Graph Based Modeling and Implementation with EER/GRAL

J. Ebert, A. Winter, P. Dahm, A. Franzke, R. Süttenbach

University of Koblenz, Institute for Software Technology, Rheinau 1,
D-56075 Koblenz, email: ebert@informatik.uni-koblenz.de

Abstract. This paper gives a cohesive approach to modeling and implementation with graphs. This approach uses extended entity relationship (EER) diagrams supplemented with the $\mathcal{Z}$-like constraint language GRAL. Due to the foundation of EER/GRAL on $\mathcal{Z}$ a common formal basis exists. EER/GRAL descriptions give conceptual models which can be implemented in a seamless manner by efficient data structures using the GraLab graph library.
Descriptions of four medium size EER/GRAL-applications conclude the paper to demonstrate the usefulness of the approach in practice.

1 Introduction

Using graphs as a means for discussing problems, as a medium for formal reasoning, or as a paradigm for data structures in software is folklore in today's computer science literature. But most of the different approaches that use graphs are not used in a coherent way.

There are *different models* in use based on undirected or directed graphs, with or without multiple edges or loops. Sometimes graph elements are typed or attributed, sometimes they are not. Mathematical graph theory usually deals only with graph structure [Har72], whereas computer science usually uses graphs where vertices are distinguishable [Meh84]. In applications, graphs are often used *without a formal basis*. This leads to problems when assertions about the models have to be proved. Furthermore, graphs are frequently implemented using non-graph-based repositories that *do not match* the conceptual graph model exactly.

In this paper, we present a *coherent and consistent approach* to using graphs in a seamless manner

- as conceptual models,
- as formal mathematical structures, and
- as efficient data structures

without any discontinuity between these three aspects.

The approach, which is called the *EER/GRAL approach* throughout this paper, is based on extended entity relationship descriptions (EER diagrams, section 3.1) which are annotated by formal integrity conditions (GRAL assertions, section 3.2) in order to specifiy graphs, which are efficiently implementable by an appropriate C++ library (GraLab, section 4). A very general class of graphs is used (TGraphs, section 2) as basis.

As opposed to [EF95], where the theoretical basis is explained, the aim of this paper is to give an introduction into the approach with emphasis on its practical applicability.

Each of the applications sketched in section 5 has been described by technical reports which are publically available[1].

2 TGraphs

To make the approach as useful as possible a rather general kind of graphs has to be treated. *TGraphs* are used as the basic class of graphs. TGraphs are

- *directed*, i.e. for each edge one has a start vertex and an end vertex,
- *typed*, i.e. vertices and edges are grouped into several distinct classes,
- *attributed*, i.e. vertices and edges may have associated attribute-value pairs to describe additional information (where the attributes depend on the type), and
- *ordered*, i.e. the edges incident with a particular vertex have a persistent ordering.

All these properties are, of course, only optional. If a certain application only needs undirected graphs without any type, attribute or ordering, the respective properties may also be ignored.

2.1 Formal Definition

TGraphs as mathematical objects are specified using the $\mathcal{Z}$-notation [Spi92].

The basic *elements* of TGraphs are *vertices* and *edges*. With respect to a vertex an edge may have a *direction*, i.e. it may occur as an out-edge or as an in-edge. Graph elements may have a type and they may have attribute-value pairs associated.

$$ELEMENT ::= vertex\langle\langle \mathbb{N} \rangle\rangle \mid edge\langle\langle \mathbb{N} \rangle\rangle$$
$$VERTEX == \operatorname{ran} vertex$$
$$EDGE == \operatorname{ran} edge$$
$$DIR ::= in \mid out$$
$$[ID, VALUE]$$
$$typeID == ID$$
$$attrID == ID$$
$$attributeInstanceSet == attrID \pfun VALUE$$

Using these basic definitions the *structure* of a TGraph consists of its vertex set, its edge set and an incidence function, which associates to each vertex v the sequence of its incident edges together with their direction.

[1] Most of them can also be found via `http://www.uni-koblenz.de/~ist`.

TGraph

$V : \mathbb{F}\, VERTEX$
$E : \mathbb{F}\, EDGE$
$\Lambda : VERTEX \nrightarrow \mathrm{seq}(EDGE \times DIR)$
$type : ELEMENT \nrightarrow typeID$
$value : ELEMENT \nrightarrow attributeInstanceSet$

$\Lambda \in V \rightarrow \mathrm{iseq}(E \times DIR)$
$\forall\, e : E \bullet \exists_1\, v, w : V \bullet (e, in) \in \mathrm{ran}(\Lambda(v)) \land (e, out) \in \mathrm{ran}(\Lambda(w))$
$\mathrm{dom}\, type = V \cup E$
$\mathrm{dom}\, value = V \cup E$

This class of graphs is very general and allows object-based modeling of application domains in an unrestricted manner.

The formal definition of TGraphs by a $\mathcal{Z}$-text admits an equally formal definition of all concepts described in this paper (e.g. the semantics of EER diagrams and GRAL) and gives the opportunity for reasoning about all kinds of properties of graphs in a common and powerful calculus.

2.2 Modeling using TGraphs

TGraphs can be used as formal models in all application areas that are subject to object-based modeling.

It is useful to adopt a general *modeling philosophy* for TGraph-based software development in order to exploit the full power of the approach. We propose to use the following rules ([EF95])

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

Of course, these rules require some modeling decisions (e.g. to decide what can be viewed as "relevant"). They help to achieve an appropriate formal graph model in the modeling process. Some examples will be shown later in this paper.

3 Graph Classes

The set of possible TGraph models for a given application is usually a subset of the set of all TGraphs, at least if the application domain has some sensible structure. This leads to the task of defining *classes of TGraphs* in a formal manner.

Here we propose to use extended entity relationship descriptions (EER diagrams) for this purpose. These diagrams may be annotated by additional restrictions (GRAL assertions).

3.1 EER Diagrams

EER diagrams are able to denote information about graph classes in a straightforward manner:

- entity types denote vertex types,
- relationship types denote edge types,
- generalizations describe a vertex type hierarchy,
- incidences between relationship types and entity types describe restrictions on the incidence structure,
- attributes describe the attribute structure of vertices and edges, depending on their type, and
- higher-level modeling constructs like aggregation and grouping add additional structural information.

Example:
Fig. 1 shows the definition of a graph class *DFD* which gives the conceptual model of dataflow diagrams, i.e. it contains the metamodel for a dataflow language. The dataflow metamodel is used to generate an editor for dataflow diagrams [Drü96] with the KOGGE-Generator, described in section 5.2.

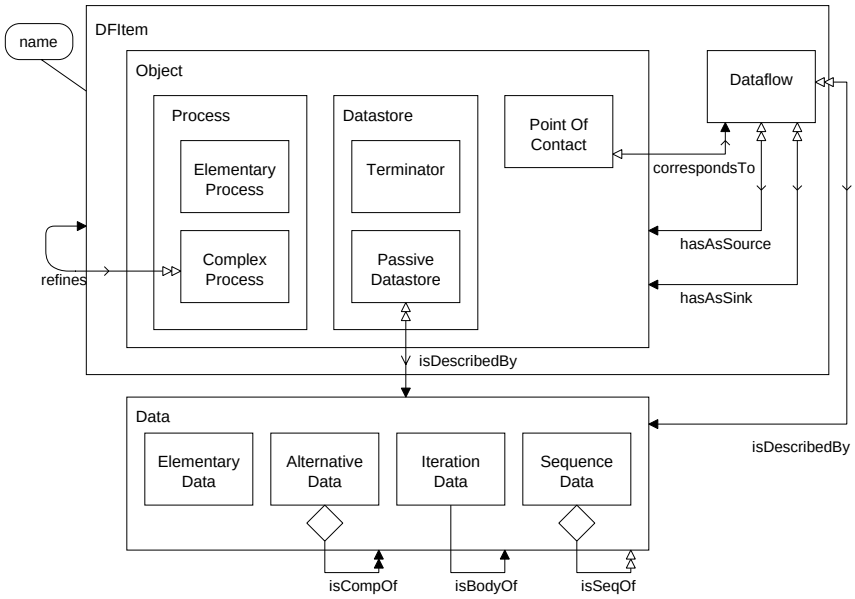

Fig. 1. EER diagram for dataflow diagrams

Dataflow diagrams describe procedural aspects through the main concepts *Process*, *Datastore* and *Dataflow*. These concepts are modeled as vertex types, which may be specialized. E.g. the concept *Process* is subdivided into *Complex Processes*, which are refined by further dataflow diagrams, and *Elementary Processes*. (Note, that specialization is depicted by inclusion of the respective rectangles of the vertex types, like in Venn diagrams.)

The relationships between these concepts are modeled using edge types. Each *Dataflow* connects exactly two *Objects*, its source and its sink. *Data* stored in *PassiveDatastores* or transported by *Dataflows* is described as regular structures in a datadictionary.

The refinement of *ComplexProcesses* by further *DFItems* is modeled using *refines*-edges. *PointOfContact*-vertices are used as surrogates for dataflows in a refinement. ■

[CEW96] describes a complete *formal semantics* of EER diagrams in terms of the TGraph class which is specified by a given diagram. This is done by defining an appropriate TGraph for the EER diagram itself. The set *SchemaGraph* of those TGraphs which describe EER diagrams is the domain of the semantic function

$$graphSpecOf : SchemaGraph \to GraphSpec$$

which assigns a graph class specification to every instance of *SchemaGraph*. (This function is formally defined using $\mathcal{Z}$.)

For a given TGraph $g \in SchemaGraph$ the result $graphSpecOf(g)$ is a specification of the set of instance TGraphs of the EER diagram described by g. The TGraph class corresponding to $graphSpecTo(g)$ is the set of all graphs h, which fulfill the specification. E.g., the TGraph class *SchemaGraph* is itself the set of TGraphs corresponding to that graph specification which is the picture of (the graph of) some meta EER diagram under *graphSpecOf*.

Since EER diagrams in practical applications are used to model the concepts of the application domain, they are also called *concept diagrams* in the following.

3.2 GRAL Assertions

EER diagrams only allow to describe the local structure of TGraphs, i.e. the types and attributes and their incidences together with only a few additional properties, like e.g. degree restrictions. In applications one has often more knowledge about the models. This knowledge can be formalized as an extension of the corresponding EER diagram.

We propose to use the $\mathcal{Z}$-like *assertion language GRAL* (GRAph specification Language), which allows to formulate further restrictions on the graph class specified by a diagram. GRAL is described in detail in [Fra96a].

GRAL assertions refer to the formal $\mathcal{Z}$-definition of TGraphs given in section 2. A GRAL assertion corresponding to an EER diagram D has the format

for G **in** D **assert**
$pred_1; \ldots; pred_k$

Here, the predicates $pred_1; \ldots; pred_k$ may be all kinds of $\mathcal{Z}$-predicates restricted only in such a way, that GRAL predicates are efficiently testable on those TGraphs which suit to the corresponding EER diagram. This efficiency is achieved

- by restricting all quantifiers to finite domains and
- by using a library of basic predicates and functions which can be computed efficiently.

A feature of GRAL which extends $\mathcal{Z}$ in the direction of TGraphs is the use of *path expressions*. Path expressions are regular expressions of edge/vertex symbols, which allow the description of paths in graphs. They are used to derive sets of vertices and to formulate reachability restrictions.
Path expressions are

- either simple, consisting of an edge symbol ($\rightharpoonup, \leftharpoonup, \rightleftharpoons$), optionally annotated with an edge type (like in $\rightharpoonup_{writes}$) and followed by a $\bullet$ symbol which may itself be annotated with a vertex type (like in $\rightharpoonup_{reviews}\bullet_{author}$)
- or composite: given two path expressions p_1, p_2
  - the sequence $p_1 p_2$,
  - the iteration p_1^* or p_1^+, and
  - the alternative $(p_1 \mid p_2)$

  are regular path expressions.

Given a path expression p and two vertices v, w,

- $v\ p$ denotes the set of vertices reachable from v along paths structured according to p
- $p\ v$ denotes the set of vertices from which v is reachable along paths structured according to p
- $v\ p\ w$ denotes the predicate that w is reachable from v along a path structured according to p

The semantics of path expressions and their application to vertices is described formally using $\mathcal{Z}$ in [Fra96b]. Since GRAL is embedded in $\mathcal{Z}$, GRAL assertions also have a $\mathcal{Z}$-compatible semantics.

Example:
The graph class *DFD* defined in fig. 1 has further properties which are shown as a GRAL assertion in fig. 2:

for G **in** *DFD* **assert**

(1) isDag(*refines*);

(2) $\{s_1, s_2 : Datastore \mid s_1 \leftharpoonup_{hasAsSink} \rightharpoonup_{hasAsSource} s_2\} = \varnothing$;

(3) $\forall p : ComplexProcess \bullet$
$p(\leftharpoonup_{hasAsSource} | \leftharpoonup_{hasAsSink}) = p \leftharpoonup_{refines} \bullet_{PointOfContact} \rightharpoonup_{correspondsTo}$

(4) $\forall c : PointOfContact \bullet$
$c \rightharpoonup_{correspondsTo} \rightharpoonup_{isDescribedBy} (\leftharpoonup_{IsCompOf} | \leftharpoonup_{isBodyOf} | \leftharpoonup_{isSeqOf})^*$
$\leftharpoonup_{isDescribedBy} (\rightharpoonup_{hasAsSink} | \rightharpoonup_{hasAsSource}) c$

(5) $\forall d : Dataflow \bullet$
$d(\rightharpoonup_{hasAsSource} | \rightharpoonup_{hasAsSink}) \bullet_{PassiveDatastore} \rightharpoonup_{isDescribedBy}$
$(\leftharpoonup_{IsCompOf} | \leftharpoonup_{isBodyOf} | \leftharpoonup_{isSeqOf})^* \leftharpoonup_{isDescribedBy} d.$

Fig. 2. GRAL Assertion for Dataflow Diagrams

Refinement of processes by further dataflow diagrams has to be cyclefree (1) and dataflows are not allowed between datastores (2). Refinement has to be structurally balanced, i.e. it has to be assured that dataflows being incident to a refined process find their correspondence in the refinement (3) as a point of contact. If a dataflow is described by a regular data description, the corresponding

point of contact has to have a conformant description (4). Accordingly, the regular descriptions of a data flow incident with a datastore, has to be conformant with the description of the datastore (5).

Balanced dataflow diagrams, an accompanying data dictionary entry and their TGraph-representation according to the graph class definition given in fig. 1 and 2 are shown in fig. 3.

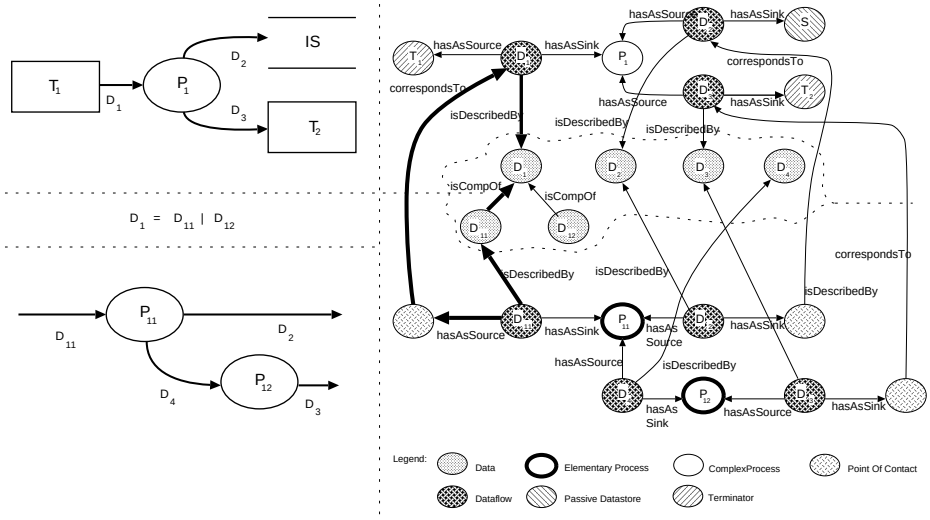

Fig. 3. Dataflow Diagram and its TGraph Representation

The emphasized arcs in the TGraph representation illustrate constraint 4 according to dataflow D_1. The other path expressions used in the integrity constraints could be followed analogeously. ■

Experience shows that regular path expressions are a powerful means for describing TGraph properties in practical applications. Some examples will be given below.

3.3 Modeling using Graph Classes

The description languages given by EER diagrams and GRAL-assertions are two aspects of a common integrated approach to specifying graph classes. Due to the common semantic basis given by their $\mathcal{Z}$-description one may use both formalisms in a *seamless manner*. It is up to the user to decide which formalism to choose in expressing knowledge about the TGraphs. EER diagrams are very well suited for formalizing local ("context-free") properties, whereas GRAL-assertions have the power to formalize even global ("context-sensitive") aspects. But there are properties (e.g. degree restrictions), which may be formulated in either way.

The *EER/GRAL modeling approach* is suited for a modular modeling process: At first one defines EER/GRAL specifications for several (smaller) graph classes. Then, these specifications are integrated

- by melting vertex types which represent the same information,
- by generalizing vertex types which represent similar information, and
- by connecting vertex types in different graph classes with additional edge types.

The GRAL assertions of the submodels are conjugated (after renaming), and additional global information may be added by further GRAL predicates.

4 Implementation

All modeling concepts described upto here have to be implemented by concrete graph software, if seamlessness of the approach shall be achieved: The GraLab (*GRAph LABoratory*) software package ([DEL94]) makes a set of C++ classes available to implement TGraphs directly and exactly as specified by EER diagrams. It allows to transform graph class definitions into vertex and edge types and into C++ classes which implement the attributes.

GraLab provides an *interface* to efficiently use and manipulate graph structures, as well as the types and attributes assigned to vertices and edges. Inside GraLab TGraphs are represented as internal data structures by symmetrically stored forward and backward adjacency lists [Ebe87].

The *structure* can be accessed and manipulated via a simple interface which includes methods to

- create and delete vertices and edges,
- traverse graphs,
- retrieve start and end vertices of edges,
- relink edges,
- change order of incidences,
- count vertices and edges, and
- retrieve edges between vertices.

This interface includes control structures for graph traversal (in the form of C++ macros) which allow a high-level programming of graph algorithms, which is very near to pseudocode used for describing graph algorithms.

The type and attribute part of TGraphs is implementable as a *type system* using GraLab. The type system contains the connection between each vertex/edge type and its (application dependent) attribute class. Each attribute class is a C++ class whose instances contain all attribute values assigned to a graph element (vertex or edge) of the type given. Furthermore, the type system implements the type hierarchy.

Attribute values can be accessed and modified via pointers to the specific attribute object. Type casts on the pointers returned are necessary to access single attribute values.

Due to the internal datastructure most of these operations requires linear afford. Creating, deleting, relinking and finding (the next) incident edge or adjacent vertex can be done in $\mathcal{O}(1)$ and the traversal of graphs depends linearly on the number of edges resp. vertices ([Ebe87]). In the projects described below graphs with more than 100 000 graph elements were handled without efficiency problems.

The GraLab software package gives the necessary completion of the EER/GRAL modeling approach with respect to implementation. Thus, all EER/GRAL models may be directly implemented by graph structures in a seamless manner.

5 Applications

The approach described in the previous chapters was developed in strong conjunction to software engineering projects and has been successfully applied in several different application areas. In the following, four projects will be sketched shortly. Each of them is described in more detail in the references given.

The examples are chosen in such a way that the following aspects of the approach are expressed: EER diagrams and GRAL assertions are used for modeling, partial models are integrated into a larger common model, and algorithmic graph theory on models is used to get efficient software.

5.1 Application: Method Modeling

Software analysis and design methods (like e.g. the *Object Modeling Technique (OMT)* of [RBP+91]) usually contain lots of different pictures with some intuitively given semantics to describe models of software systems.

An OMT model consists of three logical parts describing different views of the system to be analyzed; the *object model* describes the structure of objects, the *dynamic model* is concerned with execution aspects, and the *functional model* shows the transformation of values in the system.

The visual languages used for these pictures usually lack a formal basis. But since visual documents may be abstracted in graphs, it is possible to define the *abstract syntax* of these languages by TGraph classes. Then, the EER/GRAL description of these classes permits an integration and comparison of different visual languages as well as a formal reasoning about them.

In [BES96] an EER/GRAL formalization for OMT is given. This formalization is an *OMT metamodel*, since its instances are OMT models. There, the elements of OMT descriptions are modeled by three different graph classes, one for each model. Furthermore, these three EER/GRAL descriptions are integrated into one overall abstract model for the whole OMT-approach by merging vertex types in different (sub)models and by introducing additional edge types. The resulting OMT-model consists of an EER-Diagram with about 50 strongly connected vertextypes and more than 20 GRAL consistency constraints. The inconsistencies and incompleteness of OMT were solved by decisions of the authors. Since the descriptions allow a deliberate and formally based discussion of alternatives, it is possible to discuss these decisions on a common basis.

5.2 Application: Tool Building

Describing real systems with visual languages without the support of tools is practically infeasable because of the complexity of the systems and the methods available for description. Such a tool must help the developer with regard to

- the underlying concepts, i.e. the abstract syntax,
- the notation used, and
- general functions to develop a description conformant to a method.

The metaCASE system KOGGE (*KOblenz Generator for Graphical Design Environments*)[2] was developed to generate graphical editors for visual languages on the basis of EER/GRAL descriptions of their abstract syntax ([EC94]).

A tool for a given language, which is generated by KOGGE, is called a *KOGGE tool*. There are several KOGGE tools in use, including one for dataflow diagrams (cf. section 3) and one for the object-oriented software development method BON [NW95]. The BON-KOGGE ($\mathcal{BONSAI}$) [KU96] is used in Software Engineering education at University of Dortmund.

A KOGGE tool consists of the two physical parts – a tool description and the KOGGE base system.

The *base system* interprets the tool description at runtime in order to provide the user interface and to control its functions.

All KOGGE tools use the same base system, whereas the *tool description* is uniquely developed for any visual language. A tool description consists of three logical parts:

- an abstract syntax of the supported language,
- a set of statecharts, one for each tool operation, and
- a number of menu charts.

Since the abstract syntax inside KOGGE tools is described by EER diagrams, a KOGGE EER editor is used to specify and edit these diagrams. Analogously, a KOGGE statechart editor and a KOGGE menu editor are used to build the other parts of a tool description.

Inside KOGGE *TGraphs* are used for storing the tool specification and for representing the abstract syntax of the concrete documents, which are edited by the KOGGE tool. Both graphs are implemented using GraLab. One describes the tool itself according to the KOGGE meta EER specification, the other one represents the actual data according to the EER specification, which supplies the conceptual model of the language being edited.

An advantage of the KOGGE approach is that the abstract syntax of visual languages is given as an EER model. This allows to use the representation of an EER document inside the KOGGE EER editor as the tool specification graph of another KOGGE tool. Thus, one can develop KOGGE tools using KOGGE.

[2] The KOGGE Project was funded by the Stiftung Rheinland–Pfalz für Innovation, No. 8036-386261/112.

5.3 Application: Tour Planning

Schools for the handicapped have to organize a transportation service for their pupils who often are not able to reach the school by using e.g. public transport.

The aim of the MOTOS (*MOdular TOur Planning System*) project[3] is to develop a software component that supports *tour planning* for these schools. During the planning process, quite a lot of restrictions have to be considered that make tour planning a difficult task. Given geographical information and information on which pupils are waiting at which bus stop, MOTOS is meant to compute a set of tours that gets all pupils to their destinations while respecting all relevant constraints [GK96].

As in the other projects a TGraph class was defined in MOTOS. It is used as the (global) internal data structure for the tour planning algorithm, and represents the geographical data, the personal data, and the computed tours simultaneously.

Since MOTOS is part of a larger tour planning system, the interfaces to and from MOTOS had to be specified precisely. Basically, the MOTOS system consists of three modules:

- the *front end* that generates a MOTOS graph from the input data,
- the *planning component* that computes a suitable tour system, and
- the *back end* that hands the results over to the embedding system.

For the planning component different GRAL assertions are used: one for the initial state of the MOTOS graph, and one for the final outcome of the algorithm.

The MOTOS planning component uses well known graph algorithms like Dijkstra's shortest path algorithm, a traveling salesman algorithm for subgraphs, reachability algorithms etc. being confronted with specific aspects of the complex problem to be solved. The graph theoretical approach followed in MOTOS encouraged a kind of compositional algorithm design and enabled a quick solution quite fast. Thus, it was easy to experiment with different graph algorithms to find a good heuristic.

For MOTOS the EER/GRAL approach provided an adequate conceptual framework during the design phase. By using the GraLab, the designed data model could be implemented without much effort. Furthermore, graph theoretical concepts helped in finding a solution to the application problem which could be implemented without changing the view on the MOTOS data structure.

5.4 Application: Program Understanding

Maintenance and reuse of software requires a thorough understanding of software modules and their interdependence. It is impossible to predict all questions or classes of questions a reengineer might ask during the process of program understanding. Hence, a powerful program analysis facility is wanted which allows to answer any questions on user defined levels of granularity about programs written in different languages.

[3] The MOTOS project is a joint project of AED Süd, Meckenheim, Germany.

The GUPRO approach (*Generic Understanding of PROgrams*)[4] [EGW96], [EKW96] to program understanding is based on repositories which contain program information in graph data structures. The graph data structures can be consulted by a programming language independent analyzing mechanism.

The definition of the repository follows the modeling techniques described in this paper, namely EER/GRAL descriptions of TGraph classes are implemented by GraLab software. Thus, GUPRO is a generic approach, since EER/GRAL specifications can be used to adapt the system to different languages.

In the first part of the project an EER/GRAL specification of a heterogeneous software environment consisting of sources written in COBOL, CSP, PL/1, JCL, MFS, IMS-DBD and PSB was defined on a coarse grained level of granularity in tight cooperation with reengineers at Volksfürsorge [DFG+95]. The resulting model shows the main concepts of the different source languages and their interdependence which are used for supporting source code stocktaking. Here, GRAL assertions are used to specify additional edge types extending the abstract syntax, in order to simplify analysis.

In a first step, *isolated schemes* were defined representing the concepts of each single programming language on a fine grained level. In a second step, these single schemes were *integrated* into a common scheme by melting vertex types representing the same information, by generalization of vertex types representing similar information and by connecting vertex types with edge types.

The *GUPRO toolset* will consist of a parsing component and an analyzing component. It is implemented using GraLab functions.

The *parsing component* translates source codes into graph data structures matching the conceptual model in the EER diagram. It is generated from the programming language grammars, the user defined EER diagramms and their dependencies. The generated parser uses the GraLab library for creating instances of the conceptual model in the graph based repository [Dah95].

The *analyzing component* for language independent program analysis is also triggered by the conceptual model. An important part of the analyzing component is the query component, which allows any questions about the software stored in the repository according to the conceptual model. Retrieval of information from the repository uses a graph query language [Fra96b] suited to the graph based modeling approach described here.

Hence, GUPRO follows a closed approach of declarative conceptual program modeling using EER diagrams, storing program information in a repository using GraLab, and analyzing this repository using a query language in a consistent graph based manner.

---

[4] GUPRO is performed together with the IBM Scientific Center, Heidelberg, and the Volksfürsorge Unternehmensgruppe AG, (a german insurance company), Hamburg. GUPRO is supported by the Bundesminister für Bildung, Wissenschaft, Forschung und Technologie, national initiative on software technology, No. 01 IS 504.

6 Related Work

The main advantage of the EER/GRAL approach to graph based modeling is the coherent and consistent integration of several aspects, namely

- use of an EER dialect for declarative graph class specification,
- use of the GRAL extension of $\mathcal{Z}$ for specifying integrity constraints, and
- the efficient implementation of graphs using GraLab.

Formal semantics of ER-dialects have been defined by several authors. [Che76] already sketches a formal semantics of the basic entity relationship approach. Other sources are [NP80] and [Lie80]. An overview to the main concepts to EER Modeling including global considerations concerning derived schema components and (static) integrity constraints is given in [HK87]. [HG89] gives the semantics of a very general *EER-dialect*, which even allows entities to be attribute values. [TCGB91] discuss semantics for generalizations and specializations.

Older work on *integrity constraints* was done by [TN83]. [Len85] includes in his "semantic" entity relationship approach (SERM) integrity constraints into modells as rules. Constraints in first order logic are introduced by [Süd86] and [BT94]. Cardinality constraints are discussed by [Tha92].

Theoretical foundations for constructive *graph class descriptions* are laid by [Cou96], who uses monadic second order logic. Graph classes for concrete applications can also be specified by graph grammars (see [EK95]). PROGRES [Sch91], [SWZ95] is a language for specifying graph replacement systems, which can be used for this purpose. PROGRES also includes ER diagrams for specifying simple schemata. Thus, it includes also some declarative description elements, though they are weaker than those described here.

For the implementation of discrete structures as *internal structures* there exist efficient libraries like LEDA ([MN96]), though they are not directly adapted to such general graph types like TGraph e.g. directed an undirected graphs are stored differently, vertices are not typed, and the attribute structure is uniform for all vertices. For storing graphs persistently as *external structures* GRAS [KSW95] or PCTE [LW93] may be used. But then again TGraphs are not directly supported. Furthermore external storage leads to a tradeoff between the size of the graphs and the efficiency of graph traversals.

7 Conclusion

An overview on the EER/GRAL approach on graph based modeling was given. In order to define graph classes EER diagrams are used, which are extended by GRAL predicates. In conjunction with the GraLab C++-library this supplies a seamless way for modeling and implementation.

The modeling approach is formally based on $\mathcal{Z}$-specifications for the EER- and GRAL-definitions of TGraph classes. The approach has been successfully applied in various projects in different areas of information modeling.

Acknowledgement: The authors express their thanks for some help in compiling this paper to Ingar Uhe, Manfred Kamp, Bernt Kullbach, and Martin Hümmerich.

References

[BES96] F. Bohlmann, J. Ebert, and R. Süttenbach. An OMT Metamodel. Projektbericht 1/96, Universität Koblenz-Landau, Institut für Softwaretechnik, Koblenz, 1996.

[BT94] J. B. Behm and T. J. Teorey. Relative Constraints in ER Data Models. *R. A. Elmasri, V. Kouramajian, B. Thalheim (Eds.): Entity-Relationship Approach - ER'93, 12th International Conference on the Entity-Relationship Approach, Arlington, Texas, USA, December 15-17, 1993*, pages 46–59, 1994.

[CEW96] M. Carstensen, J. Ebert, and A. Winter. Entity-Relationship-Diagramme und Graphenklassen. to appear as Fachbericht Informatik, 1996, Institut für Softwaretechnik, Universität Koblenz-Landau, 1996.

[Che76] P. P.-X. Chen. The Entity–Relationship Model — Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[Cou96] B. Courcelle. Graph structure definition using monadic second-order languages. *In: Proceedings of the Workshop on Finite Models and Descriptive Complexity, Princeton, New Jersey, January 14-17, 1996, to appear in: AMS-DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 1996.

[Dah95] P. Dahm. PDL: Eine Sprache zur Beschreibung grapherzeugender Parser. Diplomarbeit D-305, Universität Koblenz-Landau, Fachbereich Informatik, Koblenz, Oktober 1995.

[DEL94] P. Dahm, J. Ebert, and C. Litauer. Das EMS-Graphenlabor 3.0. Projektbericht 3/94, Universität Koblenz-Landau, Institut für Softwaretechnik, Koblenz, 1994.

[DFG+95] P. Dahm, J. Fricke, R. Gimnich, M. Kamp, H. Stasch, E. Tewes, and A. Winter. Anwendungslandschaft der Volksfürsorge. Projektbericht 5/95, Universität Koblenz-Landau, Institut für Softwaretechnik, Koblenz, 1995.

[Drü96] M. Drüke. Dokumentation für den Datenflußdiagramm-Editor. Studienarbeit S 429, Universität Koblenz-Landau, Fachbereich Informatik, Koblenz, Mai 1996.

[Ebe87] J. Ebert. A Versatile Data Structure For Edge-Oriented Graph Algorithms. *Communications ACM*, 30(6):513–519, June 1987.

[EC94] J. Ebert and M. Carstensen. Ansatz und Architekur von KOGGE. Projektbericht 2/94, Universität Koblenz-Landau, Institut für Softwaretechnik, Koblenz, 1994.

[EF95] J. Ebert and A. Franzke. A Declarative Approach to Graph Based Modeling. *in: E. Mayr, G. Schmidt, G. Tinhofer (Eds.)* Graphtheoretic Concepts in Computer Science *Springer, Berlin, Lecture Notes in Computer Science, LNCS 903*, pages 38–50, 1995.

[EGW96] J. Ebert, R. Gimnich, and A. Winter. Wartungsunterstützung in heterogenen Sprachumgebungen, Ein Überblick zum Projekt GUPRO. *in F. Lehner (Hrsg.): Softwarewartung und Reengineering - Erfahrungen und Entwicklungen, Wiesbaden*, pages 263–275, 1996.

[EK95] H. Ehrig and M. Korff. Computing with Algebraic Graph Transformations - An Overview of Recent Results. *G. Valiente Feruglio and F. Rosello Llompart (eds): Proc. Colloquium on Graph Transformation and its Application in Computer Science. Universitat de les Illes Balears, 1995*, pages 17–23, 1995.

[EKW96] J. Ebert, M. Kamp, and A. Winter. Generic Support for Understanding Heterogeneous Software. Fachbericht Informatik 3/96, Universität Koblenz-Landau, Fachbereich Informatik, Koblenz, 1996.

[Fra96a] A. Franzke. GRAL : A Reference Manual. to appear as Fachbericht Informatik, Universität Koblenz-Landau, Fachbereich Informatik, Koblenz, 1996.

[Fra96b] A. Franzke. Querying Graph Structures with G^2QL. Fachbericht Informatik 10/96, Universität Koblenz-Landau, Fachbereich Informatik, Koblenz, 1996.

[GK96] S. Gossens and L. Kirchner. Projekt MOTOS Modellierung, Frontend und Backend. Studienarbeit S 410, Universität Koblenz-Landau, Fachbereich Informatik, Koblenz, Januar 1996.

[Har72] F. Harary. *Graph theory.* Addison-Wesley, Reading, Mass., 3 edition, 1972.

[HG89] U. Hohenstein and M. Gogolla. A Calculus for an Extended Entity-Relationship Model Incorporating Arbitrary Data Operations and Aggregate Functions. *C. Batini (Ed.): Entity-Relationship Approach: A Bridge to the User, Proceedings of the Seventh International Conference on Entity-Relationship Approach*, pages 129–148, 1989.

[HK87] R. Hull and R. King. Semantic Database Modelling: Survey, Applications, and Research Issues. *ACM Computing Surveys*, 19(3):201–260, September 1987.

[KSW95] N. Kiesel, A. Schürr, and B. Westfechtel. A Graph-Oriented (Software) Engineering Database System. *Information Systems, vol. 20, no. 1*, pages 21–52, 1995.

[KU96] A. Kölzer and I. Uhe. Benutzerhandbuch für die KOGGE-Tool BONsai, Version 2.0. Projektbericht 4/96, Universität Koblenz-Landau, Institut für Softwaretechnik, Koblenz, 1996.

[Len85] M. Lenzerini. SERM: Semantic Entity-Relationship Model. *P. P. Chen (ed.): Entity-Relationship Approach: The Use of ER Concept in Knowledge Representation, Proceedings of the Fourth International Conference on Entity-Relationship Approach, Chicago, Illinois, USA, 29-30 October 1985*, pages 270–278, 1985.

[Lie80] Y. E. Lien. On the Semantics of the Entity-Relationship Data Model. *P. P. Chen (Ed): Entity-Relationship Approach to Systems Analysis and Design. Proc. 1st International Conference on the Entity-Relationship Approach*, pages 155–168, 1980.

[LW93] J. Jowett L. Wakeman. *PCTE, The Standard for Open Repositories.* Prentice Hall, New York, 1993.

[Meh84] K. Mehlhorn. *Data structures and algorithms*, volume 2. Graph algorithms and NP-completeness. Springer, Berlin, 1984.

[MN96] K. Mehlhorn and S. Näher. LEDA. A Platform for Combinatorial and Geometric Computing. Technical report, Max-Planck-Institut für Informatik, 1996.

[NP80] P. A. Ng and J. F. Paul. A Formal Definition of Entity-Relationship Models. *P. P. Chen (Ed): Entity-Relationship Approach to Systems Analysis and Design. Proc. 1st International Conference on the Entity-Relationship Approach*, pages 211–230, 1980.

[NW95] J.-M. Nerson and K. Waldén. *Seamless Object-Oriented Software Architecture. Analysis and Design of Reliable Systems.* Prentice Hall, Englewood Cliffs, 1995.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, 1991.

[Sch91] A. Schürr. *Operationales Spezifizieren mit Graph Ersetzungssystemen, Formale Definitionen, Anwendungsbeispiele und Werkzeugunterstützung.* Deutscher Universitaetsverlag, Wiesbaden, 1991.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual.* International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire, UK, 2 edition, 1992.

[Süd86] N. Südkamp. Enforcement of Integrity Constraints in an Entity Relationship Data Model. Bericht 8607, Institut für Informatik und Praktische Mathematik, Christian Albrechts Universität, Kiel, September 1986.

[SWZ95] A. Schürr, A.J. Winter, and A. Zündorf. Graph Grammar Engineering with PROGRES. *W. Schäfer (Ed.): ESEC '95, 5th European Software Engineering Conference*, pages 219–234, 1995.

[TCGB91] L. Tucherman, M. A. Casanova, P. M. Gualandi, and A. P. Braga. A Proposal for Formalizing and Extending the Generalization and Subset Abstractions in the Enity-Relationship Model. *F. H. Lochovsky (Ed.): Entity-Relationship Approach to Database Design and Querying, Proceedings of the Eight International Conference on Entity-Relationship Approach, Toronto, Canada, 18-20 October, 1989*, pages 27–41, 1991.

[Tha92] B. Thalheim. Fundamentals of Cardinality Constraints. *G. Pernul, A. M. Tjoa (Eds.): Entity-Relationship Approach - ER'92, 11th International Conference on the Entity-Relationship Approach, Karlsruhe, Germany, October 7-9, 1992*, pages 7–23, 1992.

[TN83] Y. Tabourier and D. Nanci. The Occurrence Structure Concept: An Approach to Structural Integrity Constraints in the Entity-Relationship Model. *P. P. Chen (Ed.): Proc. 2nd Int. Conf. on the Entity-Relationship Approach (ER'81)*, pages 73–108, 1983.