Property Analysis of Visual Behavior Models to Code Transformation

Tamás Mészáros, Gergely Mezei

Budapest University of Technology and Economics, Hungary
mesztam@aut.bme.hu, gmezei@aut.bme.hu

Abstract: Nowadays, when visual modeling is becoming more and more popular, it is still an open issue how to model the runtime behavior (animation) of visual languages. We are currently working on a complete solution to this issue, we have specified visual languages that can describe the behavior of arbitrary metamodeled visual language and we have also provided a graph-rewriting-based transformation which processes these 'animation' models and generates executable source code. This paper shortly introduces previous work, and focuses on the analysis of the runtime properties of the transformation. We performed termination analysis on the transformation, and examined the runtime requirements of the algorithm, based on the size of the input models. We have also verified that the transformation processes topologically correct models only. We present generic techniques which are applicable not only in connection with this concrete case, but with arbitrary other graph-rewriting based model transformations.

Keywords: metamodeling, animation, termination, runtime complexity

1 Introduction

In recent years, domain-Specific Modeling (DSM) has gained increased popularity in software modeling. Domain-Specific Modeling Languages (DSMLs) can simplify the design and the implementation of systems in various domains. Domain-specific visualization helps to understand the models for domain specialists not familiar with programming. A popular way to define DSMLs is metamodeling. Metamodels define a vocabulary of model elements for a specific language by describing the available model elements, their properties and the relations between the elements. This definition is often referred to as the abstract syntax of the language. However, metamodeling is not meant to describe the visual representation, namely the concrete syntax, or the dynamic behavior (animation) of modeling items. Based on the metamodel, a default concrete syntax can be generated automatically, but the description of customized visualization – including colors, sizes and layouting - usually needs additional modeling techniques.

In [1], we have presented an integrated solution to describe the dynamic behavior of the models in a generic and visual way. In our approach, we separate the model and its animation logic, and provide visual languages to define the animation of the model elements or their visualization. The integration of the models is performed by both references between models of different domains and by the model processors. The integration of external components or frameworks into our environment is supported by a visual language and a code generator, thus the animation logic can handle all components in a uniform way.

To be able to execute the visual behavior models with high performance - instead of the runtime interpretation of the models - we generate executable source code from them and compile the source code into reusable dynamic linked libraries. We perform the code generation with graph rewriting-based [2] model transformation. The transformation itself is published in depth in [3]. This paper presents the animation framework and the transformation in a nutshell and evaluates important properties of the transformation in detail. The presented analysis techniques are not specific to this specific case, but are generally applicable in connection with any other transformations as well.

2 Background

Visual Modeling and Transformation System (VMTS) [4] is a general purpose metamodeling environment supporting n-level metamodeling. N-level means in this context that the instance models can be used as metamodels: they can be used to define model hierarchies such as meta class diagram - class diagram - object diagram. The maximum depth of these hierarchies is not limited; in VMTS, we can construct an n-level modeling chain. VMTS uses a proprietary modeling space. Models in VMTS are represented as directed, attributed graphs. In our approach, edges are attributed as well.

2.1 VMTS Animation Framework (VAF)

The VMTS Animation Framework (VAF) [1] is a flexible framework supporting the real-time animation of models both in their visualized and modeled properties. VAF separates the animation of the visualization from the dynamic behavior (animation) of the model. For instance, the dynamic behavior of a graphically simulated statechart is really different from that of a simulated continuous control system model. In our approach, the domain knowledge can be considered as a black-box whose integration is supported with visual modeling techniques. Using this approach, we can integrate various simulation frameworks or self-written components with event-driven communication. The animation framework provides three visual languages to describe the dynamic behavior of a metamodeled model and their processing via an event-driven concept. The key

elements in our approach are the *events*. *Events* are parameterizable messages that connect the components in our environment. The services of the presentation framework, the domain-specific extensions, and possible external simulation engines are wrapped with *event handlers*, which provide an event-based interface. Communication with event handlers can be established using events. The definition of event handlers is supported by a visual language. The visual language defines the event handler and the possible events. The default implementation of an event handler can be generated based on the interface of the wrapped objects [5]. The animation logic can be described using an event-driven hierarchical state machine, called *Animator*. We have designed another visual language to define these state machines. The state machine consumes and produces events. The transitions of the state machine are guarded by conditions testing the input events and fire other events after performing the transition. The input (output) events of the state machine are created in (sent to) another state machine or an event handler. The events produced by the event handlers and the state machines are scheduled and processed by a DEVS [6] based simulator engine. The event handlers and the state machines can be connected in a high-level model. The communication between components is established through ports. Ports can be considered labeled buffers. Note that both the high- and low-level languages are defined by the same metamodel, however, based on their application they can be considered as two different languages.

2.2 Processing Visual Behavior Models

The behaviour models are transformed into executable source code, more precisely, into the model of the code, then the source code is compiled and executed using a DEVS-based simulation framework. We employ a C# DOM similar to the Microsoft CodeDOM [7] to model source code. The control flow model of the transformation which processes the animation models is depicted in Figure 1. The sequence can be departed into three well-separated parts. Part (1) verifies the input models, if they are topologically correct (detailed in Section 2).

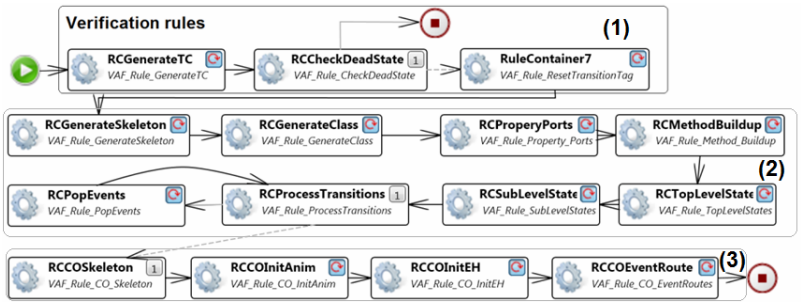


Figure 1
Transformation control flow model

The transformation processes those models only, the states of that are reachable from the start state. Part (2) creates the individual animation classes for each *Animator* element and their contained state machine, while part (3) creates the configuration class, which ties animators and event handlers together.

3 Input Model Verification

We have extended the original transformation presented in [3] with three additional rules (Figure 1, Figure 2) used to verify the input models as a first step. The transformation verifies the reachability of the state machine states. Actually, we do not verify the validity of the required event sequences to reach a state, but only the topological structure of the input model is analyzed. If a specific state is topologically unreachable, it indicates a design flaw. This is not a domain-specific problem, but applies to each simulation.

Proposition 1 The transformation processes topologically correct state machines only (in sense of each state of the input model should be reachable from the start node).

Proof:

Figure 2 depicts the rules which are used to detect unreachable states in the input models. The *GenerateTC* (*Generate Transitive Closure*) rule matches either a *StartState* or an already processed (indicated by a flag on the element) element for the *stateFrom* node. It also selects a still not processed edge for the *Transition* edge and a not processed node for the *stateTo* node. After a successful match, it sets the *Processed* flag of the *transition* edge and the *stateTo* node to *true.* The rule is executed exhaustively, thus, when it finishes, there is not a state (*stateTo*) in the input model, that has a processed neighbor with incoming edge. This is possible, if (i) each state is in processed state (meaning, that each state is reachable), or (ii) there is not an incoming edge from the processed (reachable) states into any of the unprocessed ones. The unprocessed states are not reachable from the start node, so if the *CheckDeadState* rule can mach such a node, then the input model is invalid, and the transformation exists. Otherwise each state is reachable from the start, and the transformation proceeds. □

Finally the *ResetTransitionTag* rule matches each transition, and resets the *Processed* flag of the transition and the two connected *State*s to false for later use.

a) *GenerateTC* b) *CheckDeadState* c) *ResetTransitionTag*

Figure 2
Rules detecting unreachable states

4 Termination Analysis

We use the definitions and theorems presented in [8] to make the proving method simpler. These theorems are proven to injective matches only, but this is not a problem, because the presented transformation uses injective matches only.

Definition 1 An E-concurrent production p^* is an E-based composition if there is at least one input graph G_0 with an E-related transformation $G_0 \overset{p^*}{\Rightarrow} H$.

Definition 2 Consider a possibly infinite sequence of graph productions p_i , (i=1,2,…) and a sequence of E-dependency relations (E_i, e_i^*, e_{i+1}) leading to a sequence of their E-based compositions $(p_i^* = (L_i^* \leftarrow K_i^* \rightarrow R_i^*))$ with $p_1^* = p_1$ and $p_n^* = (p_1 *_{e_1} p_2) *_{e_2} \cdots *_{e_n} p_n$

A cumulative LHS series of this sequence is the graph series L_n^* consisting of the left hand side graphs of p_n^*. Moreover, a cumulative size of series of a production sequence is the nonnegative integer series $|L_n^*|$.

Theorem 1 *A GTS=(P) terminates if for all infinite cumulative LHS sequences* (L_i^*) *of the graph productions created from the members of P, it holds that*

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

Note that we assume finite input graphs and injective matches.

Proposition: The transformation depicted in Figure 1 terminates on arbitrary finite input model.

Proof: Except for the *ProcessTransitions* and the *PopEvents* rule-pair, there are no loops in the cycle, after finishing the execution of a rule there is not a control sequence which would contain the same rule again. Therefore, we can examine the termination of the remaining rules separately.

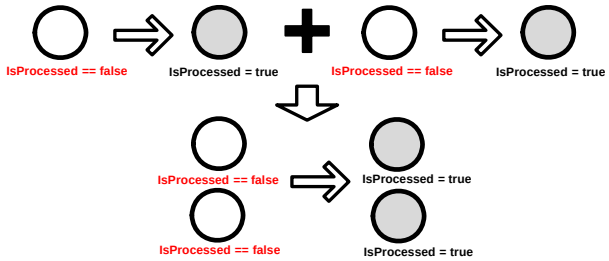
As the *CO_Skeleton* and *CheckDeadState* rules are executed only once, they do not influence the termination of the transformation.

The key of the proving method is to show, that the merging of the consecutive rule executions results in an LHS sequence that exceeds all limits. In case of the *GenerateSkeleton* rule, the processing of an *Animator* is denoted by creating an attribute reference between the *Animator* (*animator* node) and the newly created namespace declaration (*ns* node). The existence of such a reference and the connecting namespace is assigned as a negative application condition to this rule. As the *GenerateSkeleton* rule does not create or delete another attribute reference or nodes of type *Namespace*, the merging of two consecutive executions of this rule results in an LHS graph, that contains two different *Animator* nodes (the negative application condition denies to match the same *Animator* twice) (Fig. 3a). By the combination of each additional rule execution, the LHS graph will grow by another *Animator* node. Therefore, (based on Theorem 1), the rule terminates, and

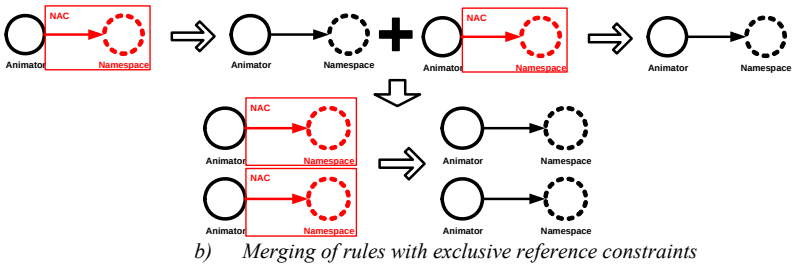
a *Namespace* node is created for each and every *Animator* node. The termination analysis of the *GenerateTC*, *ResetTransitionTag*, *Method_Buildup* and *Method_PopEvents* rules follows the same principle.

The termination of the *GenerateClass* rule can be proven on a similar basis. In this case the processing of a *Namespace* node is denoted by setting the *IsProcessed* flag to *true* on the node. The existence of this flag is assigned again to the rule as a negative application condition. Figure 3 b) depicts the combination of two consecutive *GenerateClass* rule executions. The same *Namespace* cannot be matched twice by two different executions of the same rule, thus, the LHS graph will grow by a *Namespace* and an *Animator* node (a unique *Namespace* node is created and assigned to each *Animator* node). Consequently the *GenerateClass* rule terminates as well. The termination analysis of the *Traverse, PropertyPorts, TopLevelStates, SubLevelStates, CO_InitAnim, InitEH* and *EventRoutes* rules follows the same principle.

Recall that, the *ProcessTransitions* and the exhaustive *PopEvents* rules are executed in a cycle. The termination of the *PopEvents* exhaustive rule can be proved using the presented methods independently from the processed transition; therefore, we have to prove only that the *ProcessTransition* rule can be executed for a finite number of times. As the *ProcessTransitions* rule does not match any elements created or modified by *PopEvents*, it can be examined as if it would be executed in the loop alone. This way we simplify the execution to the exhaustive way, where the termination is ensured by the application of an *IsProcessed* flag again. □



a) Merging of rules with exclusive attribute constraints



b) Merging of rules with exclusive reference constraints

Figure 3
E-concurrent production of consecutive executions of the same rule

4 Complexity Analysis of the Transformation

In this section, we analyze the algorithmic complexity of the transformation. As the rewriting phase of a rule requires constant time, we examine only the time required by the matching phase of the rules.

The transformation engine generates an execution plan [9] for each rule based on the LHS patterns. The execution plan defines exactly how the matcher traverse the host graph and in which order it matches the elements of the LHS graph. The matching order highly influences the complexity of the matcher. In this chapter, we perform complexity analysis based on the generated execution plan.

Definition 1 Let n_T mean the number of nodes of type T found in the input model.

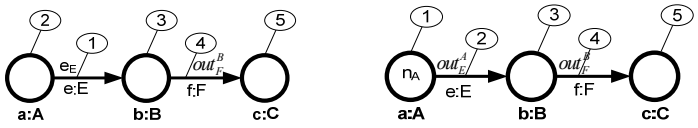
Definition 2 Let e_T mean the number of edges of type T found in the input model.

Definition 3 Let in_E^T mean the maximum number of incoming edges of type E in node of type T

Definition 4 Let out_E^T mean the maximum number of outgoing edges of type E in node of type T

Note: It is evident, that $in_E^T \leq e_T$ and $out_E^T \leq e_T$

In Figure 4, we present the notation, that describes the execution plan of a rule in a



a) matching starts at an edge *b) the matching starts at a node*

Figure 4
Execution plans

visual way. Figure 4 a) illustrates the case when the matching starts at the edge e, the nodes a and b are matched, then the edge f and finally the node c. As an edge exactly defines its endpoints, their matching cost can be considered constant. The matching of the two edges can be performed in $O(n_E * out_F^B)$. (In the worst case, we have to check every E-typed e edges, and every f edges outgoing from every possible b.) Figure 4b) illustrates the case, when the matching starts at the node a, then continues through e to b, then through f to c. The complexity of this execution plan is $O(n_A * n_E * n_F)$.

In the following we analyze each rewriting rule of the transformation, and evaluate their complexity separately and then aggregate the results.

GenerateSkeleton:

The rule matches animator nodes in an exhaustive way. The matching of a single *Animator* node can be performed in $O(n_{Animator})$ time. The rule is executed once for each *Animator*, since new *Animator*s are not created, the execution of the rule requires $O(n_{Animator}^2)$ time. Ideally, iterating through a set of nodes can be performed in $O(n)$. However, the applied transformation engine does not support the execution of a rule for each elements of a type, we can achieve the same functionality by executing the rule in the exhaustive way and marking already processed elements.

GenerateClass:

The rule matches an *Animator* and a *Namespace* node in an exhaustive way. The rule also prescribes an attribute reference from the *Animator* node towards the *Namespace*. As this reference is not part of the metamodel (it is used only during the transformation), we cannot navigate along it, and the existence of it can be verified only after matching the two nodes. Therefore, one matching can be performed in $O(n_{Animator}*n_{Namespace})$ time (Figure 5). The rule is executed once for each *Animator*, and as $n_{Animator}=n_{Namespace}$ (one *Namespace* is created for each *Animator*, and there are no *Namespace*s in the output model by default), the total execution time is $O(n_{Animator}^3)$.

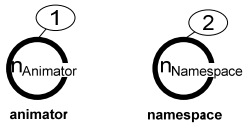


Figure 5
Execution plan for *GenerateClass*

PropertyPorts

Figure 6 illustrates the execution plan for the *PropertyPorts* rule.

Note, that there is exactly one *g* edge of type *TypeMemberContainment* incoming into *method_init*, thus it (and *class_anim*) can be matched in constant time. One match can be found in $O(n_{AnimatorPortContainment}*n_{StatementCollectionContainer})$, the rule is executed once for each *AnimatorPort*. As $n_{AnimatorPort}=n_{AnimatorPortContainment}$ and $n_{StatementCollectionContainer}=n_{Animator}$ (three methods, three *StatementCollection-Containers* are created for each animator, and the output model did not contain such an element by default) the complexity of the rule is $O(n_{AnimatorPort}^2*n_{Animator})$.

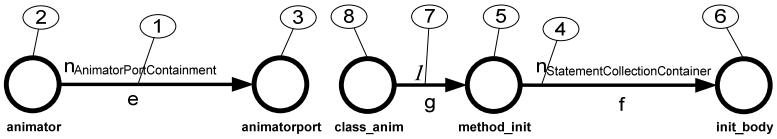


Figure 6
Execution plan for *PropertyPorts*

MethodBuildup

Figure 7 illustrates the execution plan of *MethodBuildup*. Similarly to the case of *GenerateClass*, the connection between the two nodes can be discovered only after matching both of them, thus the execution time is $O(n_{Animator} * n_{TypeDeclaration})$, this simplifies to $O(n_{Animator}^2)$ as $n_{Animator}=n_{TypeDeclaration}$.

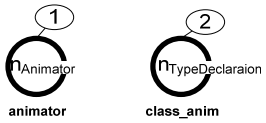


Figure 7
Execution plan for *MethodBuildup*

TopLevelStates, SubLevelStates, ProcessTransitions

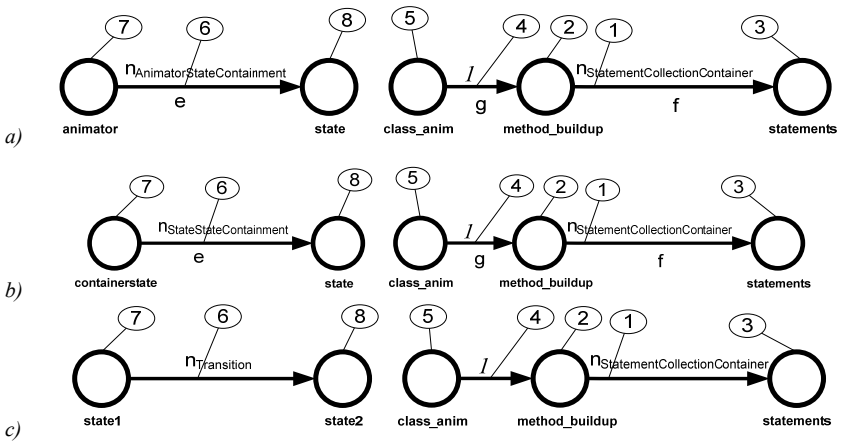


Figure 8
Execution plans for *a*) *TopLevelStates b) SubLevelStates* and c) *ProcessTransitions*

The matched patters of the *TopLevelStates*, *SubLevelStates* and *ProcessTransitions* rules (Figure 8) are similar to that of the *PropertyPorts* rule. However, because of the different element types, the execution order has been slightly changed.

One match can be found in $O(n_{AnimatorsStateContainment} * n_{StatementCollectionContainer})$ time in the top-level case, $O(n_{StateStateContainment} * n_{StatementCollectionContainer})$ in the sub-level case and $O(n_{Transition} * n_{StatementCollectionContainer})$ in case of the *ProcessTarnsitions* rule. The *TopLevelStates* and *SubLevelStates* rules are executed once for each top-level state ($n_{AnimatorStateContainment}$) respectively once for each sub-level state ($n_{StateStateContainment}$) (*ProcessTransitions* is executed once). As $O(n_{StatementCollectionContainer})=O(n_{Animator})$,

the aggregated execution times are $O(n_{AnimatorStateContainment}^{2} * n_{Animator})$, $O(n_{StateStateContainment}^{2} * n_{Animator})$ and $O(n_{Transition} * n_{Animator})$.

PopEvents

The *PopEvents* rule (Figure 9) receives the *transition* and the *class_anim* element as a parameter, thus they can be matched in *O(1)*.

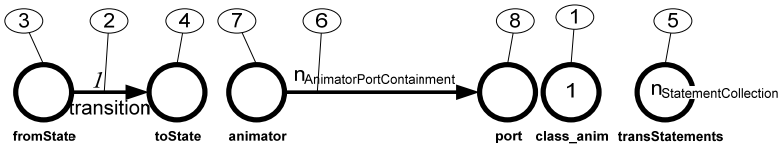


Figure 9
Execution plan of the *PopEvents* rule

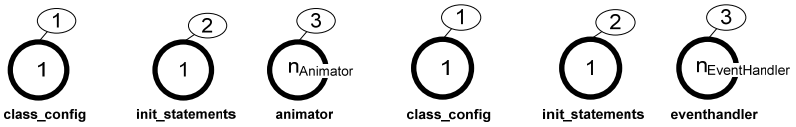
The rule is executed once for each *AnimatorPortContianment* (*Port*), the overall execution time is thus $O(n_{AnimatorPortContainment}^{2} * n_{StatementCollection})$.

The loop consisting of *ProcessTransitions* and *PopEvents* is executed for each transition, the execution time of the complete loop is $O(n_{Transition}^{2} * n_{Animator}^{2} * n_{AnimatorPort}^{2})$ ($n_{StatementCollectionContainer} = n_{StatementCollection}$ and $n_{AnimatorPortContainment} = n_{AnimatorPort}$ and $n_{StatementCollection} = n_{Animator}$).

CO_Skeleton

The rule does not match a node, but creates new elements only. It is executed once, thus it requires *O(1)* time.

CO_InitAnim, CO_InitEH



a) *CO_InitAnim* b) *CO_InitEH*

Figure 10
Execution plans for *CO_InitAnim* and *CO_InitEH*

The two rules receive the *class_config* and the *init_statements* elements as parameters from the *CO_Skeleton* rule, thus, their matching can be performed in *O(1)* (Figure 10). The remaining nodes can be matched in $O(n_{Animator})$ and $O(n_{EventHandler})$. They are executed once for each *Animator* respectively for each *EventHandler*, thus the aggregated execution times are $O(n_{Animator}^{2})$ and $O(n_{EventHandler}^{2})$.

CO_EventRoutes

The *initStatements* node is passed as a parameter to the *CO_EventRoutes* rule, therefore, it has a constant match time. The container edges and nodes for *port1* and *port2* can also be matched in constant time, as there exists exactly one container (either *Animator* or *EventHandler* for each port). Thus, the complexity of the rule is $O(n_{EventRoute})$.

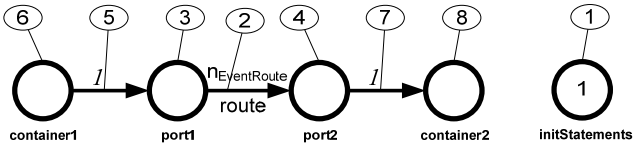


Figure 11
Execution plan for *CO_EventRoutes*

GenerateTC, ResetTransition

The execution plan of the *GenerateTC* and *ResetTransition* rules is depicted in Figure 12. Both rules are executed once for each transition, thus their execution time is $O(n_{Transition}^2)$.

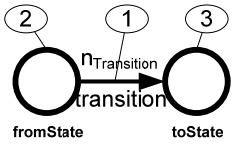


Figure 12
Execution plan for the *GenerateTC* and *ResetTransition* rules

CheckDeadState

The rule tries to match a single *State* (which was not processed by *GenerateTC*), and is executed once, therefore it finishes in $O(n_{AnimState})$.

Aggregated complexity of the transformation

To be able to summarize the runtime requirements, we have to note that $n_{AnimatorPort} \geq n_{Animator}$ (each animator contains at least one port to be able to communicate with the environment). Furthermore, $O(n_{Transition}) \geq O(n_{AnimatorStateContainment})$ *and* $O(n_{Transition}) \geq O(n_{StateStateContainment})$, because $n_{Transition} \geq ½\ n_{AnimState}$ (an input state machine consist of connected graph components nested hierarchically and each component contains at least one transition, thus in each component $n_{Transition} \geq ½\ n_{AnimState}$, which is the case of two states and one transition). So the runtime requirements with the highest exponents are: $O(n_{Transition}^2 * n_{Animator}^2 * n_{AnimatorPort}^2)$, $O(n_{EventHandler}^2)$ and $O(n_{EventRoute})$. In a typical case, the state machine is much larger, than the high level connecting model with the *Animators* and *EventHandlers*, therefore, the most significant component is the only loop in the control flow graph with the complexity of $O(n_{Transition}^2 * n_{Animator}^2 * n_{AnimatorPort}^2)$.

Conclusions

In [3] we have presented the model transformation which converts visual behavior models to source code. In this paper we have analyzed the runtime properties of the transformation. We have shown that the transformation processes only those models the states of which are topologically reachable. We have proven that the transformation terminates regardless of the input model. We have also evaluated the runtime complexity of the transformation. We have shown, that the transformation can be executed in $O(n_{Transition}^2 * n_{Animator}^2 * n_{AnimatorPort}^2)$. Although, the presented results are specific to this concrete transformation, the applied techniques can be used at the analysis of other graph-rewriting based transformations as well.

Acknowledgement

This paper has been supported – in part – by the Mobile Innovation Center.

References

[1] T. Mészáros, G. Mezei, H. Charaf. *Engineering the Dynamic Behavior of Metamodeled Languages*. Simulation, Special Issue on Multi-Paradigm Modeling, 2009

[2] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, Berlin, illustrated edition, 2006

[3] T. Mészáros, T. Levendovszky, G. Mezei. *Code Generation with the Model Transformation of Visual Behavior Models*. Proceedings of the 3rd International Workshop on Multi-Paradigm Modeling. Denver, US. 2009

[4] VMTS Team. Visual Modeling and Transformation System Website 2009. http://vmts.aut.bme.hu

[5] T. Levendovszky, T. Mészáros. *Tooling the Dynamic Behavior Models of Graphical DSLs*. In In proceedings of the 13th International Conference on Human-Computer Interaction. San Diego, USA, July 2009

[6] B. P. Zeigler, T. G. Kim, H. Praehofer. *Theory of Modeling and Simulation*. Academic Press, Inc., Orlando, FL, USA, 2000

[7] Microsoft CodeDOM website: http://msdn.microsoft.com/en-us/library/650ax5cx.aspx

[8] T. Levendovszky, U. Prange, H. Ehrig, *Termination Criteria for DPO Transformations with Injective Matches*, ENTCS 175. Vol. 4, pp. 87-100, 2007

[9] Batz, G. V., Kroll, M., Geiß, R.: *A First Experimental Evaluation of Search Plan Driven Graph Pattern Matching*, Proceedings of the 3rd Intl. Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE '07), Springer, 2008