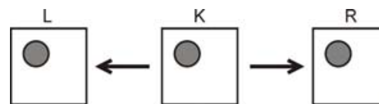

Figure 12
Rewriting pattern no. 2

With the rewriting rule in Figure 12, we can describe the behavior of the *Process Decision Node Recursive Step* rule, because it contains exactly one *DecisionNode* element in LHS and RHS, but does not modify it. As mentioned in Section 3, with external causalities it is guaranteed that the rule *Process Decision Node Recursive Step* and the rule *Process Decision Node Last Step* matches the same *DecisionNode* element of the input model, that we have matched with the first *Process Decision Node Step 0* rule in the loop. Therefore the composition of two *Process Decision Node Recursive Step* rules can described as in Figure 13. This

results that Figure 13 also presents the composition of the application of rules *Process Decition Node Recursive Step*.

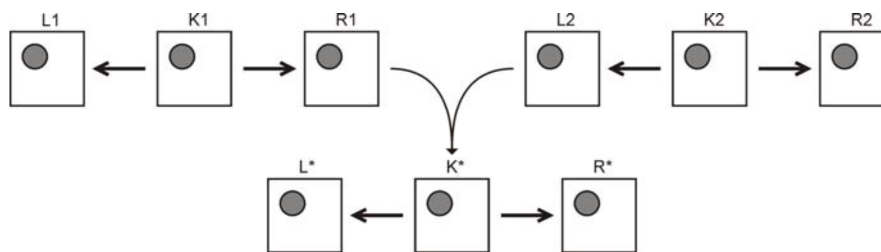

Figure 13
Composition of two pattern no. 2 rewriting rules

The pattern in Figure 12 also represents the rule *Process Decision Node Last Step*, therefore the composition of rules of the rest of the loop (all rules, except the first *Process Decision Node Step 0* rule) can be described with pattern no 2 rule in Figure 12. The whole loop itself is the composition of the first rule and the rest of the loop, which can be seen in Figure 14.

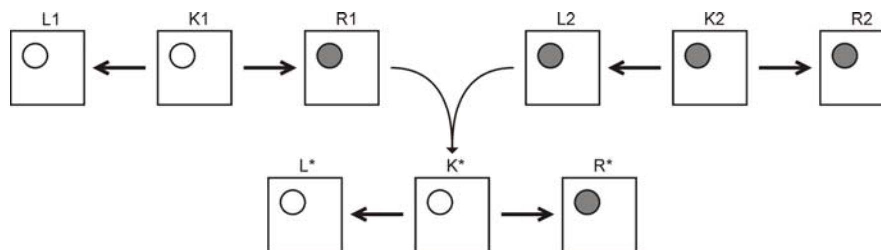

Figure 14
Composition of a pattern no. 1 and a pattern no. 2 rule

Similarly to the proof of lemma 4.1 now we have to produce the composition of pattern no 2. rules and in the same way, we can guarantee that the number of applications is finite and therefore the loop is terminating.

Since both possible loop in Box A terminates, the whole control flow of the box itself also terminates. ■

Lemma 4.4 The *fork node* processing part of the transformation (box B, in Figure 1) terminates for every valid UML activity diagram input model.

Proof This lemma can be proved in the same way as lemmas 4.3, since the operation of Box B is the same as the operation of Box A. ■

Proposition 4.5 The UML to CSP transformation terminates for every valid UML activity diagram input model.

Proof Each rule, which is applied only once (the not exhaustive rules) terminates. Lemma 4.1 and 4.2 guarantee that the exhaustive application of each exhaustive rule separately terminates. Lemma 4.4 and 4.5 guarantee that the *decision node* and the *fork node* processing parts of the transformation terminates (Box A, and Box B). It is proven that the transformation is the sequence of separately terminating processes and therefore the whole transformation itself terminates. ■

Conclusions

In this paper we have presented the analysis of a transformation from UML activity diagrams to CSP models. We proved that the transformation terminates for every valid input model and therefore it can generate CSP expressions for all UML activity diagrams.

References

[1] Object Management Group: Unified Modeling Language, version 2.1.1 (2006), http://www.omg.org/technology/documents/formal/uml.htm

[2] Hoare, C.A.R.: Communicating Sequential Processes, Prentice Hall International Series in Computer Science, Prentice Hall (April 1985)

[3] D. Harel, B. Rumpe: Modeling Languages: Syntax, Semantics and all that Stuff (or, What's the Semantics of "Semantics"?) (July 18, 2004)

[4] D. Bisztray, K. Ehrig R. Heckel: Case Study: UML to CSP Transformation, AGTIVE 2007 Graph Transformation Tool Contest

[5] M. Asztalos, L. Lengyel, T. Levendovszky, H. Charaf: Graph Transformation Contest - UML to CSP Transformation AGTIVE 2007 Graph Transformation Tool Contest

[6] M. Asztalos: UML to CSP Transformation with VMTS, http://vmts.aut.bme.hu

[7] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer: Fundamentals of Algebraic Graph Transformation. Springer, 2006

[8] T. Levendovszky, U. Prange, H. Ehrig.: Termination Criteria for DPO Transformations with Injective Matches, Graph Transformation for Verification and Concurrency, 2006

[9] G. Rozenberg.: Handbook on Graph Grammars and Computing by Graph Transformation, Foundations, Vol. 1 World Scientific, 1997

[10] Object Management Group: UML 2.0 Object Constraint Language Specification, http://www.omg.org/

[11] D. Plump: Termination of graph rewriting is undecideable, Fundamental Informatica, Vol. 33(2), pp. 201-209, 1998