Electronic Notes in Theoretical Computer Science 44 No. 4 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume44.html 15 pages

# Visual Design of Software Architecture and Evolution based on Graph Transformation

C. Ermel, R. Bardohl and J. Padberg

*Institut für Softwaretechnik und Theoretische Informatik*
*Technische Universität Berlin*
*Germany*[1]

**Abstract**

The paper suggests a two-level approach to describe visually software architectures and their evolution. One visual modeling formalism is used to describe the architecture level while another is used to model the behavior of each component (component specification level). Graph transformation is applied for both levels to describe the modeling formalisms and the model evolution in a formal way. The graph transformation based visual modeling approach GenGEd allows the designer to define the concrete and abstract syntax of each formalism. Thus, the choice of the visual formalisms is not restricted to existing ADLs or modeling languages but new visual languages can be defined by the user according to the problem domain. The architecture and component specifications are related over their abstract syntax. Here, it is possible to enforce coherence between the two levels while the user is changing the model in the editor generated by the GenGEd tool. The ideas are illustrated by a small example using UML-like class diagrams for the architecture and Petri net like networks for the component behavior.

## 1 Introduction

The increasing complexity of software systems calls for methods and tools for easing their development. In component-based software engineering, problems like component interaction/migration and the overall design (architecture) of the system must be dealt with.

The software architecture research field is quite new (since the early 90ies) and there still does not exist a consensus on what should be the abstract description of a software architecture. However, guidelines have been already provided. In particular, it is accepted that an architecture definition decomposes into the types of elements *component* (computation unit or data store),

[1] E-mail: {lieske,rosi,padberg}@cs.tu-berlin.de

*connector* (interaction unit) and *configuration* (architecture) [14]. As the notion of a component varies in literature, we here give the (implementation oriented) definition of Szyperski [19] which covers the most important aspects for the development of complex systems:

*"A software component is a unit of computation with contractually specified interfaces and explicit context dependencies only. A software component can be developed independently and is subject to composition by third parties."*

An appropriate architecture representation has to follow this decomposition paradigm and to support component interactions using an appropriate visualization. A visual tool environment for the visual description of an architecture should be available. Moreover, the tool environment should allow the user to perform syntactical checks on his model concerning e.g. the import/export relations between the component interfaces.

Architecture Description Languages (ADLs) [2] offer a well-defined set of notations for the description of an application architecture. Basically, an ADL allows the developer to describe the abstract organization of his system in terms of coarse-grained architectural elements, independent of the element's implementation details.

Apart from the static architecture description, a major challenge in developing and maintaining distributed software is *Evolutionary Change*, namely the problem how to handle changes in an existing system's structure (software evolution). Often, it is desirable to extend existing software in an unforeseen way, e.g. the addition, removal or change of components or entire subsystems should be possible without having to plan for this design step before the system is started.

In [15], Kramer and Magee state that the key feature of change management is a clean separation of functional concerns of individual application processing components on the one hand and architectural coordination of components on the other hand. Additionally, we expect that a method for change management should have a formal background to allow mathematical reasoning and serve as a basis for tool support. The system architecture and the internal behavior of components should be specified by appropriate visual modeling languages. The method should relate changes on the architectural description level to the underlying component specifications and vice versa to allow syntactical checks of change properties.

Hence, we suggest an approach of employing visual modeling techniques on two levels, the architecture and the component specification. Evolution steps are specified as rule-based modifications on both levels. The focus of our concept is the independence from a specific visual modeling technique, taking into account the large variety of common architecture description languages and visual specification techniques. The user therefore may define a subset of an existing ADL or own visual modeling languages adapted to his purpose. Thus, we aim at a generic visual modeling approach for model evolution at the design level.

Our approach is based on GenGEd [1], a visual modeling approach supporting the syntax specification of arbitrary visual modeling languages. GenGEd is based on the formal specification technique *Algebraic Graph Transformation* [3], thus allowing rule-based modifications of the system behavior as well as the architecture. The formal basis also gives rise to analyzing properties of components and their connections (see e.g. [21] for a first discussion). GenGEd admits the visual specification of a visual language and (up to now) automated editor generation for diagrams of this language. Graphical constraint solving techniques are used to compute the layout of diagrams. GenGEd has been successfully applied to a variety of visual modeling techniques, including simplified versions of UML class diagrams, statecharts, Nassi-Shneiderman diagrams and Petri nets.

Figure 1 illustrates our two-level concept. For the architecture level we use simple graphs. The component specification level allows arbitrary visual models and their relations. These visual models may be given in terms of graphs (then we have distributed graphs in the sense of [20]), Petri nets, algebraic specifications etc. In order to tackle the problem of changing large and complex systems we identify two different kinds of change. 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. Figure 1 shows a graph transformation rule specifying a global change step. The left-hand side of a rule ($L$) specifies the parts of a graph the rule can be applied to. If the right-hand side ($R$) does not contain some of these parts, they are deleted by the rule. If $R$ contains new parts, these are added. Parts common to both sides are preserved by the rule.

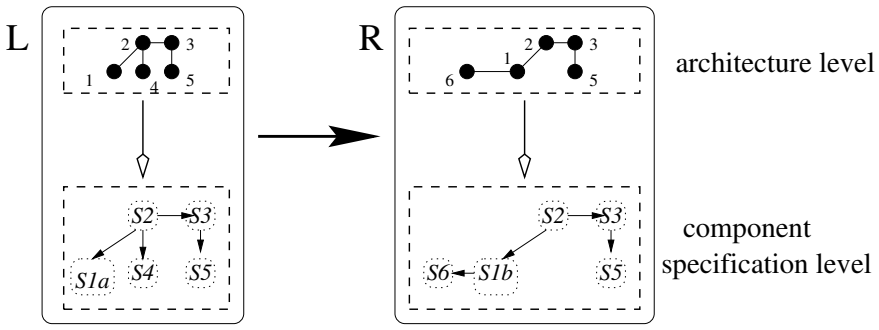


Fig. 1. Two-level approach to model evolution

The rule in Figure 1 describes changes on both levels. At the architectural level, node 4 is deleted and node 6 is added. Correspondingly, the specifications of their subsystems are deleted (*S4*) and added (*S6*) at the specification level. The specification *S1a* in $L$ is changed to the specification *S1b* in $R$.

The paper is organized as follows: In Section 2, we illustrate our ideas of rule-based model evolution using the specification of a simplified UNIX print server as example. We use a language similar to UML class diagrams as architecture design language and Petri nets to specify the component behavior.

Section 3 presents the use of GenGEd for our approach and discusses the formal backgrounds of our concepts.

*Acknowledgements*

This work is supported by the German Research Council (DFG) as part of the work of the Forschergruppe Petri Net Technology, by the German-Brazilian Cooperation for Graphical Support of Formal and Semi-Formal Methods for Software Specification and Development (GRAPHIT), and by the German Federal Ministry of Education and Research (BMBF) as part of the research project Continuous Engineering for Evolutionary I&C Infrastructures. We also would like to thank our anonymous referees for their most valuable hints.

## 2 Example: A Print Server

A component called PrintServer is part of a (distributed) operating system and offers services to print documents. To realize these services, the component relies on other components' services. A system user calls a service directly by a system command or indirectly via an application. A print job is queued in a printer queue (invisible for the user) and is processed later. The user may call for a list of his currently running print jobs (service lpq), add documents to the printer queue (service lpr) or remove a single job from the list of his print jobs (service lprm). The class PrintServer offers (exports) these services. For their realization, a PrintServer component needs the user administration service checkAccess from the SystemAdmin component to check whether a user is authorized to print or not. If not, an error message is sent to the user by email using the mail service offered by the MailSystem component. The components Printer and PriorityQueue offer services for printing and for adding/removing print jobs to/from the priority queue.

From this scenario, we derive the classes for components of our system yielding the architecture level illustrated in Figure 2.

A graphical notation similar to UML class diagrams is employed. Components offer services via export interfaces which are based on a contract principle: an ADT-like signature [4] specifies in the traditional way the names and required service calls. Services needed from other components are listed in a component's import interface. Here, only method names are given, not their complete signature. Hence, a component is divided into the three blocks *export interface* (upper block), *component name* (middle) and *import interface* (lower block). Association arrows represent import/export relations on component services. We do not consider inheritance relations here in order to keep our example simple.

For the specification of the component behavior, we use Open Algebraic High-Level Nets [17]. Each of the PrintServer component's offered services (an operation in a class signature in the architectural level) as well as inter-

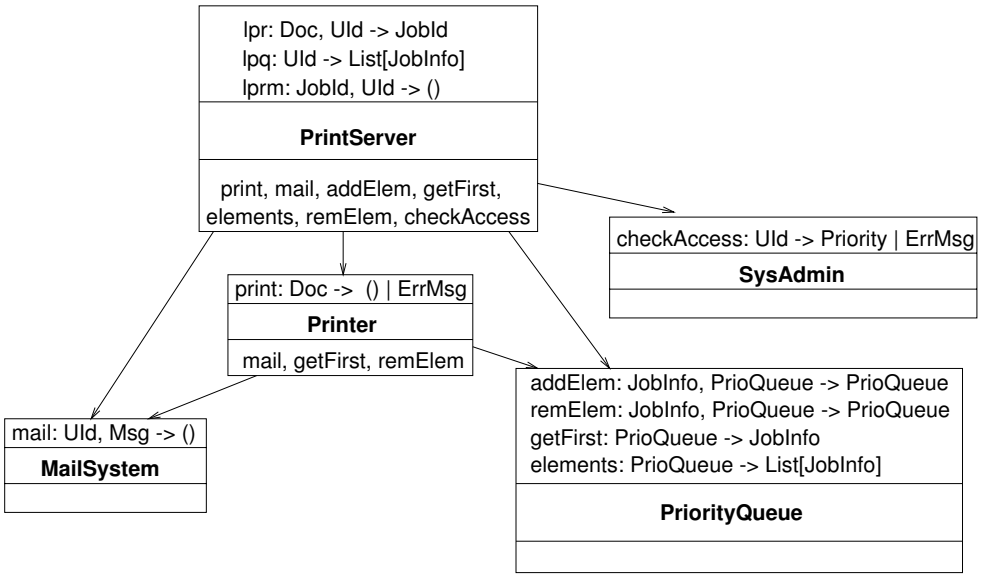


Fig. 2. Architecture level of the print server example

nal services are represented as so-called service nets at the specification level. Hence, a component class corresponds to a number of service nets. Input and output parameters of a service call are made explicit in the corresponding service nets by special input and output places. The service nets show concurrency and synchronization of methods as well as the exclusive or shared use of resources. There may be a hierarchy of service nets starting by the interface services on top which rely on internal services again represented as service nets. In our example, the component PrintServer is specified by four service nets, three for the exported services lpr, lprm and lpq and one for the internal service printFirst realizing the printing of the first job in the priority queue. The use of services from other components is depicted as gray transitions with the same name as the corresponding service of the exporting component. Analogously, shared resources (e.g. the priority queue) are represented by equally named gray places in different nets. Operations represented as arc inscriptions in algebraic high-level nets are formally defined in corresponding algebraic data type specifications (see e.g. [16]). As we chose self-explaining operation names (or variables) we here omit the formal data type specifications for our nets.

The service net for lpr (Fig. 3) shows how the user access to a printer is checked by the SystemAdmin method checkAccess at first. If the user is allowed to print, checkAccess computes a priority, and an object of type JobInfo is generated containing the user id, the document and a new job id. This JobInfo object is added to the priority queue (a shared place which is also used in all other PrintServer service nets). The method returns the job id or, if the user is not allowed to print, an error message is sent to him by email.

In the net for lpq (Fig. 4) a list of current job infos in the priority queue is read by the method elements from the component PriorityQueue. The user

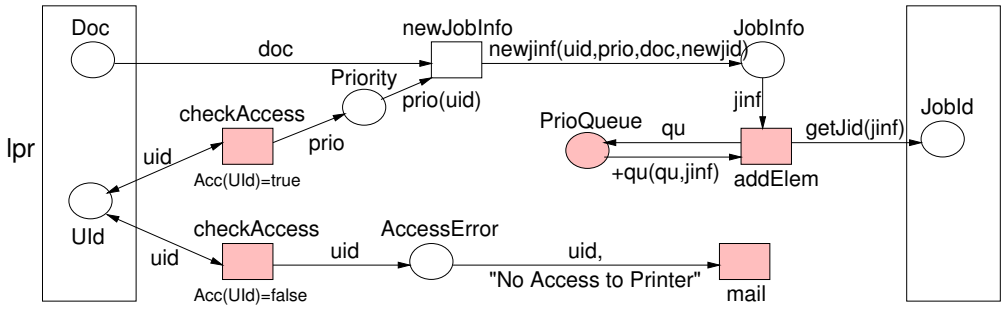


Fig. 3. Service net for the method lpr

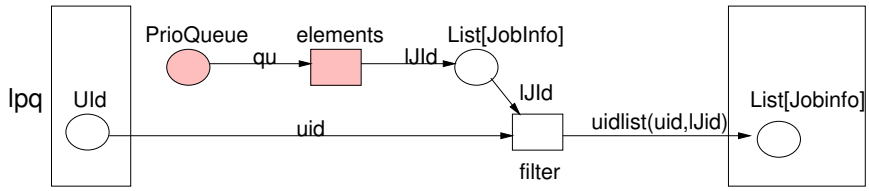


Fig. 4. Service net for the method lpq

ids of the entries in this list are compared to the current user id on the input place and a new list containing only those entries with the same user id is computed and returned by the method.

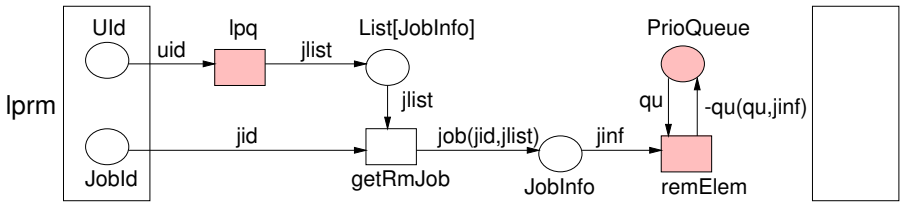


Fig. 5. Service net for the method lprm

The net for lprm (Fig. 5) uses the method lpq to compute a list of the user's print jobs in the priority queue. From this list, the entry with the corresponding job id is filtered and removed from the queue.

The internal service net for printFirst (not depicted here) realizes the printing and removing of the first job of the priority queue. This service is independent of system user calls and is running continuously – in parallel to the other services – as long as there are jobs in the queue.

Evolution steps are modeled by changes within a service net (local changes) that do not alter the input and output parameters, or by changes of the architecture (global changes), e.g. by removing/adding components or by changing a method signature. Global changes at the architecture level influence the component specification level. Whenever components are added or removed, method nets have to be added or removed, too. When a method signature is changed, the service has to be adapted accordingly. Examples for global changes are e.g. the addition of a Scanner component class (plus the needed service nets), or the removal of the MailSystem component. Here, all mail service transitions also have to be removed out of service nets. A local change

would be to alter the behavior of the lpq method to return the complete job list of the priority queue and not only a single user's jobs. Here, only the internal net is changed whereas the method's signature remains unchanged.

As we use class diagrams at the architectural level we realize an abstract and open design: The addition of a new printer to our system would not change the model because a printer is an instance (object) of the Printer component class.

Obviously, a combination of both levels is necessary to decide whether a change has global effects or not and to perform consistency checks. For example, it should be possible to ensure that the methods in export interfaces at the architecture level are consistently refined to service nets at the component specification level and that a method's signature corresponds to its input/output places in the corresponding service net. Hence, in the next section we introduce the GenGEd approach to formally specify model evolution and show how the combination of both levels is realized and forms the basis for checking dependencies in model evolution steps.

## 3 The Visual Environment GenGEd

GenGEd is a visual environment supporting the visual specification of visual languages [1]. The visual specification is used to generate a graphic editor for the manipulation of visual sentences over the specified language. Visual sentences are diagrams (or instances) of a visual language. The specification of visual languages is given by a visual alphabet and a visual grammar. We distinguish the abstract syntax and the concrete syntax of a visual language. The abstract syntax offers the basic relation into the application domain. Here, it is possible to define semantic dependencies of language elements. The concrete syntax describes the layout of symbols and their links. For the transformation of instances (diagram manipulation) we use the Single Pushout approach to algebraic graph transformation [3]. This approach is implemented by the Agg system [5]. The vertices' graphical attributes are calculated by constraint solving mechanisms using the ParCon constraint solver [9]. Both Agg and ParCon are integrated in the GenGEd environment [1].

Our approach integrates two visual modeling languages using GenGEd. The integration is realized by the abstract syntax. Here, in addition to the language's elements, items are specified used for the combination of both languages. In our print server example from Section 2, these are the language of class diagrams for the architectural level and the Petri net language for the specification of component behavior. Thus, it is possible to relate items of the architecture level to items of the component specification level. For example, an exported method at the architectural level is always related to a service net at the specification level.

*Defining Language Elements: The Visual Alphabet*

Fig. 6 illustrates the visual alphabet of the two-level language. In the mid-

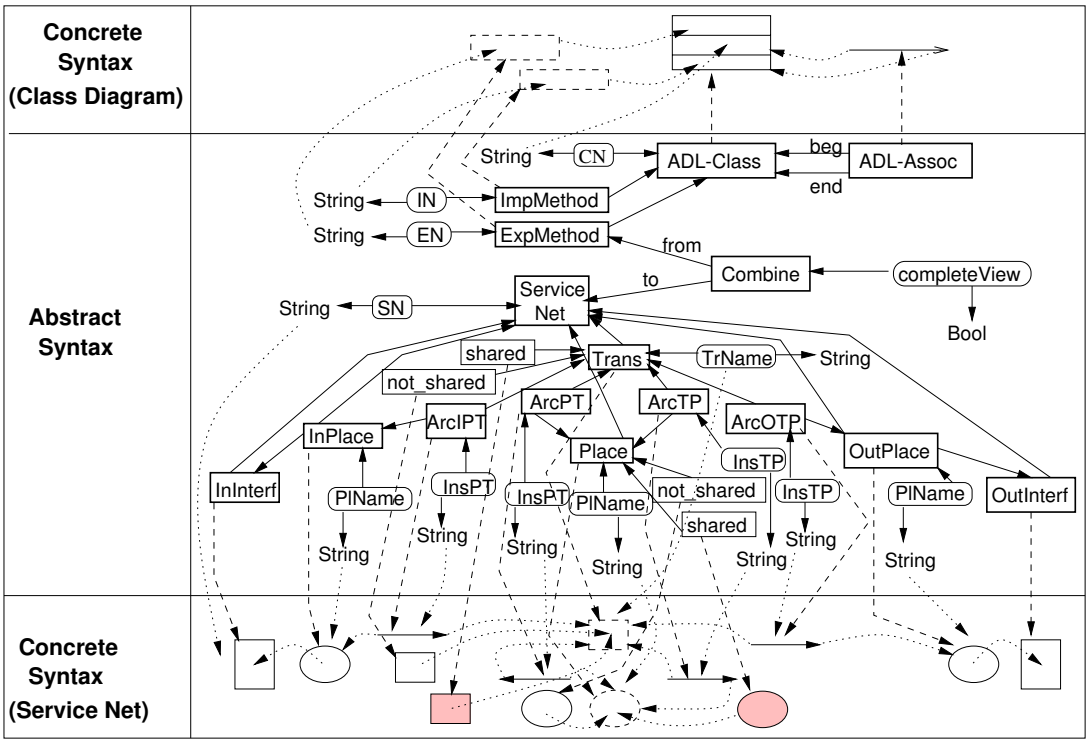


Fig. 6. Visual alphabet of the two-level language combining Class Diagrams and Petri Nets.

dle of Fig. 6, the abstract syntax of all language elements is illustrated. All lexical symbols comprising the abstract syntax are visualized by rectangles and all data attribute symbols by rounded rectangles. The upper part of the abstract syntax area contains the elements of the class diagram language and the lower part elements of the Petri net language. Both are connected along the element Combine that relates each exported method (element ExpMethod) to a service net (element ServiceNet). We have equipped the item Combine with an attribute symbol containing a boolean value completeView. This value is controlled by user interaction in the intended graphical editor for our combined language. The user interaction should be realized in the graphical user interface by a button the user can press if he wants to see the Petri nets for the methods (completeView = true) or to hide it and only see the architectural level (completeView = false). Thus, a simple view concept controlled by the abstract syntax can be realized as well.

The graphical layout of the abstract elements is shown in the top and the bottom of Fig. 6. In the GenGEd approach, the relation between abstract and concrete syntax is modeled by graphical attributes for each symbol and link in the abstract syntax. The graphic attribution is visualized by dashed arrows in Fig. 6. Graphical constraints between symbol graphics are visualized by dotted

arrows. For example, we require that an arrow between two components (the layout of ADL-Assoc) always begins (respectively ends) at the corresponding rectangles (the layout of ADL-Class). A shared place or transition is visualized as gray coloring of the corresponding item. This is modeled by two possible language items shared and not_shared connected to the place and transition symbols. The item shared is depicted as gray object, the item not_shared as white one, with the same position as the underlying ellipse (place) or rectangle (transition). The visual rules ensure that exactly one instance of shared or not_shared is connected to one place or transition instance.

Formally, instances over a visual alphabet are graphs over the alphabet type graph satisfying the constraint satisfaction problem (Csp) which is derived from the alphabet. The computed solution of the Csp affects the sizes and the positions of the graphics. Fig. 7 illustrates one instance. We omit the inscriptions in the Petri net for better readability. This instance has one component named C at the architecture level containing one service method (test). The service net for this method consists of one input and one output place and a shared transition in between. Both levels are visible (completeView = true).

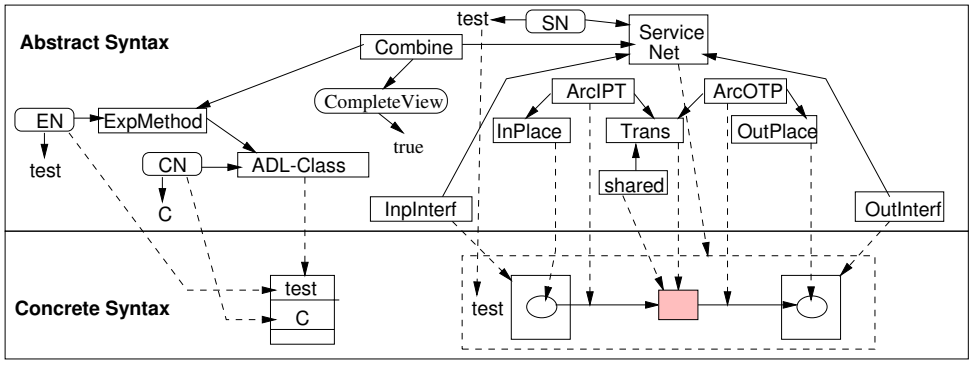


Fig. 7. One instance of the two-level language.

*Ensuring Consistency: The Visual Grammar*

Instances also occur in visual grammars that allow for syntax-directed editing in an intended graphical editor. The grammar rules ensure that only syntactically correct visual instances can be maintained in the editor and that certain consistency conditions are satisfied by the visual sentences (diagrams) drawn in the editor. The left and right-hand sides L and R of a rule may contain variables that are mapped to concrete values when the rule is applied to a visual sentence. Moreover, a rule may contain a negative application condition (NAC) specifying a situation that must not occur if the rule is to be applied.

Fig. 8 illustrates a rule which supports the insertion of a class symbol together with a class name. The NAC ensures that there is not yet a class with the same name.

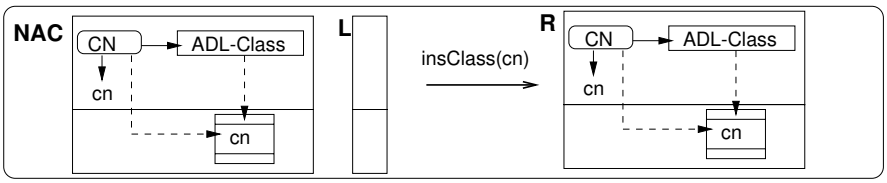


Fig. 8. Rule for inserting a class symbol

The rules that realize the combination of both levels and ensure consistency, are the rules insExpMeth and insImpMeth allowing to add an export or an import method to a class. The rule insExpMeth is coupled to the generation of a service net (see Fig. 9). We use the shorthand em for export method in Fig. 9 for the concatenation of the strings en (export method name) + s1,s2,..., sn (parameter sorts of the method) + s (target sort). Rule insImpMeth is com-

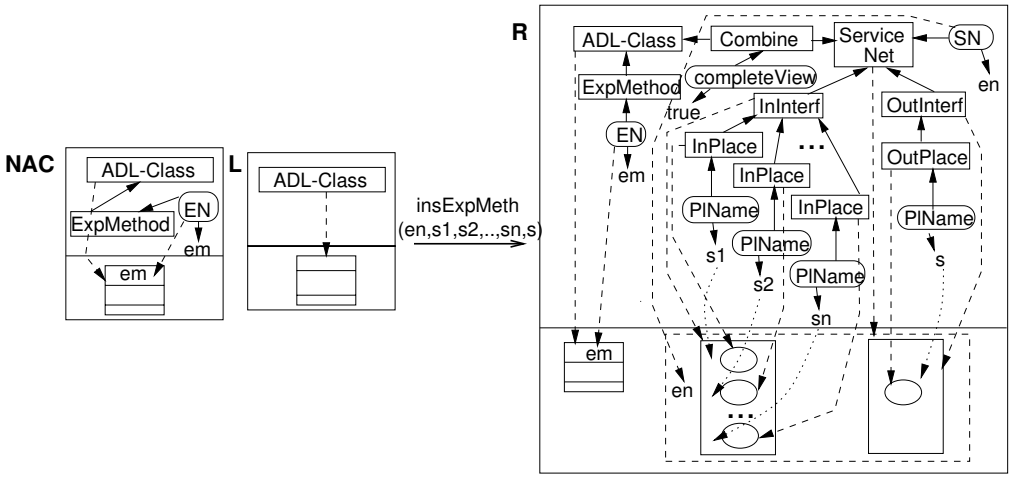


Fig. 9. Rule for inserting an export method

bined to the insertion of an association between two class symbols. Here, two cases are distinguished. The first rule is applied if there exists already an association: the import method name is added to the import section of of the class. The second rule is applied if no association arc exists to the class exporting the method: here, the association is inserted and the method is added to the import section.

At the Petri net level specifying a service net requires rules to insert places, transitions and arcs labeled by names and arc inscriptions. Fig. 11 shows the rules for inserting a place. Here, it is important to keep the color of shared / not shared objects consistent. We need three rules for different situations: Rule insNewPlace(sn,pln) inserts a place pln in net sn. The place named pln did not exist in any service net before; hence it is marked as not shared. Rule insShareFirstTime(sn,pln) inserts a place that is shared with a place of the same name of another service net. Up to now this place has been not shared. The rule therefore marks both places as shared after the insertion. Rule insSharedPlace(sn,pln) inserts a place that is shared with more than one other places (already marked as shared). This new place also is marked as

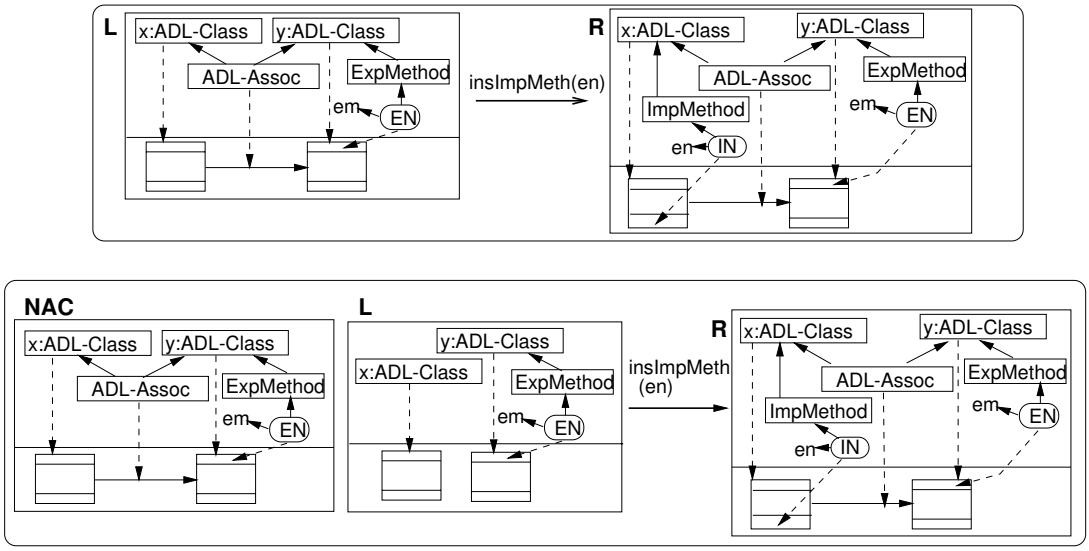


Fig. 10. Rules for inserting an import method

shared by the rule.

We here do not depict all rules (e.g. we omit the insertion rules for transitions and arcs), but their definition works analogously to the depicted rules. Additionally, rules deleting and changing elements in the editor have to be included in the grammar.

Summarizing, using the GenGEd approach, an evolution step is a graph transformation induced by the application of one or more rules to a graph (a sentence of our visual language) that models a system on the two levels architecture and component specification. The rules ensure that the evolution steps are consistent (e.g. no method is added in the architectural level without a corresponding service net being generated). Additional checks may be added by enhancing the abstract alphabet and the grammar rules.

## 4 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 over architecture description languages (ADLs) based on components and connectors can be found in [14]. In our example we use architecture design diagrams related to UML class diagrams. Work on architecture descriptions based on (extended) UML diagrams is done in [13] and [18] where the aim is to validate architectural properties by formal means.

In Object Coordination Nets (OCoNs) [7] a UML architecture description is combined with Petri nets specifying the component behavior. This idea

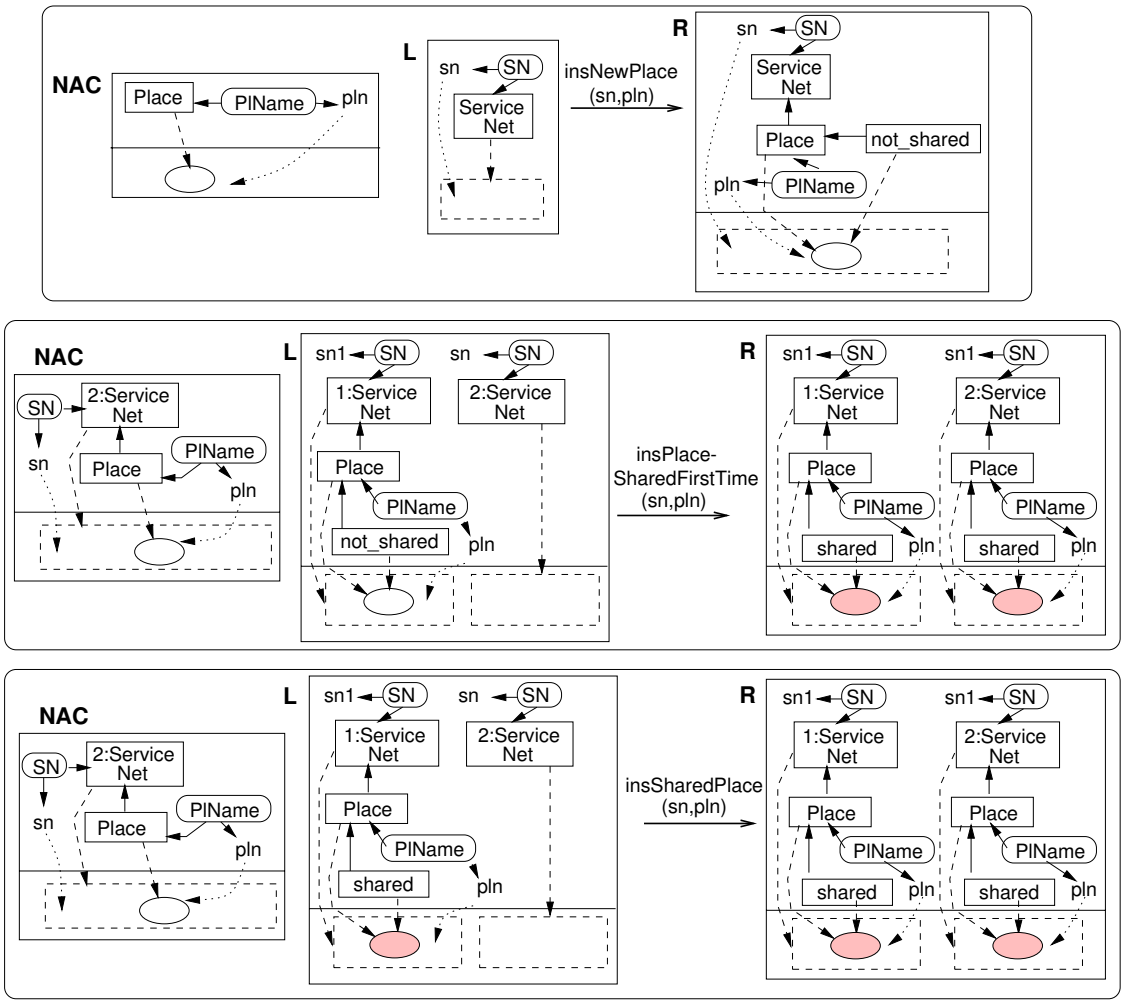


Fig. 11. Rules for inserting a place

serves as example for our two-level approach presented in this paper. The OCoN tool 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 tool 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 [22]. 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 [20], 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 [8] integrating distributed graph grammars and consistency checking rules. Our two-level ap-

proach is based on this work but allows more flexible visualization techniques than graphs for both levels.

Related approaches are given in [6,11,12] 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 evolution", i.e. changes are performed in the model editor, but our editing rules incorporate the checking of consistency conditions. Another formal approach to software system evolution is given in [10]. 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.

## 5 Conclusion

We introduced our concept of visual model evolution and illustrated the basic notions using a small print server example. The main ideas are on the one hand a concept of visually representing a system on two levels and on the other hand a rule-based approach for the description of consistent system evolution. We then have reasoned about the benefits of a visual environment for the employment of visual modeling techniques at two levels of abstraction discussing the GenGEd approach and its advantages in this context. In general, existing tools supporting visual modeling are restricted to a fixed visual modeling language. Moreover, they are often complex and not easy to use because they are intended to cover as many software engineering steps as possible. The advantage of the GenGEd approach is to support the generation of a small application specific visual modeling environment.

Some future work is needed to use the formal basis of our approach to formalize and check e.g. invariants of component behavior and interrelations in an appropriate way. Accordingly, we will enhance the GenGEd tool environment in order to model and check system behavior. More examples and larger case studies using different visual modeling techniques will be investigated to validate the usefulness of our approach towards a rapid prototyping environment for visual model evolution.

## References

[1] R. Bardohl. *Visual Definition of Visual Languages based on Algebraic Graph Transformation*. PhD thesis, Technische Universität Berlin, 1999. Published by Verlag Dr. Kovac, 2000.

[2] P. Donohe, editor. *Software Architecture*. Kluwer Academic Publishers, 1999.

[3] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic Approaches to Graph Transformation II: Single Pushout Approach and Comparison with Double Pushout Approach. In

G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, volume 1: Foundations*, chapter 4, pages 247–312. World Scientific, 1997.

[4] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer Verlag, Berlin, 1985.

[5] C. Ermel, M. Rudolf, and G. Taentzer. The AGG-Approach: Language and Tool Environment. In H. Ehrig, G. Engels, J.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, volume 2: Applications, Languages and Tools*, pages 551–603. World Scientific, 1999.

[6] H. Fahmy and R. Holt. Using Graph Rewriting to Specify Software Architectural Transformations. In *Proc. of Automated Software Engineering (ASE 2000)*, 2000.

[7] 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, July 1998.

[8] 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.

[9] P. Griebel. *Paralleles Lösen von grafischen Constraints*. PhD thesis, University of Paderborn, Germany, February 1996.

[10] 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.

[11] D. Hirsch, P. Inverardi, and U. Montanari. Graph Grammars and Constraint Solving for Software Architecture Styles. In *Proc. Int. Software Architecture Workshop (ISAW'98)*, 1998.

[12] D. Hirsch, P. Inverardi, and U. Montanari. Modeling Software Architectures and Styles with Graph Grammars and Constraint Solving. In *Proc. First Working IFIP Conference on Software Architecture (WICSA1'99)*, 1999.

[13] C. Hofmeister, R. Nord, and D. Soni. *Describing Software Architecture in UML*, pages 145–159. Kluwer Academic Publishers, 1999.

[14] 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.

[15] J. Kramer and J. Magee. Analysing Dynamic Change in Software Architectures: A Case Study. In *Proc. of 4th Int. Conference on Configurable Distributed Systems (ICCDS'98), Annapolis, USA*, 1998.

[16] J. Padberg, H. Ehrig, and L. Ribeiro. Algebraic High-Level Net Transformation Systems. *Mathematical Structures in Computer Science*, 5:217–256, 1995.

[17] J. Padberg, L. Jansen, H. Ehrig, E. Schnieder, and R. Heckel. Cooperability in Train Control Systems Specification of Scenarios Using Open Nets. *Journal of Integrated Design and Process Technology*, 2000.

[18] H. Störrle and M. Wirsing. Formal Analysis of Architectural Models. In *Proc. GI-Workshop "Rigorose Entwicklung software-intensiver Systeme"*, pages 33–42, Berlin, 2000.

[19] C. Szyperski. *Component Software.* Addison Wesley, Reading/MA, 1998.

[20] G. Taentzer. Distributed graphs and graph transformation. *Applied Categorical Structures*, 1999.

[21] G. Taentzer and H. Ehrig. Semantics of distributed system specifications based on graph transformation. In *GI Workshop "Rigorose Entwicklung software-intensiver Systeme", Berlin*, 2000.

[22] 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.