
Application of Graph Transformation to Visual Languages

- State of the Art and Further Ideas -

Roswitha Bardohl

Technical University Berlin

Sekr. FR 6-1, Franklinstr. 28, D-10587 Berlin

E-mail: rosi@cs.tu-berlin.de

Technical Report 97 – 10

March, 1997

Contents

1	Introduction	2
2	Visual Programming	4
2.1	[Shu86]: Visual Environments and Visual Languages	4
2.2	[Men95]: Visual Programming Systems	5
2.3	Graph Transformation based Visual Language and Visual Environment . .	7
3	Graph Transformation and Visual Languages	9
3.1	Scheme Evolution in Object-Oriented Models - A Graph Transformation Approach	10
3.2	<i>DiaGen</i> - A Generator for Diagram Editors based on a Hypergraph Model	12
3.3	Graph Rewrite Systems and Visual Database Languages	14
3.4	How to represent a Visual Program ?	15
4	Further Ideas	18
4.1	Diagrammatic and Graph Representation	18
4.2	Conceptual Ideas for Tool Support	20
4.3	Reconsideration of Section 3	22
5	Conclusion	24
A	References	25

1 Introduction

Within the last years more and more graphical software tools and environments came into the foreground supporting several tasks. One well known example are CASE tools supporting software development by diagrammatic techniques, like class diagrams used to model static aspects of a software system, or e.g. state transition diagrams used to model dynamic aspects. Common to all graphical software tools is the fact, that they offer mainly a visual language instead of a textual one.

We are looking towards an environment supporting the definition as well as the use of visual languages. The difference to common CASE tools is based on the use of formal specification techniques, especially graph grammars and graph transformation systems. We want to concentrate on graph grammars for the definition of a visual language, and graph transformation systems allowing the manipulation of sentences over the defined visual language by application of rules corresponding to the given grammar. Such a visual language can define a certain diagram class as e.g. given by the object model notation in [RBP⁺91]. Furthermore, it is conceivable to get a prototype from the formal specification. Within section 3 of this paper we present the state of the art concerning the application of graph transformation to visual languages.

Graph Grammars and Graph Transformation

Sentences of visual languages, i.e. diagrams, may often be regarded as collection of pictorial objects like *circles*, *arrows* or *strings* with spatial relations like *above* or *includes* between them, i.e. their underlying structure is a kind of directed graph. This is the reason why graph grammars are a natural means for defining the abstract syntax of visual languages. Furthermore, graph grammars can be used as single formalism to describe on the one hand the structure of visual sentences, and on the other hand to describe all kinds of operations on them. The application of such operations is called *graph transformation*, providing rule-based manipulation on graphs. Furthermore, graph transformation is a formally defined calculus based on set theory, algebra or category theory, and generalizes the notion of graph grammars.

More precisely, graph transformation is described by the application of graph productions modeling the permitted actions on graphs which can e.g. represent states of a software system or model data structures. Moreover, graph transformation defines a relation on graphs which can be iterated arbitrarily yielding the transformation process. In this way a set of productions gets an operational semantics.

Tools based on Graph Transformation

A tool based on graph transformation (also called *graph transformation machine*) contains usually a graph editor and an interpreter. The graph editor allows the definition of graph productions, which consist of a left and a right hand side, respectively, where both sides are represented by graphs. The interpreter then interprets the graph productions such that the user is able to manipulate the actual working graph.

An editor usually supports the graphical representation of graph productions and a (work-

ing) graph, mouse/menu-driven interfaces, and different views on graphs, like different sections of a graph, or different levels of abstraction as in the AGG-System ([LB93]).

The interpretation of graph productions can be regarded as simulation by test. If the simulation works well it is conceivable to generate a prototype from the defined graph productions.

In section 4 of this paper we present some further ideas going beyond the state of the art presented in section 3. Our further ideas aim at the development of a generator for diagram editors, where the generator as well as the diagram editors are part of a graph transformation based environment. Therefore it is not only necessary to support the rule-based definition of graph grammar productions defining the abstract syntax of a visual language, but additionally to support the rule-based definition and generation of graphics, i.e. diagram elements corresponding to a certain diagram class. Such a definition forms the basis for the generation of diagram editors where syntax directed diagram drawing is supported by application of the grammar rules.

Further aspects have to be considered with respect to the generation of diagram editors, like mouse/menu driven user interfaces, the support of view definitions, layout, etc.

Organization of this paper

Within section 2 visual programming is discussed in general, following two different works. The first work is given by [Shu86], which is concerned with visual environments and visual languages (see section 2.1). Within section 2.2, [Men95] is shortly presented which is concerned with visual programming systems in general, and furthermore, which is based on [Shu86]. On the one hand both works provide an overview about the complexity of the topic, and on the other hand this overview clarifies the application area we want to consider. Because a lot of items are used, which we cannot adapt for our work without some difficulties, we offer some definitions within section 2.3.

In section 3 we give a brief overview of some already existing graph transformation based approaches, which forms the basis for our further ideas, presented in section 4. There, some ideas are presented towards a graph transformation based visual environment for visual languages. Concluding remarks are given in section 5.

2 Visual Programming

The item *Visual Programming* in general means the usage of graphical techniques in connection with programming. But what does it mean exactly?

Within the work of Shu in [Shu86] *Visual Programming* is decomposed into the parts *Visual Environment* where graphical techniques and selections e.g. by mouse click are used for program development, debugging, information retrieval, etc., and *Visual Languages* where languages are developed to manage picture informations as well as to support visual interaction and to program with visual expressions. In the following these items are explained.

2.1 [Shu86]: Visual Environments and Visual Languages

A *Visual Environment* supports the program development, provides debug facilities, makes it possible to ask for information and to represent them, and supports software development and its understanding. It includes the following:

Visualization of the system development. This includes the visualization of requirements, specifications, design decisions, etc. which is necessary if a program should be changed or maintained. This kind of visualization can include several documents, e.g. diagrams, programs, textual documents, etc., each of them can be hierarchical combined. The user can select one document to reach the next level of detail.

Visualization of a program and its behavior. Several user views on a program are possible including several levels of detail. The visualization then consists on e.g. Nassi-Shneiderman diagrams, Module-Interconnection diagrams, data type diagrams, expression trees, flow graphs, etc.

The behavior of a program can be visualized as follows: several generated flow diagrams will be visualized together with an execution stack, where the expression just in work will be special marked.

Visualization of informations. Data or informations are stored within databases, but they are represented to the user in graphical form. The environment supports different operations on the graphics, like *zoom*, *rotate*, etc. Furthermore, a *direct manipulation* possibility exists for information retrieval using a graphical view of the underlying database.

At all, visual environments show a new way to deal with software, but they do not supply new approaches concerning language aspects or program constructs. *Visual Languages* allow the processing of visual informations, support visual interactions, and permit the programming with visual expressions:

Processing of visual informations. Picture informations, coming from different application areas like biology, physics, chemistry, etc., are represented by logical as well as physical pictures.

Logical pictures are defined by three relational tables, one includes the informations of the picture objects, another one the picture outlines and the third table the picture pages, whereas physical pictures are stored within a separate store. The handling of logical and physical pictures will be done by an efficiently working picture information system.

Questions to the underlying database must be given textual, only the result will be visualized which is supported by a corresponding visual language.

Support of visual representation and interaction. Data are stored in a database but presented to the user in a graphical view. The specification of a view is given textual by input of size, color, position, attributes, etc. of icons. Interaction is given by defining functions on graphical objects (icons).

Programming with visual expressions. This is equivalent to a so called area of *visual programming languages*, where the language has some visual representations as means for programming. For the assessment of such languages it is to distinguish between the language level, the scope of the language, i.e. the application area, and the extent of the visual expressions.

Non-procedural languages (highest level) permit the user to define what a program has to do and not how it is to do. Procedural languages (lower level) force the user to define exactly the steps to be executed. The scope of languages ranges from general and widely applicable to specific and narrowly applicable. Visual expressions include the meaningful visual representations (text, tables, icons, diagrams, graphs, etc.) used as language components to achieve the purpose of programming.

In his work [Shu86] Shu proposed an analytic approach to assess visual programming languages. This approach is graphically given by the triangle given on the left hand side of figure 1. The triangle shows the three dimensions corresponding to the language level, the scope and the visual extent.

2.2 [Men95]: Visual Programming Systems

The triangle presented in [Shu86] provides a basis for the assessment of visual programming systems (short VP systems) in [Men95]. A similar triangle is presented showing three dimensions for the characterization of visual programming systems (see right box of figure 1). Within this work it is not distinguished between *Visual Environment* and *Visual Language* as given by [Shu86]. The visualization is only concerned with scientific data as e.g. given by biology, whereas *Visual Languages* are seen to be one kind of executable specifiers allowing the user to define new language primitives. Both, the visualization as well as the specification is part of the item *purpose* of a VP system.

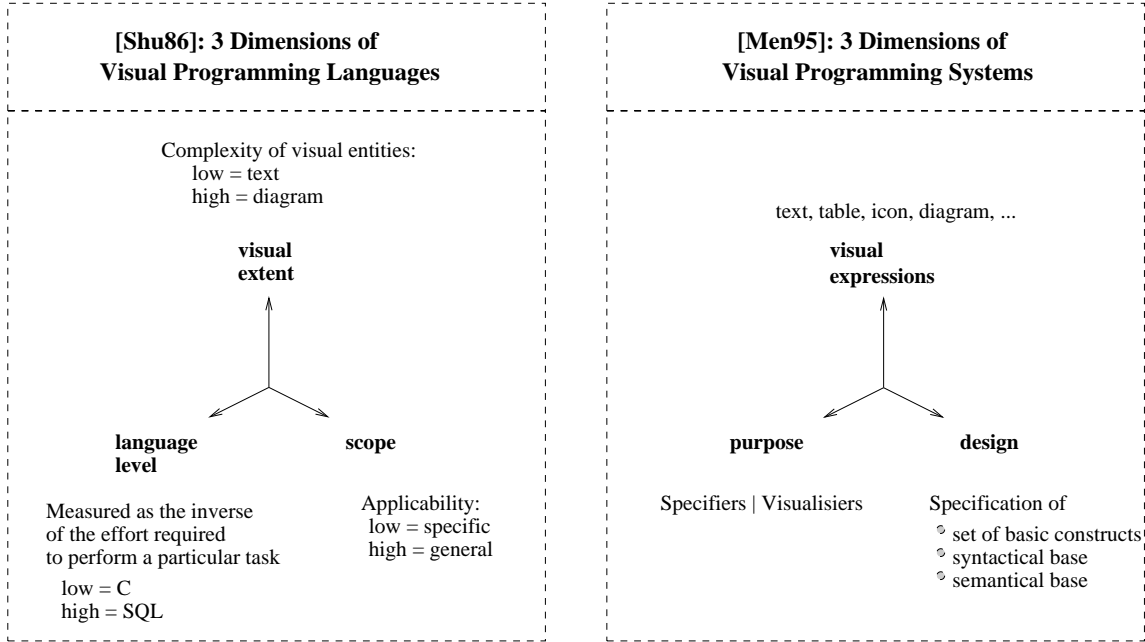


Figure 1: The three dimensions of VPLs and VPSs

Equivalent to [Shu86] are the possible expressions (text, ..., diagrams), but they are dependent on the purpose of a VP system and not restricted to *programming with visual expressions*. Furthermore, some topics are mentioned concerning the design of VP systems. In the following we give a brief overview of the three dimensions concerning the design, the purpose and expressions of visual programming systems:

The design of a visual programming system needs a specification of a semantical base, a syntactical base, and a set of basic constructs. The syntactical base deals with the expressions text ... diagrams, as already mentioned by Shu. The semantical base includes control-flow, data-flow, constraint-based, logic-based and procedural-based paradigms.

Procedural-based systems convert their diagrams into an underlying programming language. In a control-flow system a user manipulates iterations, conditionals and sequence expressions, which are necessary e.g. to model flow chart systems. Control is implicit in data-flow systems (e.g. Petri-nets) where each expression manipulated by the user describes a set of data source, perhaps a set of conditions, and actions to perform when all data sources are available and when the conditions are satisfied. In a constraint-based system the user visually specifies the invariants for each expression. At runtime a constraint-solver permits manipulations which do not violate the invariants.

The purpose of a visual programming system may be the visualization of some data, or the possibility of specifying, perhaps both. Specifiers are e.g. screen painters as

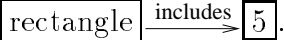
VISUAL BASIC (interactive point and click environment for placing window widgets on a screen; automatical generation of the layout code for the widgets; semantics and interactions on the widgets are given by the developer using a textual language) or design-only tools like HOTDRAW (interactive point and click environment for placing design notation icons/diagrams on a screen). This kind of specifiers is called non-executable specifiers. Executable specifiers are divided into visual specification shells (translating e.g. diagrams into the underlying programming language) and visual languages, allowing the user to specify new languages primitives.

Text, simple forms, tables, icons, diagrams, etc. define the visual expressions within a visual programming system and are in increasing order corresponding to the purpose of a system: Table expressions make extensive use of the position of the cells e.g. a spreadsheet cell can be the sum of the cell above. In icon-based systems users can click on the icon to access a menu of services. Moving the icon on the screen can represent the transfer of data or the application of some function to some data. In diagrammatic systems users create a diagram by linking up visual components offered from a palette. Normally, a visual programming system uses a combination of many of the expressions mentioned above.

2.3 Graph Transformation based Visual Language and Visual Environment

Nearly ten years are between the above mentioned articles ([Shu86], [Men95]). A lot of ideas and notions are presented, but it is difficult to adapt these notions exactly for our work. One reason may be that the term *programming* is placed into the foreground, another one that a lot of application areas are considered.

At all, we want to concentrate on a graph transformation based environment supporting the definition of visual languages as well as the insertion of visual sentences. I.e., for each visual-language definition an editor is generated, allowing the manipulation of sentences corresponding to the defined visual language by application of graph transformation rules. But, what is a visual language, an environment, graphics, etc.? To avoid confusion we will clarify now the notions we use within the following sections.

A graphic is built up by graphical primitives, like rectangle, text, circle, line, arrow, etc. together with corresponding attributes for size, color, etc. and, if possible, graphical parameters as e.g. given by a rectangle including a graphic. We will use directed graphs as underlying data structure for the definition of graphics, such that e.g. a rectangle including the number 5 can be represented by .

The display of graphics is called physical layout and is represented on a monitor in a two-dimensional fashion. This representation is supported by a presentation layer which

translates the high-level graphics definition into display instructions for a virtual display.

An icon is represented by a pictogram, which is simply a stylized symbol. Normally, a user clicks twice on an icon and something will happen, e.g. a window is opened. Further functions are imaginable: the icon supports a user defined operation where arguments are expected. With the double click a widget will be opened for the users' input.

A visual language for diagrams is given by two structures. One describes the information specific part of a language, we will call the abstract syntax or logical structure. The other one is concerned with the graphical part, we will call view or graphical structure. Both structures are defined by directed graphs, and must be partly connected. That means, the logical structure of language specific constructs have to be connected with graphical structures consisting of graphics as mentioned above. As we will see in section 3.4 the connection will be supported by a coupled graph grammar.

Furthermore, a visual language on a higher level defines a visual environment. It must support the above mentioned definition of a visual language for diagrams as well as the definition of several abstraction levels, and several views for language constructs. Moreover, this higher level visual language must include mechanisms to define e.g. user interaction for the diagram editors, which will be automatically generated from a specification (as mentioned in section 3.2).

A visual programming language is a visual language for diagrams together with a translation mechanism which translates a visual sentence, i.e. a diagram, into a programming language, e.g. C++, Eiffel, etc. Within this paper we do not concentrate on this topic.

A visual environment consists on several tools supporting different tasks, and follows the laws of the higher level visual language as mentioned above. In our sense, it should provide a possibility for incrementally working, and dynamical changes.

A visual environment we have in mind supports several layers. That means, several editors and interpreters are necessary for several depending tasks. As already mentioned, the graphical structure of language specific parts must be defined and connected with these parts, which requires one editor. A further editor supports the definition of a visual language by the graphical means. This definition will be supported by an interpreter which allows only syntax directed editing. From the visual language' definition a diagram editor is generated allowing the input of several diagrams, i.e. visual sentences. Within our first approach, only syntax directed diagram editing will be considered which is supported by a suitable interpreter.

3 Graph Transformation and Visual Languages

Now we introduce several approaches, all concerned in some way with visual languages in connection with graph grammars and graph transformation respectively:

1. Claßen/Löwe/Erdmann in [CLWW94, CL95, EC96, CGL96]:
Scheme Evolution in Object-Oriented Models - A Graph Transformation Approach.
Use of attributed graphs;
modifications are described by graph transformation rules;
without considerations wrt. graphical structure.
2. M.Minas/Viehstädt in [Min93, VM94, MV95, VM95]:
DiaGen - A Generator for Diagram Editors Based on a Hypergraph Model.
Use of attributed hypergraphs;
context free hypergraph replacement (restricts number of diagram classes);
with considerations wrt. graphical structure.
3. M.Andries/G.Engels in [And96, AE97]:
Graph Rewrite Systems and Visual Database Languages.
Use of attributed and labeled graphs (PROGRES [Zün92, SZW95]);
defines syntax and semantics of a visual database language;
without considerations wrt. graphical structure.
4. Rekers/Schürr/Engels/Andries in [Rek94, Sch94, RS95b, RS95a, AER96, RS96]:
How to represent a Visual Program ? and
A graph based framework for the implementation of visual environments
Use of attributed and labeled graphs (PROGRES [Zün92, SZW95]);
defines syntax of visual languages, graphical parsing, and a graph based framework;
with considerations wrt. graphical structure.

This is a collection of approaches relevant for our work because they offer theoretical as well as implementational aspects. Theoretical in the sense that syntax and semantics of visual languages are considered, and transformations are discussed with respect to how to transform the graphical structure of e.g. diagrams to graph structures. Implementational aspects are concerned wrt. generators, and more detailed it is considered how to define and implement a visual language using the graph grammar approach. All these approaches regard diagrams or more general diagrammatic development techniques.

3.1 Scheme Evolution in Object-Oriented Models - A Graph Transformation Approach

Within the first article ([CLWW94]) a transformation of the language specific part of an E/R-diagram (which is called object scheme) into an attributed graph signature is presented. Such a signature can be represented by a graph as given in figure 2, which illustrates the abstract syntax of a visual sentence. Instances of an E/R-diagram (called object structures) are regarded as algebra wrt. the attributed graph signature. Integrity constraints are transformed into a set of first-order formulas. They have to be fulfilled by application of graph productions allowing the modification of instances, as given by the insertion of entities or relationships. The result of the transformation process defines the abstract syntax of an E/R-diagram in our sense.

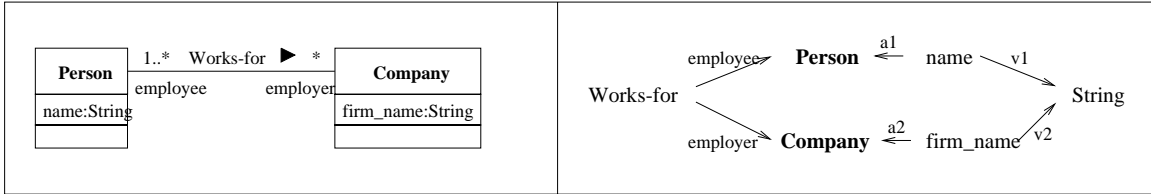


Figure 2: Graphical and graph representation of an E/R-diagram

[CLWW94] forms the basis for [CL95], where a formal framework for object oriented models (short OO-models) in general is presented. A model as given by e.g. an E/R-diagram, which is used in the area of semantic data modeling, or a method of description as given by Rumbaugh [RBP⁺91] or Booch [Boo94], used in the area of OO-analysis and OO-design, is an algebra wrt. the attributed graph signature. Each model provides the language means (e.g. a graph grammar for the input of E/R-diagrams) corresponding to the meta model, which is the class of all models, i.e. algebras.

As shown in [CLWW94] the attributed graph signature represents the basic building blocks of OO-models, which defines the abstract syntax in our sense. A flexible way to handle object models is presented, allowing the simultaneous modification of instances (called object structures) and object schemes (in our sense this is the abstract syntax graph of a sentence). Such modifications are given by graph productions and represented on the same level, e.g. class diagrams and instances, to facilitate simultaneous modification of schemes and object structures as illustrated in figure 3.

One object scheme is e.g. given by figure 2, which is a description in a special model, in this case it corresponds to a class diagram. The model is represented by an algebra T wrt. the attributed graph signature together with a type function, the typing of base algebra elements (e.g. $t : Person \rightarrow Class$ as shown in figure 4).

An object structure is then an instance of a scheme, i.e. concrete objects that conform to the given object scheme. An object structure wrt. a scheme T is represented by an algebra

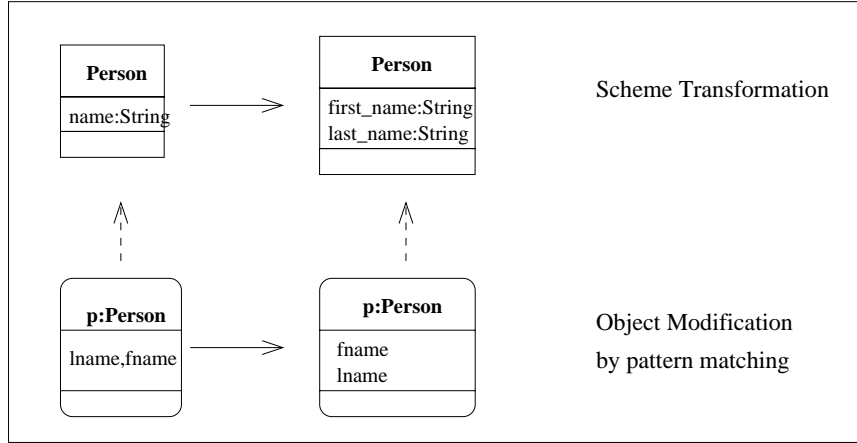


Figure 3: Example scheme/object rule

B, the (untyped) base algebra, together with a type function $t : B \rightarrow T$ describing the typing of base algebra elements. The type function must be compatible with the operations in the category of all attributed graphs, i.e. it must be a homomorphism in this category.

In [EC96] a class library for the meta model (ALPHA) is implemented where each model, i.e. a description technique is seen as an algebra T wrt. the attributed graph signature. It is further mentioned that other graphical modeling techniques as OO-notions simply require a different (model) algebra (as ALPHA-algebras) as the basis for typing of corresponding schemes which has to be examined.

[CGL96] extends the work of [CL95] by adding temporal logic to formulate temporal conditions, which arise often in information systems. A suitable graphical notation for temporal conditions is not given until now.

Within this work some diagrammatic description techniques are interpreted to be algebras in the category of models corresponding to an algebraic graph signature. This category defines a meta model in the area of algebraic graph theory, which is given by attributed graph signatures, and corresponding algebras and homomorphisms ([LKW93]). Graph transformation follows the single pushout approach as defined in [Löw93].

This work offers maybe a theoretical background and partly some implementational aspects, we could adapt for our work. As already mentioned above, the models define the abstract syntax of visual sentences, but not a visual language as we require. Furthermore, we miss the treatment of graphical structures in connection with the language specific parts, although within the implementation design patterns ([GHJV95]) are used serving the following properties:

- independence of the elements' kind and its depicted form,
 - support for people working together
- A Scheme can be decomposed into subschemes, which can be treated independently,*

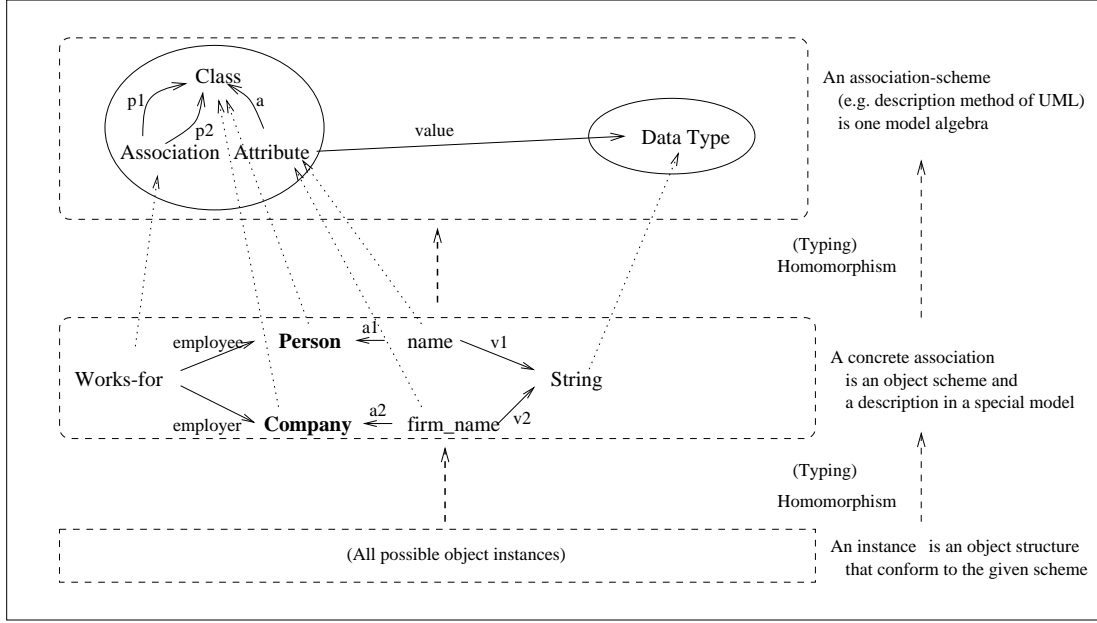


Figure 4: Hierarchical levels of ALPHA algebras

and

- including mechanisms for the synchronization of views.

Another topic we have to regard is the single pushout approach used for graph transformation, but the answer of the question if the double pushout approach is more comprehensive than the single pushout approach is beyond the scope of this paper.

3.2 *DiaGen* - A Generator for Diagram Editors based on a Hypergraph Model

The preceding work is [Min93] where a specification of diagram editors based on hypergraph grammars is discussed together with automatic layout adjustment. In [VM94] the specification is extended by event automatas for the description of interaction in graphical user interfaces. The ideas have been implemented leading to a generator for diagram editors, which is called *DiaGen* and first presented in [MV95]. An editor is generated from a user defined specification [Min93], which have to include a description for the class of diagrams (hypergraph grammar and transformations) as well as for the behavior of the generated editor, i.e. user interactions are described by a textual notation of an event automata. The description of the behavior includes syntax directed editing, automated (layout adjustment) but by the user changeable layout, direct manipulation, and execution respectively animation of diagrams, e.g. special marking of automatas' states, within the editor.

Constituents of the specification are

- a context free hypergraph grammar for a certain diagram class to describe the diagram syntax, as e.g. for Nassi-Shneiderman diagrams (NSD), and suitable transformation rules (on derived structures),
- layout conditions which are attached to grammar productions (the rules are given by in-equalities relating attributes of a production),
- layout informations for terminal symbols,
- definition of group types to deal with a set of graphical objects which is necessary e.g. for moving, and
- definition of an automata describing interaction in the graphical user interface (key and mouse events).

From the given specification, e.g. for NSD's, an editor is generated, processing (only syntactical correct) NSD's. The generated editor supports direct manipulation and execution by special marking if this is specified. The latter one is desirable e.g. for automatas, where a change from state to state is given by a transition. Figure 5 illustrates the environment of *DiaGen*.

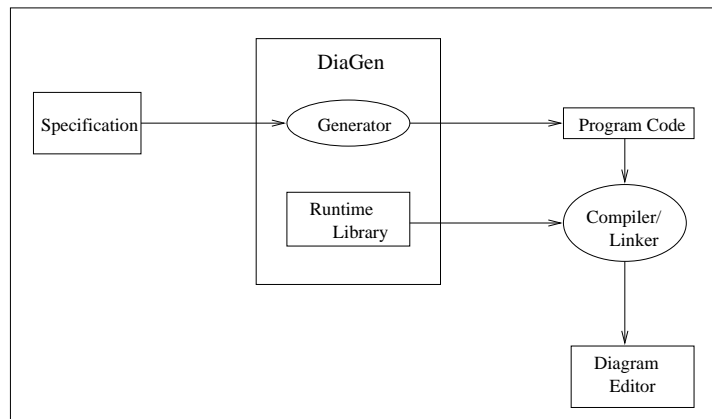


Figure 5: DiaGen - A generator for diagram editors

Until now, the specification must be given textual. It is mentioned that this should be improved by adding a graph parsing algorithm which supports also graphical input, i.e. specifications can be given graphically. But in the moment it is first tried to support free editing within a generated editor. Further extensions are mentioned concerning a visual programming environment. That is, generated editors supporting several applications should be put together to one visual programming environment.

This work shows how diagram editors can be automatically generated. The generator is implemented as a traditional compiler which takes a textual description as input. Formalisms

for all necessary activities are provided, but reusability concepts for the specifications are not regarded. Furthermore, it seems to be very difficult to specify complex analysis or execution activities. *DiaGen* uses one hypergraph for all notions, i.e. for the view (graphical structure) as well as for the logical structure of visual sentences, together with its derivation tree (graph transformation rules). In-equality constraints concerning the layout of some language constructs are connected with grammar productions of the corresponding hypergraph grammar. In our opinion the use of one graph structure for all notions makes it difficult to handle dynamical changes of views for one logical structure, i.e. for language specific parts. But this work treats all the other aspects which are necessary for the generation of diagram editors.

3.3 Graph Rewrite Systems and Visual Database Languages

In [And96] two ways are shown for the usage of graph rewrite systems for defining the syntax as well as the semantics of a visual database language. Several languages are introduced, one language is concerned with queries which is defined using graph rewriting (called *Graph Oriented Query Language (GOQL)*), another one is a language for database manipulations (called *Graph Oriented Database language (GOOD)*) which is defined as graph rewriting. The query language considers extended entity-relationship models (short EER), whereas the manipulation languages considers only E/R-models (short ER). Both kind of languages only regard the abstract syntax, i.e. the logical structure of visual languages, whereas the graphical structure is not taken into account.

With respect to the query language GOQL/EER it is said that a query consists on a collection of concrete database components which is put into a pattern describing the desired information. The pattern has to match with one database instance to provide the desired informations which are given by the corresponding attributes. The syntax of GOQL/EER is defined constructive by graph rewrite rules of a graph grammar. The semantics is defined operational by a translation of GOQL/EER queries into the textual structured query language SQL/EER which is formally defined. This will be done by attribute evaluations within the graph rewrite rules of the graph grammar.

Some graphical queries have limitations if they become more complex than textual queries e.g. by aggregate functions. Therefore a hybrid query language is defined which is syntactically a union of GOQL/EER and SQL/EER. Identifier of node types in GOQL/EER are the mechanism to link the textual parts of a hybrid query to the graphical part. The semantics of an HQL/EER query is defined by translating it to SQL/EER using an algorithm presented in [And96].

With respect to the manipulation language GOOD/ER it is said that database manipulations need the ability to structure a number of rules into a program where a program is given by a graph rewrite system. More concrete, for manipulations some basic operations are defined as for the insertion and deletion of entities and relationships respectively. A GOOD/ER program is than a sequence of some basic operations. The syntax of GOOD/ER

is purely declarative defined by adding or update of sentences' parts. The semantics is defined denotational by an algorithmic description of the resulting instance after application of same basic operations or a program.

The difference between both languages is the expressive power. That is, negative conditions cannot be expressed within a GOQL/EER pattern, while in GOOD/ER negation is in a sense a "simulation" using deletion. More general, GOQL/EER can only express conjunctive queries, whereas GOOD/ER is a generically complete language. This stems from the fact that, while GOOD/ER uses pattern matching combined with the powerful paradigm of graph rewriting, GOQL/EER uses only pattern matching.

This work shows not only a way to describe the syntax and semantics of visual languages, i.e. visual database languages, it additionally offers a formally defined hybrid query language, which connects graphical and textual expressions. Furthermore, it illustrates a rudimentary "life-cycle" for the construction of a graph grammar specification for a visual language (not discussed here), and provides some open issues for future research mainly concerned with graph rewriting and visual languages.

3.4 How to represent a Visual Program ?

The earliest work we found was [Rek94], where graph grammars are used to define the logical structure (called abstract syntax) as well as to define the view (called spatial relations) of diagrammatic expressions. [Sch94] provides triple graph grammars to hold theses two structures, which is said to define a visual language. Within following articles graphical (context sensitive) parsing ([RS95b, RS95a]) is regarded, and it is more concrete defined how to represent a visual program ([AER96]) corresponding to the two structures mentioned above. Figure 6 illustrates the several representations of a visual program.

Within the last contribution ([RS96]) a graph based framework for the implementation of visual environments is presented. The aim of this framework is to introduce an architecture for storing visual sentences within an environment which supports several kinds of editing (syntax directed, free and layout editing). The architecture of the framework is based on an abstract syntax graph (ASG) holding the logical structure of a visual sentence, and a spatial relation graph (SRG) holding the view (graphical structure) of a visual sentence. Furthermore, the architecture is supported by a coupled graph grammar to define, build and relate these two structures. Figure 7 illustrates the proposed data structure wrt. a finite automata as example.

Usually, a syntax directed graphical editor supports drawing of diagrams by providing language specific editing commands leading to correct diagrams at all time. A more lenient grammar for the graphical structure can be defined to improve the user interface, where the original grammar is a subset of. Layout editing as e.g. given by move commands do not change the interpretation (the meaning) of a diagram and must be controlled by

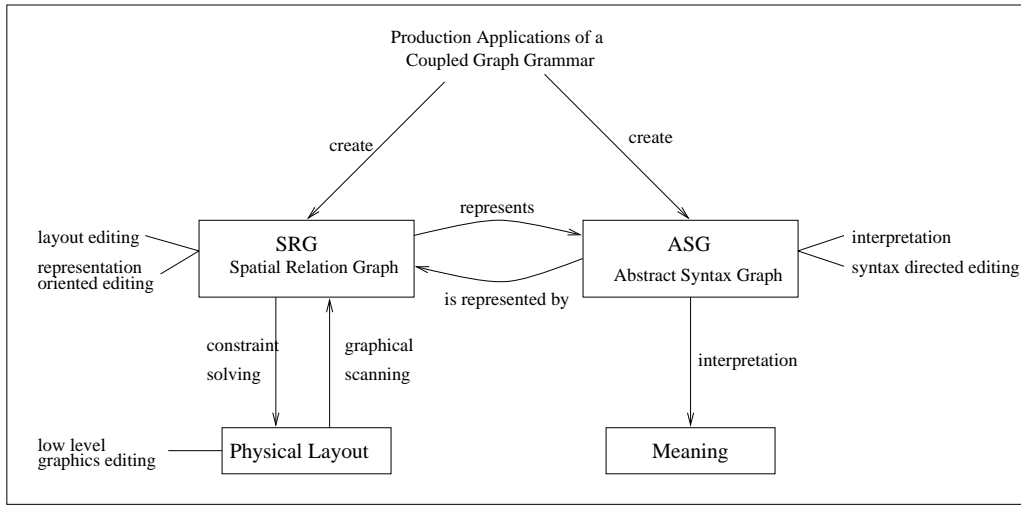


Figure 6: Representations of a visual program

a constraint solver (the graphical structure graph forms a constraint system). Graphical scanning is necessary if the graphical editor allows free editing. These facts are reflected within the provided visual environments' architecture:

The central components are abstract data types for the graphical structure and logical structure graphs. These graph data types can be either stored within a graph-oriented database system or an object-oriented database system with a graph-oriented layer on top of it. For analysis and execution activities additional graph data types and tools may be added, which are specified by means of graph grammars. Unparser and parser components are necessary to keep both structures (graphical and logical structure) in a consistent state, whenever they are modified. For maintaining the graphical structure's layout in a consistent state, a constraint solver is needed which combines constraint solving techniques with standard layout algorithms. Furthermore, a displayer (and a scanner if free editing should be possible) is necessary for keeping the widgets of a user interface toolkit and the corresponding graphical structure graph in a consistent state.

Two integration layers are necessary to combine these components. The first one controls the views and the user interface, the second one coordinates any necessary interaction between offered tools.

Some of the concepts are partially specified within the PROGRES environment where a single formalism is used to specify a class of directed attributed graphs together with all needed operations for editing, analysis and execution activities. The attributed graphs represent the abstract syntax graphs, i.e. the logical structure. But the view, i.e. the graphical structure graph is represented by a Tk-layout graph, and the physical layout (what a user sees) by Tk/Tcl-widgets. This is the reason why neither specific classes of graphical structure graphs nor tool specific editing activities on these graphs can be specified. Furthermore, until now it exists no support for free editing and parsing.

This work offers the basis to hold several graph structures, one for the logical structure and one for several graphical structures (spatial relations) representing visual sentences. Furthermore, it includes remarkable items wrt. graphical parsing and more important, a graph-based system architecture. Many of these ideas we will adapt and try to integrate them with our further ideas as well as with the other works presented in this section.

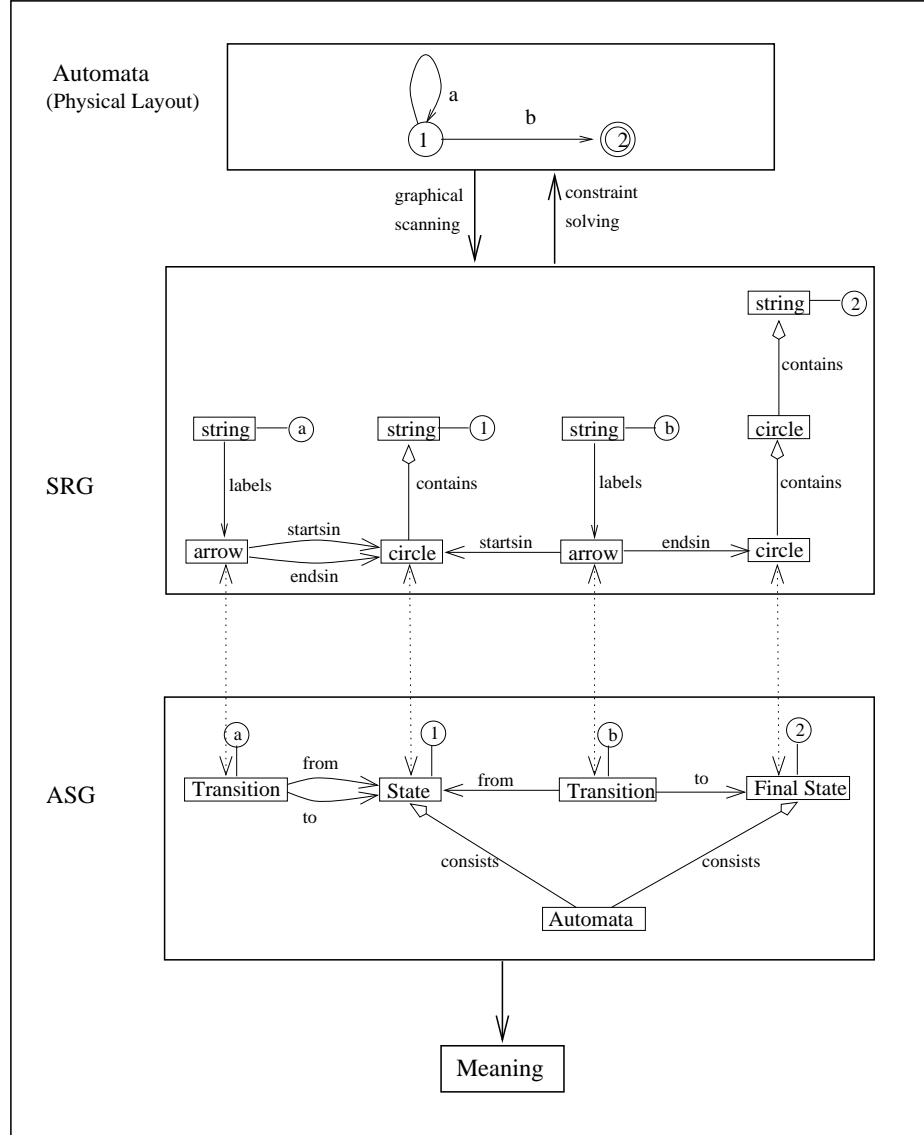


Figure 7: The proposed data structure: ASG + SRG

4 Further Ideas

Within the next sections we will discuss some issues that go beyond the state of the art presented above. One of the topics is concerned with the description of (semiformal) diagrammatic elements by graph structures, which is given in section 4.1.

Different diagrams of a certain diagram class can be seen as several visual sentences corresponding to one visual language. The definition of a visual language should be supported by a tool. Corresponding ideas are presented within section 4.2. Furthermore, preceding from a visual-language definition we consider the generation of graphic editors supporting diagram drawing.

Because a lot of work is already done, where some of them are mentioned in section 3, we will use and connect them within our work. That is, on one hand the **generation** of diagram editors (see section 3.2), and on the other hand the possibility to define several **views** (see section 3.4). A further topic is to embed this work into the area of algebraic graph theory (see section 3.1), and to offer **syntax** and **semantics** of defined visual languages (see section 3.3).

4.1 Diagrammatic and Graph Representation

The question arises what kind of diagrams or diagram classes respectively, are used for software design and how to model them by graph structures. Within this section we consider mainly the language specific parts of a visual language, i.e. the logical structure, and the physical layout, i.e. that what a user sees.

With respect to different modeling techniques several diagram classes exist. One well known kind of modeling technique is that of data structure definition where e.g. Entity-Relationship diagrams are used within the area of semantic data modeling, or class diagrams for object oriented design. We are sure that these and further different diagram classes can be modeled by graph structures.

Figure 8 illustrates how an E/R-diagram, shown in the left hand side, can be modeled by a directed attributed graph, which is by itself graphically represented. The entity types **E1** and **E2** correspond to nodes which are represented as circles, including the node attributes. Several possibilities exist concerning the relationships. Once it is possible to translate the relationship of an E/R-diagram into a directed edge, graphically represented by an arrow (shown within the middle of figure 8), and marked with the relation name as edge attribute. Another possibility is to model the relation of the E/R-diagram as node, where the relation name is the node attribute (right hand side of figure 8).

Corresponding to algebraic graph theory, the graph structure given in the middle offers implicitly the source and target function of the edge, whereas the graph structure within the right hand side shows explicitly these functions.

Using E/R-diagrams or class diagrams several levels must be considered, a user has to deal

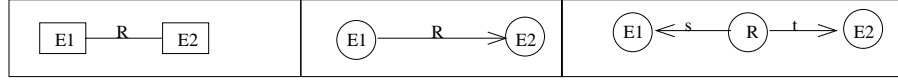


Figure 8: Diagrammatic and graph representations of an E/R-diagram

with. One level is the drawing of diagrams where only the name of an entity type and relationship type respectively, is given. An extension can be the insertion of attributes (and maybe the notation of methods which occur within class diagrams of object oriented design techniques). Such extensions can be seen as different development or refinement steps. A further level we have in mind handles with instances of the given E/R-diagram. Both levels should provide transformation rules the user can apply, e.g. to insert or delete an entity type, or to insert or delete an instance where additional constraints like integrity constraints have to be considered.



Figure 9: Diagrammatic and graph representation of a decision unit

A flow diagram is a further modeling technique including several language constructs, like the decision units as illustrated in figure 9. The left hand side shows the graphically represented decision unit which is modeled by a hypergraph, shown within the right hand side of figure 9. The language constructs of flow diagrams are similar to that of e.g. Nassi-Shneiderman diagrams but different wrt. the graphical layout. We have in mind, that it should be possible to build up one logical structure for these language constructs, which can be connected with several graphical structures.

Another example for a diagrammatic expression is a binary association combined with an association class which is part of the Unified Modeling Language (short UML), whose graphical and graph representation is illustrated by figure 10. The graphical representation of the underlying graph structure shown in the middle includes edges on edges. This graph structure can be translated into an directed attributed graph which is given in the right hand side of figure 10. Corresponding to algebraic graph theory this graph can be viewed just as given within the middle of figure 8 with implicitly source and target functions.

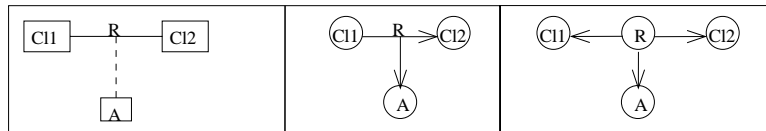


Figure 10: Diagrammatic and graph representation of a binary association

As already mentioned, we are sure that it is possible to transform several diagrammatic expressions into graph structures. What we need is a meta model, maybe similar to the work presented in section 3.1, also based on algebraic graph theory. Furthermore, several aspects have to be regarded concerning the visual environment we have in mind. That is first, the development of a visual language consisting of a logical structure, which is to be connected with a graphical structure. And second, the functionality the generated diagram editor for these visual language should have.

4.2 Conceptual Ideas for Tool Support

Until now we have concentrated only on diagrams wrt. the design of software. The question arises, if it is possible to develop a tool based on graph structures, which supports additionally the generation of executable code. On the one hand, code can be generated for a tool non-experts of formal specification techniques as given by graph transformation, can deal with, that is a certain diagram editor. On the other hand, it is conceivable that code can be generated for a given diagram which can be seen as the semantics of one diagram.

Furthermore, our main question is, whether it is possible to offer an incremental environment which is based on graph transformation. Incremental in the sense that we start with attributed typed graphs as underlying data structure, and some basic functionality we have to define, but the exact definition is out of the scope of this paper. One of the basic functionality must be the possibility to define graph productions for a visual language. Therefore, several levels must be regarded, because we would like to define a visual language by graphical means. This fact requires a tool allowing the definition of a graphical structure which is to be connected with some language constructs of the visual language' logical structure. Such a tool maybe consists of a type editor, allowing to define graphical elements for nodes and edges respectively, and furthermore, to define connections corresponding to the operations of the underlying signature. This fact will be illustrated in figure 11, where a type editor and a connection editor is illustrated corresponding to the automata as presented in figure 7.











Type Editor: sort symbol assignments	Connection Editor: operation symbol assignments
<div>State </div> <div>Final State </div> <div>Transition </div>	<div>source and target: </div> <div>attribution:  </div> <div> </div> <div> </div>

Figure 11: Example: Type and connection editor

This tool, we will call graphic box, supports not only the definition of the layout, but additionally the definition of connection points which is supported by the connection editor.

From the user definition given within the graphic box, a production editor is generated. Within this production editor the graphical means are available for the definition of several grammar productions. These grammar productions serve as the basis to generate a graphic editor supporting the syntax directed editing of diagrams corresponding to the given productions.

Figure 12 illustrates some system components we have in mind, and we will concentrate on in the future.

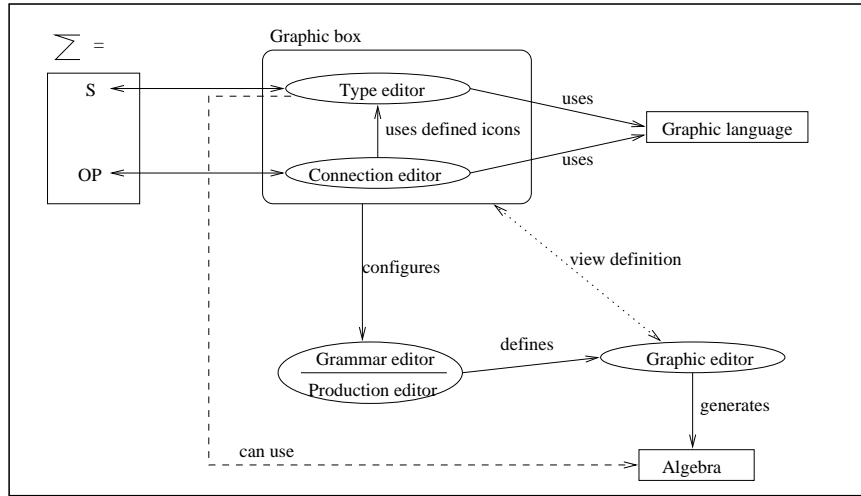


Figure 12: Some system components

Graphic language Graphical objects correspond to simple forms, like *Text*, *Ellipse*, *Rectangle*, *Line*, etc. Each object consists of a name and attributes describing their physical appearance in detail. The composition of graphical objects can be supported by graph grammar rules defining a graphical language. This language should provide several constructs to connect graphical objects, e.g. *a rectangle includes a graphic*, *a line is marked by a graphic*, *a graphic is concatenated with a graphic* etc. In this context we will try to adapt and to extend language concepts of \mathcal{GVT} ([Bar96]) which is already successfully used within the ACT-environment.

Graphic box Within the graphic box, the user assigns a graphic to each sort symbol of the underlying signature which we interpret as a type, and a connection to each operation symbol. The given assignments configure the target system by generating corresponding editing rules which can be applied within the grammar editor allowing the definition of a visual language.

A type editor permits the user to define own symbols which consist of a collection of simple graphical objects. Furthermore, the user should be able to define a functionality for each type, which can be e.g. the input of concrete integer values for *int*.

The connection editor supports the definition of connections, i.e. the definition of operations. This is mapped onto spatial relations between graphical elements. In case of the discussed example (see figure 11) one connection is e.g. that the “circle *includes* an integer value”.

Further connections must be defined for the *source* and *target* operation, and the operations wrt. the data type *Nat*. In case of an automata the *source* operation is mapped onto “the transition *starts_in* a circle”, whereas the *target* operation is mapped onto “the transition *ends_in* a circle”.

Grammar editor The grammar editor supports the graphical input of a graph grammar defining a concrete visual language, e.g. for graph or automata drawing.

Graphic editor The diagram editor defines the real graphic editor. This editor supports the application of grammar rules, i.e. the drawing of sentences corresponding to the defined visual language. Furthermore, a given diagram can be used by the type editor for further development steps, e.g. for building hierarchical structures.

4.3 Reconsideration of Section 3

Let us first reconsider section 3.4, where a data structure is proposed for the logical structure (abstract syntax) of a visual language together with its corresponding layout which is given by a graphical structure (spatial relations).

If we assume to have a graphic language like $\mathcal{GV}\mathcal{T}$ we are able to construct such expressions like $\boxed{Circle} \xrightarrow{\text{includes}} \boxed{String}$ as used within the example given by figure 11. Furthermore, the assignments given within the graphic box, i.e. type and connection editor, support the connection of terminal symbols with a graphical structure. This is used for defining the abstract syntax of a visual language which automatically builds up the language for the graphical structure. In this context we are able to handle implicitly both, the logical structure graph and the corresponding graphical structure graph allowing to draw correct diagrams within the generated graphic editor. A suitable constraint system supports the output, i.e. the layout on the screen.

The separation of logical and graphical structure provides the basis for the definition of several views, i.e. one user would like to draw circles, another user favours rectangles. View definitions do not change the logical structure, i.e. the abstract syntax, and therefore the definition of several views should be supported by our tool.

With respect to *DiaGen* which is mentioned in section 3.2 we additionally want to develop a generator. The main difference between this approach and our ideas is the fact that we want to provide an incremental approach with the possibility of dynamical change, and not only to generate one diagram editor. I.e. we want to be able to reuse some given graphics for further development steps.

What we have not discussed up to now are specification elements which are necessary for our system, e.g. user interaction like key and mouse events, or layout conditions which are attached to grammar rules within the *DiaGen*-approach. These items will be

discussed in the future. By the side, as already mentioned before, we are looking towards a possibility for code generation. E.g. an automata describing user interactions can be drawn within a generated graphic editor for automatas. The generation of executable code can be interpreted to be the implementation of the given automata, and furthermore, to be its semantics.

Concerning section 3.1, we would like to serve a meta model underlying all the tools we have in mind, too. As already illustrated in the beginning of this section, several diagrammatic structures can be mapped onto graph structures. We are looking forward that graph structures can be used as a meta model for our work.

Concerning section 3.3, the syntax of a visual language is defined constructively by graph rewrite rules of a graph grammar. The semantics can be defined by a translation of sentences over the visual language (given within the generated graphic editor) into a specification language which is formally defined.

5 Conclusion

Within this paper, we presented some approaches concerned with visual languages in general and the application of graph transformation to visual languages in particular. The visual languages based on graph transformation are all restricted to diagrammatic expressions used for the design of software components.

Furthermore, some ideas are discussed for an environment supporting the graphical definition of visual languages based on graph transformation. We are sure, a tool support for these ideas would improve graph transformation leading to a better acceptance of formal specification techniques by practitioners.

But for the improvement of graph transformation the availability of graphical programming is not sufficient enough. Further extensions like the availability of an animation component is conceivable. An animation component could be used on the one hand to visualize and animate several development steps, and on the other hand it could be used to animate the operational semantics of defined functions. The latter one is called *simulation* which can be seen as a kind of verification by test, but it could also be used for demonstrational aspects.

In the future we would like to develop a uniform generator for the development of editors for visual languages based on graph transformation. This generator, which we will define conceptually, should serve as a basis for tool support. The supporting tool by itself supplies a visual language development environment. It should be generic for the adaption to certain application areas as e.g. given by distributed systems. At all, the concept of a visual graph-transformation-based development environment should support the definition of a visual language, static consistency checks (we will integrate an existing tool) and the execution (by an interpreter or compiler). Therefore we want to concentrate on the following issues

- Specification of a concrete *visual language* using graph grammars.
This visual language is the basis for our start system.
- Conceptual modeling of systems using a *visual language* which is based on graph transformation (diagrams, OOA-models, etc.).
This is a visual language, an expert of formal specification techniques has to define, aiming in the generation of an algebra editor.
- Conceptual programming using a *visual language*, where the visual program is mapped into programming / executable code.
This fact may lead to an independent system, non-experts can be used.

A References

References

- [AE97] M. Andries and G. Engels. *A hybrid query language for the extended entity relationship model*. Journal of Visual Languages and Computing, 8(1), 1997. Special Issue on Visual Query Systems, to appear.
- [AER96] M. Andries, G. Engels, and J. Rekers. *How to represent a Visual Program ?* In [TVL96], 1996.
- [And96] M. Andries. *Graph Rewrite Systems and Visual Database Languages*. PhD thesis, Leiden University, P.O. Box 9512, 2300 RA Leiden, The Netherlands, February 1996.
- [Bar96] R. Bardohl. *Graphical Support for Prototyping of Algebraic Specifications by GVT - Language Description and Users's Manual*. Technical Report 96-11, Technical University Berlin, Franklinstr. 28/29, D-10587 Berlin, Germany, April 1996.
- [Boo94] G. Booch. *Object Oriented Analysis and Design*. Benjamin/Cummings, 2nd edition, 1994.
- [CGL96] I. Claßen, M. Gogolla, and M. Löwe. *Dynamics in Information Systems: Specification, Construction, and Correctness*. Technical Report TR 96-01, Technical University of Berlin, 1996.
- [CL95] I. Claßen and M. Löwe. *Scheme Evolution in Object-Oriented Models - A Graph Transformation Approach*. In ICSE '95 - 17th Workshop on Formal Methods Application in Software Engineering Practice, Seattle, Washington, April 1995.
- [CLWW94] I. Claßen, M. Löwe, S. Waßerroth, and J. Wortmann. *Static and Dynamic Semantics of Entity-Relationship Models Based on Algebraic Methods*. In B. Wolfinger, editor, *Innovationen bei Rechen- und Kommunikationssystemen*, pages 2-9. 24. GI-Jahrestagung/13. IFIP Congress, Springer, 1994.
- [EC96] S. Erdmann and I. Claßen. *Alpha - A Class Library for a Metamodel based on Algebraic Graph Theory*. In *Proc. Fifth International Conference on Algebraic Methodology and Software Technology*, LNCS. Springer, 1996.
- [GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

- [LB93] M. Löwe and M. Beyer. *AGG - an implementation of algebraic graph rewriting*. In C. Kirchner, editor, Proc. Rewriting Techniques and Applications, volume 690 of *LNCS*, pages 451–456. Springer, 1993.
- [LKW93] M. Löwe, M. Korff, and A. Wagner. *An algebraic framework for the transformation of attributed graphs*. In M.R. Sleep, M.J. Plasmeijer, and M.C. van Eekelen, editors, Term Graph Rewriting: Theory and Practice, pages 185–199. John Wiley & Sons Ltd., 1993.
- [Löw93] M. Löwe. *Algebraic approach to single-pushout graph transformation*. *Theoretical Computer Science*, 109:181–124, 1993.
- [Men95] T. Menzies. *Frameworks for Assessing Visual Languages*. Technical Report TR 95-35, Monash University, Dep. of Software Development, December 1995.
- [Min93] M. Minas. *Spezifikation von Diagrammeditoren mit automatischer Layoutanpassung*. In H.Reichel, editor, Proc. 23. GI-Jahrestagung, Dresden, Reihe 'Informatik aktuell', pages 334–339. GI, September 1993.
- [MV95] M. Minas and G. Viehstaedt. *DiaGen: A Generator for Diagram Editors Providing Direct Manipulatin and Execution of Diagrams*. In [VL'95], 1995.
- [RBP⁺91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented Modeling and Design*. Prentice-Hall, 1991.
- [Rek94] J. Rekers. *On the use of Graph Grammars for defining the Syntax of Graphical Languages*. Technical Report tr94-11, Leiden University, Dep. of Computer Science, 1994. Also available from `ftp.wi.leidenuniv.nl:/pub/cs-techreports/ as tr94-11.ps.gz`.
- [RS95a] J. Rekers and A. Schürr. *A Graph Grammar Approach to Graphical Parsing*. In [VL'95], 1995.
- [RS95b] J. Rekers and A. Schürr. *A Parsing Algorithm for Context-Sensitive Graph Grammars*. Technical Report tr95-05, Leiden University, Dep. of Computer Science, 1995. Also available from `ftp.wi.leidenuniv.nl:/pub/cs-techreports/ as tr95-05.ps.gz`.
- [RS96] J. Rekers and A. Schürr. *A graph based framework for the implementation of visual environments*. In [VL'96], 1996.
- [Sch94] A. Schürr. *Specification of Graph Translators with Triple Graph Grammars*. In Proc. 20th International Workshop on Graph-Theoretic Concepts in Computer Science, volume 903 of *LNCS*, pages 151–163. Springer, 1994.

- [Shu86] C.N. Shu. *Visual Programming Languages: A Perspective and a Dimensional Analysis*. In S.K. Chang, T. Ichikawa, and P.A. Ligomenides, editors, *Visual Languages*, pages 11–34. IEEE Computer Society Press, 1986.
- [SZW95] A. Schürr, A. Zündorf, and A. Winter. *Visual Programming with Graph Rewriting Systems*. In [VL’95], 1995.
- [TVL96] *Proc. of the AVI’96 Workshop Theory of Visual Languages*, Gubbio, Italy, May 30. 1996. Available at URL <http://www.cs.monash.edu.au/~berndm/TVL96/tvl-home.html>.
- [VL’94] *Proc. IEEE Workshop on Visual Languages*, St. Louis, Missouri, October, 4-7 1994. IEEE Computer Society Press.
- [VL’95] *Proc. IEEE Workshop on Visual Languages*, Darmstadt, Germany, September, 5-9 1995. IEEE Computer Society Press. Available at URL: <http://www.computer.org:80/conferen/vl95/asTALKHP.html>.
- [VL’96] *Proc. IEEE Workshop on Visual Languages*, Boulder, Colorado, September 1996. IEEE Computer Society Press.
- [VM94] G. Viehstaedt and M. Minas. *Interaction in Really Graphical User Interfaces*. In [VL’94], pages 270–277, 1994.
- [VM95] G. Viehstaedt and M. Minas. *DiaGen: A Generator for Diagram Editors Based on a Hypergraph Model*. In *Proc. 2nd International Workshop on Next Generation Information Technologies and Systems (NGITS’95)*, pages 155–162, Naharia, Israel, June 1995.
- [Zün92] A. Zündorf. *Implementation of the Imperative / Rule Based Language PROGRES*. Technical Report AIB 92-38, RWTH Aachen, 1992.