Transforming Specification Architectures by GenGED*

Roswitha Bardohl, Claudia Ermel, Julia Padberg

Institut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin, Germany
{rosi,lieske,padberg}@cs.tu-berlin.de

Abstract. This contribution concerns transformations of specification architectures which are diagrams of sub-specifications. The graph of a diagram presents the architecture: the nodes correspond to the sub-specifications and the edges to specification morphisms. We do not fix a specific visual specification technique, so this approach is in the tradition of high-level replacement systems. We discuss how to transform such specification architectures and distinguish local and global changes. The main emphasis of this contribution is the specification and transformation of specification architectures using GenGED. In GenGED, a visual language (VL) is defined by a visual alphabet and a visual syntax grammar. We define a VL for specification architectures by composing VLs for graphs and for P/T nets enhanced by Petri net morphisms. From this VL definition a syntax-directed editor is generated supporting the editing of consistent specification architectures. Local and global changes of a specific specification architecture then can easily be defined as transformation rules in our VL and visualized in the GenGED environment.

1 Introduction

The need for continuous development of software systems results mainly from the changing demands of the market and the technological advances. We suggest a layered approach that allows the simultaneous description of a system on an architecture and a specification level. Both levels are presented using an adequate visual modeling technique. Based on this approach we suggest a rule-based description of the model's evolution.

Two-Level Visual Design of Distributed Systems Software architectures describe the different ways a system can be built. The larger a system is the more important this level of description is. Otherwise the detailed specification of the system, namely the different models, are indispensable. Our approach integrates these two levels and hence shows their relation. We propose a two-level representation for both the architecture and the specification of subsystems. This allows

* This work is part of the joint research project "DFG- Forschergruppe Petri-netz-Technologie", supported by the German Research Council (DFG), and the GRAPHIT project, supported by DLR and CNPq.

the abstract representation as a graph as well as the detailed specification of the subsystems based on some adequate visual specification technique. The architecture is given as a specific graph whereas at specification level the models are given in terms of a visual modeling technique. Fig. 1 illustrates the two-level concept. The above graph represents the architecture, whereas the graph in the bottom represents the specification level given by different visual models and their relations. These visual models may be given in terms of graphs (then we have distributed graphs in the sense of [14]), Petri nets, algebraic specifications etc. The meaning of such a specification architecture is the overall specification it is intended to describe. Hence we define the semantics of a specification architecture as the composition of the subspecifications according to the architecture graph.

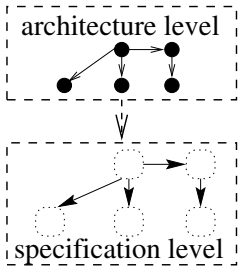

Fig. 1. Two Levels

Rule-Based Model Evolution In order to tackle the problem of changing large and complex systems we use rules to transform our two-level description. These rules can obviously describe two different kinds of transformation. Changing the architecture implies changes on the specification level as well. Changes of the models at the specification level may but need not induce changes of the architecture. Hence we introduce global changes that cause effects on both levels (cf. Fig. 2). In contrast, local changes only work on the specification level. We suggest only to conceive global changes as evolution steps. Moreover we can distinguish local rules that describe synchronous changes of several subspecifications.

Our approach generalizes the advantages of graph transformation to other specification techniques. We sketch the basic ideas of rule-based modification in terms of high-level replacement systems. The left-hand side L of a rule specifies the parts to be deleted and the right-hand side R those to be added. Note that in contrast to the graph transformation approach rules and transformations are *not* used for the description of the system behavior but for the description of its *changes*. The architecture level is represented by diagrams, where entities describing the subsystems are related to the specification of the corresponding subsystems at the specification level. We can distinguish two kinds of rules: local and global rules. Local rules imply identities at the architectural level but changes at the specification level. Global rules imply changes on both levels. Fig. 2 illustrates the basic idea. There we describe a change on both levels. A node on the architectural level is deleted and correspondingly the specification of that subsystem is deleted at the specification level. Moreover, a subsystem is added. Hence a new specification is added as well and the specification S is replaced by S'. The semantics of the

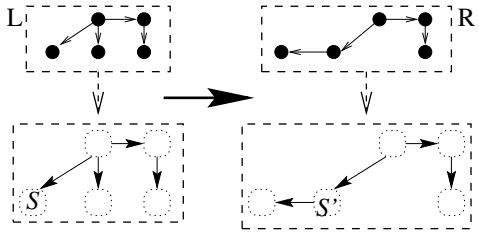

Fig. 2. Global Rule

specification architecture is preserved. This means that the composed specification from the resulting specification architecture is the same as transforming the semantics of the source architecture. This ensures the compatibility with the usual transformation of specifications in terms of high-level replacement systems.

Related Work Several different research areas overlap with our work including architecture design techniques, architecture transformation, distributed systems engineering and evolutionary system development. As the main focus of our work lies in the evolution of visual models, the areas of software visualization and visual languages also relate to our approach. An overview on architecture description languages (ADLs) based on components and connectors can be found in [11].

In Object Coordination Nets (OCoNs) [5] a UML architecture description is combined with Petri nets specifying the component behavior. The OCoN environment supports the visual development of OCoNs but not their relation to the architectural level. Other examples for visualizing architecture and (restricted forms of) their evolution can be found in ADL environments [2]. None of these tools allow a generic description of the visual model as we suggest in our approach using GenGED.

We model evolution steps by graph transformation which is also subject to considerable research. Software architecture reconfiguration based on graph transformation is presented by Wermelinger and Fiadeiro in [15]. They introduce a uniform algebraic framework based on category theory where an architecture is given as a graph whose nodes are refined to programs. Reconfiguration steps are modeled by conditional graph rewriting rules. In [14], Taentzer introduces Distributed Graph Transformation. In this formal specification technique an architecture level (network graph) and a component level (local graphs) are distinguished. This work is extended in [7] integrating distributed graph grammars and consistency checking rules. Our two-level approach is based on this work but allows more flexible visualization techniques than graphs for both levels.

Related approaches are given in [6, 4, 9, 10] where software architecture graphs are transformed to adapt them to new requirements or to reduce the component interrelations. In our paper, we restricted to "editor transformation", i.e., changes are performed in the model editor. Another formal approach to software architecture transformation is given in [8]. Here properties of component interrelations (i.e., invariants and dependencies) are formalized by a modal logic to enable consistent modifications in evolution steps. These invariants also might be expressed within graph rules for visual modeling.

2 Formal Foundations

For our approach of a *Two-Level Visual Design of Distributed Systems* we use the basic ideas of the algebraic approach to graph transformations in order to suggest a rule-based description of system changes and evolution. Here we concentrate on the basic concepts of our approach and do not discuss their formal representation;

this is the topic of another paper submitted to ICGT [12]. So, this approach is based on [14].

In [14] the dynamic network topology of a possibly open distributed system is described on the architectural level whereas the evolving data and system structures in the local subsystems are given on the specification level. The specification level is related to the architectural level via common interfaces. Again, different visual software architecture modeling languages [13] should be supported by our approach for the architectural level, as well as common visual modeling languages for the specification level. Therefore, we generalize [14] in allowing arbitrary specification techniques instead of graphs for the specification level. Analogously to distributed graph transformations [14] the architectural and the specification level can be related by functors. The architectural level is represented by diagrams, where entities describing the subsystems can be related by corresponding diagram functors.

Main Concept 1 (Specification Architecture).
A specification architecture consists of an architecture graph $G = (G^N, G^E, s, t)$ and of specifications and specification morphisms. To be more precise we have for each node $i \in G^N$ a specification $\Delta(i)$ in a given specification category **Cat**. For each edge $e \in G^E$ from the source $s(e) = i$ to the target $t(e) = j$ there is a specification morphism $\Delta(e) : \Delta(i) \to \Delta(j)$ in the specification category **Cat**. This relation is expressed by a diagram functor $\Delta : \mathbf{FG} \to \mathbf{Cat}$ where **FG** is the category of finite graphs. $\Delta : \mathbf{FG} \to \mathbf{Cat}$ presents the diagram of the architecture graph G in the specification category **Cat**. $\triangle$

The semantics of an architecture can be considered as the composition of the sub-specifications. The gluing of specifications is achieved by the colimit construction (the precise definition is given in [12]). The involved morphisms constitute the way the gluing is done.

Main Concept 2 (Semantics of a Specification Architecture).
The semantics of a specification architecture $\Delta : \mathbf{FG} \to \mathbf{Cat}$ is given by the gluing of all sub-specifications along the morphisms. This can be achieved in the following way: We first construct the disjoint union of all sub-specifications for each node. Then we glue recursively those parts of sub-specification that are the target of morphisms with the same source. $\triangle$

Subsequently we discuss these two concepts in terms of an abstract example. That is we do not assume a specific specification technique; it might be graphs, Petri nets, algebraic specifications, COMMUNITY programs, or something else. The main feature is that we have distinct abstraction levels of representation. Namely, we present an architecture graph and corresponding diagrams of specifications, that is the left column in Fig. 3. Its formal denotation is depicted in the middle column and its functorial presentation in the right column. The rows denote the following: the top row depicts the architecture graph, the middle row the specification diagram and the bottom row the composed system, i.e. the semantics.

In Fig. 3 there is a simple graph in the category **FG**. This graph consists of nodes $\{1, 2, 3\}$ and the edges in between. It is the architecture graph which is mapped by the diagram functor Δ to the specification diagram $\Delta(G)$ in category **Cat**. The specification diagram $\Delta(G)$ consists of specifications $sp1$, $sp2$ and $sp3$ and two specification morphisms in between. The semantics is given in the lowest row. It is the result of the gluing, namely it is the specification $sp4$. A concrete example, where Petri nets are used as specifications, is illustrated in Fig. 4.

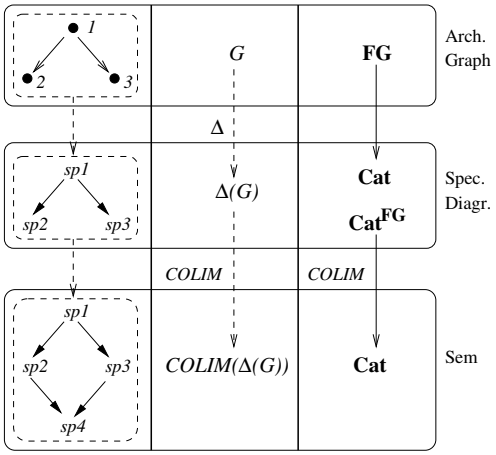

Fig. 3. Abstraction Levels of Specification Architectures

The transfer to high-level replacement (HLR) systems causes a representation independence in the sense that we may choose the specification technique. As in [14] we need to ensure specific conditions in order to have pushouts of specification architectures or more precisely pushouts of diagram functors. Nevertheless, we have a HLR-category (namely HLR0), and we have transformations provided that the pushout conditions are satisfied. In [14] gluing condition and applicability are given explicitly. We obtain the well-known transformations in the double-pushout approach. We distinguish two levels, namely the level of the architecture graph and the level of the specifications. Hence we can qualify different types of rules; those that leave the architecture intact are called local rules. Those rules that change the architecture graph are called global. Global rules necessarily change the diagram of specifications and hence may change the specifications themselves. Analogously to [14] we can identify further those rules that describe synchronized changes of sub-specifications as special class of local rules. Examples are given subsequently in Section 3.

The compatibility with the semantics of the architecture, that is the composition of all sub-specifications, is ensured in [12]. This result is crucial as it relates our approach to the usual transformation of specifications. Hence it guarantees that the result of an architecture transformation is the same as the corresponding transformation of the composed specification.

Main Result in [12]

(Compatibility of Semantics Construction with Transformation). Given a transformation of a specification architecture $\Delta_G \overset{p}{\Longrightarrow} \Delta_H$ with $p = (\Delta_L \leftarrow \Delta_K \rightarrow \Delta_R)$ then we have as well a transformation

$$COLIM(\Delta_G) \overset{COLIM(p)}{\Longrightarrow} COLIM(\Delta_H)$$

with $COLIM(p) := (COLIM(\Delta_L) \leftarrow COLIM(\Delta_K) \rightarrow COLIM(\Delta_R))$. $\triangle$

3 Example

As running example we use the well-known specification of a producer/consumer system. This example is (like the reader/writer protocol) one of the basic models for communication-based systems: two independent agents (the *producer* P and the *consumer* C) communicate via a channel (the *buffer* B). The producer sends messages (writes) to the channel, and the consumer receives (reads) them from the channel[1].

In our example illustrated in Fig. 4 the sub-specifications are place/transition (P/T) nets, so the corresponding category is the category of P/T nets. The nets are the objects and the P/T nets morphisms are the edges in between. According to Concept 1 the specification architecture consists of an architecture graph, specifications and specification morphisms: each node of the architecture graph indicates a sub-specification ($\Delta(i)$), namely a producer P, a consumer C and a buffer B, and each edge corresponds to a specification morphism in the corresponding category. The semantics of the specification architecture is given by the composition of the specifications along the given morphisms.

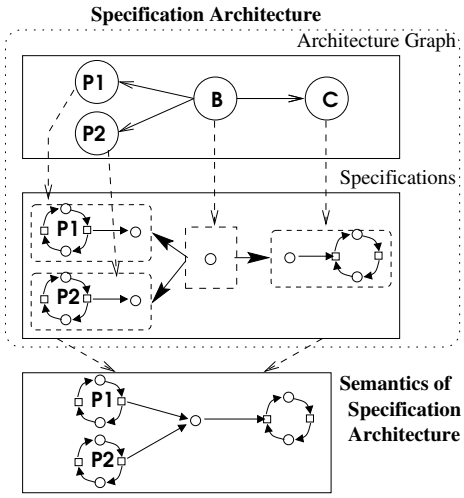

Fig. 4. A Producer/Consumer System

Now we extend the producer/consumer example introducing a global and a local rule. As mentioned before local rules induce changes on the specification level only, whereas global rules imply changes on both levels, namely the specification level and the architecture level. Global rules usually are rules that allow for the insertion and connection of new producers, consumers, and buffers.

Fig. 5 illustrates a DPO-rule for inserting a new consumer. Please note that the rule consists of architecture graphs (and graph morphisms) as well as specifications (and specification morphisms) according to Concept 1. The rule's left-hand side L demands for its application the existence of a buffer. This buffer is preserved by the rule

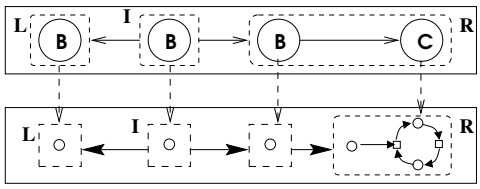

Fig. 5. Global Rule adding a new Consumer

[1] Please note that in our example the arrows describe *"being used"* relations.

indicated by the component I. By the right-hand side R a new consumer C is inserted and connected to the buffer.

Similar to the rule in Fig. 5 are rules for inserting a new buffer or a new producer. Deletion of components is achieved reversing the corresponding rules, i.e. exchanging the left- and the right-hand sides of the rules.

In Section 2 we mentioned the possibility to describe local changes of sub-specifications. Fig. 6 illustrates such a rule changing the specification of a producer. The rule needs for its application the existence of a specific producer specification presented in the rule's left-hand side L (the producer loads in each production cycle arbitrarily one of the two buffers). The two buffers and a part of the producer are preserved indicated by I (one transition and some edges are removed). By applying this rule a new transition and edges are inserted as shown in the rule's right-hand side R (in each cycle both buffers are loaded).

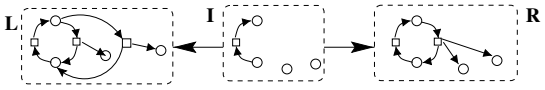

Fig. 6. Local Rule changing the Specification of a Producer

As stated in the main result in Section 2 we obtain the same result, independent whether we first construct the semantics and then apply the rules, or if we first apply the rules and then construct the semantics. For illustration let us consider Fig. 7. Fig. 7 (a) shows the application of the global rule in Fig. 5 to the specification architecture in Fig. 4. This yields the insertion of a new consumer called $C2$. Fig. 7 (b) shows the corresponding semantics.

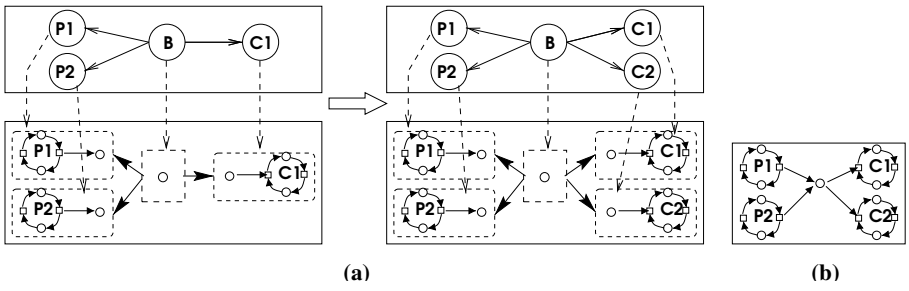

Fig. 7. Application of the Global Rule in Fig. 5

The brief example of this section illustrates the formal basis of our approach. For examples using other specification techniques, the reader is referred to the discussion in [12]. In the following section we suggests a graphical editor for two-level architectures using GenGED .

4 Specifying Two-Level Architectures using GenGED

The GenGED approach developed at the TU Berlin [1] allows for the generic description of visual languages (VLs) and visual environments based on VL specifications. The simplest form of a VL specification we consider here consists of a visual alphabet and a visual syntax grammar. The definition of a VL specification is based algebraic graph transformation and graphical constraint solving. VL sentences (diagrams) can be derived by applying the grammar rules in the syntax grammar to its start diagram.

In general a diagram consists of a set of *symbol graphics* that are spatially related. We offer graphical constraints for these spatial relationships. Symbol graphics and graphical constraints concern the layout of diagrams, called *concrete syntax*. The logical part of a diagram (its symbols independent of the concrete layout) is called *abstract syntax*. The combination of both syntactical levels, called *visual syntax* level, is represented by attributed graphs.

4.1 The Visual Alphabet

A *visual alphabet* establishes a type system for *symbols* and *links*, i.e. it defines the vocabulary of a VL. It can be represented as a type graph. Here as well we distinguish the abstract and the concrete syntax level. Symbol graphics and graphical constraints specify layout conditions. In addition to logical (data) attributes like, e.g., a name for a place in a Petri net, symbol graphics define a further kind of attributes for all abstract symbol nodes.

Graphical constraints specify layout conditions. They are given by equations over constraint variables denoting the positions and sizes of graphical objects. The set of all constraint variables and constraints define a constraint satisfaction problem (Csp) that has to be solved by an adequate variable binding in a diagram conforming to the alphabet.

We now develop the visual alphabet for the two-level specification architecture language. We begin by defining two separate alphabets, one for a special kind of graphs (the architectural level) and one for place/transition (P/T) nets according to the specification level. As a second step we enhance the P/T net alphabet in order to be able to express P/T net morphisms between different nets and combine the two alphabets.

Definition 1 (Alphabet for Architecture Graphs).
In architecture graphs, we have only the symbol types Node and Edge. Nodes are drawn as circles and may be attributed by strings (their names) which are placed inside the node circles. Edges are directed arcs connecting two nodes. The visual alphabet of the language of architecture graphs is shown in Fig. 8 (a). The dashed arrows mark the connections of the abstract syntax and the concrete syntax level. Link constraints are illustrated by dotted arrows between the symbol layouts. △

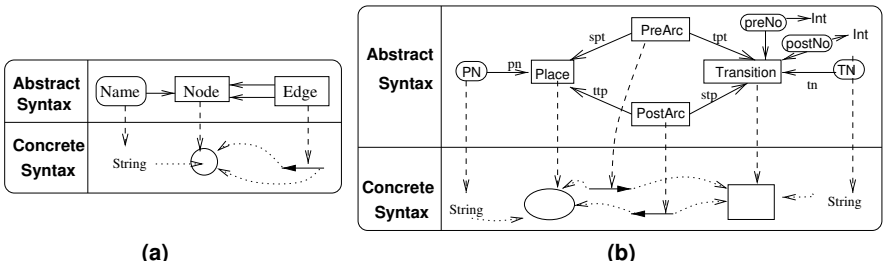

Fig. 8. Alphabets for Architecture Graphs (a) and for P/T Nets (b)

Definition 2 (Alphabet for P/T Nets).
The visual alphabet of the P/T net language, called P/T net alphabet, is illustrated in Fig. 8 (b). We have attribute symbols for place names PN and for transition names TN which are linked to the symbol types Place resp. Transition. Each name is given by a String data type. To compute the number of places in a transition environment (pre domain and post domain), a transition symbol carries the integer attributes preNo and postNo. These numbers are needed later to control the insertion of mappings between transitions.

We distinguish arcs that run from places to transitions (PreArc) and arcs that run from transitions to places (PostArc). Both kind of arcs have a certain source and target symbol where they are linked to (depicted by the edges spt, tpt, ttp, stp, short for source/target of place-transition arc resp. transition-place arc). To keep the alphabet simple, we restrict to unmarked P/T nets where the uniform arc weight is "1", and therefore arc inscriptions are omitted. The constraints force a specific layout of nets typed over the P/T net alphabet. For example, one constraint ensures that the place name is always "near" the ellipse (the symbol graphic for Place symbols). △

Sentences over the P/T net alphabet defined so far are unmarked P/T nets (to keep the example simple; for a specification of marked P/T nets in GenGED including their firing behavior see [3]). The visual language we aim to specify, also should provide means to express morphisms between different P/T nets. This means, we have to enhance the alphabet from Def. 2 to include sentences consisting of more than one net, and to allow morphism between different nets. We call such a relation of different P/T nets *P/T net systems*. Hence, for the definition of an alphabet for our two-level specification architecture language, we combine the alphabets for architecture graphs and for P/T net systems.

Definition 3 (Alphabet for Two-Level Language).
We introduce the symbol Net into our P/T net alphabet where an instance in a sentence is linked to all objects (e.g. places, transitions) belonging to the same net. The symbol Net is visualized by a dashed frame around all its net objects. Furthermore, we introduce the symbol Morphism linking one Net symbol to another. Such a morphism is visualized by a double arrow from one dashed Net frame to another.

On the Petri net level a morphism is given by mappings from all net objects of the source net to net objects in the target net. The mappings have to be type compatible (places are mapped to places, transitions to transitions and arcs to arcs of the same type). I.e., mappings have to be compatible with structure (the source/target node of an arc is mapped to the source/target node of the arc's image) and they have to preserve the firing behavior (the transitions are mapped to transitions with the same number of ingoing and outgoing arcs only). We ensure the type compatibility by introducing the symbol types PlMap, TrMap, PreMap, PostMap for the different net objects. The other conditions are ensured by the grammar rules introduced later for modifying diagrams of the language.

The resulting alphabet for P/T net systems now is combined with the alphabet for architecture graphs, i.e. each node from the architecture level is linked to a net at the specification level. The complete alphabet of the combined languages is shown in Fig.9.

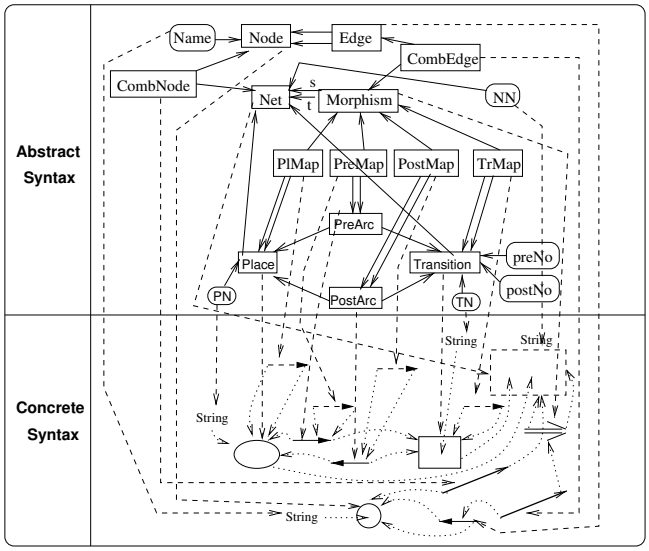

Fig. 9. Alphabet for the Two-Level Language

△

An example for a sentence over the two-level alphabet is shown in Fig. 10. At the architecture level we have two nodes, namely a producer and a buffer which are connected by an edge. Accordingly there are two P/T nets at the specification level related by a morphism: the P/T net modeling the producer is related to the second P/T net consisting of one place modeling the buffer. The morphism consists of one mapping of type PlMap between the buffers of the P/T nets, only.

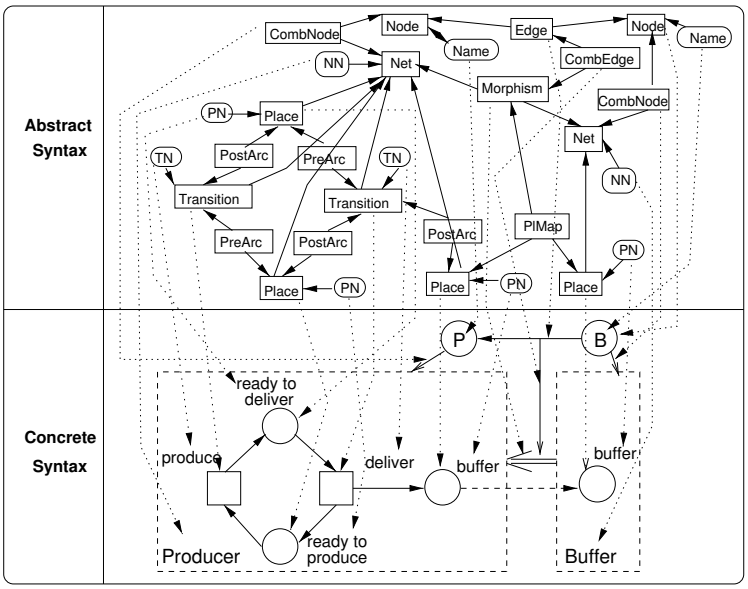

Fig. 10. Visual Sentence over the Two-Level Alphabet

4.2 The Visual Syntax Grammar

The visual alphabet depicted in Fig. 9 is the basis to define the syntax grammar for our two-level language for architecture specifications. The syntax grammar is represented by an attributed graph grammar: it consists of a *start diagram* and a finite set of *rules*. The start diagram and both sides of a rule are diagrams typed over the alphabet, as well as the diagrams which can be derived by applying grammar rules. In the rules we use in addition to the left-hand rule sides so-called negative application conditions (NACs) which restrict the application of a rule. An NAC is a graph containing a forbidden graph pattern. The rule must not be applied to a sentence if there is a match from the NAC to the sentence, i.e. the forbidden pattern is found in the sentence. Moreover, rule applications can be restricted by boolean conditions over attributes.

In Def. 4 we define the syntax grammar for the two-level language combined by graphs and P/T net systems. This definition has to be fixed once and allows the generation of a syntax-directed editor for two-level models (specification architectures) in GenGED. An example for a global syntax rule is the rule adding a new component node which means at the same time to add a new (initially empty) specification net at the specification level. Examples for local syntax rules are rules adding places or transitions to an existing P/T net.

Let us consider the local rules first. They operate only on the specification level and can thus be considered as syntax rules for the VL of P/T nets over the alphabet shown in Fig. 8 (b).

Definition 4 (P/T Net Syntax Grammar).
Fig. 11 illustrates a syntax grammar for our P/T net language based on the visual alphabet in Fig. 8 (b). In this P/T net grammar the start sentence consists of the empty net (i.e. a single Net node).

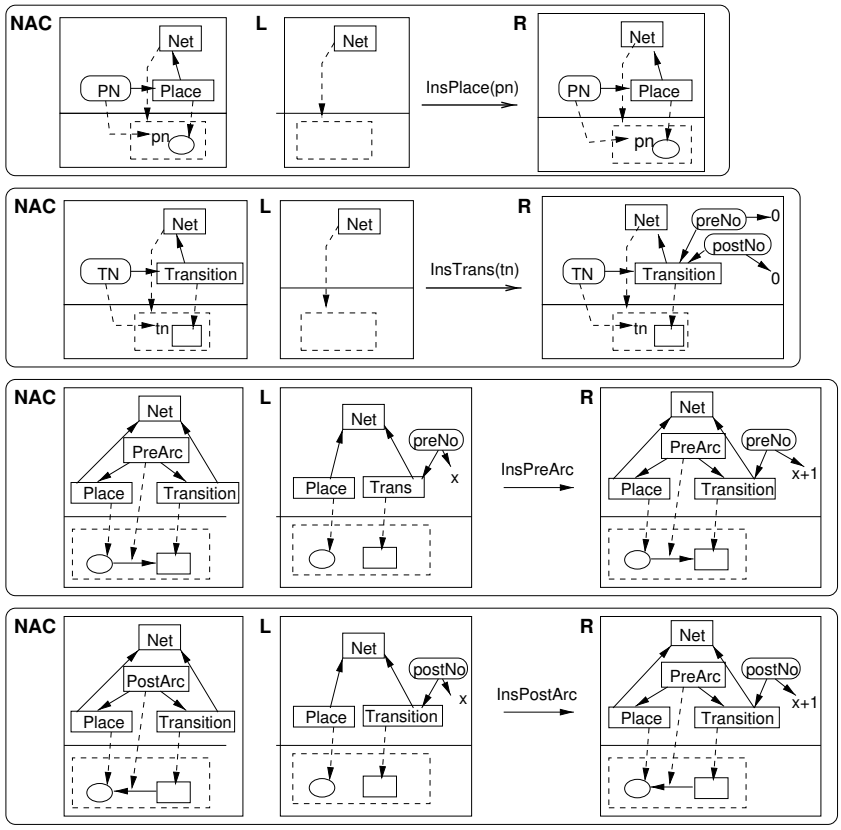

Fig. 11. Visual Syntax Grammar for the P/T Net Language

The first rule InsPlace(pn) supports the insertion of a place together with a place name; the NAC requires that a place with the user-defined name given in the parameter variable pn is not existing so far in the net the rule is applied to. The second rule analogously supports the insertion of a transition symbol. Here the integer attributes preNo and postNo are initialized by 0 as the newly inserted transition is not yet connected to any places. The next two rules allow for the insertion of arcs, either running from a place to a transition (insPreArc) or running from a transition to a place (insPostArc). The respective counter is incremented for the transition. The NACs forbid the application if there is already such an arc.

Graphical constraints (dotted arcs in Fig. 11) ensure that arcs connect places and transitions in a proper way and a name of a net object is placed near the object. △

A VL is generated by applying the syntax grammar rules. Up to now we defined the VL of P/T nets consisting of all diagrams over the P/T Net Alphabet as given in Fig. 8 (b) which can be derived from the start diagram by the local syntax grammar rules given in Fig. 11. Let us now extend our visual language by global rules concerning both the architecture level and the P/T net level.

Definition 5 (Syntax Grammar for Two-Level Models).
The start graph of the combined syntax grammar for two-level models is empty reflecting that the editing process starts with an empty editor panel. Fig. 12 shows the global grammar rules.

Rule InsComponent inserts a new component to the architecture level combined to the insertion of a new (empty) net specification. The NACs ensure that there exist no node and no net in the specification so far with the same names as the currently inserted ones (uniqueness of names). The other rules deal with the insertion of mappings and morphisms. Note that we do not provide a rule to insert a morphism. This is done implicitly by the mapping-inserting rules. For the insertion of mappings between places or transitions of different nets we distinguish two cases: either there is already a morphism between the two corresponding nets (due to previous mappings) or there is no morphism (first insertion of a mapping). In case a mapping is inserted for the first time, the morphism between the two corresponding nets has to be inserted together with an edge connecting two nodes at the architecture level (see e.g. rule InsP|Map1). If a morphism already exists, the mapping simply is added, but the architecture level is not changed (see e.g. rule InsP|Map+). This distinction is realized by the respective NACs and works analogously for all types of mappings.

The rules InsTrMap+ and InsTrMap1 for the insertion of transition mappings are analogical to the insertion of place mappings but contain additionally a rule application condition that ensures that a mapping between transitions is inserted only if the number of places in the pre domain preNo and the number of places in the post domain postNo are the same for both transitions such that the firing behavior of the source net is preserved by the net morphism. We only depict InsTrMap1 in Fig. 12, the step to InsTrMap+ is obvious. Note that arcs can be mapped only if there exist mappings between their start and end nodes (see rule InsPreArcMap). The analogical rule InsPostArcMap is omitted in Fig. 12.

The complete syntax grammar for two-level models now consists of the union of the set of global rules as explained above and the set of local grammar rules shown in Fig. 11. △

The visual two-level language thus consists of all diagrams over the two-level alphabet shown in Fig. 9 which can be derived from the empty start diagram by the complete syntax grammar consisting of the local rules (see Fig. 11) and the global rules (see Fig. 12).

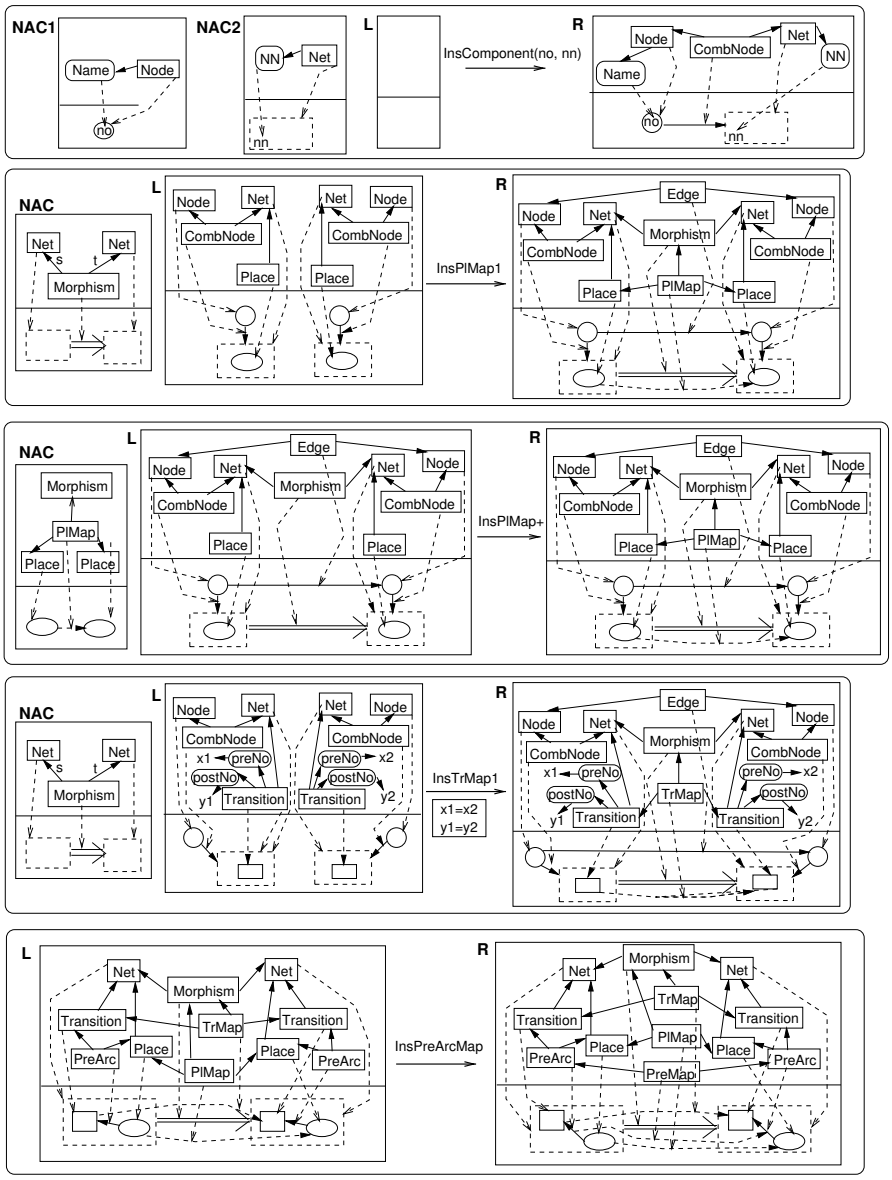

Fig. 12. Global Grammar Rules for Two-Level Modeling Language

In order to define specification transformation steps in GenGED, the syntax rules of our two-level language can be used to define the left-hand and right-hand sides of the desired transformation rules. Thus a specific model transformation grammar can be visually defined as well, and rules like the transformation rule adding a new consumer (see Fig. 5) may be constructed and applied in the GenGED environment.

5 Conclusion

We have presented an approach for architectures of specifications that is based on diagram functors. The emphasis of this paper has been on the illustration of the main concepts and the implementation of the example within the GenGED environment. We have given a simple example, namely a producer/consumer system. First the main concepts of our approach have been exemplified there, then we have used this example to illustrate the visualization of specification architectures using the GenGED environment.

This approach is general enough to provide a framework for various specification techniques. It can be employed for textual as well as graphical ones. Examples of specification architectures comprise architectures of COMMUNITY programs [15], distributed graph transformation systems [14], specification architectures of algebraic high-level nets [12]. Hence the question of implementation of changes can be attributed to the question of the specification techniques. Obviously a programming language as COMMUNITY is closer to the real implementation as a P/T net. So it is clearly an important and challenging task to ensure compatibility of model transformation with its implementation. But it is dependent on the underlying specification technique. For methodological questions it needs several case studies using various specification techniques to extract a specification independent process model. Hence we have not yet concentrated on this question, but merely have distinguished between local, synchronizing and global transformations. Further research can either consider specific specification techniques or concentrate on the general approach. The first case includes all semantic aspects, as consistent changes of behavior, as preservation of properties, and as mentioned above compatibility with realization. The second case focuses on structural questions as compatibility with (categorical) structuring techniques, as parallel and sequential independence, and so on.

References

1. R. Bardohl. GenGEd – *Visual Definition of Visual Languages based on Algebraic Graph Transformation*. Verlag Dr. Kovac, 2000. PhD thesis, Technical University of Berlin, Dept. of Computer Science, 1999.
2. P. Donohe, editor. *Software Architecture*. Kluwer Academic Publishers, 1999.
3. C. Ermel, R. Bardohl, and H. Ehrig. Generation of Animation Views for Petri Nets in GENGED. In Ehrig et al (eds.), *Advances in Petri Nets: Petri Net Technologies for Modeling Communication Based Systems*, Springer, LNCS, 2002. To Appear.
4. H. Fahmy and R. Holt. Using Graph Rewriting to Specify Software Architectural Transformations. In *Proc. of Automated Software Engineering (ASE 2000)*, 2000.
5. H. Giese, J. Graf, and G. Wirtz. Modeling Distributed Software Systems with Object Coordination Nets. In *Proc. Int. Symposium on software Engineering for Parallel and Distributed Systems (PDSE'98),Kyoto, Japan*, pages 107–116, jul. 1998.
6. Holger Giese and Jörg P. Wadsack. Reengineering for Evolution of Distributed Information Systems. In Scott Tilley, editor, *3rd International Workshop on Net-Centric Computing (NCC 2001), May 14, 2001; Toronto, Canada*, May 2001.

7. M. Goedicke, T. Meyer, and G. Taentzer. ViewPoint-oriented Software Development by Distributed Graph Transformation: Towards a Basis for Living with Inconsistencies. In *Proc. 4th IEEE Int. Symposium on Requirements Engineering (RE'99), June 7-11, 1999, University of Limerick, Ireland*. IEEE Computer Society, 1999. ISBN 0-7695-0188-5.
8. M. Große-Rhode, R. Kutsche, and F. Bübl. Concepts for the Evolution of Component-Based Software Systems. Technical Report TR-2000/11, FB Informatik, TU Berlin, 2000.
9. D. Hirsch, P. Inverardi, and U. Montanari. Graph Grammars and Constraint Solving for Software Architecture Styles. In *Proc. ISAW'98*, 1998.
10. D. Hirsch, P. Inverardi, and U. Montanari. Modeling Software Architectures and Styles with Graph Grammars and Constraint Solving. In *Proc. Working IFIP Conference on Software Architecture*, 1999.
11. V. Issarny, L. Bellissard, M. Riveill, and A. Zarras. Component-Based Programming of Distributed Applications. In *Distributed Systems*, pages 327–353. Springer-Verlag, LNCS 1752, 2000.
12. J. Padberg. Formal Foundation for Transformations of Specification Architectures. *Submitted*, 2002.
13. M. Shaw and D. Garlan. *Software Architecture - Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
14. G. Taentzer. Distributed Graphs and Graph Transformation. *Applied Categorical Structures*, 4(4):431–462, December 1999.
15. M. Wermelinger and J. Fiadeiro. A Graph Transformation Approach to Software Architecture Reconfiguration. In H. Ehrig and G. Taentzer, editors, *Proc. Joint APPLIGRAPH and GETGRATS Workshop on Graph Transformation Systems (GRATRA'00)*. TU Berlin, FB Informatik, TR 2000-2, 2000. Accepted to Journal of Science of Computer Programming.