From Graph Transformation to Rule-based Programming with Diagrams*

Berthold Hoffmann

FB 3 – Technologiezentrum Informatik, Universität Bremen
Postfach 33 04 40, D-28334 Bremen, Germany
hof@tzi.de

Abstract. Graph transformation is a well studied computational model for specification and programming. In this paper we outline a path that can be taken in order to turn graph transformation into a rule-based language for programming with diagrams. In particular, we discuss how data abstraction and functional abstraction can be achieved in the setting of graphs, by minimal extensions of the underlying graph and transformation model.

1 Introduction

The rule-based transformation of graphs is a well studied field of theoretical computer science, see [25]. Graph transformation (also known as graph grammar theory, and graph reduction) has been applied successfully for modelling software systems and studying their behaviour, see [12].

So far, PROGRES [28] is the most successful programming language and system based on graph transformation. It supports functional abstraction, control structures (including backtracking), and encapsulation. However, PROGRES and other graph transformation languages still have some deficiencies:

- Graphs, their central data structures, are *flat*; they may not contain other graphs as components. The concept of *aggregation* is missing.
- Most of their programming concepts have only been added as textual constructs, on top of the graph transformation mechanism. Thus relevant parts of the languages are no longer *graphical and rule-based.*

We believe that a graph transformation language needs both features if it shall compete with visual object-oriented programming languages. Therefore we extend graphs by an aggregation concept, and lift a simple graph transformation mechanism to this model. Then we introduce functional abstraction, control, typing, and graph-oriented encapsulation, by slight modifications to the graph

* This work has been partially supported by the ESPRIT Working Group *Applications of Graph Transformation* (APPLIGRAPH). To appear in: M. Nagl, A. Schürr (eds.): *Proc. Workshop on Applications of Graph Transformation* (AGTIVE'99), Lecture Notes in Computer Science, April 2000.

model and transformation mechanism. The resulting language proposal is still completely graphical and rule-based.

The paper is organized as follows: Graphs and a simple way of graph transformation are introduced in section 2, and extended by a concept for graph aggregation in section 3. Concepts for functional abstraction and control are proposed in section 4, and some ideas concerning typing are outlined in section 5. Then, in section 6, we show how subgraphs and transformations can be encapsulated in classes. We conclude with some remarks on related and future work.

Due to space limitations, the presentation is informal. The concepts are explained by a running example concerned with the graphical representation of queues and queue operations.

2 Graph Transformation

We introduce a notion of graphs, and graph transformation that is simple, and yet expressive enough to form the basis for specification and programming.

2.1 Graphs

Graphs represent relations between entities as *edges* between *nodes*. Usually, edges link two nodes. We, however, allow edges that link *any number* of nodes, and *label* them so that different relations, of any arity, can be represented in a single graph. We also allow that a sequence of nodes, called *points*, may be designated as the interface of a graph at which it may be glued with other graphs. In the literature, such graphs are known as *pointed hypergraphs* [16, 7].

Example 1 (Queue Graphs). The structure of queue graphs is represented by two kinds of edges: a Q-labelled edge is linked to the begin and end node of a chain of I-labelled edges; every I-labelled edge is in turn linked to the begin and end node of an *item graph* that is stored in the queue.

Figure 1 shows how graphs are depicted in this paper. Nodes are drawn as circles, and filled if they are points. Edges are drawn as rectangles around their label, and are connected to their attachments by lines that are ordered counter-clockwise, starting at noon. The rectangles for binary edges with empty labels "disappear" so that they are drawn as lines from their first to their second linked node.

We abstract from some graph features although they are important in practice:

- *Typing* is only considered as far as it makes our constructs and definitions well-defined. Section 5 discusses some further issues.
- *Attributes* are values that may be associated to nodes and edges in order to represent non-structural properties of a graph. We do not consider attributes here although they will be used in implementations, e. g. for computing the layout of graphs.

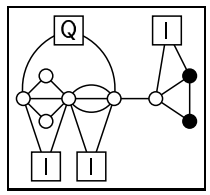


Fig. 1. A pointed graph

- *Notation* and *layout* of graphs is often taylored towards a particular application domain. Such conventions for the drawing of nodes and edges define *diagram languages*. We restrict ourselves to the graph notation explained above, and refer to DiaGen [22] for a system that allows to specify diagram languages for the kind of graphs considered here.

2.2 Rules and Transformation

We use a simple kind of gluing graph transformation [9] that is compatible with a resticted form of substitution-based graph transformation [17].

A *graph transformation rule* $t : P \to R$ (*rule* for short) consists of a *pattern graph* P, and a *replacement graph* R. A transformation step from a host graph G to some graph H via a graph transformation rule t is written $G \Rightarrow_t H$ and proceeds as follows:

- *Match* the pattern graph P, i. e. find a subgraph P' in G that is a copy of P.
- *Check* that every node in P' which is linked to an edge outside P' corresponds to a point of P. (Otherwise, the clipping described below would leave some edges with *dangling* links.)
- *Clip* P' by removing it up to its points, to obtain the *context graph* C.
- *Glue* a copy R' of the replacement graph R to C by identifying the points of P' with the corresponding points of R', to obtain the *transformed graph* H.

A graph may host several matches, of several rules. Thus graph transformation is nondeterministic in general. This gives a potential for concurrency: Several matches of rules can be replaced in parallel if they are *independent* in a certain sense, see e. g. [9].

Graph transformation with a set $\mathcal{T}$ of graph transformation rules considers sequences of sequential transformation steps in arbitrary order, and of arbitrary length. (We ignore concurrency, as an independent parallel step corresponds to a sequence of sequential steps.) If there is a transformation sequence $G_0 \Rightarrow_{t_1} G_1 \Rightarrow_{t_2} \cdots \Rightarrow_{t_n} G_n$, we write $G_0 \Rightarrow^*_{\mathcal{T}} G_n$, and say that $\mathcal{T}$ *transforms* G_0 *to* G_n.

Graph transformation can be used to define *graph languages*, analogous to Chomsky grammars, as the set of all *terminal* graphs G into which $\mathcal{T}$ transforms some distinguished *start graph* S, where terminality is usually defined by the absence of certain ("nonterminal") labels in a graph.

Graph transformation can be used to specify a *function* on graphs, like term rewriting [19] specifies functions on terms, by taking an arbitrary graph as input, and transforming it as long as possible. This function is *partial* if certain graphs can be transformed infinitely, and *nondeterministic* if a graph may be transformed in different ways.

It is this last way of using graph transformation that is the basis for programming with graph transformation, but language generation is useful too, for typing (see section 5).

Example 2 (Queue Graph Transformation). Figure 2 shows a rule that *dequeues* the first item graph of a queue graph, Figure 3 shows how this rule transforms the queue graph in Figure 1. The occurrences of the pattern and replacement graphs in the host graph, and transformed graph are drawn with fat lines.

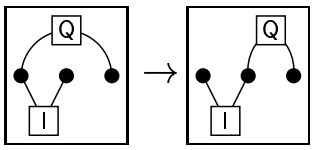


Fig. 2. A dequeuing rule

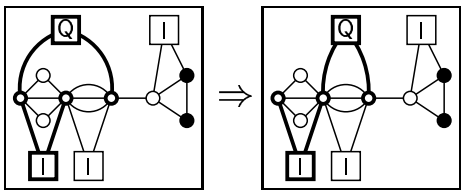


Fig. 3. A dequeuing step

This representation of queue graphs is not really adequate. It provides no general answer to the question: *What is the item graph linked to some l-edge?* In Figure 3, we could assume that an item frame designates all nodes and edges connected to both its begin and end node. Then, queues could only be used to store connected graphs, which might be too restrictive. (We could link l-edges to *all* nodes that belong to the item graph; then it would still be open *which* of the edges between those nodes in the host graph belong to the item graph.)

3 Structured Graphs

The graphs considered so far are *composite* values, but *flat*: Their components, nodes and edges, are primitive; none of them may be a graph again. That is the

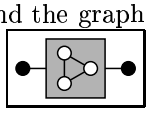
general problem when graphs shall be composed from subgraphs, like the queue and item graphs in Example 2 above. To overcome this limitation, we introduce compound edges that may contain graphs, and extend graph transformation correspondingly.

3.1 Hierarchical Graphs

A *hierarchical graph* consists of a graph as considered before, called its *top-level graph*, wherein some edges, called *frame edges* (or just *frames*), *contain* graphs that may be hierarchical again.

The hierarchical graphs that occur in rules may furthermore contain *variable edges* as placeholders for graphs. Variable edges bear distinguished labels, called *variable names*. A mapping $\beta = \{X_1 \mapsto G_1, \ldots, X_n \mapsto G_n\}$ that associates hierarchical graphs G_i with variables names X_i $(1 \leq i \leq n)$ is called a *binding*. The *instantiation* of a hierarchical graph G according to a binding β is denoted by $G\beta$, and obtained by removing every X_i-edge x in G, and gluing a copy of $\beta(X_i)$ to the nodes that were linked to x.

Example 3 (Hierarchical Queue Graphs). For a hierarchical graph representation of queues, we turn the Q- and I-edges of Examples 1 and 2 into frames that contain queue and item graphs, respectively. Frames are rectangles (like ordinary edges), with their contents drawn inside; they are filled in different shades of grey, and their labels are omitted. Variable names appear in italics.

Figure 4 shows two queue graphs. The graph on the left hand side contains a variable X, and the graph on the right hand side is its instantiation with the binding $\left\{ X \mapsto \right.$  $\left. \right\}$.

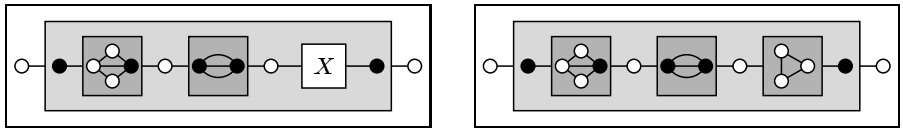


Fig. 4. Two hierarchical queue graphs

In this representation, item graphs are always complete (clippable) subgraphs that are disjoint to each other. This helps to maintain the consistency of the representation. Note that item frames may contain graphs of any arity; in Figure 4, they have 1, 2, or 0 points.

To keep the presentation simple, we do not consider *compound nodes* that may contain hierarchical graphs. Such nodes can be simulated by unary frame edges pointing to plain nodes. In a real programming language, however, compound nodes should be supported, if only for symmetry.

3.2 Hierarchical Graph Transformation

In a *hierarchical graph transformation rule* (*hierarchical rule*, for short) $t : P \to R$, the hierarchical pattern and replacement graphs P, R may contain variables, but every variable name occurring in R must occur in P as well, and every variable name may occur at most once in P.

The transformation of hierarchical graphs is then performed as follows:

- *Match* the top-level of the pattern graph P, either on the top-level of the host graph G, or recursively in the contents of some of its frames; then match the contents of every frame in P recursively with the contents of the corresponding frame in G.
- *Bind* the variables in P during matching, e. g. constuct a binding β such that $P'\beta$ is a subgraph of G.
- *Check* whether $P'\beta$ is clippable.
- *Clip* $P'\beta$ to obtain the context graph C.
- *Glue* an instantiated copy $R'\beta$ of the replacement graph R to C.

It should be noted that the *Bind* step requires *graph parsing* which is not defined in general. However, for the typing considered in section 5, parsing algorithms exist, even if they are not efficient.

Example 4 (Hierarchical Queue Graph Transformation). Figure 5 shows two hierarchical graph transformation rules for enqueuing and dequeuing. Figure 6 shows an enqueuing transformation, followed by a dequeuing transformation.

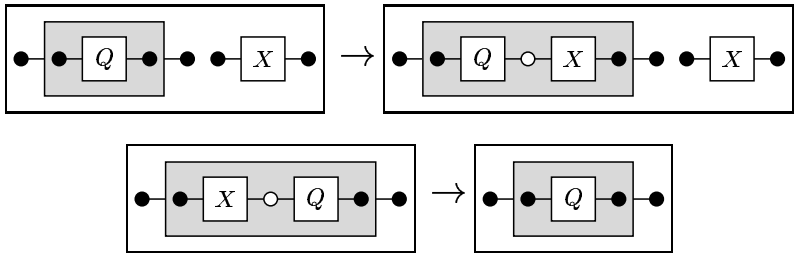


Fig. 5. Hierarchical rules for enqueuing (top) and dequeuing (bottom)

The variables Q and X binds queue graphs, and item frames, respectively. Enqueuing *duplicates* an item frame with its entire contents, by duplicating the variable X in its replacement graph; dequeuing *deletes* an item frame, again with its contents, by deleting X in its replacement graph.

Note that the time required for enqueuing and dequeuing does not depend on the length of the queue, wheras at least one of these operations would need at least logarithmic effort in a term rewriting implementation [6].

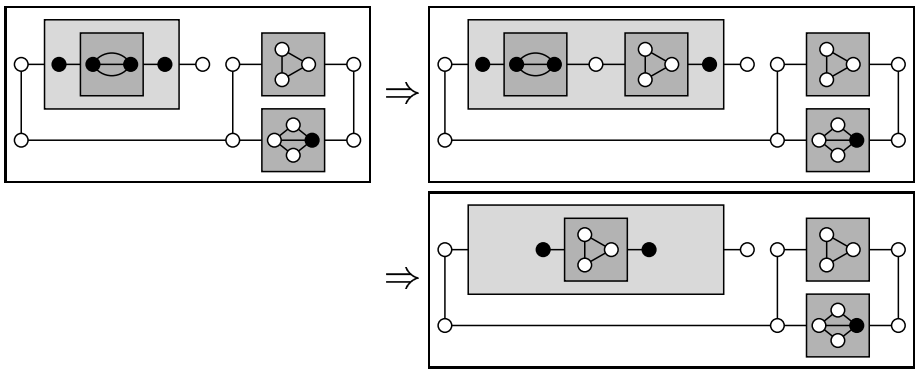


Fig. 6. An enqueuing, followed by a dequeuing transformation

4 Abstraction and Control

Graph transformation provides no explicite way to compose transformations from simple ones, and no means to control the order in which rules shall be applied. We introduce transformation predicates as a means to structure and parameterize transformations. We extend rules by application conditions, and show that this can be used to specify control flow.

4.1 Transformation Predicates

A graph transformation rule t can only "call" other rules by indicating, in its replacement, places where other rules shall be applied later. This can be done by inserting a p-labelled edge e in the replacement of t that appears in the pattern of a rule t' that shall be applied there. The label p can then be considered as the name of t', and e's links indicate the parameters to which it shall be applied.

We distinguish certain labels as *predicate names*. A *graph transformation predicate* (or just *predicate*) consists of a predicate name p that is associated with a set of graph transformation rules, called its *body*. Every pattern in the body of p contains exactly one p-labelled edge. An edge labelled by a procedure name is called a *button edge* (or just *button*), and is depicted as an oval.

Predicates are *called* by inserting buttons into the start graph of a transformation, or into the replacement graphs of rules and predicates.

A predicate is *applied* by applying a rule of its body to one of its calls in the host graph. A predicate is *evaluated* by applying it, and evaluating all predicates that are called in its replacement, recursively.

The links of a button point to the *parameters* of the transformation predicate. A parameter can be just a *node*, or an *edge* with its linked nodes. In particular, such an edge can be a frame that contains a *graph parameter* (as in Example 5 below), or a button that denotes a *predicate parameter* (as in Example 6 below). Thus buttons are *meta edges*; they may not only have links to nodes, but also *meta links* to other edges or to meta edges.

4.2 Success and Failure

The application of a predicate is very similar to the application of simple graph transformation rules. However, for a transformation predicate, the following question arises: *What happens if a predicate is called, but none of its rules applies?* This situation can be handled in one of the following ways:

- The transformation predicate, or its call, is considered *erroneous*, and transformation aborts.
- The call is considered to *fail*, and cancelled so that another rule may be applied instead.
- The call is considered to *succeed*, and transformation may continue.

The first interpretation is that of functional languages, and the latter ones are used in logical languages. We allow both. In any case, buttons are always removed during the transformation because they are meta edges that are just introduced to control the program's execution, but shall not be part of the graphs it computes.

Success, failure and abortion of a transformation predicate are specified in its body: We require an *otherwise* definition (starting with a "$/\!/$" symbol), followed by one of the symbols "$+$", "$-$", or "$\perp$", for success, failure, and abortion, respectively.

So we refine the application of predicates as follows: Whenever it turns out that no rule applies to a button, the predicate's otherwise definition is interpreted as described above.

Example 5 (A Graph Transformation Predicate). In Figure 7 the rule of Example 4 is re-specified as a transformation predicate dequeue that is parameterized by queue frames.

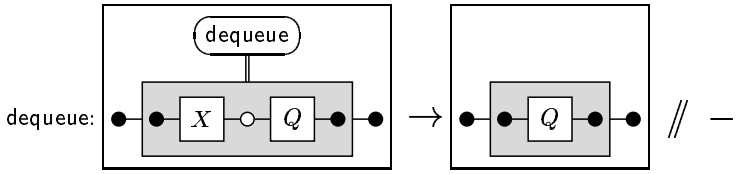


Fig. 7. The transformation predicate dequeue

The body of dequeue contains a single rule; its otherwise definition "$/\!/ \rightarrow -$" leads to failure if it is applied to an empty queue.

4.3 Application conditions

A transformation predicate can be applied as soon as one of its pattern graphs matches a clippable part of the host graph. All predicates inserted by the application can then be called in arbitrary order. However, the application will often

succeed only if some of these predicates evaluate successfully. It makes sense to evaluate them first, before attempting to evaluate the rest.

We extract these "critical" predicates calls as *application condition*, and denote a *conditional rule* as $t : P \,[\!]\, A \to R$. It is applied as follows: If the pattern graph P matches, its application condition A is glued to the host graph, and evaluated completely. Only if this succeeds, the pattern occurrence P' is replaced by a copy R' of the replacement graph R; otherwise, the rule is not applicable, and the remainders of A are removed.

Figure 8 below shows a predicate with a conditional rule.

4.4 Control

Transformation predicates already provide two simple control mechanisms:

- Pattern matching and *otherwise* definitions allow for case distinction.
- Applicability conditions specify which predicate calls in a rule are evaluated first.

With recursion, and by using *combinator predicates* that have predicate parameters, this suffices to specify control within the language.

Example 6 (A Control Combinator). Figure 8 shows a control combinator normalize that applies to a transformation predicate denoted by the variable T, evaluates T as an application condition, and, if that succeeds, calls itself recursively. As T shall bind to predicate calls with any number of parameters, we use the dot notation to indicate that T links to a varying number of nodes. Where the T-button is used as a predicate parameter, it is *disguised* as an ordinary edge by drawing a frame around its button. This prevents it from evaluation while it is "carried around" (in the pattern and replacement graph of the rule).

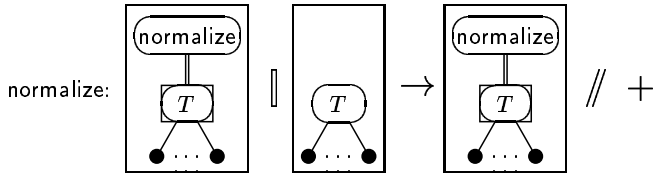


Fig. 8. The control combinator normalize

In Figure 9, normalize is applied to a disguised call of dequeue. Every application of normalize removes one item frame by evaluating dequeue as an application condition, until the queue frame contains no item frame, and dequeue fails. The empty queue graph is represented by a single node; the numbers 1 and 2 attached to it shall indicate that this node is the first, as well as the second point of the queue graph.

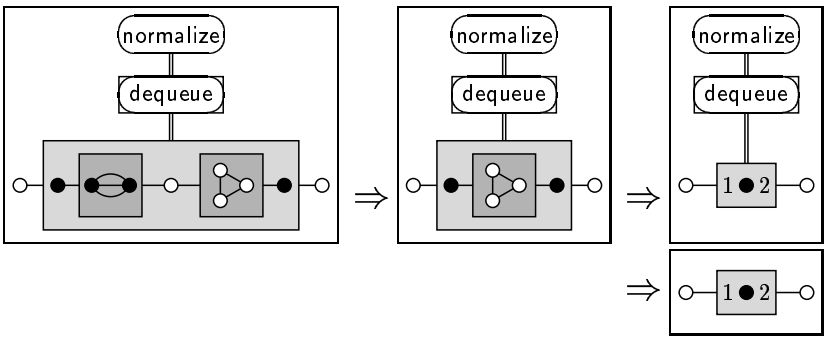


Fig. 9. An evaluation of normalize

The use of conditional rules is crucial for the termination of normalize: Had the call of T been inserted in the replacement graph, normalize could loop in its recursion without ever applying T. If defined as above, normalize will only loop if T does not terminate.

Other imperative control structures like while loops, or functional combinators like map and reduce as in Haskell [23] can be defined in a similar way. The most common of them should be predefined in the language.

5 Typing

Typing specifies how the data of a program is structured, and establishes rules for applying operations to data. These rules can be checked, preferably just by inspecting the program, *before* executing it, in order to ensure that it is consistent.

5.1 Graph Structures

In modern programming languages like Haskell [23], a type definition

```
Intlist ::= Nil | Cons Int Intlist
```

specifies the structure of data recursively. We introduce similar definitions for the structure of graphs.

A distinguished set of labels is used as *type names*, and the *structure* of a *graph type* named T is specified by a *graph structure definition* of the form

$$T ::= G_1 \mid G_2 \mid \ldots \mid G_n$$

where the graph T consists of a T-edge with its linked nodes, and the G_i are graphs that may contain type edges again. Graph structure definitions can be considered as predicates that generate the *graph values* of a type.

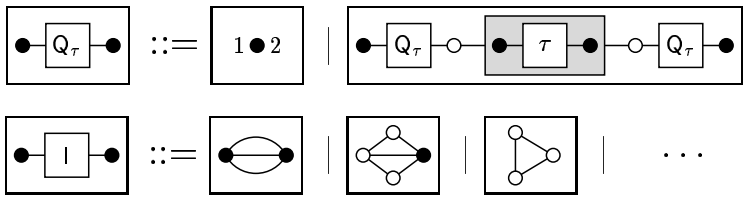


Fig. 10. The structure of queue and item graphs

Example 7 (Typing Rules for Queue and Item Graphs). Figure 10 defines the graph structure of queue and item graphs.

These graphs may be contained in queue and item frames, respectively. Queue graphs may be bound to queue variables like Q, whereas the variable X in the previous examples may be bound to a frame containing an item graph. The type Q_τ of queue graphs is *generic*. The type parameter τ can be instantiated by any graph type, e. g. to to the type Q_I used in the previous examples.

The rules used for graph structure definitions are a well-studied special case of *context-free* graph transformation, see [16, 7]. Type checking thus amounts to *context-free graph parsing*, e. g. as implemented in DiaGen [22].

5.2 Predicate Signatures

The *signature* of a transformation predicate shall specify to which kind of parameters it applies. As predicates are represented as graphs, their signature can be specified by graph structure definitions for a type π of predicates.

Example 8 (Signature of Queue Predicates). Figure 11 specifies the signatures for the predicates used in our examples.

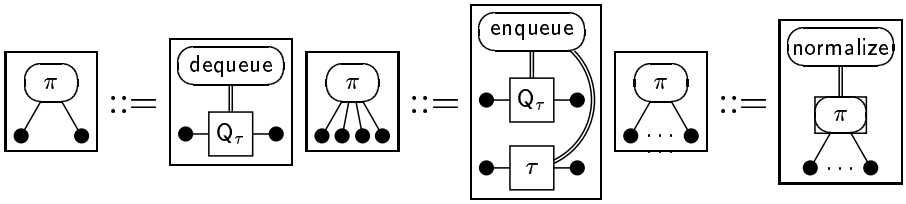


Fig. 11. The signature of queue predicates

The predicate type π has a varying number of parameter nodes so that the rules of the graph structure definition have different left hand sides. All predicate calls occurring in the examples of this paper can be derived with these rules (together with those of Figure 10). The predicate variable T used in example 6 is of type π.

5.3 Fine-Grained Typing

So far, edges are the only graph components that are typed explicitely, by labelling them with different symbols. In a real programming language, nodes should be typed in the same way. A particular type of edge could then be restricted to have a certain number of links, to nodes of certain types, as in typed graphs [4].

The *degree* of nodes, e. g. the number of links to some node, could be restricted as well, typically by cardinality expressions like 1, 0..1, 1..n, 0..n. Similarly, the overall number of nodes or edges contained in a graph could be restricted. Such *cardinality constraints* are allowed in PROGRES [28].

Pointed graphs could be specified to have a certain number of points, of certain types. Then the links of a frame, and the points of its contents could be required to correspond in their number, and their types. A similar correspondence could be required for bindings, between variable edges and graphs, and for rules, between pattern and replacement graphs.

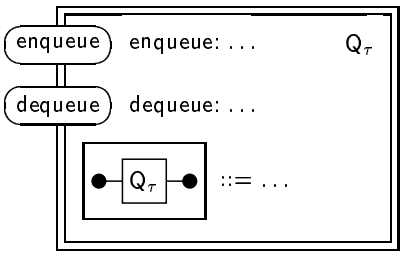
6 Encapsulation

Programming–in–the–large relies on the encapsulation of features in modules so that only some of them are visible to the public, and the others are protected from illegal manipulation. We sketch how graph classes and objects may be added to our proposal, and refer to packages that allow to group classes into subsystems.

6.1 Graph Classes and Objects

A *graph class* defines a graph *type* (denoted by the name of the class), and declares graph transformation predicates as its *methods*. The name of this type, which equals that of the class, and some designated methods are *public*. The structure of the type, and the other methods, are *private*.

Example 9 (The Queue Class). In Figure 9 we encapsulate primitive operations on queues within a class.



In this small example, all methods are public. However, the graph structure is visible only inside the class definition, thus adhering to the principle of *data abstraction.*

Graph objects are frames that contain a graph of some graph class C. In other classes, an object can only be manipulated by invoking a method of C.

At first glance, information hiding seems to contradict the expectation that the objects of a visual program shall be completely visible to a user. However, information hiding applies only to the program, not to the graphs manipulated in it. This can be defined by views, see [11].

6.2 Graph Packages and Libraries

Experience with object-oriented languages (e. g. Java [2]) has shown that the rather fine-grained encapsulation concept of classes should be complemented with a simple coarse-grained *package* concept that allows to group a set of related classes in a subsystem. Such a concept can be easily added along the lines of [18], as it just structures the namespace of a program, whithout changing the evaluation.

Then libraries of *graph classes* can be easily predefined. Also non-graphical values like numbers and strings can be considered as predefined "graph" classes if the language allows graphs to be visualized in a non-standard way, e. g. as text. Then, the language is purely graphical, at least on the conceptual level. (B. Meyer [21] states such a *purism principle* for object-oriented languages).

7 Conclusion

In this paper we have proposed how a simple notion of graph transformation can be developed towards a programming language. Basically, this has been achieverd by distinguishing several kinds of edges:

- *Frames* allow graphs to be structured hierachically so that hierarchical subgraphs may be linked via their points.
- *Variables* bind subgraphs in rules so that they may be deleted or duplicated in a single rule application.
- *Buttons* allow graph transformations to call each other recursively, with parameters. Based on *application conditions* and *higher-order predicates*, control structures can then be defined in the language itself.
- *Types* define the structure of graph languages and the signature of predicates.
- *Objects* are frames containing graphs of some type, with methods defined by a graph class. Classes provide for a fine-grained data-driven module concept.

The extensions are graphical, rule-oriented, object-oriented, with some logical and functional flavour (backtracking, and higher order predicates). These ideas could become the kernel of a "complete" graph transformation language that overcomes major deficiencies of today's graph transformation languages.

Related Work

Structured graphs have already been proposed by several authors: The *hierarchical graphs* of [24, 13] have compound nodes. Pratt [24] considers only language generation similar to Example 7, and Engels and Schürr [13] do not consider transformation at all. Schneider [27] considers graphs that have (simple) graphs as node and edge labels. The graphs of the old AGG system [20] support a rigid *layering*: Graphs, and the mappings between them can be viewed and manipulated as nodes and edges on the next layer of abstraction. This helps to structure the systems rather than the graph values.

Predicates exist in PROGRES [28], but without graph and predicate parameters. The language also provides textual logical and imperative control structures. FUJABA [15] has graphical control structures, similar to UML [26].

Modules have recently been added to PROGRES; they support functional and data abstraction, but not graph aggregation. We are currently not aware of any other language or language proposal that features graph aggregation and classes. However, the new AGG system [14] and the FUJABA system [15] allow to use object-oriented concepts of their implementation language Java. After all, graph structuring can then be realized by implementing plain graph objects as node or edge attributes of other graph objects.

Future Work

This work is closely related to GRACE [1], a design activity for an approach-independent graph-centered specification and programming language. *Specification* issues have been ignored here, and complete *independence* of particular notions of graphs and graph transformation had to be given up because hierarchical graphs require certain properties of graphs. However, although this paper is based on *hypergraphs* [16] and the *gluing approach* [3], it is not completely approach-specific: Nesting can be defined by adding points, frames and buttons to any kind of graph that has nodes, and a suitable notion of subgraph matching; several transformation approaches can easily be implemented with the rules and predicates proposed here. So we hope that our ideas will be fruitful for GRACE too.

The precise definitions of the concepts presented in this paper has been started in [8] (for frames and hierarchical graph transformation), and will be continued. Some more concepts, like *concurrency* and *distribution*, have still to be considered. For plain graph transformation, these concepts have been studied in [29] and [30] so that there is some hope that these results can be extended to our model.

The *visualization* of graphs, rules, predicates and classes is still very elementary in this paper. A lot of work has to be done in this area if graph transformation shall become a really attractive paradigm for programming and specification. Last but not least, such a language has to be *implemented* with a comprehensive program development environment.

Acknowledgements

The author is grateful to the members of the GRACE group, who stimulated this research, and to Manfred Nagl and Andy Schürr, who provided helpful comments on an earlier version of this paper.

References

[1] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. Graph transformation for specification and programming. *Science of Computer Programming*, 34:1–54, 1999.

[2] Ken Arnold and James Gosling. *The Java Programming Language*. Java Series. Addison-Wesley, Reading, Massachusetts, 1998. 2nd edition.

[3] Andrea Corradini, Hartmut Ehrig, Reiko Heckel, Michael Löwe, Ugo Montanari, and Francesca Rossi. Algebraic approaches to graph transformation part I: Basic concepts and double pushout approach. In Rozenberg [25], chapter 3, pages 163–245.

[4] Andrea Corradini, Hartmut Ehrig, Ugo Montanari, and Julia Padberg. The category of typed graph grammars and its adjunction with categories of derivations. In J. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, *gragra*, volume 1073 of *Lecture Notes in Computer Science*, pages 56–74. Springer, 1996.

[5] Andrea Corradini and Ugo Montanari, editors. *Proc. Joint* COMPUGRAPH/SEMAGRAPH *Workshop on Graph Rewriting and Computation*, number 2 in Electronic Notes in Theoretical Computer Science, `http://www.elsevier.nl/locate/entcs`, 1995. Elsevier.

[6] Frank Drewes. An optimal term rewrite implementation of queues. Report 5/93, Univ. Bremen, 1993.

[7] Frank Drewes, Annegret Habel, and Hans-Jörg Kreowski. Hyperedge replacement graph grammars. In Rozenberg [25], chapter 2, pages 95–162.

[8] Frank Drewes, Berthold Hoffmann, and Detlef Plump. Hierarchical graph transformation. In Jerzy Tiuryn, editor, *Foundations of Software Science and Computation Structures (FOSSACS 2000)*, Lecture Notes in Computer Science. Springer, March 2000. To appear.

[9] Hartmut Ehrig. Introduction to the algebraic theory of graph grammars. In Volker Claus, Hartmut Ehrig, and Grzegorz Rozenberg, editors, *Proc. Graph Grammars and Their Application to Computer Science and Biology*, number 73 in Lecture Notes in Computer Science, pages 1–69. Springer, 1979.

[10] G. Engels and G. Rozenberg, editors. *Proceedings 6th International Workshop on Theory and Application of Graph Transformation (TAGT'98)*, Technical Report tr–ri–98–201. Universität GH Paderborn, 1998. To appear in Lecture Notes in Computer Science, Springer, 2000.

[11] Gregor Engels, Hartmut Ehrig, Reiko Heckel, and Gabriele Taentzer. A view-based approach to system modelling based on open graph transformation systems. In Engels et al. [12], chapter 16, pages 639–668.

[12] Gregor Engels, Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. II: Specification and Programming*. World Scientific, Singapore, 1999.

[13] Gregor Engels and Andy Schürr. Encapsulated hierachical graphs, graph types, and meta types. In Corradini and Montanari [5].

[14] C. Ermel, Michael Rudolf, and Gabriele Taentzer. The AGG approach: Language and environment. In Engels et al. [12], chapter 14, pages 551–603.

[15] Thorsten Fischer, Jörg Niere, Lars Turunski, and Albert Zündorf. Story diagrams: A new graph grammar language based on the Unified Modelling Language and Java. In Engels and Rozenberg [10], pages 112–121. To appear in Lecture Notes in Computer Science, Springer, 2000.

[16] Annegret Habel. *Hyperedge Replacement: Grammars and Languages.* Number 643 in Lecture Notes in Computer Science. Springer, 1992.

[17] Annegret Habel and Detlef Plump. Unification, rewriting, and narrowing of term graphs. *Electronic Notes in Theoretical Computer Science*, 2, 1995.

[18] Reiko Heckel, Berthold Hoffmann, Peter Knirsch, and Sabine Kuske. Simple modules for GRACE. In Engels and Rozenberg [10], pages 158–165. To appear in Lecture Notes in Computer Science, Springer, 2000.

[19] Jan Willem Klop. Term rewriting systems. In S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 1–116. Oxford University Press, 1992.

[20] Michael Löwe and Martin Beyer. AGG — an implementation of algebraic graph rewriting. In Claude Kirchner, editor, *Proc. Rewriting Techniques and Applications*, number 690 in Lecture Notes in Computer Science, pages 451–456. Springer, 1993.

[21] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, New York, 1997. 2nd edition.

[22] Mark Minas. Specifying diagram languages by means of hypergraph grammars. In *Proc. Thinking with Diagrams (TwD'98), Aberystwyth, UK*, pages 151–157, 1998.

[23] John Peterson and Kevin Hammond (eds.). *Report on the Programming Language Haskell, Version 1.4*, 1997. Available via `http//www.haskell.org`.

[24] Terrence W. Pratt. Pair grammars, graph languages and string-to-graph translations. *Journal of Computer and System Sciences*, 5:560–595, 1971.

[25] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. I: Foundations.* World Scientific, Singapore, 1997.

[26] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modelling Language Reference Manual.* Object Technology Series. Addison Wesley, 1999.

[27] Hans-Jürgen Schneider. On categorical graph grammars integrating structural transformations and operations on labels. *Theoretical Computer Science*, 109:257–274, 1993.

[28] Andy Schürr, Andreas Winter, and Albert Zündorf. The PROGRES approach: Language and environment. In Rozenberg [25], chapter 13, pages 487–550.

[29] Gabriele Taentzer. *Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems.* Dissertation, TU Berlin, 1996. Shaker Verlag.

[30] Gabriele Taentzer and Andy Schürr. DIEGO, another step towards a module concept for graph transformation systems. In Corradini and Montanari [5].