Towards a Step Semantics for Story-Driven Modelling

Géza Kulcsár
Technische Universität Darmstadt
geza.kulcsar@es.tu-darmstadt.de

Anthony Anjorin
Chalmers | University of Gothenburg
anjorin@chalmers.se

Graph Transformation (GraTra) provides a formal, declarative means of specifying model transformation. In practice, GraTra rule applications are often *programmed* via an additional language with which the order of rule applications can be suitably controlled.

Story-Driven Modelling (SDM) is a dialect of programmed GraTra, originally developed as part of the Fujaba CASE tool suite. Using an intuitive, UML-inspired visual syntax, SDM provides usual imperative control flow constructs such as *sequences*, *conditionals* and *loops* that are fairly simple, but whose interaction with individual GraTra rules is nonetheless non-trivial. In this paper, we present the first results of our ongoing work towards providing a formal *step* semantics for SDM, which focuses on the *execution* of an SDM specification.

1 Introduction and Motivation

Graph Transformation (GraTra) provides a formal, declarative means of specifying how graph-like structures can be manipulated and changed. This is useful in numerous application domains including the specification of model transformations, a central task in Model-Driven Engineering (MDE) [18]. Although GraTra rules can simply be applied as long and as often as they are applicable, in practice more control over rule application is often required for complex transformations. This can be achieved in many ways including grouping and ordering rules in layers [15], or by providing a dedicated additional language, with which GraTra rules can be suitably "programmed" [1, 8, 19, 22].

Story-Driven Modelling (SDM) [22] is a dialect of programmed GraTra introduced by Zündorf and released as part of the Fujaba CASE tool suite. Originally inspired by PROGRES [19], SDM combines object-oriented concepts and GraTra rules in a single, formal and integrated language, with an intuitive, UML-inspired visual concrete syntax. To provide control over rule application, SDM introduces usual imperative control flow constructs including sequences and conditionals. These constructs are relatively simple, but their interaction with the embedded, normal GraTra rules is nonetheless non-trivial and involves a series of careful design decisions.

The current formalisation of SDM semantics by Zündorf [22] is based on pairs of graphs representing input-output pairs of GraTra rule applications, first defined for single GraTra rules and then extended to combinations of rule applications, programmed by an iterative control flow. This denotational style of formalisation is useful to determine the correctness of results of rule applications. Nevertheless, although Zündorf mentions typical optimization techniques for SDMs such as binding variable names to input model elements to reduce the search space of subsequent GraTra rule applications, the exact semantics regarding the combination of conditionals and such bindings remains largely unspecified. Such a "high-level" semantics can be advantageous from a specification point of view, but for GraTra researchers and tool developers aiming to provide, e.g., static analyses for SDM specifications, a more detailed *execution-based* semantics is also required as the set of output models is not available for every possible input.

Alexander Heußner, Aleks Kissinger, Anton Wijs (Eds.):
2nd Graphs as Models workshop (GaM 2016)
EPTCS 231, 2016, pp. 42–56, doi:10.4204/EPTCS.231.4

In this paper, therefore, we propose to complement [22] with an operational semantics for SDM that focuses on the control flow and execution of an SDM specification. To this end, we define a *step semantics* whose step concept relies on GraTra rule applications, ensuring that the resulting formalisation is particularly comprehensible for our target audience: researchers and tool developers in the GraTra community. We demonstrate this on a fundamental set of SDM constructs, which we shall refer to as *basic SDM*, consisting of single rule applications, sequences, conditionals and head-controlled loops. We define the set of syntactically valid combinations of these constructs by means of a graph grammar that generates exactly the valid control flows of basic SDM. Based on this, we provide a further set of GraTra rules whose application yields a step concept for our semantics.

2 Related Work

The initial, and to the best of our knowledge, only formal SDM specification is provided by Zündorf in [22] and is based on the semantics for PROGRES [19]. Technical reports on SDM such as [5] go a long way towards clarifying SDM syntax and semantics and are probably more readable and accessible to end users than any formal specification. For researchers and tool developers, however, whose aim is to provide new tool support or extend the language, such semi-formal documentation is not precise enough, especially concerning control flow constructs and their interaction with embedded GraTra rules.

Formalisation techniques. Dynamic Meta-Modelling (DMM) is introduced by [7] as a visual approach to specify the dynamic behaviour of (visual) languages. It is closely related to and can be viewed as a generalisation of Graphical Operational Semantics (GOS) [4]. Both approaches are inspired by Structural Operational Semantics (SOS) [16] and they employ GraTra rules to induce a transition system representing the specified semantics. These techniques have been applied in [10, 11] to formalise UML collaboration diagrams and the pattern language used in Fujaba. In this paper, we focus more on the control flow constructs of SDM and less on the embedded patterns, which we restrict to normal GraTra rules without integrating advanced object-oriented constructs.

SDM specifications show some resemblance to UML activity diagrams, for which an executable step semantics has been proposed in [13], inspired by a similar approach to a statechart semantics [9]. The DMM approach has been applied to the semantics of activity diagrams in [20], where the semantics of Petri-nets is taken as an initial point for comparison. The authors conclude that despite superficial similarities, the intention of higher-level UML constructs notably diverges from a Petri-net semantics. Although these approaches are similar to our token and step concepts, nondeterministic pattern matching, failure as a branching condition, as well as variable binding are not considered directly in the case of activity diagrams (as Petri-nets are also missing analogous concepts). Moreover, the formalisms of [9, 13] do not involve GraTra concepts (even if an encoding into GraTra would be possible). An analysis of activity diagrams based on GraTra has been proposed in [12] involving an object flow concept and a rule-based semantics. Nevertheless, in contrast to our approach concentrating on execution, the proposed semantics focuses on the resulting graphs and the properties of the involved GraTra rules.

As demonstrated by [22], it is possible to provide a denotational semantics for SDM. This can be achieved by mapping an SDM specification to the set of all pairs of possible input and output graphs. While this can be viewed as an elegant and compact formalisation, we believe that a complementary operational approach that uses GraTra along the lines of [4, 7] is crucial for further research and development on SDM. Similar arguments apply to other approaches including, for example, abstract state machines (cf. [3] for a comparative study).

Alternatives to SDM. Existing and established GraTra tools take different approaches to enabling programmed GraTra. PROGRES [19] and Viatra [2], similarly to SDM, provide a dedicated control flow language, rich and complex enough to completely replace any host language. Approaches such as EMF-IncQuery [21] and Groove [8] have chosen to concentrate fully on providing a rich pattern language (in the case of EMF-IncQuery even without side effects) for GraTra. These approaches rely on a host language (Java, Prolog, Xtend) for cases where users wish to additionally control GraTra rules. Approaches such as Henshin [1] are in-between, focussing primarily on patterns but still providing a relatively simple and high-level language such as *transformation units* [14] to program individual GraTra rules.

3 Preliminaries

In this section, we recapitulate the basics of GraTra based on [6, 17], and briefly recall the motivation and background of SDM, introducing standard terminology required for the paper with a simple example.

Graph transformation. Graphs are ubiquitously used for capturing different structures in various domains as they are mathematically tractable but still comprehensible and can be easily visualised. In this paper, a graph is defined as a tuple $G = (V_G, E_G, src_G : E_G \to V_G, trg_G : E_G \to V_G)$, V_G being the set of *nodes*, E_G the set of *edges*, src_G and trg_G the source and target functions, assigning edges to their source and target nodes, respectively. Given graphs G and H, a *graph morphism* $m : G \to H$ is a pair of functions $m_V : V_G \to V_H$ and $m_E : E_G \to E_H$ preserving connectivity, i.e., $m_V(src_G(e_G)) = src_H(m_E(e_G))$ for each $e_G \in E_G$ and analogously for target functions. A morphism is *injective* if both functions are injective.

Type graphs are often used to enhance graphs with additional structure by assigning types to nodes and edges. A type graph is a distinguished graph TG. A *typed graph (G, t) over TG* is a graph G together with a graph morphism $t : G \to TG$. A *typed graph morphism* is a morphism $m : (G,t) \to (H,t')$ preserving typing, i.e., $t'_V(m_V(v_G)) = t_V(v_G)$ and $t'_E(m_E(e_G)) = t_E(e_G)$ for each $v_G \in V_G$ and $e_G \in E_G$.

In this paper, we use the Single Pushout (SPO) approach to GraTra, which is formalised in a categorical framework (cf. [17] for formal details). The SPO approach to GraTra utilizes the notion of a *partial graph morphism*. A partial graph morphism $p : G \rightharpoonup H$ is a graph morphism $dom(p) \to H$, where $dom(p)$ is a sub-graph of G. The definition can be extended to the notion of *partial typed graph morphism* analogously to typed graph morphisms.

A *graph transformation rule* (or simply *rule*) is a partial graph morphism $r : L \rightharpoonup R$ where the graphs L and R are also called the *left-hand side* and the *right-hand side* of the rule r, respectively. A *match* of this rule in a graph G is a total injective morphism $m : L \to G$. The set of valid matches can be further constrained by *negative application conditions* (NAC). A NAC is a morphism $n : N \to L$ which forbids a match m if there is an image of N in G via m. The *application of a rule r* at match m in graph G, denoted as $G \overset{r,m}{\Longrightarrow} H$ (or simply $G \overset{r}{\Longrightarrow} H$ if the match is not relevant), transforms G to the graph H as seen in the diagram to the right, also called a *pushout* (PO) diagram. Intuitively, the pushout diagram defines rule application as follows: the images $m(L \setminus dom(r))$ of elements not in the domain of r are deleted from G and the elements $m'(R \setminus r(L))$ not in the image of L in R are added to G, yielding the graph H. Again, the generalisation to the typed case is straightforward. Practically, the side effect of using SPO rules (compared to the Double Pushout approach) is that if nodes are deleted by a rule application, their incident edges are also deleted, even if they are not explicitly in the deletion sub-graph derived from r. A graph grammar $GG = (G_S, \mathsf{R})$ consists of a *start graph* G_S and a set of graph transformation rules R. The *language generated by GG* contains those and only those graphs which can be reached via a rule application or a chain of rule applications from the start graph, i.e., is a set of graphs $L_{GG} = \{H \mid \exists G_S \overset{r_1}{\Longrightarrow} ... \overset{r_n}{\Longrightarrow} H\}$ where $r_1, ..., r_n \in \mathsf{R}$ and $n \geq 0$.

$$\begin{array}{ccc} L & \xrightarrow{r} & R \\ {\scriptstyle m}\downarrow & (PO) & \downarrow{\scriptstyle m'} \\ G & \xrightarrow{r'} & H \end{array}$$

Story-Driven Modelling. Story-Driven Modelling (SDM) has been originally proposed as a dialect of programmed GraTra, which enhances declarative GraTra rules with a control flow, describing the application order of the rules and including basic imperative control structures such as conditionals (if-then-else) and for-each loops.

In this paper, an *SDM specification*, referred to as a *story diagram*, consists of a *control flow graph* and a mapping of control flow nodes to *story patterns*. A control flow graph is a graph consisting of control flow nodes and edges defining the imperative control structure. A story pattern is a typed GraTra rule as defined previously. We say that a story pattern is contained in a control flow node if the latter is mapped to the former. In this case, the control flow node is referred to as a *story node*.

SDM execution starts at a special control flow node, referred to as a *start node* and continues along the edges of the control flow graph according to the semantic rules of SDM, which we shall define in this paper. *Story node execution* is formalized as an attempt to apply its contained story pattern (which is a GraTra rule) on the underlying input graph, which can either be successful (if the rule is applicable) or fail. SDM execution terminates if it arrives at a special control flow node referred to as a *stop node*, or if the execution of some story pattern fails unexpectedly (what this exactly means will be formalised by our semantic rules). A story diagram is thus a program that takes as input a graph G and outputs another graph G', which can be derived from G via a chain of GraTra rule applications $G \overset{r_1,m_1}{\Longrightarrow} G_1 \overset{r_2,m_2}{\Longrightarrow} G_2 \Longrightarrow \ldots \Longrightarrow G'$ that is compatible with the specified control flow graph of the story diagram.

Example. An *ordered list* consists of nodes (objects) linked to each other via single *next* edges (links), where the next reference of the last object in the list is undefined (null). Figure 1 depicts the visual specification of a simple operation called *deleteNextObject()* as a story diagram, which deletes a node from a list while retaining the linked structure of the list. It also guarantees that the current list object is not the last one in the list after execution.

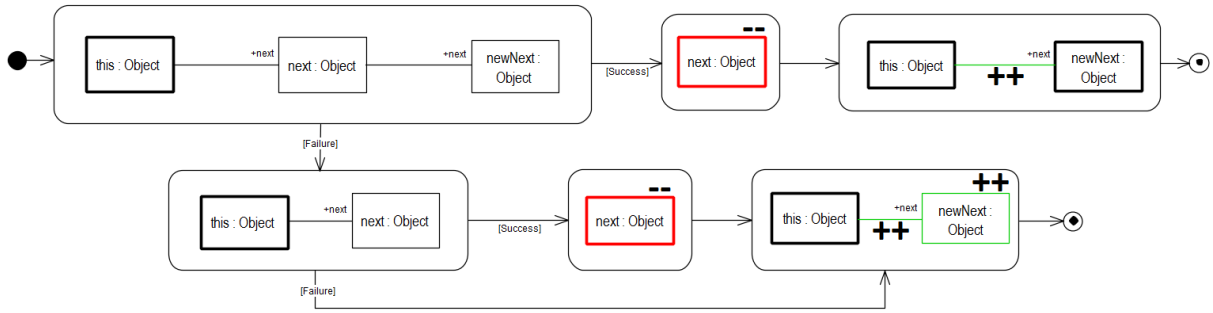


Figure 1: Example SDM method specification: *deleteNextObject()*

A story pattern $r : L \rightharpoonup R$ is represented compactly by merging L and R as follows: black elements constitute $L \cap R$ and are retained, red elements with a "--" markup constitute $L \setminus R$ and are deleted, while green elements with a "++" markup constitute $R \setminus L$ and are created by the rule. Finally, note that the first story pattern of *deleteNextObject()* has a partial match already consisting of a node *this* that is mapped to the object on which *deleteNextObject()* is invoked, in the usual object-oriented fashion. Such nodes are referred to as *bound* variables and are denoted with a bold border.

The story diagram consists of a *start node* (filled circle, on the left), two *stop nodes* (circle with a dot, on the right) and *story nodes* in between. The execution of the story diagram starts at the start node, follows the first control flow edge (depicted as an arrow) leading to a story node which is a conditional. It has two outgoing arrows, one labelled with *Success* and one with *Failure*.

If the story pattern in the conditional node is successfully matched, the control is passed over along the *Success* arrow, otherwise along *Failure*. Note that the two branches of this conditional never meet again and they end in different stop nodes. The *Failure* branch (bottom row) starts with a conditional again (thus demonstrating a nested conditional), but is different from the first one as the branches join again in the story node just before the bottom stop node. If execution arrives at the second conditional, it means that the current object is followed by at most one object. In the second conditional, we check if at least one *next* object exists. Note that it is allowed to have the variable `next` as unbound again as we know that the binding in the first conditional was unsuccessful. If we manage to match at least this smaller structure, then we follow the *Success* arrow, deleting `next` and proceeding to the last story node. If `this` has no *next* object at all, we skip to the last story node along *Failure*. Independent of the executed branch, `this` now stands without a follower—which we create in the last control flow node.

Denotational SDM semantics. In [22], Zündorf defined an SDM semantics for the Fujaba tool suite, inspired by and partly based on the PROGRES approach [19]. The basic element of this semantics is a single story node, which serves as a basis for the semantics of compound structures (sequences, conditionals, loops). In this denotational approach, the semantics of story node execution is given in terms of pattern matching in an input graph. Particularly, the semantics $Sem(n)$ of a story node n containing GraTra rule r is a set of graph pairs $Sem(n) = \{(G,H) \mid G \stackrel{r}{\Longrightarrow} H\}$. Thus, the semantics of a single step is based on GraTra rule applications which corresponds to our approach. Note that in the present paper, we do not explicitly specify the semantics of a single pattern matching step in terms of attaching input objects to the variables. In this respect, our technique is analogous to the one in [22] where it has been thoroughly specified. Through the denotational semantic style, SDM remains closely connected to GraTra theory, the resulting semantic domain is simple, and can be effectively used for testing an SDM method against, e.g., pairs of input-output models in a model transformation scenario.

In some cases, however, this approach might not capture the required level of detail. For instance, in applications of programmed GraTra in general and SDM in particular, the intended behaviour of an SDM method is mostly that in case a rule cannot be applied, the execution terminates immediately and optionally, the user gets informed about which pattern failed to match. The handling of termination requires an operational viewpoint and is not captured by the denotational approach.

Particularly, as a consequence and crucial difference, the semantics according to [22] of story node n with rule r and input graph G is $Sem(n) = \{(G,G)\}$ if r is not applicable to G. In a sequential composition of story nodes starting with story node n, therefore, this means that the input graph remains unchanged after n, but leaves open the possibility of executing further story nodes in the sequence after n.

Furthermore, in the case of conditionals where the branches join at some later point, the denotational semantics according to [22] does not distinguish between the branches regarding rule applications. This distinction, however, is essential to precisely define the concept of *variable bindings*, i.e., matches of one story node that can be extended by subsequent story nodes. This, again, represents an operational semantic viewpoint and can be suitably handled by using *scopes* as introduced in Section 4. For example, one might choose to allow bound variables within a branch but not after the branches join, regardless of if the conditional node was successful or not. This distinction and corresponding degrees of freedom in fixing such detailed design choices is not possible in the denotational semantics.

The aforementioned constructs (sequential composition, conditionals, head-controlled loops) are all part of our basic SDM language and are described in the following Section 4, where we also recall the denotational semantics of each construct and discuss relevant differences in more detail.

4 SDM Syntax and Semantics

In this section, we present two typed graph grammars, one for the syntax and one for the semantics of basic SDM. We specify the syntax graph grammar in a compact form (Sec. 4.1), as the primary focus of our paper is on the semantics. The syntax grammar generates syntactically valid control flows but does not specify the mapping of story patterns to story nodes. For the story patterns, we assume that they are properly typed over the input type graph and that all bindings are marked such that each bound variable has a previous occurrence in the story diagram where it was matched.

In Sections 4.2–4.6, we define the semantics of the basic SDM constructs on the basis of a type graph (Sec. 4.2) also via standard GraTra rules. We characterize our approach as a *step semantics* in order to emphasize its focus on operational steps of the execution and to distinguish it from a fully-fledged SOS (where other concepts such as observable actions, labelling and equivalence mostly play an important role as well). It is important to define(i) how a *state* of the system (here, a state of the story diagram execution) is characterized and (ii) which operations are explicitly included in the semantics, i.e., what is the notion of an observable *step* in the semantics as expressed by the semantic rules. These notions are also covered in Section 4.2. Afterwards, we present the semantic rules by incrementally adding the following constructs to the already specified language fragment: initialization (Section 4.3), pattern matching (Section 4.4), sequential chains of control flow nodes (Section 4.5), as well as conditional control flow nodes and head-controlled loops (Section 4.6).

4.1 Control Flow Syntax

In this section, we define a graph grammar $GG_{Syntax} = (G_s, \mathsf{R}_{Syntax})$, which generates the set of all valid control flow graphs of basic SDM. According to this specification technique, a control flow is considered as a graph (also called *control flow graph*) that represents the imperative control structure of a story diagram (without considering story patterns). We define GG_{Syntax} by specifying a *start graph* G_s (considered as the minimal valid control flow) and a set R_{Syntax} of GraTra rules which specify the possible expansions of the start graph resulting in valid control flow graphs. To guarantee the validity of the resulting control flow graphs, we specify the rules in a way that every possible rule application to a valid control flow graph yields also a valid one.

Both the start graph and the GraTra rules are typed over the type graph TG_{Syntax}, depicted in Figure 2. The types `CFNode`, `StopNode` and `StartNode` have a common parent type, `AbstractNode`, from which they inherit all the edge types. These edge types are: *next* for sequences of control flow nodes, as well as *success* and *failure* for the two branches of conditional nodes. All the edge types are represented as self-edges of the abstract type `AbstractNode` as such edges in a control flow run from one control flow node to another. Introducing this abstract node enables a simplified, compact syntax grammar.

Figure 3 depicts the start graph G_s which is, consequently, the minimal valid control flow graph. G_s consists of a sequence of a start node, a story node, and a stop node. Table 1 shows the rules in R_{Syntax} for the success case. For all rows apart from the first and the second, there are additional, completely symmetric rules for each one depicted, formed by substituting failure for success and vice versa, as success and failure are dual concepts. We also distinguish between different branch configurations as seen, e.g., in the third and fourth rows. We merge these two possibilities into a single row in the last two cases due to space restrictions. The rules are defined according to the standard GraTra notions of left-hand side (L) and right-hand side (R) graphs as introduced in Section 3 – nevertheless, as each rule shares a single common L graph, we do not include it for each rule but only specify it once in Figure 4. For the sake of clarity, we additionally use the SDM visualization of deletion and creation and mark the

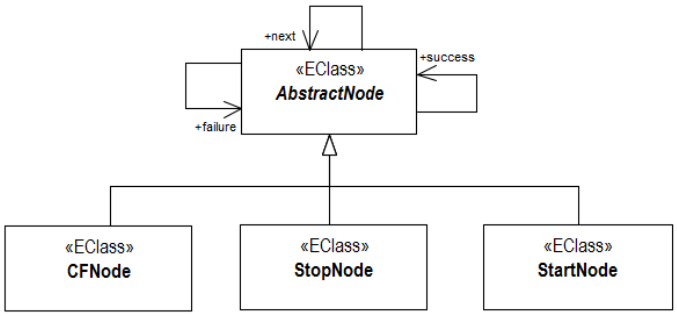


Figure 2: Type graph TG_{Syntax} of the control flow syntax graph grammar GG_{Syntax}

elements to be deleted in L with red (resp. `--`) and the elements to be created in R with green (resp. `++`). The rules depicted in Table 1 only give rise to well-formed, i.e., valid control flow graphs. We start from a minimal, but complete control flow graph G_s. In each step, we either insert a simple sequential story node or introduce a conditional, possibly with a loop. Conditionals without loops are handled in rows 2-4 and *while* loops in rows 5-7. Considering also the symmetric rules not depicted here, R_{Syntax} consists of 16 rules: 1 for sequential composition, 1 for joining conditional branches, 4 for non-joining conditional branches and 10 for *while* loops.

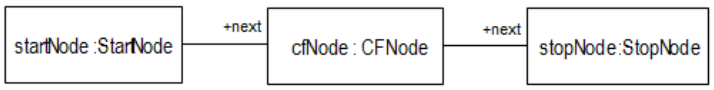


Figure 3: Start graph G_s of GG_{Syntax}

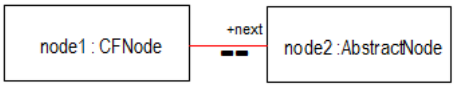


Figure 4: The left-hand side L shared by all rules in Table 1

4.2 Concept of the Semantics

Figure 5 depicts the type graph for story diagrams that we use to specify our semantics. The central elements of a story diagram are story nodes, modelled by the type `CFNode`. The class `CFNode` is also responsible for connecting the syntactic (Figure 2) and the semantic type graphs (being a concept in both). `CFNodes` might be ordered into linked sequences (defining their execution order) using the next reference of the `AbstractNode` type (Figure 2). `CFNodes` are contained in `Scopes`, as expressed by the containment arrow (a bidirectional arrow with a diamond on the container side). Basically, scopes represent the different blocks of the control flow and are ordered hierarchically. For instance, the scopes of conditional branches are sub-scopes of the *parent scope*, where the branches originate from. There is always exactly one *root scope* to which the first story node belongs.

A `Scope` contains control flow variables, which represent model elements. The correlation of these three elements (left of the dashed line) can be summarized as: the `CFVariables` in a given `Scope` are those variables which are valid in the `CFNodes` of the same scope.

On the right-hand side of the dashed line, we can see the dynamic constructs used by the semantic rules. In each story diagram being executed, there is a single `PositionToken` that is attached to the current `CFNode` to be executed. `VariableBindings` represent the local variables of a scope, whose names are bound to an object in the input graph. This attachment is represented by a `VariableBinding` and a `Variable`, the latter representing our interface to the nodes in the input graph, which are not explicitly visible from the semantics. Thus, this triple (together with a `CFVariable`) captures a mapping *loc* of local variables to input graph nodes $loc : CFVariable \rightarrow Variable$.

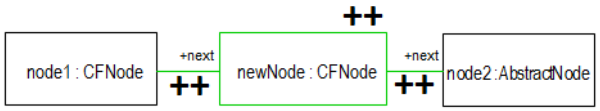
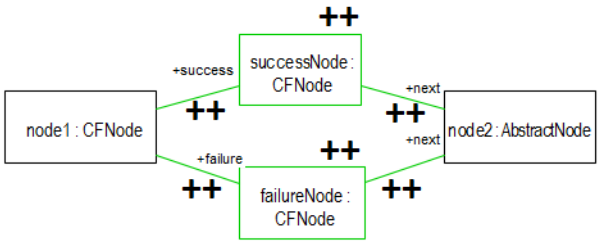
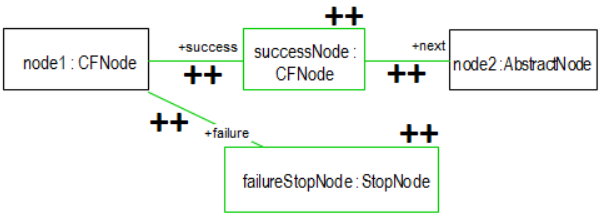
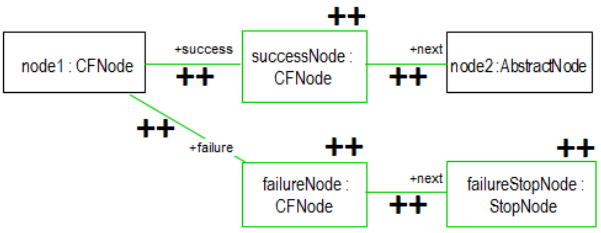
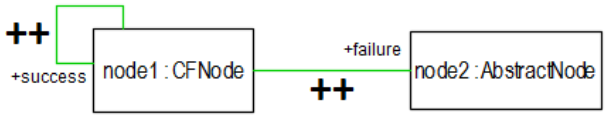
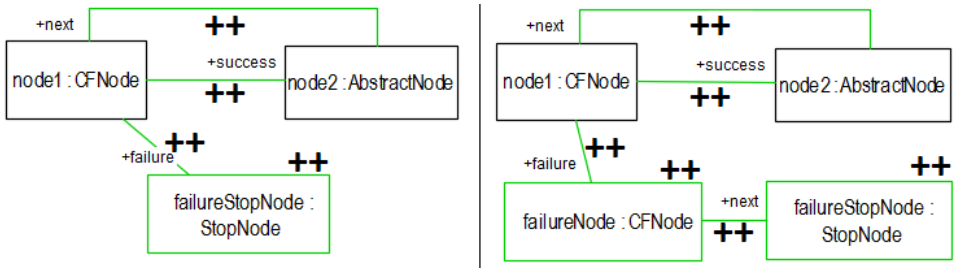
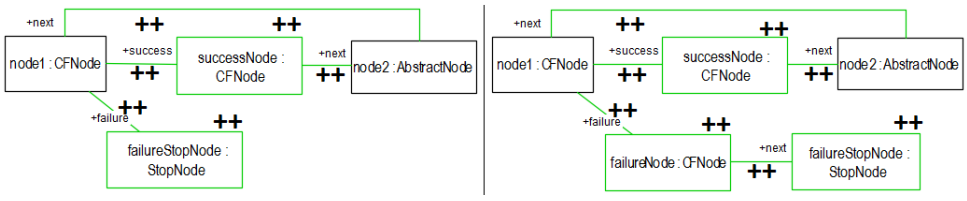
Description	Right-hand side of syntactic rule (common left-hand side in Figure 4)
Inserting a node sequentially	
Conditional with joining branches	
Conditional with opening up a *Failure* branch containing only a stop node	
Conditional with opening up a *Failure* branch containing a control flow node and a stop node	
Direct loop along *Success*	
Loop consisting of two nodes along *Success* (both *Failure* variants)	
Loop consisting of three nodes along *Success* (both *Failure* variants)	

Table 1: Graph grammar for control flow syntax

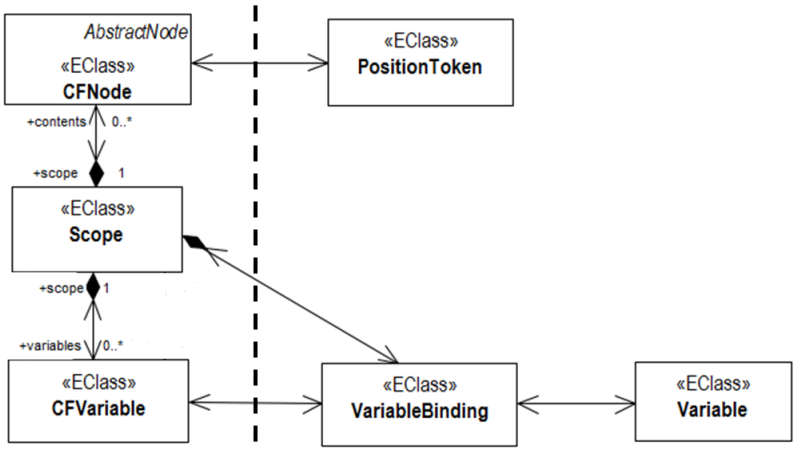


Figure 5: Type graph for SDM states

A graph typed according to Figure 5, together with a control flow graph and story patterns constitute a valid *state* of a story diagram execution. Such a valid state, together with the actual input graph (which might have already undergone some manipulations), constitutes a configuration of the story diagram execution. To avoid confusion, we stick to this terminology and refer to the semantic model of the story diagram execution as *state*, to the input graph under manipulation further as model or input model, and to the pair of state and model as *configuration*. A *step* of the semantics is a transition from one state to another which constitutes: (i) modifying the input model according to the pattern in the actual control flow node, (ii) identifying the next control flow node and shifting the token there or terminating and (iii) optionally modifying other state elements such as bindings.

How such a step is actually specified is the main concern of the following sections. It is important to note that there is no one-to-one correspondence between GraTra rules and semantic steps. In our semantics, a semantic step consists of the application of multiple GraTra rules. All possible applications of these rules in a state constitute together one semantic step leading to the next state. As the possible applications are independent of each, their joint application and the next state are deterministic.

Technical remarks. The two type graphs presented (syntax and semantics) can be considered as a single one joined by the type `CFNode`. We thus use types from both for specifying the rules. We assume that the scopes and control flow variables of a story diagram have been analysed and created before executing the semantics (hence scopes can be used as existing elements in the rules). This analysis and creation of scopes is possible on the control flow graph and the patterns in the control flow nodes without additional information; the corresponding algorithm to accomplish this is, however, out of scope.

4.3 Initialization

In this section, we define *initialization* which, given a control flow graph with story patterns, creates the initial state for the semantics. In an initial state, the position token is set to the first control flow node after the start node and default bindings are added to the root scope. The resulting state, together with an unmodified input graph considered as user input, constitute the initial configuration for the execution. We also interpret the necessary preparations of creating the fixed `this` variable as part of the initialization.

Semantics. Figure 6 depicts the initialization semantics consisting of two rules. In the left rule (applied exactly once), the position token is created and set on the first control flow node (the one directly con-

nected to the start node in the control flow graph). In the right rule, for the initial variables in the root scope (in basic SDM, only for `this`), the bindings are created and also attached to the root scope. Note that initialization does not have an analogue in the higher-level denotational approach [22] as there is no need to handle these operational details.

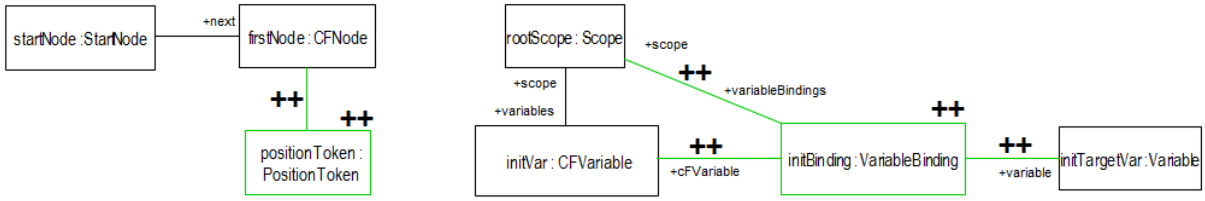


Figure 6: Initialization

4.4 Pattern Matching

In this section, we consider a single control flow node with a story pattern. When the execution of the story diagram proceeds to a story node, its story pattern is executed.

Semantics. The semantics of pattern matching requires special handling and follows the same principles for our semantics and the denotational approach [22], as both rely on GraTra rule applications. Although our step semantics focuses on the control flow, the actual model transformation is still performed by the story patterns in the control flow nodes. As soon the position token is set to point to a new control flow node, the matching of the pattern contained in the control flow node is performed. There are two cases possible: a match can be found and model objects become bound via the variable names in the pattern, or no match can be found. In the former case, new variable bindings are created (unbound black and green elements) and some are deleted (red elements); in the latter case, a failure is reported.

We abstract away the actual matching and rule application process (which is performed according to the SPO approach), as from a semantic point of view, we are only interested in the result, i.e., which values (model objects) are attached to control flow variables. Moreover, we do not explicitly include the modifications which the input model undergoes in case of a successful match; we consider this step as a part of the pattern matching process and we assume that we get information on which variables are newly bound and which have to be deleted. To capture the effects of pattern matching in our semantics, we introduce a new type `PatternInvocation`. It is contained by the control flow node and represents the result of the pattern matching process regarding variables, i.e., which unbound variables become bound and which bindings get deleted.

Example. Figure 7 depicts two subsequent control flow nodes from Figure 1 on the left hand side, and a corresponding representation of how the results of the rule applications appear in the semantic model. The effect of the deletion in the first control flow node is shown in the middle, where the corresponding variable is added to the pattern invocation `pattern1` with a *destructedVariables* reference. Creating a new object in the second control flow node is shown on the right, where the corresponding variable is added to `pattern2` with a *constructedVariables* reference. Note that deletion and creation might also be combined in a pattern and there might be multiple variables to delete/create.

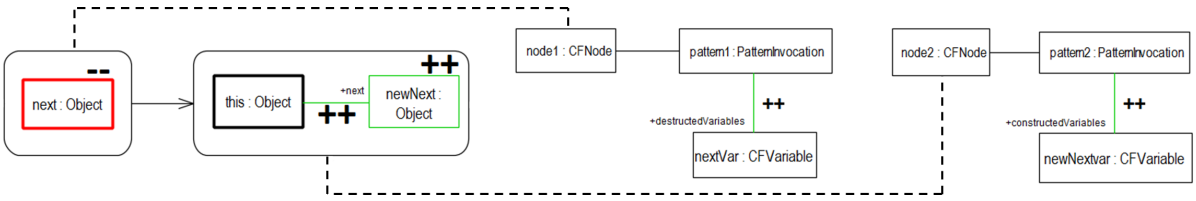


Figure 7: Result of pattern executions

4.5 Sequential Chain of Story Nodes

In this section, we consider simple, sequential chains of control flow nodes, which are executed in a strict order and where it is not allowed to create loops in the chain. The chain might be arbitrarily long, but it is strictly sequential, i.e., it contains no loops. In a complete, valid story diagram where no other construct is allowed (i.e., a specification in the SDM sub-language that consists only of sequential chains), the chain starts with a start node and ends with a stop node. From a semantic point of view, we regard the stop node as a control flow node having no pattern and no outgoing arrow. When the position token arrives at such a node, no rule fires any more and the execution terminates.

Semantics. Denotationally [22], the semantics of two subsequent control flow nodes n_1 and n_2 with rules r_1 and r_2, respectively, is given in terms of input and output graphs as $Sem(n_1,n_2) = \{(G,H) | \exists G' : G \overset{r_1}{\Longrightarrow} G' \overset{r_2}{\Longrightarrow} H\}$ (longer chains are defined by induction). The most important difference compared to a more detailed step semantics is that bindings are not handled explicitly. The semantic definition of sequential composition only requires that there is *an* intermediate graph between the two applications, but does not assign the resulting pairs to their respective matches. As mentioned before, this might be an appropriate level of abstraction for testing and verification tasks but might be too high-level for tool developers.

Regarding our step semantics, subsequent control flow nodes in a sequential chain all belong to the same scope. A step in our semantics corresponds to a shift of the position token from the current control flow node to the next one if pattern matching was successful. We also have to handle the case where no match for the pattern can be found. In basic SDM, this case is handled by terminating with an error. We model this error by detaching the position token from the control flow to clearly distinguish between normal and erroneous states, as otherwise, the position token is always attached to a control flow node. We do not show graphically the fairly trivial rule of deleting the edge between the position token and the actual control flow node. The step for handling a match is depicted in Figure 8. Note that the binding update rules (middle and right ones) are executed for every relevant variable correspondingly marked in the pattern invocation step.

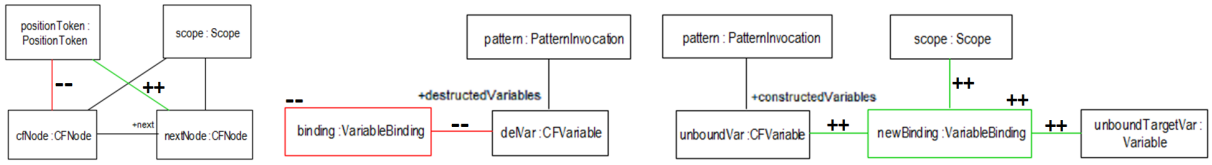


Figure 8: *seqSuccess*: token shift and binding updates after successful pattern matching

Example. Considering the two subsequent control flow nodes on the left of Figure 7, after the first node has been executed, the position token is passed to the second one and the binding of nextVar gets deleted, corresponding to the middle rule in Figure 8.

4.6 Conditional Control Flow Nodes and Head-Controlled Loops

In this section, we extend our language by a conditional construct. *Conditional control flow nodes* have exactly two outgoing arrows leading to other control flow nodes, one labelled with *Success*, and the other with *Failure*. In the case of a successful matching process, the *Success* edge will be followed, otherwise the *Failure* edge is taken. It is important to note that the notion of conditionals in SDM is not completely analogous to the traditional conditional statements of imperative programming: choosing the next block to be executed does not depend on evaluating a boolean expression over already bound variables, but depends rather on the result of a pattern matching process, during which new variables might become bound and existing bindings deleted.

A special form of a conditional construct is when one of the branches returns to the conditional node, resulting in a head-controlled loop, also simply known as a *while* loop. After the recurring branch has been executed, the control is passed to the conditional node again, taking the possibly changed model as input; the conditional is then re-evaluated independently of the previous iteration, just as in a traditional imperative language. Another special case that must be explicitly handled is when two branches *join* again at a control flow node. We shall discuss different strategies of handling variable bindings.

Semantics. In the denotational approach [22], the semantics of a conditional *if* n_1 *then* n_2 *else* n_3 (where n_1, n_2, n_3 are control flow nodes with rules r_1, r_2, r_3, respectively) is: $Sem_{if}(n_1,n_2,n_3) = Sem(n_1,n_2)$ if $\exists G' : G \overset{r_1}{\Longrightarrow} G'$ with input G, and $Sem(n_1,n_3)$ otherwise. Note that it is assumed in this semantics that SDM execution always continues even after a failing match. This does not allow for the distinction that in the case of sequential nodes, a failing match should result in abrupt termination while in the case of a conditional, execution continues along the *Failure* branch. Our approach handles both cases explicitly.

Denotationally, the semantics of a *while* loop: *while* n_1 *do* n_2, with control flow nodes n_1, n_2, rules r_1, r_2, and input graph G is given as $Sem_{while}(n_1,n_2) = Sem(n_1,n_2,$ *while* n_1 *do* $n_2)$ if $\exists G' : G \overset{r_1}{\Longrightarrow} G'$ and $Sem(n_1)$ otherwise. Note that this specification only handles looping along success, whereas our step-based approach can also handle a "negative loop" where the recurring branch goes along failure.

Regarding our step semantics, in order to define the relations between the conditional node and subsequent nodes, we make use of an additional reference type between two control flow nodes: *success* and *failure* (as shown in Figure 2). A more significant difference is that in the case of conditionals, we now utilize different scopes. Both branches of a conditional have their own separate scopes, whose *parent scope* is the scope of the conditional node. The branch scopes get dynamically created as execution proceeds to them for the purpose of handling *while* loops. Deciding which bindings they should inherit and pass on to their parent scope is an interesting design decision, especially if the branches join at some later point; we elaborate on this subject to demonstrate the capabilities of our approach as well as to emphasize possible alternative semantics and ambiguities.

As soon as the execution of a branch of a conditional is completed, i.e., execution of a node that is reachable from both branches of the conditional commences, we revert to the parent scope (the scope of the conditional node). In the conditional node itself or somewhere in one of the branches, we might, however, have updated some bindings. The case of variables that are newly bound (or created) *in only one branch* is clear: such bindings must lose their validity when the branch scope is exited (as is the case in standard imperative languages), and are not passed on to the parent scope. For variables that are newly bound or created *in both branches*, one could take different approaches for cases where this can be guaranteed statically. We suggest a conservative approach where such bindings also lose their validity, i.e., identical variable names do not necessarily represent the same variable. An optimistic approach, however, where such variables do become available after the branches are merged also makes sense, implying that the conditional represents different ways to bind essentially the same variable.

The case of deletion is equally interesting: If a variable is already bound before the conditional node, it would also be available after the branches have joined as these parts of the story diagram belong to the same scope. If this variable is, however, deleted *in any* of the branches, this could result in invalid bindings in the parent scope. We suggest again a conservative approach, where such bindings are removed from the parent scope, as opposed to an optimistic approach, where such bindings are retained, possibly leading to failures at runtime.

Figure 9 depicts the rules for handling a conditional with a successful match (i.e., the *Success* branch). Note that all the objects, particularly the scopes, are uniquely determined as they are attached to their respective control flow nodes whose connection is unique through the *success* reference. The upper left rule creates the scope for the success branch and passes the token to its first node. In the other rules, `successScope` always denotes this scope just created. The upper right rule copies the bindings of the parent scope to the fresh sub-scope. The binding updates of the conditional node (rules in the bottom row) are then performed in the sub-scope `successScope`. Note that the upper rules can be used for handling an unsuccessful matching in a conditional if we substitute failure for success in each relevant element (all bindings are inherited from the parent scope, token is shifted, but no bindings are updated). Creating a fresh scope for conditional branches upon entering them is crucial for handling *while* loops, where one of the branches returns to the conditional node. In this case, the pattern in the conditional node gets matched again and we proceed accordingly. Creating a fresh scope each time prevents bindings in branch scopes from appearing in subsequent iterations.

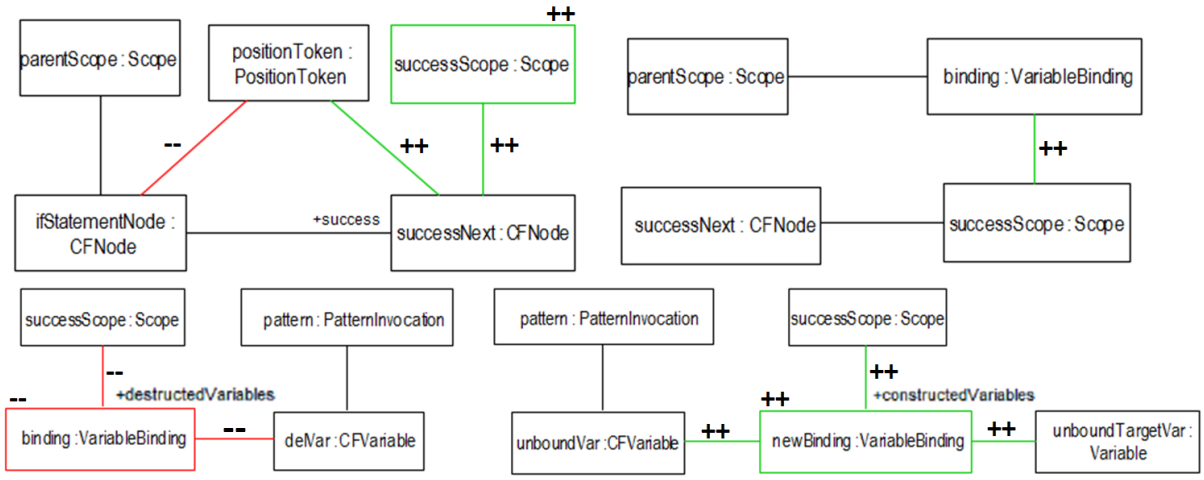
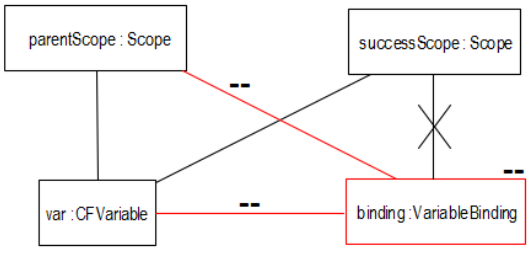


Figure 9: Handling a match of the conditional node

The "conservative" rule for handling deletion inside a conditional branch is depicted for the success branch in the figure to the right below (analogously for failure). It specifies that all variables bound in the parent scope, for which there is no binding in the success scope (NACs are represented by crossed-out edges), must be invalidated in the parent scope, making these bindings unavailable after the join.

Specifying an "optimistic" rule to make all elements matched in *both* branches available as bound in the parent scope after the join is fairly straightforward. Although we do not specify this rule and other possible variants in the present paper, the aforementioned considerations demonstrate the advantages of an explicit step-based semantic approach in clarifying such details.



Example. The bottom row in Figure 1 represents a conditional where the branches join. If the match on the left is successful, a binding for `next` is created which is used in the *Success* branch to delete this very object. If there is no match for `next`, we directly move on to the join node on the right. According to our semantics, this node belongs again to the same scope as the conditional, which means that there is definitely no valid binding for `next`.

5 Conclusion and Future Work

In this paper, we have presented a novel specification approach for a step semantics of SDM, where the steps correspond to GraTra rule applications in a provided graph grammar. Although we only examined basic SDM constructs, we have demonstrated the potential of our specification technique by presenting an alternative to a state-of-the-art semantics and by discussing possible design decisions (conservative vs. optimistic) concerning allowed variable bindings before and after conditionals (and while loops).

As also illustrated by our work, there is no definitive approach for defining language semantics and a detailed, step-based semantics can complement a high-level, denotational one. The domain of the latter does not include anything more than graphs resulting from a chain of rule applications, which makes it suitable to analyse standard GraTra notions such as confluence in a programmed scenario. However, our complementary approach is tailored to guide and facilitate providing programmed GraTra tooling in practice, where details hidden in the denotational semantics often play a crucial role. The price we pay is a substantially more elaborate machinery, dealing with finer details of SDM execution.

Nonetheless, our initial results presented in this paper have proven that the size of the semantic specification remains manageable even in the case of conditionals and loops; compound constructs can be specified by reusing parts of simpler steps. We do not expect the complexity of the semantic steps to explode when considering further constructs. We have experimentally implemented the semantic rules from this paper in the graph transformation tool eMoflon, realizing the rules on top of the actual SDM meta-model of eMoflon. This is promising regarding the practical applicability of our approach, e.g., for static analysis or refactoring.

The most obvious and immediate item of future work is to expand the semantics to cover further SDM language constructs such as SDM method calls and *for-each* style loops. These extensions can be based on the semantics presented in this paper. Another promising field of future investigation is extending SDM with a notion of multi-threading, where the single threads are SDM method executions. As a relevant application area, research on graph-based networking could profit from such an extension in various ways.

From a theoretical point of view, we plan to investigate the possibilities which arise from using a semantic style that is already close to well-known *operational* semantics. In particular, we plan to elaborate on a *trace* notion for SDM based on the defined steps and transition labelling based on GraTra rule applications. These concepts would enable us to leverage advanced techniques such as defining a suitable equivalence concept for SDM.

References

[1] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause & Gabriele Taentzer (2010): *Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations*. In: *MODELS 2010*, *LNCS* 6394, Springer, pp. 121–135, doi:10.1007/978-3-642-16145-2_9.

[2] András Balogh & Dániel Varró (2006): *Advanced Model Transformation Language Constructs in the VIATRA2 Framework*. In: *SAC 2006*, ACM, pp. 1280–1287, doi:10.1145/1141277.1141575.

[3] Egon Börger (2005): *Abstract State Machines: a Unifying View of Models of Computation and of System Design Frameworks*. *Ann. Pure Appl. Logic* 133(1-3), pp. 149–171, doi:10.1016/j.apal.2004.10.007.

[4] Andrea Corradini, Reiko Heckel & Ugo Montanari (2000): *Graphical Operational Semantics*. In: *ICALP Satellite Workshops*, pp. 411–418.

[5] Markus von Detten, Christian Heinzemann, Marie Christin Platenius, Jan Rieke, Dietrich Travkin & Stephan Hildebrandt (2012): *Story Diagrams - Syntax and Semantics*. Technical Report, University of Paderborn.

[6] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange & Gabriele Taentzer (2006): *Fundamentals of Algebraic Graph Transformation*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, doi:10.1007/3-540-31188-2.

[7] Gregor Engels, Jan Hendrik Hausmann, Reiko Heckel & Stefan Sauer (2000): *Dynamic Meta Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML*. In: *UML 2000*, *LNCS* 1939, Springer, pp. 323–337, doi:10.1007/3-540-40011-7_23.

[8] Amir Hossein Ghamarian, Maarten de Mol, Arend Rensink, Eduardo Zambon & Maria Zimakova (2012): *Modelling and Analysis using GROOVE*. *STTT* 14(1), pp. 15–40, doi:10.1007/s10009-011-0186-x.

[9] David Harel & Amnon Naamad (1996): *The STATEMATE Semantics of Statecharts*. *ACM Trans. Softw. Eng. Methodol.* 5(4), pp. 293–333, doi:10.1145/235321.235322.

[10] Reiko Heckel & Stefan Sauer (2001): *Strengthening UML Collaboration Diagrams by State Transformations*. In: *FASE 2001*, *LNCS* 2029, Springer, pp. 109–123, doi:10.1007/3-540-45314-8_9.

[11] Reiko Heckel & Albert Zündorf (2001): *How to Specify a Graph Transformation Approach - A Meta Model for Fujaba*. *Electr. Notes Theor. Comput. Sci.* 44(4), pp. 41–51, doi:10.1016/S1571-0661(04)80942-3.

[12] Stefan Jurack, Leen Lambers, Katharina Mehner, Gabriele Taentzer & Gerd Wierse (2009): *Object Flow Definition for Refined Activity Diagrams*. In: *FASE 2009*, *LNCS* 5503, Springer Berlin Heidelberg, pp. 49–63, doi:10.1007/978-3-642-00593-0_4.

[13] Christoph Knieke & Ursula Goltz (2010): *An Executable Semantics for UML 2 Activity Diagrams*. In: *Proc. of FML*, ACM, New York, NY, USA, pp. 3:1–3:5, doi:10.1145/1943397.1943400.

[14] Hans-Jörg Kreowski, Sabine Kuske & Grzegorz Rozenberg (2008): *Graph Transformation Units - An Overview*. In: *Concurrency, Graphs and Models*, *LNCS* 5065, Springer, pp. 57–75, doi:10.1007/978-3-540-68679-8_5.

[15] Michael Löwe & Martin Beyer (1993): *AGG - An Implementation of Algebraic Graph Rewriting*. In Claude Kirchner, editor: *RTA-93*, *LNCS* 690, Springer, pp. 451–456, doi:10.1007/978-3-662-21551-7_36.

[16] Gordon D. Plotkin (2004): *A Structural Approach to Operational Semantics*. *J. Log. Algebr. Program.* 60-61, pp. 17–139, doi:10.1016/j.jlap.2004.03.009.

[17] Grzegorz Rozenberg, editor (1997): *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations*. World Scientific Publishing Co., Inc., River Edge, NJ, USA.

[18] Douglas C. Schmidt (2006): *Guest Editor's Introduction: Model-Driven Engineering*. *Computer* 39(2), pp. 25–31, doi:10.1109/MC.2006.58.

[19] Andy Schürr, A. J. Winter & Albert Zündorf (1999): *The PROGRES-Approach: Language and Environment*. In: *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2*, World Scientific, pp. 487–550, doi:10.1142/9789812815149_0013.

[20] Harald Störrle & Jan Hendrik Hausmann (2005): *Towards a Formal Semantics of UML 2.0 Activities*. In: *SE 2005, Fachtagung des GI-Fachbereichs Softwaretechnik*, pp. 117–128.

[21] Zoltán Ujhelyi, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, Benedek Izsó, István Ráth, Zoltán Szatmári & Dániel Varró (2015): *EMF-IncQuery: An Integrated Development Environment for Live Model Queries*. *SCP* 98(1), pp. 80–99, doi:10.1016/j.scico.2014.01.004.

[22] Albert Zündorf (2002): *Rigorous Object Oriented Software Development*. Habilitation thesis, draft version 0.3, University of Paderborn.