State Diagrams in UML: A Formal Semantics using Graph Transformations

or

Diagrams are nice, but graphs are worth their price*

Martin Gogolla Francesco Parisi Presicce

University of Bremen, Computer Science Department (FB3)
Postfach 33440, D-28334 Bremen, Germany
gogolla@informatik.uni-bremen.de

University of Rome 'La Sapienza', Computer Science Department (DSI)
Via Salaria 113, I-00198 Roma, Italy
parisi@dsi.uniroma1.it

Abstract

We show how to transform UML (Unified Modeling Language) state diagrams into graphs by making explicit the intended semantics of the diagram. The process of state expansion in nested state diagrams is explained by graph transformations in three steps: (1) adding boundary nodes introducing a precise interface for the state to be expanded, (2) expanding the state, and (3) removing the boundary nodes. The general idea of approaching the semantics of UML diagrams by graph transformations is applicable to other forms of UML diagrams as well. The main advantage of the graph transformation approach is the closeness between the (mathematical) graph representation and the (UML) diagram representation.

Keywords: UML, formal semantics, graph transformation, state diagram, nested state, stubbed transition.

1 Introduction

One of the descriptions of what UML is supposed to be is given in one of its original documents [BJR97b]:

> *The Unified Modeling Language (UML) is a general-purpose visual modeling language that is designed to specify, visualize, construct and document the artifacts of software systems. The UML is simple and powerful. The language is*

*Work reported here has been carried out during a visit of the first author at the Dipartimento di Scienze dell' Informazione supported by a grant from Universitá di Roma 'La Sapienza'.

based on a small number of core concepts that most object-oriented developers can easily learn and apply. The core concepts can be combined and extended so that expert object modelers can define large and complex systems across a wide range of domains.

The UML effort [BJR97c] is an offspring of the object-oriented development methods OMT [RBP$^+$91], OOSE [JCJO92], and OOA/OOD [Boo94]. UML comprises a number of diagram forms used to describe particular aspects of software artifacts: the diagram forms can be divided depending on whether they are intended to describe structural or behavioral aspects. One particularly important diagram form concerning behavior is the state diagram. State diagrams (or statecharts as they are also called) are derived from the original proposal [Har87] and are modified in order to include object-oriented features.

In the UML semantics document [BJR97b], the description of UML diagrams is divided into three parts, namely (1) the abstract syntax, (2) the well-formedness rules for the abstract syntax, and (3) the semantics. The abstract syntax is provided by UML diagrams, the well-formedness rules are formulated in the special constraint language OCL, and the semantics is described primarily in precise natural language. In some cases, however, this semantics is not clear enough. In other cases the use of more formal description techniques may improve the readability of the corresponding documents. For the development of tools (like editors, analyzers, code generators, validation systems, verification systems) a formal semantics, at least an operational one, is a prerequisite.

Our approach to UML state diagram semantics can be seen as an intermediate step between the original UML diagrams and a general comprehensive semantical framework. With the graph notation we try to be as close as possible to the original UML representation, but we achieve a representation forcing an unambiguous interpretation. On the basis of the resulting graphs (the semantics of the UML state diagrams) various semantical frameworks like temporal logics, streams, or (again) graph transformation systems (among many other approaches) can be applied. Thus our target language, i.e. unnested state diagrams, is an intermediate language to which semantics can be given in a variety of ways. Such unnested state diagrams serve to explain the semantics of the UML high-level constructs like nesting to the UML modeller in terms of language features the modeller already uses for its own modelling business.

Our topic is related to other works, in particular to approaches for treating the UML semantics, but none is based on the simple and intuitive graph transformation approach such as ours. We can roughly divide the approaches into specification-based or graph-based.

Specification-based approaches: There is work on the basis of well-established traditional approaches to specification like Z and VDM: [FBLPS97] focuses on the UML type system, a general integration of Z with object-orientation is discussed in [Ebe97], and in [Lan96, BLM97] an object calculus enhancing the expressibility of object-oriented notations has been proposed. Other approaches treat in detail the UML predecessor OMT [BCV96, WRC97], and in particular class diagrams [BC95] in connection with Larch are discussed. [BHH$^+$97] sketches a general scenario for most UML diagrams without going into technical details (for example stubs in state diagrams, which we consider in detail, are not mentioned in that paper).

Graph-based approaches: In [MSP94] graphs are employed but explicitly for statecharts, and the introduced graphs and graph transformations are well-suited for the statechart computational model only. In the graph grammar community there is work on the application of this field to actor systems [JLR93, Kor94] which are inherently close to object-oriented approaches. Graph grammars have also been successfully been applied to the semantics [WG96] of an object specification language which has many similarities to UML [GR97].

The structure of the rest of this paper is as follows. In Sect. 2 we indicate the features of UML state diagrams and what the resulting graphs look like. In Sect. 3 the process of deriving graphs from diagrams is explained in more detail by describing the respective graph transformation systems. Section 4 introduces the formal background on graph transformation systems, and in Sect. 5 more advanced UML state diagram features like stubs are handled. Sect. 6 contains concluding remarks.

2 From UML Diagrams to Graphs by Example

In this section, we show how to transform UML state diagrams into graphs, making explicit the intended semantics of the diagram. In doing so, the resulting visual representation becomes independent of the specific layout and of the relative position of the components of the diagram. The transformation from diagrams to graphs is illustrated via an example taken from [RBP$^+$91] and dealing with the automatic transmission of a car: the internal structure of one node of the diagram in Fig. 1 is made explicit in the diagram in Fig. 2.

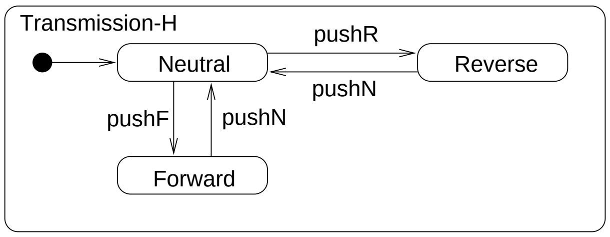

Figure 1: Car transmission - UML high level diagram

The statechart called Transmission-H in Fig. 1 consists of three states (Neutral, Forward, Reverse) and one pseudo-state (represented by a black dot) used to indicate which internal state is entered when there is a transition to Transmission-H. The four labelled arrows denote the possible transitions from one internal state to another one by performing corresponding actions. The graph corresponding to the Transmission-H diagram is depicted in Fig. 3 where the states are denoted by (labelled) nodes and the transitions by (labelled)

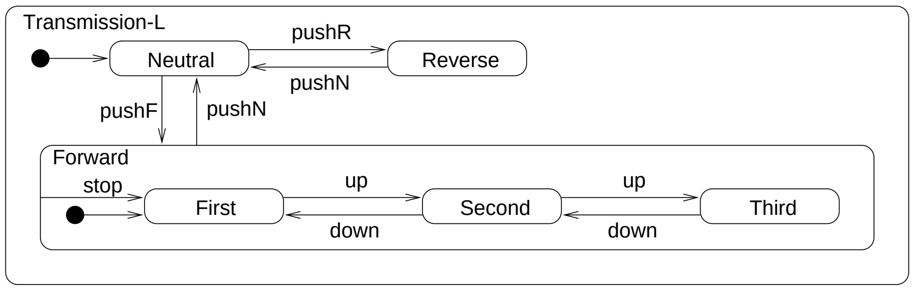

Figure 2: Car transmission - UML low level diagram

arcs (when referring to state diagrams, we use the notions state and arrow, where referring to graphs we speak of nodes and arcs). This translation is trivial since the semantics of Transmission-H is already explicit in the diagram (another example to follow will point out that a high level UML diagram and the corresponding graph are not necessarily isomorphic).

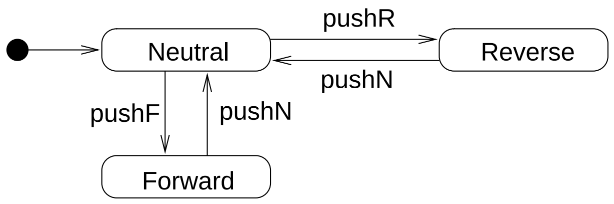

Figure 3: Car transmission - High level graph

The state diagram Transmission-L (a lower level representation of the system, since it shows details on the structure of the state labelled Forward) contains semantical information in implicit form, and therefore the corresponding graph looks somewhat different. In Fig. 2, the arrow labelled pushF from the state labelled Neutral to the boundary of the substate Forward indicates a transition to the initial state of Forward, i.e., the state labelled First, to which the (unique) pseudo-state (represented by the black dot) is connected. In the graph depicted in Fig. 4 and corresponding to the state diagram in Fig. 2, there is a direct arc (labelled pushF) from the node labelled Neutral to the node labelled First.

Similarly, but *not* symmetrically, the arrow labelled pushN from the boundary of the nested state Forward to the state labelled Neutral indicates a transition from *every* state in Forward to Neutral. In the resulting graph, there is an arc labelled pushN from each of the nodes First, Second, and Third to the node labelled Neutral. Analogously, the intended semantics of the arrow labelled stop from the boundary of Forward to First is a transition from every internal state of Forward to First: in the resulting graph, there is an arc labelled stop from each of the nodes First, Second, and Third to the node labelled First.

It is worth noticing (again) the asymmetric intended meaning of the arrows touching the boundary of a state: the arrows pointing to the boundary indicate a specific state (denoted by the pseudo-state) while the arrows pointing away from the boundary reflect

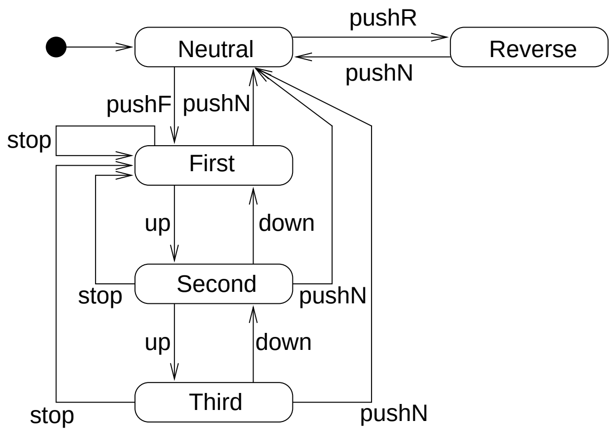

Figure 4: Car transmission - Resulting low level graph

the internal structure of the state since it represents as many transitions (all with the same name) as there are internal states.

3 Changing Levels of Abstraction by Example

As mentioned in the previous section, the arrows in a diagram pointing to a boundary have a different meaning than the arrows pointing away from a boundary. In order to "restore" the symmetry between entry and exit arrows, we transform, via simple rewriting rules, the graph corresponding to the high level diagram by explicitly introducing boundary nodes. Such boundary nodes are the technical vehicle for representing the boundaries of nested states. They allow the "graph flattening" to be explained independently of the context. So there is a clear separation between internal structure and context. Each entry/exit arrow is replaced by one entry/exit boundary node, and all the boundary nodes represent the interface between a given state and those connected to it. After this generic transformation is performed, the actual substitution of a state with its internal structure is obtained by using an "application specific" transformation ("exploding the chosen state"). After applying the specific transformation, other simple rewriting rules are applied to systematically remove the boundary nodes to obtain the graph corresponding to the low level diagram.

Returning to the example of the previous section and selecting from Fig. 3 the state labelled Forward, one boundary node is introduced for the arrow labelled pushF and one for the arrow labelled pushN, using the two rules in Fig. 5, respectively. The result of applying these rules to the graph in Fig. 3 is shown in Fig. 6.

Our rules consist of three parts: for each rule, the leftmost graph represents the part to be deleted, the rightmost graph the part to be added, and the central graph the part unchanged which is used to connect the transformed part to the subgraph not affected by the transformation. The leftmost graph is also the part which is mapped to an actual graph when such a rule is applied. In this example, L is a variable label which corresponds

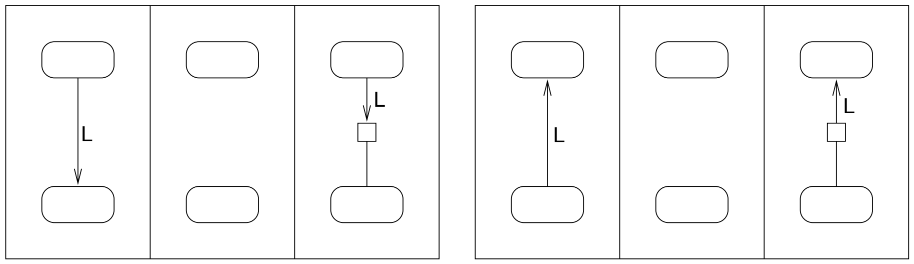

Figure 5: Graph transformation system for introducing the boundary nodes

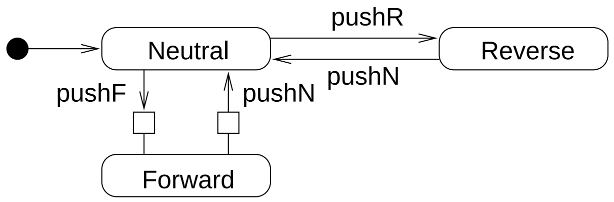

Figure 6: Car transmission - Making boundaries explicit in the high level graph

to pushF for the left rule and to pushN for the right rule.

One further remark concerning the rules should help reading them. Although the layout of the graphs in the rules is irrelevant for their application, we have followed certain layout guidelines for rules handling boundary nodes: larger *top* nodes stand for nodes of the input graph lying *outside* the nested UML state in consideration, large *bottom* nodes (if present) represent output graph nodes *inside* the considered nested UML state, and auxiliary boundary nodes (if present) are displayed in the middle by small nodes. These conventions for boundary nodes are also obeyed for treating the other auxiliary nodes to be introduced below, namely stub nodes and final nodes.

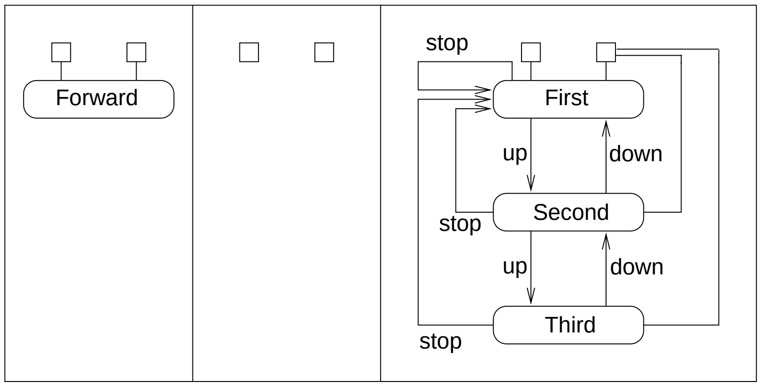

Figure 7: Graph transformation system for the car transmission example

The rule in Fig. 7 represents the application specific transformation for the car transmission example: the left side is the state to be expanded with its two boundary nodes, the right side the internal structure of the state, and the central part consists of the two boundary nodes only, which are the interface with the rest of the graph. The application of the rule in Fig. 7 to the graph in Fig. 6 generates the graph in Fig. 8 (having boundary nodes in contrast to the graph in Fig. 4).

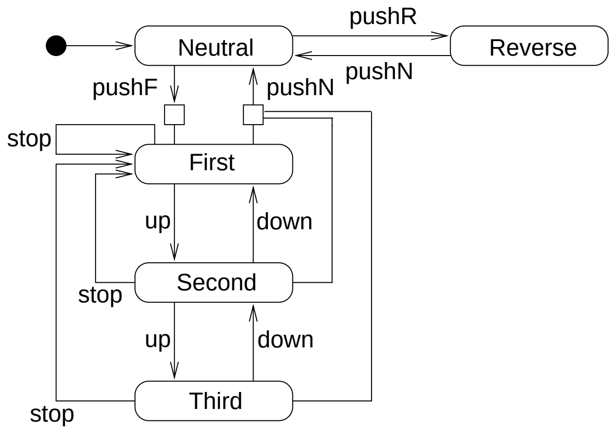

Figure 8: Car transmission - Applying the rule in the high level graph

Having performed the substitution, the rules in Fig. 9 are applied to remove the boundary nodes. The top rule is responsible for arcs entering an expanded subgraph, and the two bottom rules treat arcs leaving such a subgraph.

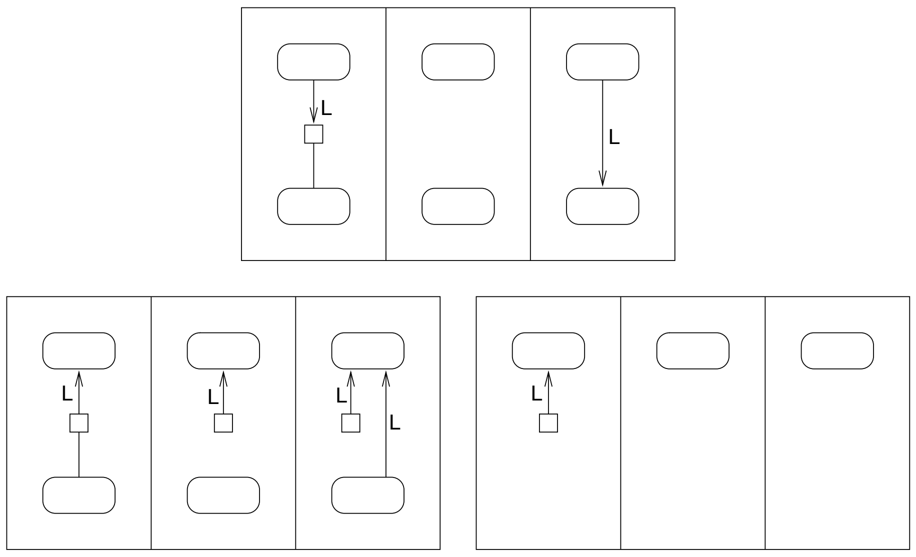

Figure 9: Graph transformation system for removing the boundary nodes

The top rule in Fig. 9 is just the inverse of the left rule in Fig. 5. But it is not possible to use the inverse of the right rule of Fig. 5 to remove boundary nodes because of the asymmetry in the meaning of entry and exit arrows: the specific substitution in Fig. 7 has introduced several states from which to exit to Neutral. So the bottom left rule is used to replace connections to the (exit) boundary node with direct transitions, and only after the boundary node has been bypassed by all the "internal" nodes, the boundary node can be removed by the bottom right rule (the formal framework for graph rewriting presented in the next section guarantees that the rules can only be applied in the stated order). The result of eliminating the boundary nodes (which served as the interface for the application dependent production) from the graph in Fig. 8 is of course the graph in Fig. 4.

4 Background on Graph Transformation Systems

We briefly review the basic notions of the algebraic approach [Ehr79, PPEM87] to graph transformations necessary to explain the formal model on which our notation is based. The elements (arcs and nodes) of our graphs are labelled with names from appropriate sets. Let us call C_A and C_N the sets of labels used for arcs and nodes, respectively. As in most programming languages, certain identifiers (representing "variables") can be associated to values from a certain domain (representing the "type" of the variable). To distinguish identifiers that represent constants from those that represent variables, we introduce a structure on, say C_A, via a binary relation $\leq$, where $a < x$ $(:\Leftrightarrow\ a \leq x \wedge a \neq x)$ denotes the fact that the variable x can be replaced by the constant a but not vice versa. For our development, a "flat" structure is sufficient, where we denote with O the constants and with V the variables, $C_A = O \cup V$ and $a < x$ for $a \in C_A$ and $x \in V$ (this idea has been generalized in [PPEM87]).

Definition (C-colored graph) *Let $C = (C_A, C_N)$ be a pair of color alphabets.*

1. *A* **C-colored graph** *G is a six-tuple $(G_A, G_N, s, t, m_A, m_N)$, consisting of:*

   - *sets G_A and G_N, called the set of arcs and the set of nodes, respectively;*
   - source *and* target *mappings $s : G_A \rightarrow G_N, t : G_A \rightarrow G_N$;*
   - arcs *and* nodes *coloring mappings $m_A : G_A \rightarrow C_A, m_N : G_N \rightarrow C_N$.*

   A graph $G' = (G'_A, G'_N, s', t', m'_A, m'_N)$ is a subgraph of a graph G if $G'_A \subseteq G_A$, $G'_N \subseteq G_N$, and all the mappings s', t', m'_A, m'_N are restrictions of the corresponding ones from G.

2. *A* **graph morphism** *$f : G \rightarrow G'$ is a pair $(f_N : G_N \rightarrow G'_N, f_A : G_A \rightarrow G'_A)$ such that*

   - *$f_N \circ s = s' \circ f_A$; $f_N \circ t = t' \circ f_A$ (the structure is preserved);*
   - *$m'_A \circ f_A \leq m_A$; $m'_N \circ f_N \leq m_N$ (the labels can be changed).*

   as in the diagram in Fig. 10 which commutes componentwise.

$$\begin{tikzcd}
& G_A \arrow[dl, "m_A"'] \arrow[r, shift left, "s"] \arrow[r, shift right, "t"'] \arrow[dd, "f_A"'] & G_N \arrow[dr, "m_N"] \arrow[dd, "f_N"] & \\
C_A & & & C_N \\
& G'_A \arrow[ul, "m'_A"] \arrow[r, shift left, "s'"] \arrow[r, shift right, "t'"'] & G'_N \arrow[ur, "m'_N"'] &
\end{tikzcd}$$

Figure 10: Graph morphism

For simplicity, in our examples we have denoted the components (arcs and nodes) with their labels. All the nodes and arcs without an explicit label in the rules must be considered as labelled with a variable which can be instantiated with the appropriate name when the rule is applied.

Although in our examples all morphisms are inclusions (with the exception of changing labels), the theory is valid also for general morphisms.

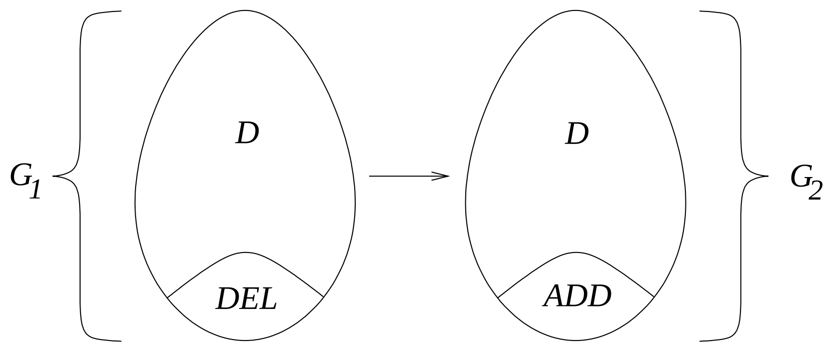

Figure 11: Single graph rewriting step

As pictured in Fig. 11 the idea, common to every theory on graph rewriting, is to realize, from an initial graph, a sequence of transformations, following rules allowing a subgraph DEL of G_1 to be replaced by another graph ADD resulting in G_2, leaving unchanged the subgraph D of G_1 not involved in the deletion of DEL. Every transformation of this kind requires the specification of how ADD must be connected to the graph D (called *embedding*).

Definition (Graph rule) *A graph rule p is a pair $(L \leftarrow K \rightarrow R)$ of graph morphisms, where L, R and K are called the left side, the right side and the interface of the rule, respectively.*

A rule specifies that a graph L must be replaced by a graph R, using an interface graph K (whose arcs and nodes are the gluing items) for the embedding.

Definition (Direct Derivation) *Given a rule $p = (L \leftarrow K \rightarrow R)$ with $l : K \rightarrow L$*

injective, and a morphism $g_1 : L \to G_1$, a direct derivation from G_1 to G_2 via p and g_1, denoted by $(p, g_1) : G_1 \Rightarrow G_2$ or by $G_1 \Rightarrow_{p,g_1} G_2$, consists of the double pushout in Fig. 12 (in comparison to Fig. 11 we have $DEL = g_1(L)$ and $ADD = g_2(R)$).

$$\begin{tikzcd}
L \arrow[d, "g_1"'] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "c"'] & R \arrow[d, "g_2"] \\
G_1 & D \arrow[l, "l'"] \arrow[r, "r'"'] & G_2
\end{tikzcd}$$

Figure 12: Double pushout construction for rule application

The morphism g_1 denotes the presence of the lefthand side L of the rule p in G_1. The graph D denotes the context, the part of the graph G_1 not affected by the rewriting and to which R is attached via the interface K. Recall that for simplicity all morphisms can be interpreted as inclusions. So r and c indicate that the interface graph K is part of both the graph R to be added and the graph D (of G_1) not affected by the transformation and used for the gluing. The morphisms l' and r' indicate that D is present both before and after the derivation. In the application of the left rule in Fig. 5 to the graph in Fig. 3, the morphism g_1 maps the top (bottom) node to the node labelled Neutral (Forward) and the arc with the variable name L to the arc labelled pushF.

Definition (Derivation)

A derivation $G \Rightarrow_P^* H$ is a sequence of direct derivations

$$G = G_1 \Rightarrow_{p_1} \ldots \Rightarrow_{p_{n-1}} G_n = H$$

using rules $p_i \in P$.

Notice that the derivation diagram is symmetric and therefore, if $(p, g_1) : G_1 \Rightarrow G_2$, then $(p^{-1}, g_2) : G_2 \Rightarrow G_1$ where $p^{-1} = (R \leftarrow K \to L)$.

The applicability of a rule $p = (L \leftarrow K \to R)$ to a graph G_1 is determined by the existence and properties of a total morphism $g_1 : L \to G_1$. The existence of a pushout complement D for a given g_1 depends on the following gluing conditions being satisfied [Ehr79].

Theorem (Gluing Conditions)

Given $p = (L \leftarrow K \to R)$ and $g_1 : L \to G_1$, let

$$ID_{g_1} = \{x \in L : \exists x' \in L, x \neq x', g_1(x) = g_1(x')\}$$

$$DANG_{g_1} = \{n \in L_N : \exists e \in G_{1_A} - g_{1_A}(L_A) \textit{ such that } g_{1_N}(n) = s(e) \textit{ or } g_{1_N}(n) = t(e)\}$$

Then the pushout complement D exists iff $DANG_{g_1} \cup ID_{g_1} \subseteq l(K)$

The condition $ID_{g_1} \subseteq l(K)$ guarantees that a D can be found so that G_1 is exactly the gluing of L and D. The condition $DANG_{g_1} \subseteq l(K)$ guarantees that D is “well formed” and that there are no arcs without one of the end nodes. For example, the rule at the bottom right of Fig. 9 cannot be applied to Fig. 8 because it would leave three arcs (leaving the nodes First, Second and Third) without one of the end nodes. This rule can be applied only after three applications of the rule at the bottom left of Fig. 9.

This (apparent) limitation of the traditional double pushout approach can be overcome either by using single pushouts [Löw93] or by using "restricting derivation sequences" [PP93]. In this context, we take advantage of the gluing conditions to guarantee that certain rules are applied in the appropriate order.

It is possible to give a graph grammar in terms of terminal and nonterminal alphabets, a finite set of rules and an initial graph to generate the language of graphs corresponding to diagrams. In this paper, we are only interested in transformations rules where we consider terminal graphs those corresponding to diagrams (as shown in Sect. 2) and nonterminal those with explicit boundary nodes.

5 Transitions to Nested States

After having explained what basic UML state diagrams look like and how the corresponding translation into a graph is achieved, we continue by discussing more advanced UML state diagram modeling features, namely the so-called stubs. In the example in Fig. 13 (taken from [BJR97a]) the nested state W contains two stubs represented by two fat vertical bars indicating that there is (1) a transition from outside the nested state into the nested state, indicated by the arrow labelled p, and (2) a transition from inside the nested state to the outside, indicated by the upper unlabelled arrow leaving state W. These stubbed transitions are understood to be exceptional ones in the sense that they are different from a normal entry to a nested state via the initial node and different from a normal exit from a nested state via the final node. The diagram expresses that there is more structure in the state W and gives the possibility to access this state by the stub notation. The stubs can be thought of as place holders or formal parameters for substates of the state W.

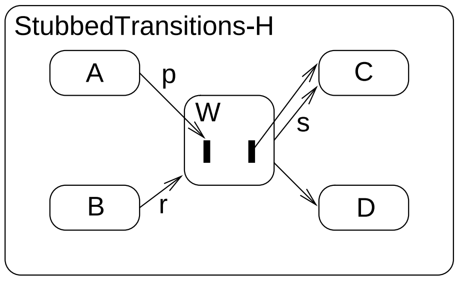

Figure 13: Stubbed transitions - UML high level diagram

Another UML language feature (introduced in this example in a silent way) is the use of unlabelled arcs. They represent transitions from the nested state W to states C and D.

The actions belonging to the transitions are (in the UML philosophy) connected with its source states which are hidden inside the nested state W and therefore not displayed in the high level graph in Fig. 13: the action associated with the upper unlabelled arrow leaving the stub is to be specified inside the nested state W; the action associated with the regular unlabelled exit arrow to state D is the action leading to the final state which therefore must be present. This "action" model for unlabelled arcs is made precise below by our graph transformation approach.

On the UML level, the refinement of the diagram in Fig. 13 is presented in Fig. 14. The state W is expanded into a more elaborate structure, and the stubs are mapped to actual states. In this example both stubs are mapped to the single state E. The arrows to and from the stubs are passed to the actual parameter state. As in the car transmission example in Fig. 2, the expanded nested state must have an initial state, and unlike the car transmission example it has a final state.

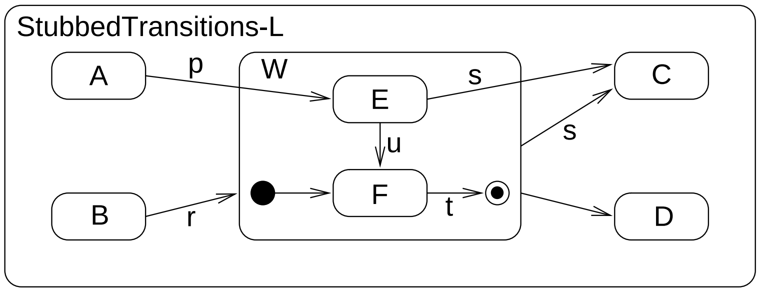

Figure 14: Stubbed transitions - UML low level diagram

Earlier we mentioned that the transformation from an UML diagram to a corresponding graph is not always an one-to-one correspondence. The graph in Fig. 15 makes explicit the stub connection to the nested state W by introducing stub nodes represented again by fat vertical bars. The stub nodes of Fig. 13 are "moved outside" the nested state, the connection between the nested state and the stub nodes is represented by undirected arcs, and the arrows to and from the stubs are preserved.

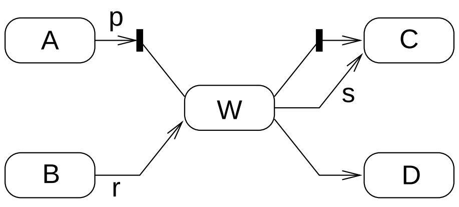

Figure 15: Stubbed transitions - Making stubs explicit in the high level graph

In the next graph as pictured in Fig. 16, we introduce boundary nodes for making explicit the relationship between the node to be expanded and its environment. As in the car transmission example, we use the graph transformation system given in Fig. 5 to introduce these boundary nodes on the respective arcs. Unlike the car transmission example where no final state was specified, we need here the graph transformation system as shown in Fig. 17 allowing to generate nodes representing the final state. Such nodes are generated from unlabelled arcs.

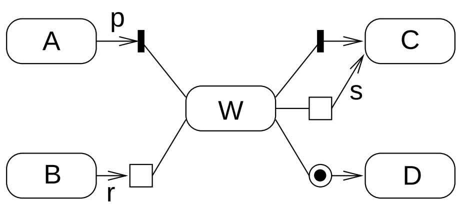

Figure 16: Stubbed transitions - Making boundaries explicit in the high level graph

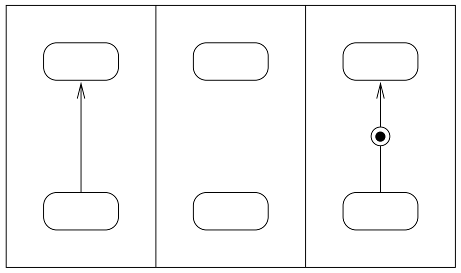

Figure 17: Graph transformation system for introducing the final node

One short remark concerning the role of the graph in Fig. 15 seems to be in order. This graph is an intermediate step between Fig. 13 and Fig. 16. In Fig. 15 in a sense “boundary” nodes are introduced for the stub nodes; therefore these nodes are called stub nodes which are introduced on the arcs (in Fig. 13) to the stubs. Fig. 16 afterwards introduces the normal boundary nodes not related to stubs. Thus, nodes corresponding to the stubs are introduced for the same reason for which boundary nodes are used, namely context independence.

After having applied these transformation steps, we are now in a position to specify the application dependent rule corresponding to the desired state expansion as given in Fig. 18. The node W is expanded into the two nodes E and F with a transitions along the arc u, a transition labelled t from node F to the final node, and a transition labelled s from node E to a stub node. Applying this rule to the graph of Fig. 16 yields the graph as given in Fig. 19.

The last step is now the removal of all auxiliary nodes, i.e., boundary, stub, and final nodes. By means of the already explained transformation system in Fig. 9 for removing boundary nodes, and another simple, analogous system (Fig. 20) for removing stub nodes as well as an analogous system (Fig. 21) for removing final nodes we can transform the intermediate result in Fig. 19 into the final graph as pictured in Fig. 22. Being precise, we would have to add one (simple) transformation system for removing duplicate arcs

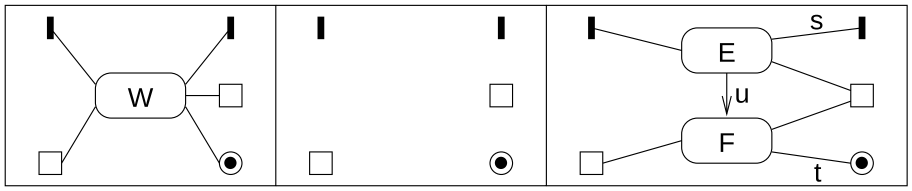

Figure 18: Graph transformation system for the stubbed transitions example

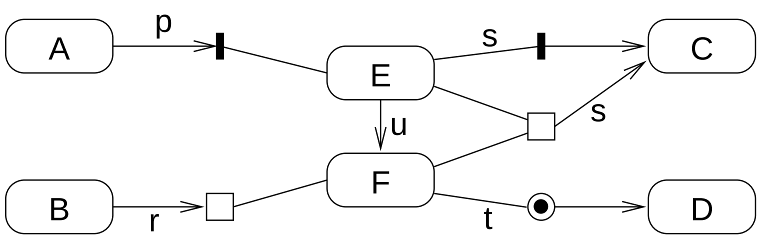

Figure 19: Stubbed transitions - Applying the rule in the high level graph

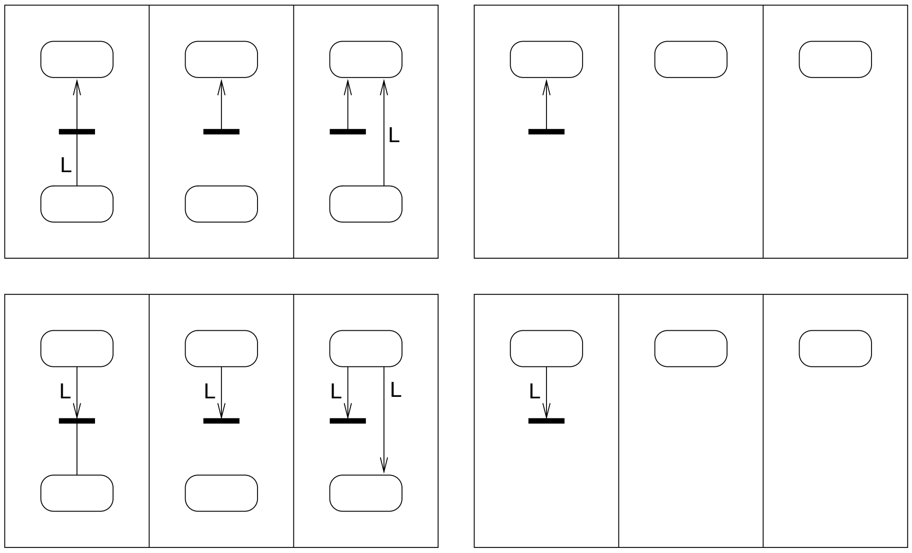

Figure 20: Graph transformation system for removing the stubs

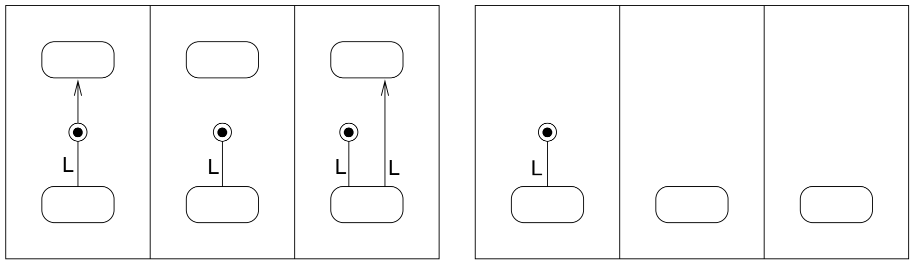

Figure 21: Graph transformation system for removing the final node

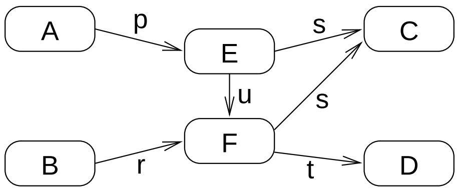

Figure 22: Stubbed transitions - Resulting low level graph

having the same label (in the stubbed transition example we end up with *two* arcs from node E to node C both labelled s).

One closing remark should clarify the role the rules play and who is responsible for defining them. The rules for introducing and removing boundary and final nodes and for removing stub nodes (Fig. 5, 9, 17, 20, and 21) are fixed once and for all UML models, i.e. they come from the method. The introduction of stub nodes is done directly on the basis of the UML state diagram (Fig. 13) by moving the stubs "outside" the nested state. The rules for state expansion (Fig. 7 and 18) come from the UML modeller (specifier). They can however (at least partly) be deduced from the low- and high-level UML diagrams. Thus, the specifier needs to worry about the internal structure only of a state: thanks to boundary, stub, and exit nodes, the specifier can ignore the context in which they will be used. The method developer provides the rules to allow the "graph flattening" which are based on the rules provided by the specifier.

6 Conclusion

We have presented an intuitive way of achieving a normal form for nested UML state diagrams by means of graph transformations. The value of this approach is in the better understanding of the semantics in terms of the notation itself. The main advantage we see in our language of mathematical graphs is the closeness to the original UML diagrammatic representation. Proceeding this way, we preserve as much as possible the intuitiveness of the UML representation, but we additionally obtain a precise semantics in terms of automata. The process of developing a graph from a diagram is done in a systematic way by applying graph transformations.

This approach is applicable to other forms of UML diagrams as well. The general idea of the overall approach is to rewrite UML diagrams to canonical graph representation. Apart from applying this to state diagrams, this can be done (for instance) with class diagrams reflecting the structural aspects of an UML specification, and it can be done (for instance) with sequence diagrams particularly suited for representing certain snapshots of object life cycles.

Acknowledgements

Thanks to Mark Richters for commenting a draft of this paper. The remarks of the referees have helped to improve our presentation.

References

[BC95] R. Bourdeau and B. Cheng. A Formal Semantics for Object Model Diagrams. *IEEE Transactions on Software Engineering*, 21(10):799–821, 1995.

[BCV96] E. Bertino, D. Castelli, and F. Vitale. A Formal Representation for State Diagrams in the OMT Methodology. In K.G. Jeffery, J. Kral, and M. Bartosek, editors, *Proc. Seminar Theory and Practice of Informatics (SOFSEM'96)*, pages 327–341. Springer, Berlin, LNCS 1175, 1996.

[BHH$^+$97] Ruth Breu, Ursula Hinkel, Christoph Hofmann, Cornel Klein, Barbara Paech, Bernhard Rumpe, and Veronika Thurner. Towards a Formalization of the Unified Modeling Language. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proc. 11th European Conf. Object-Oriented Programming (ECOOP'97)*, pages 344–366. Springer, Berlin, LNCS 1241, 1997.

[BJR97a] Grady Booch, Ivar Jacobson, and James Rumbaugh, editors. *UML Notation Guide (Version 1.1)*. Rational Corporation, Santa Clara, 1997. `http://www.rational.com`.

[BJR97b] Grady Booch, Ivar Jacobson, and James Rumbaugh, editors. *UML Semantics (Version 1.1)*. Rational Corporation, Santa Clara, 1997. `http://www.rational.com`.

[BJR97c] Grady Booch, Ivar Jacobson, and James Rumbaugh, editors. *UML Summary (Version 1.1)*. Rational Corporation, Santa Clara, 1997. `http://www.rational.com`.

[BLM97] J.C. Bicarregui, Kevin Lano, and T.S.E. Maibaum. Objects, Associations and Subsystems: A Hierarchical Approach to Encapsulation. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proc. 11th European Conf. Object-Oriented Programming (ECOOP'97)*, pages 324–343. Springer, Berlin, LNCS 1241, 1997.

[Boo94] Grady Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin Cummings, Redwood City, 1994.

[Ebe97] Jürgen Ebert. Integration of Z-Based Semantics of OO-Notations. In Haim Kilov and Bernhard Rumpe, editors, *Proc. ECOOP'97 Workshop on Precise Semantics for Object-Oriented Modeling Techniques.* Technische Universität München, Informatik-Bericht TUM-I9725, 1997.

[Ehr79] Hartmut Ehrig. Introduction to the Algebraic Theory of Graph Grammars. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *Proc. Int. Workshop Graph-Grammars and Their Application to Computer Science and Biology*, pages 1–69. Springer, Berlin, LNCS 73, 1979.

[FBLPS97] R. France, J.M. Bruel, M. Larrondo-Petrie, and M. Shroff. Exploring the Semantics of UML type structures with Z. In H. Bowman and J. Derrick, editors, *Proc. 2nd IFIP Conf. Formal Methods for Open Object-Based Distributed Systems (FMOODS'97)*, pages 247–260. Chapman and Hall, London, 1997.

[GR97] Martin Gogolla and Mark Richters. On Combining Semi-Formal and Formal Object Specification Techniques. Technical Report, University of Bremen, 1997.

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex System. *Science of Computer Programming*, 8(3):231–274, 1987.

[JCJO92] Ivar Jacobsen, Magnus Christerson, Patrik Jonsson, and G. G. Overgaard. *Object-Oriented Software Engineering.* Addison-Wesley, 1992.

[JLR93] Dirk Janssens, M. Lens, and Grzegorz Rozenberg. Computation Graphs for Actor Grammars. *Journal of Computer and System Science*, 46(1):60–90, 1993.

[Kor94] M. Korff. Single Pushout Transformations of Equationally Defined Graph Structures with Applications to Actor Systems. In H.J. Schneider and H. Ehrig, editors, *Proc. Workshop Graph Transformations in Computer Science*, pages 234–248. Springer, Berlin, LNCS 776, 1994.

[Lan96] Kevin Lano. Enhancing Object-Oriented Methods with Formal Notations. *Theory and Practice of Object Systems*, 2(4):247–268, 1996.

[Löw93] M. Löwe. Algebraic Approach to Single-Pushout Graph Transformation. *Theoretical Computer Science*, 109(1,2):181–224, 1993.

[MSP94] Andrea Maggiolo-Schettini and Adriano Peron. Semantics of Full Statecharts Based on Graph Rewriting. In H.J. Schneider and H. Ehrig, editors, *Proc. Workshop Graph Transformations in Computer Science*, pages 265–279. Springer, Berlin, LNCS 776, 1994.

[PP93] Francesco Parisi-Presicce. Single versus Double Pushout Derivations of Graphs. In E.W. Mayr, editor, *Proc. 18th Int. Workshop Graph-Theoretic Concepts in Computer Science (WG'92)*, pages 248–262. Springer, Berlin, LNCS 657, 1993.

[PPEM87] Francesco Parisi-Presicce, Hartmut Ehrig, and Ugo Montanari. Graph Rewriting with Unification and Composition. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Proc. Int. Workshop Graph Grammars and Their Application to Computer Science*, pages 496–514. Springer, Berlin, LNCS 291, 1987.

[RBP$^+$91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, 1991.

[WG96] Annika Wagner and Martin Gogolla. Defining Operational Behavior of Object Specifications by Attributed Graph Transformations. *Fundamenta Informaticae*, 26(3,4):407–431, 1996.

[WRC97] Enoch Y. Wang, Heather A. Richter, and Betty H. C. Cheng. Formalizing and Integrating the Dynamic Model within OMT. In *Proc. 19th Int. Conf. on Software Engineering (ICSE'97)*, pages 45–55. ACM Press, 1997.