A Declarative Approach to Graph Based Modeling

Jürgen Ebert Angelika Franzke

March 1994

Abstract

The class of TGraphs, i.e. typed, attributed, and ordered directed graphs, is introduced as a general graph class for graph based modeling. TGraphs are suitable for a wide area of applications. A declarative approach to specifying subclasses of TGraphs by a combination of a schematic graphical description and an additional constraint language is given. The implementation of TGraphs by an appropriate software approach is described.

1 Introduction

Modeling parts of the real world is a wide area of applications of graph theoretic concepts in practice. E.g. in the software design process the structure of the information to be kept and handled by the system is often describable by graphs. In order to describe which graphs are correct models and which are not, an approach to describing classes of graphs (graph languages) is necessary. Since graphs are to the same extent

- ▷ expressive pictures,
- ▷ formal models, and
- ▷ efficient data structures,

a well-based approach to working with graphs is necessary, which covers all three aspects in a consistent manner.

In this paper, we introduce the class of TGraphs as a general graph class, which by being very general is suitable for a wide area of applications. We give a declarative approach to specifying subclasses of TGraphs by a combination of a schematic graphical description and an additional constraint language, and we show how TGraphs can be implemented appropriately. The declarative approach for describing graph classes, allows to check the correctness of a graph at any point of time. Here the constraints are forced by the graph module itself and not by the application program.

The paper is organized as follows: In section 2 the class of TGraphs is described and a formal definition is given[1]. Based on this definition, section 3 introduces our approach to graph based modeling. We describe a specification language suitable to describe TGraph languages - i.e. subclasses of the general class - from a conceptual point of view, by using extended entity-relationship modeling in section 4 and an appropriate constraint description language in section 5. Section 6 certifies that the formal and conceptual descriptions may be implemented accordingly. Section 7 compares this work with related approaches, especially with graph grammar techniques. The approach is illustrated by an example chosen from a software engineering context, namely by a class of graphs that are models for state charts.

2 Basic Definitions

Graph models are to a certain extent abstractions of an object-oriented view of the world. When using graphs, objects are usually represented by vertices, whereas relationships are modeled by edges. To enhance the modeling power of graphs, vertices and edges may be classified into different types, and (depending on their types) they also may have attributes. To control graph traversal order in the context of graph algorithms an additional order can be put on the edges incident with a given vertex. Thus, a general class for modeling is the class of ordered directed graph with typed and attributed vertices and edges, called **TGraphs** in the following.

An **ordered directed graph** is defined by its set of vertices V and its set

[1] Throughout the text, a $\mathcal{Z}$-like notation is used. For an introduction to $\mathcal{Z}$, the reader is refered to [Diller 90] or [Spivey 92].

of edges E. The incidence relation between vertices and edges is a function Λ, which maps each vertex $v \in V$ to a sequence of all edges $e \in E$ that are incident to v. For each edge e in $\Lambda(v)$ also the direction of e with respect to v (e may be an ingoing or an outgoing edge) is specified. In order for G to be a correct graph, it has to be assured that every edge in E is mapped to exactly one vertex as an outgoing edge and to exactly one vertex as an ingoing edge.

To model **types**, a universe of type identifiers is assumed. An **attribute** (instance) is defined to be an ordered pair consisting of an attribute identifier and a corresponding attribute value. Then, a TGraph is an ordered directed graph extended by two functions. *type* assigns a type identifier to the vertices and edges, whereas *value* is an assignment of finite sets of attribute instances. Here, it is demanded that every vertex and every edge has exactly one type, whereas graph elements may or may not have attribute instances:

$$
\begin{aligned}
&UNIVERSE ::= vertex\langle\langle\mathbb{N}\rangle\rangle \mid edge\langle\langle\mathbb{N}\rangle\rangle\\
&VERTEX == \operatorname{ran} vertex\\
&EDGE == \operatorname{ran} edge\\
&DIR ::= in \mid out
\end{aligned}
$$

$$
\begin{aligned}
&[ID, VALUE]\\
&typeID == ID\\
&attrID == ID\\
&attributeInstanceSet == attrID \twoheadrightarrow VALUE
\end{aligned}
$$

TGraph

$$
\begin{aligned}
&V : \mathbb{F}\, VERTEX\\
&E : \mathbb{F}\, EDGE\\
&\Lambda : VERTEX \nrightarrow \operatorname{seq}(EDGE \times DIR)\\
&type : UNIVERSE \nrightarrow typeID\\
&value : UNIVERSE \nrightarrow attributeInstanceSet\\
\hline
&\Lambda \in V \rightarrow \operatorname{iseq}(E \times DIR)\\
&\forall\, e : E \bullet \exists_1\, v, w : V \bullet (e, in) \in \operatorname{ran}(\Lambda(v)) \land (e, out) \in \operatorname{ran}(\Lambda(w))\\
&\operatorname{dom} type = V \cup E\\
&\operatorname{dom} value = V \cup E
\end{aligned}
$$

Note, that this definition of a TGraph defines a very general class of graphs. A TGraph may contain loops and multiple edges, it may be interpreted as being undirected or unordered by ignoring the *DIR*-value or the ordering implied by the seq-operator. Edges are first-class objects which allows also edge-oriented algorithms. It is possible to derive most graph definitions used in a practical contexts by loosening the definition of a TGraph.

3 Graph Based Modeling

In the previous section, the general schema for TGraphs has been introduced. These graphs can be used as a modeling structure in many different contexts. We used them e.g. to model many different kinds of languages (string languages and graphical languages) in building software design and software reengineering tools, as well as for representing the subject matter aspects and the learner model inside a tutor system. Furthermore street maps, bus tours and the like were respresented with appropriately defined graph classes inside a tour planning tool.

To model reality by TGraphs, the following principles should be adhered to:

- ▷ each identifiable and relevant object is represented exactly once by a vertex,
- ▷ each relationship between objects is represented exactly once by an edge,
- ▷ similar objects and relationships are assigned a common type,
- ▷ informations on objects and relationships are stored in those attribute instances that are assigned to the corresponding vertices and edges, and
- ▷ an ordering of relationships is expressed by edge order.

This approach is very close to object-oriented information modeling of reality. Note, that objects and relationships of a common type will always have the same kind of information assigned and that objects are modeled by vertices whereas their relationships (including occurrences of an object in some context) are modeled by edges.

Example. To illustrate these modeling principles, a TGraph model for state charts is presented in the following. State charts are a graphical language introduced by David Harel ([Harel 87]) for describing reactive systems. They provide a visual formalism which extends finite state descriptions by introducing hierarchy and orthogonality as additional concepts. Figure 1 shows a sample state chart.

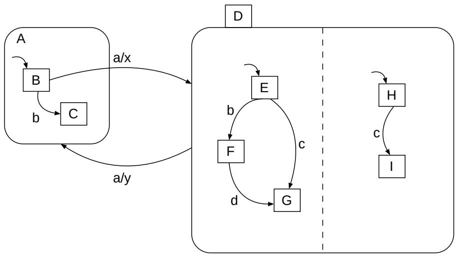

Figure 1: A Sample State Chart

To model state charts inside a tool, a conceptual analysis shows, that there are four different kinds of entities, namely *blobs* (rectangles), *transitions* (arcs), *events*, and *responses* (actions). *Blobs* may be either *elementary* (like $B, C, E, ...$) or composite (like A and D). Composite blobs may be *xor-blobs* (like A) containing a set of other (elementary or composite) blobs or they may be *and-blobs* (like D), which contain at least two unnamed xor-blobs separated by dashed lines. Each xor-blob (as well as the state chart itself) contains exactly one so-called *start blob*, which is marked by a small arrow. Transitions connect blobs and are annotated by an *event symbol* (like a, b, and c) and an optional *response symbol* (like x and y), which are separated by a slash character from each other. Some blobs and all events and responses are described by strings, i.e. they a carry a string-type attribute.

Following these modeling decisions every concrete state chart may be viewed as an SCGraph, i.e. as a TGraph, which reflects its abstract syntactical structure. Figure 2 shows the syntax graph corresponding to the state chart in figure 1. (For the sake of readability the edge types have only been differentiated coarsely in this figure.) $\Diamond$

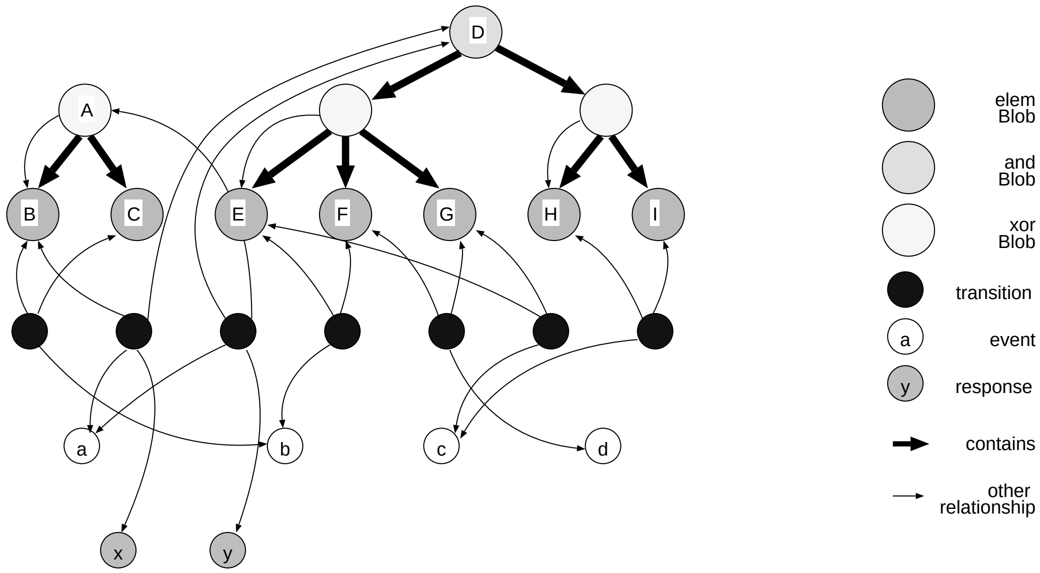

Figure 2: A TGraph Representation of a State Chart

4 Schematic Description

To document the class of legal models according to some modeling decisions, a suitable specification language for TGraph classes is needed. A graphical specification of the graph class for modeling state charts is shown in figure 3. The diagram captures all the informations on vertex and edge types gathered so far. It is an (extended) entity relationship(ER) description. Here a diagram is used to define a graph class, i.e. the class of TGraphs representing a state chart. Thus, we use ER-diagrams as a specification language for graph classes[2]. Like EBNF grammars may be used to describe and discuss string languages, ER-diagrams serve as a description of graph languages. Note, that they are not read as a generative scheme, but as a means of declarative description.

[2]A complete definition of our notation is given in [CaEbWi 94], where also a precise graph based semantics for ER-diagrams is defined, i.e. the class of graphs defined by an ER-diagram is specified.

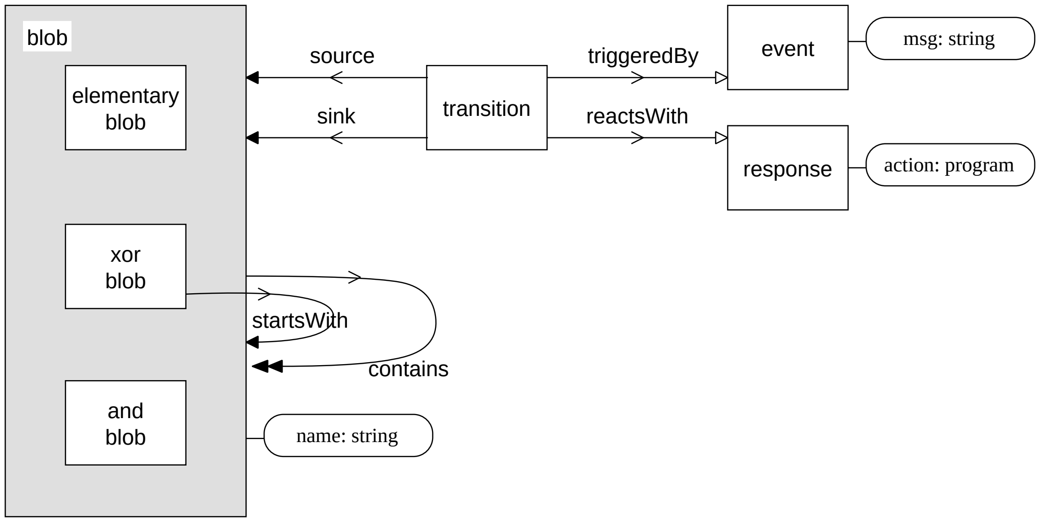

Figure 3: ER-Description of SCGraphs

The entity types defined in an ER-diagram correspond to vertex types, while relationship types represent edge types. The incidence structure shown in the diagram imposes structural constraints on the class of TGraphs defined thereby. A TGraph matches this specification if

- ▷ the type of every vertex or edge is defined in the ER-Diagram and
- ▷ for every edge the incidence information defined in the ER-diagram is respected.

4.1 Type Systems

In the following, we will formalize the notion of a TGraph matching a specification given by an ER-diagram. To do this, we introduce the terms **type system** and **incidence system**, which are abstractions of what is described by an ER-diagram. Given these definitions, the relations $\models_{\mathbf{typeSystem}}$ and $\models_{\mathbf{incSystem}}$ between graphs and type systems are defined. These relations specify whether a given TGraph matches the type restrictions imposed by a type system, as well as the incidence information.

From a modeling point of view, objects of one type share common properties. Thus, when defining a type t, one should also define the attributes that characterize the objects of t. Moreover, it is often useful to build up type hierarchies. Thus, when defining a set of valid types, one has also to specify a subtyping relation. This is reflected by the following definition of the schema *typeSystem*.

$$domID == ID$$
$$attributeSchema == attrID \leftrightarrow domID$$

$$\begin{array}{|l} \hline \textit{typeSystem} \\ typeDefinitionSet : typeID \twoheadrightarrow attributeSchema \\ isA : typeID \leftrightarrow typeID \\ \hline (\_isA\_) \in \mathrm{dom}\, typeDefinitionSet \times \mathrm{dom}\, typeDefinitionSet \\ \hline \end{array}$$

Example. In figure 3 the entity types and the relationship types form the domain of *typeDefinitionSet*. *blobs* have an attribute schema consisting of the attribute identifier *name* paired with the domain identifier *string*. Accordingly *events* and *responses* have non-empty attribute schemes. All other type identifiers have the empty scheme associated. The types *elementaryBlob*, *xorBlob* and *andBlob* are subtypes of *blob*, which is expressed by the inclusion of their respective rectangles. ◊

A TGraph G matches a given type system, if the following conditions hold:

- ▷ All types that are assigned to vertices or edges are defined in the type system.
- ▷ If a vertex or an edge has type t, then the value of this vertex or edge is correct with respect to the attribute schemes assigned to t and all superclasses of t in the corresponding type system.

If we define t_1 to be a subtype of t_2, we express that every object of type t_1 has also the properties of an object of type t_2, i.e. that the objects of type t_1 inherit the attributes defined for objects of type t_2. The set of all attribute definitions which belong to a type t due to inheritance is described by the following function (here, *union* should be defined appropriately as the union of sets in its argument):

$$
\begin{array}{|l}
allAttributesOfType : typeSystem \rightarrow (typeID \rightarrow attributeSchema) \\
\hline
allAttributesOfType = \\
\quad \lambda\, typeSystem;\ t : typeID \mid t \in \operatorname{dom} typeDefinitionSet \bullet \\
\quad\quad union(\{t' : \operatorname{dom} typeDefinitionSet \mid t\ isA^*\ t' \bullet \\
\quad\quad\quad typeDefinitionSet(t')\})
\end{array}
$$

The relation $\models_{\mathbf{attrSchema}}$ describes the fact, that the attribute instances of a graph element are correct with respect to an attribute schema. (Note that the definition of $\models_{\mathbf{attrSchema}}$ does *not* imply that there is an instance for every definition in the given schema.)

$$
\begin{array}{|l}
carrier : domID \rightarrow \mathbb{P}\, VALUE
\end{array}
$$

$$
\begin{array}{|l}
\_ \models_{\mathbf{attrSchema}} \_ : attributeInstanceSet \leftrightarrow attributeSchema \\
\hline
\forall\, I : attributeInstanceSet;\ S : attributeSchema \bullet \\
\quad I \models_{\mathbf{attrSchema}} S \Leftrightarrow \\
\quad\quad \forall\, i : I \bullet \\
\quad\quad\quad \exists_1\, d : S \bullet \\
\quad\quad\quad\quad first(i) = first(d) \wedge \\
\quad\quad\quad\quad second(i) \in carrier(second(d))
\end{array}
$$

This definition implies, that an *attributeInstance* may only have values for attribute identifiers, which appear only once in the *attributeSchema*. Thus, the following definition solves conflicts due to multiple inheritance by forbidding conflicting attributes on the instance level.

$$
\begin{array}{|l}
\_ \models_{\mathbf{typeSystem}} \_ : TGraph \leftrightarrow typeSystem \\
\hline
\forall\, TGraph;\ typeSystem \bullet \\
\quad \theta TGraph \models_{\mathbf{typeSystem}} \theta typeSystem \Leftrightarrow \\
\quad\quad \operatorname{ran} type \subseteq \operatorname{dom} typeDefinitionSet \\
\quad\quad \forall\, obj :\ V \cup E \bullet \\
\quad\quad\quad value(obj) \models_{\mathbf{attrSchema}} \\
\quad\quad\quad\quad allAttributesOfType(\theta typeSystem)(type(obj))
\end{array}
$$

4.2 Incidence Systems

The edges of a TGraph specified by an ER-diagram may be incident only with vertices, whose types fit to the description. The schema *incidenceSystem* specifies restrictions on the incidence structure of a TGraph. Given an edge type t, $incidences(t)$ determines the types of the start and goal vertices.

$$
\begin{array}{|l}
\textit{incidenceSystem} \\
\hline
\textit{typeSystem} \\
\textit{incidences} : \textit{typeID} \nrightarrow (\textit{typeID} \times \textit{typeID}) \\
\hline
\mathrm{dom}\, \textit{incidences} \subseteq \mathrm{dom}\, \textit{typeDefinitionSet} \\
\mathrm{ran}\, \textit{incidences} \subseteq \mathrm{dom}\, \textit{typeDefinitionSet} \times \mathrm{dom}\, \textit{typeDefinitionSet} \\
\hline
\end{array}
$$

The relation $\models_{\mathbf{incSystem}}$ defines whether a given TGraph G satisfies an incidence system. This is the case, if for every edge e of G the typing conditions for its start and its goal vertex are respected. (Here, inheritance has to be taken into account, too.)

$$
\begin{array}{|l}
\_ \models_{\mathbf{incSystem}} \_ : \textit{TGraph} \leftrightarrow \textit{incidenceSystem} \\
\hline
\forall\, \textit{TGraph};\ \textit{incidenceSystem} \bullet \\
\quad \theta\textit{TGraph} \models_{\mathbf{incSystem}} \theta\textit{incidenceSystem} \Leftrightarrow \\
\qquad \forall\, e : E;\ t : \textit{typeID} \mid \textit{type}(e) = t \bullet \\
\qquad\quad \textit{type}(\alpha(e)) \in \{\, t' : \textit{typeID} \mid \\
\qquad\qquad t'\ \textit{isA}^*\ \textit{first}(\textit{incidences}(t)) \,\} \wedge \\
\qquad\quad \textit{type}(\omega(e)) \in \{\, t' : \textit{typeID} \mid \\
\qquad\qquad t'\ \textit{isA}^*\ \textit{second}(\textit{incidences}(t)) \,\}
\end{array}
$$

Example. From Figure 3 on may derive that $incidences(source) = (transi-$ $tion, blob)$ holds, which e.g. implies, that the types of a goal vertex v of a *source*-edge e may only be *blob*, *elementaryBlob*, *xorBlob*, or *andBlob*. $\Diamond$

4.3 ERSpecifications

The specification given by an ER-diagram consists of a type system and a corresponding incidence system. A TGraph matches a specification if it matches the type and incidence claims specified therein.

$$
\begin{array}{|l}
\hline
\textit{ERSpecification} \\
\textit{typeSystem} \\
\textit{incidenceSystem} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\_ \models_{\mathrm{ERSpecification}} \_ : TGraph \leftrightarrow ERSpecification \\
\hline
\forall\, TGraph;\ ERSpecification \bullet \\
\qquad \theta TGraph \models_{\mathrm{ERSpecification}} \theta ERSpecification \Leftrightarrow \\
\qquad\qquad \theta TGraph \models_{\mathrm{typeSystem}} \theta typeSystem \land \\
\qquad\qquad \theta TGraph \models_{\mathrm{incSystem}} \theta IncidenceSystem
\end{array}
$$

Thus, the class of TGraphs given by an ERSpecification *Spec* is the set of those TGraphs G with $G \models_{\mathrm{ERSpecification}} Spec$.

5 Additional Constraints

So far, it has been discussed how to define classes of TGraphs by specifying vertex and edge types and restrictions on possible incidences in an ER-diagram, as well as how to define appropriate attribute schemes for objects and relations. Often, this information alone does not suffice to describe an application domain in a satisfying manner. Hence, the diagram still has to be extended by additional predicates which specifiy additional constraints to complete the specification language.

Some of these predicates, like e.g. degree restrictions might also be expressed graphically in the diagram, but more elaborate (especially global) constraints must be stated explicitly.

Example. In state charts, for instance, the set of *contains*-arcs defines a tree-like hierarchy on blobs. This is not expressed in the diagram. In this sense, the specification of the class of TGraphs modeling state charts is still incomplete. In graph theoretical terms, the described constraint is given by the following predicate:

$$\mathsf{isTree}(\mathsf{eGraph}(\mathsf{edges}(contains)))$$

Here, $\mathsf{edges}(contains)$ is the set of all edges e with $type(e) = contains$. eGraph returns the graph induced by a set of edges, and isTree is a predicate on graphs. Further conditions on state charts are: a blob is either elementary, or an xor-blob, or an and-blob, elementary blobs are not refined, xor-blobs contain exactly one start blob, and and-blobs consist of xor-blobs, only:

$$
\begin{array}{l}
\forall\, q : \mathsf{nodes}(blob) \bullet \\
\quad type(q) \in \{elementaryBlob, xorBlob, andBlob\} \wedge \\
\quad \textbf{if } type(q) = elementaryBlob \textbf{ then } \mathsf{outdegree}(q, contains) = 0 \wedge \\
\quad \textbf{if } type(q) = xorBlob \textbf{ then } \mathsf{outdegree}(q, startsWith) = 1 \wedge \\
\quad \textbf{if } type(q) = andBlob \textbf{ then } (q \rightarrow_{\text{contains}}) \subseteq \mathsf{nodes}(xorBlob)
\end{array}
$$

◊

To describe additional constraints on graph classes the constraint language GRAL ([EbeFra 92]) has been developed. GRAL is a $\mathcal{Z}$-like notation to describe structural properties of TGraphs, which can be translated into efficient algorithms that test the property described ([CapFra 91]).

A GRAL predicate is a first order predicate logic term containing only elements, which can be tested by polynomial algorithms. Basic elements of GRAL are

- ▷ a set of predicates to describe basic properties of graph elements (like $\mathsf{isIsolated}, \mathsf{isSink}$), or graphs (like $\mathsf{isTree}, \mathsf{isConnected}, \mathsf{isBipartite}$)
- ▷ a set of functions to compute sets of vertices (like nodes) and sets of edges (like edges) of a given type, or induced subgraphs (like eGraph).
- ▷ predicates and functions on sets and numbers,
- ▷ logical operators (like $\wedge, \vee$) and quantifiers (like $\forall$), which are allowed to range over finite sets, and
- ▷ (regular) path expressions.

A path expression describes a path in a graph in an abstract manner. A simple path expression consists of a single edge ($\leftarrow, \rightarrow, \rightleftarrows$). Each edge symbol may be annotated with restrictions on its type and attributes (e.g. $\rightarrow_{\text{contains}}$). Analogously, restrictions on nodes reached by following an edge can be expressed (e.g. $\rightarrow_{\text{contains}} \bullet_{\text{elem}}$). Simple path expressions may be combined to form more complex ones. Here, only regular structures (sequence, alternative, iteration) are allowed. Path expressions may be applied in postfix notation to vertices, thus delivering the set vertices which are reachable via a path matching the description, or they may denote predicates, if they are applied to two vertices in infix notation. Sets defined by path expressions may be computed by appropriate search algorithms in $\mathcal{O}(\#E \cdot s)$ time, if s is the number of symbols in the path expression ([CapFra 91]).

Example. To describe the set of all transitions in a state chart that are enabled through a given event e by a blob q, the following expression suffices:

$$\{\ t : \mathsf{nodes}(transition) \mid q \leftarrow^{*}_{\mathbf{contains}} \leftarrow_{\mathbf{source}}\ t \rightarrow_{\mathbf{triggeredBy}}\ e\ \}$$

◊

Since GRAL - as a constraint description language - allows also to define the type system and the incidence structure of graph classes. It depends on the user, where he puts the borderline between the ER-like description and the GRAL-part. We recommend to use the diagrammatic description as far as possible and to use GRAL only for the addition of further constraints, which are not expressible graphically by the ER-dialect chosen.

6 Programming With Graph Based Models

Up to now, we described the class of graphs used for modeling and gave a an approach to define subclasses as model classes for practical applications which consists of a combination of a graphical and a textual description. In this section we sketch how graph based models specified in this way may be implemented.

TGraphs may be handled efficiently by graph algorithms written in a pseudocode according to the algorithmic interface described in figure 4. Using an implementation technique like the one described in [Ebert 87] a direct implementation of pseudocode algorithms is possible, without any intermediate transformation. If the traversal operations are implemented in optimal time, complexity analysis of algorithms also carries through to the final implementation.

procedure init ()	initializes the graph as empty
for all v **in** V **do** ... **end**	processes all vertices v
for all e **in** E **do** ... **end**	processes all edges e
for all e **in** Λ(v) **do** ... **end**	processes all edges e incident with vertex v
for all e **in** Λ^+(v) **do** ... **end**	processes all edges e going out of vertex v
for all e **in** Λ^-(v) **do** ... **end**	processes all edges e going into vertex v
procedure vertexCount(): nat0	returns the number of vertices
procedure edgeCount(): nat0	returns the number of edges
procedure isEdge (v, w: vertex): edge	returns an edge from v to w
procedure areNeighbours (v, w: vertex): edge	returns an edge between v and w
procedure alpha (e: edge): vertex	returns the start vertex of edge e
procedure omega (e: edge): vertex	returns the end vertex of edge e
procedure this (e: edge): vertex	returns the vertex from whose Λ-sequence e was taken
procedure that (e: edge): vertex	returns the other vertex of e with respect to this (e)

Figure 4: Algorithmic Interface for TGraphs

The algorithmic interface has been realised by C++-software package ([DaEbLi 94]), which may be used for programming with TGraphs. This package works on the edge-oriented basis described in this paper, where the data type edge includes the edge object plus its direction. This allows to view a graph as being undirected or directed without any change of its representation. The view on the graph is only determined by the interface operations that are used.

Internally graphs are represented by symmetrically stored forward and backward adjacency lists ([Ebert 87]). The sets of attribute instances belonging to vertices and types are implemented by application dependent C++-objects, thus using the multiple inheritance concept of the language to implement the analogous concept in TGraphs.

Since according to the modeling approach given above ER-diagrams themselves may easily be modeled as ERGraphs (according to the (meta)-ER-schema in figure 5), the graph interface can be realised in such a way that creation and deletion of edges are controled by a constraint checker, which consults the ERGraph of the graph class with every updating operation.

If G is an ERGraph, which describes a graph class, its *entity* vertices correspond to vertex types and its *relationship*, *role*, and *grouping* vertices

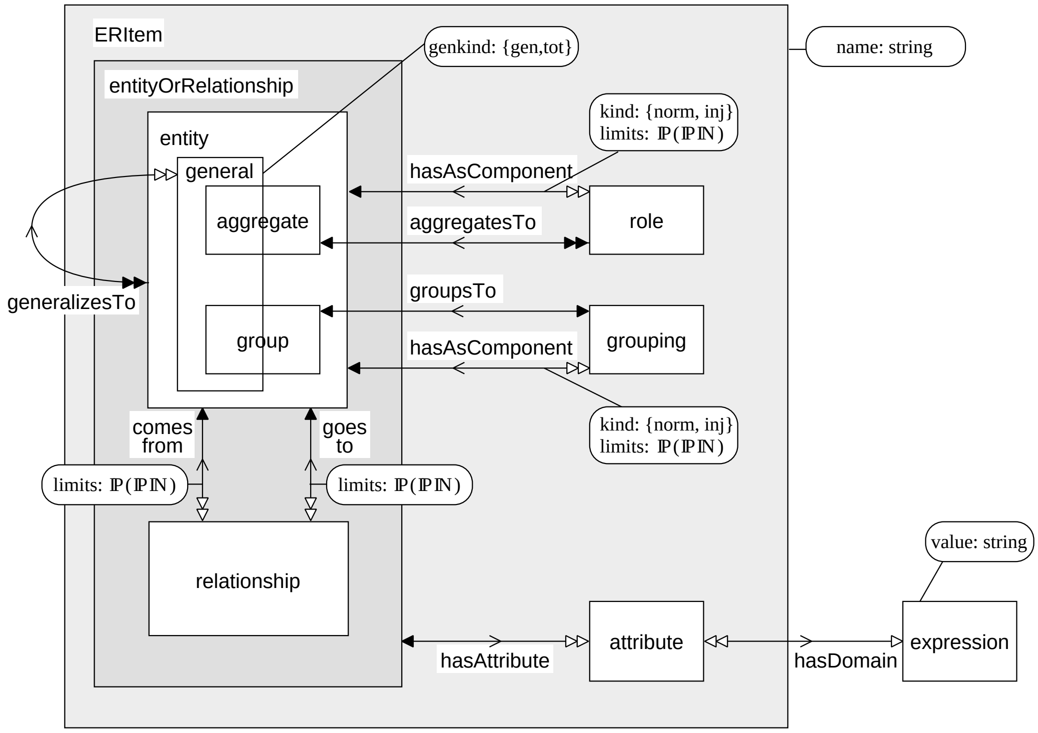

Figure 5: Schema for ERGraphs

correspond to edge types. The *generalizesTo* edges give the *isA*-relation. Furthermore, the *hasAsComponent*, *aggregatesTo*, *groupsTo*, *comesFrom*, and *goesTo* edges contain the incidence information, and the subgraphs induced by the sets of *entityOrRelationship*, *attribute*, and *expression* vertices describe the attribute schemes[3].

In addition to the checking via the ER-diagram, the checkings procedure derived from the GRAL predicates may be activated every time, a modification of the graph is performed. Thus, the declarative description of TGraph classes is directly usable in consistency assurance.

[3][CaEbWi 94] contains a formal description of the type system and the incidence system defined by an ER-diagram via its ERGraph.

7 Conclusion and Related Work

In this paper we gave an overview on our approach of using graph theory and graph algorithms in a consistent software technological manner in practical applications. A formal definition of the class of TGraphs was given together with a declarative approach of specifying subclasses for modeling purposes by giving their type and incidence structure graphically and by supplying a constraint specification language, which is translatable into efficient test procedures. Since this language includes regular path expressions, it already leads quite far. All aspects are implementable accordingly without any gap. Thus TGraphs build a framework which uniformly supports the aspects of expressiveness, formality, and efficiency.

In the KOGGE project ([Ebert 94]) software requirement and design documents are modeled inside a Meta-CASE-tool by TGraphs. Here, graphs are used as internal data structures exactly in the way described above. The modelling classes are specified using ER-diagrams and GRAL, and the repository component of the system assures that all specified integrity constraints are automatically kept by the system.

Since the class of TGraphs is defined in terms of objects, relations, attributes, etc. it fits very well into today's object-oriented way of thinking. Thus, TGraphs models are object-oriented structural models. Their definition shows also a way to formalizing structural aspects of object-oriented description.

The approach of characterizing graph classes by ER-diagrams, which have a widespread use in information modeling in database systems, may as well be read as a definition of a graph based data model for databases, including a powerful specification language for consistency conditions, namely GRAL. Comparable work has been done by e.g. Paredaens and others ([GyPaGu 90]) and Consens and Mendelzon ([ConMen 90]), where the emphasis lies on the definition of suitable query languages while we concentrate on the modeling process.

Foundations of declarative specification for graph languages have been explored by Courcelle ([Courcelle 90]). He showed that graph properties describable in monadic second order logic (MSOL) are efficiently testable for large graph classes. The specification language proposed in this paper is less powerful then MSOL (by offering sufficient expressive power for practical applications). We can guarantee that our declarative specification lead to efficient implementations, which is also shown by our translator in a constructive manner.

The concept of declarative graph class specification contrasts with graph grammar approaches, which assure constraints on graphs by specifying the operations, which build or update the graph. The graph grammar based approach has been used in the IPSEN project by Nagl and others ([EnLeNa 92] where graph grammar based specifications are used in the development of software engineering environments. In this context, the operational specification language PROGRESS ([Schuerr 91]) has been defined that allows users to specify even complex graph transactions. Furthermore, implementation is supported by a powerful graph database system.

In the graph grammar based approach to graph class specification it is quite easy to implement tools that control the construction of graphs matching the specification (e.g. syntax directed editors). On the other hand, given a more declarative specification it is easier to test whether a given graph is correct with respect to the specification. Test procedures for specified graph properties may be embedded in a module managing the graph structure and thus be separated from application programs. Any test may be invoked at any point in time, on system or user demand. Thus, flexible strategies for consistency control might be implemented. It is depends on the goals of graph based modeling in an application context which kind of specification should be used. From a modeling point of view, the declarative approach seems to be more intuitive or natural since the user is able to describe the class of graphs he thinks of without having to think of operations.

References

[CaEbWi 94] Martin Carstensen, Jürgen Ebert, Andreas Winter. *Entity–Relationship–Diagramme und Graphenklassen.* to appear as: Fachbericht Informatik 1994. Universität Koblenz 1994.

[CapFra 91] Carla Capellmann, Angelika Franzke. *GRAL: Eine Sprache für die graphbasierte Modellierung.* Universität Koblenz, Diplomarbeit, 1991.

[ConMen 90] Mariano P. Consens, Alberto O. Mendelzon. *GraphLog: a Visual Formalism for Real Life Recursion.* In: Proc. 9th Symposium on Principles of Database Systems. New York: ACM Press, 1990, pp. 417-424.

[Courcelle 90] Bruno Courcelle. *Graph rewriting: An algebraic and logic approach.* In: Jan van Leeuwen (ed.): Handbook of Theoretical Computer Science, Vol. B. Amsterdam: Elsevier 1990. pp. 193 - 242.

[DaEbLi 94] Peter Dahm, Jürgen Ebert, Christoph Litauer. *Das EMS-Graphenlabor.* in preparation.

[Diller 90] Anthony Diller. *$\mathcal{Z}$: An Introduction to Formal Methods.* Wiley 1990.

[EbeFra 92] Jürgen Ebert, Angelika Franzke. *Specification of a Graph Based Data Model for a CASE Tool.* Universität Koblenz, Fachbericht Informatik 5/92

[Ebert 87] Jürgen Ebert. *A Versatile Data Structure For Edge-Oriented Graph Algorithms.* Communications ACM 30 (1987,6), 513-519.

[Ebert 93] Jürgen Ebert. *Efficient Interpretation of State Charts.* in: Zoltán Ésik (Ed.). *Fundamentals of Computation Theory (FCT '93), Szeged, Hungary.* Berlin: Springer, LNCS 910, 1993, pp. 212-221.

[Ebert 94] Jürgen Ebert. *KOGGE: A Generator for Graphical Design Environments.* in preparation.

[EnLeNa 92] Gregor Engels, Claus Lewerentz, Manfred Nagl, Wilhelm Schäfer, Andreas Schürr. *Building Integrated Software Development Environments. Part I: Tool Specification.* ACM Transactions on Software Engineering and Methodology 1 (1992, 2), 135 - 167.

[GyPaGu 90] M. Gyssens, J. Paredaens, D. van Gucht. *A graph-oriented object database model.* In: Proc. 9th Symposium on Principles of Database Systems. New York: ACM Press, 1990, pp. 417-424.

[Harel 87] David Harel. *Statecharts: a Visual Formalism for Complex Systems.* Science of Computer Programming 8 (1987,3), 231-274

[Schuerr 91] Andreas Schürr. *Operationales Spezifizieren mit programmierten Graphersetzungssystemen.* Wiesbaden: Deutscher Universitätsverlag, 1991.

[Spivey 92] J.M. Spivey. *The Z Notation - A Reference Manual.* New York: Prentice Hall, 1992.