UNIVERSITÄT KOBLENZ · LANDAU

**Graph Based Modeling and
Implementation with EER/GRAL**

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

11/96

Universität
Koblenz-
Landau

Fachbereich
Informatik

infko

Fachberichte
INFORMATIK

Graph Based Modeling and Implementation with EER/GRAL

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

April 30, 1996

Abstract

This paper gives a closed approach to modeling and implementation with graphs. This approach uses extended entity relationship (EER) diagrams complemented 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 the different approaches that use graphs are mostly not used in a coherent way.

There are *different models* in use based on undirected or directed graphs with or without multiple edges of loops. Sometimes graph elements are typed or attributed, sometimes not. Mathematical graph theory is usually plainly on graph structure [Har72], whereas computer science usually uses graphs where vertices are distinguishable [Meh84]. In applications graphs are often used *without a formal basis*, what 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

---

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

> ▷ as conceptual models,
> ▷ as formal mathematical structures, and
> ▷ as efficient data structures

without any fractures between these three aspects.

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

To demonstrate the usefulness of the approach four medium-size applications are described (section 5), which successfully used this approach.

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 needs only 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.

$$
\begin{array}{ll}
ELEMENT ::= vertex \langle\!\langle \mathbb{N} \rangle\!\rangle \mid edge \langle\!\langle \mathbb{N} \rangle\!\rangle & [ID,\ VALUE] \\
VERTEX == \operatorname{ran} vertex & typeID == ID \\
EDGE == \operatorname{ran} edge & attrID == ID \\
DIR ::= in \mid out & attributeInstanceSet == attrID \nrightarrow VALUE
\end{array}
$$

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.

$$
\begin{array}{|l}
\underline{\ TGraph\ }\\
\quad V : \mathbb{F}\ VERTEX \\
\quad E : \mathbb{F}\ EDGE \\
\quad \Lambda : VERTEX \nrightarrow \mathrm{seq}(EDGE \times DIR) \\
\quad type : ELEMENT \nrightarrow typeID \\
\quad value : ELEMENT \nrightarrow attributeInstanceSet \\
\hline
\quad \Lambda \in V \rightarrow \mathrm{iseq}(E \times DIR) \\
\quad \forall\, e : E \bullet \exists_1 v, w : V \bullet (e, in) \in \mathrm{ran}(\Lambda(v)) \wedge (e, out) \in \mathrm{ran}(\Lambda(w)) \\
\quad \mathrm{dom}\, type = V \cup E \\
\quad \mathrm{dom}\, value = V \cup E
\end{array}
$$

This class of graphs is very general and allows object-based modeling of applica-
tion 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 strength 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 afford some modeling decisions (e.g. to decide what can be
viewed as "relevant"). They help to arrive at 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 in 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 straight-forward manner:

  ▷ entity types denote vertex types,
  ▷ relationship types denote edge types,
  ▷ generalisations 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: Figure 1 shows the definition of a graph class *DFD* which gives the conceptual model of dataflow diagrams, i.e. the metamodel for a dataflow language. Dataflow diagrams describe procedural aspects by use of 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 *DefaultProcesses*, 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 with edge types. Each *Dataflow* connects exactly two *Objects*, its source and its sink. *Data* stored in *Datastores* or transported by *Dataflows* are described as regular structures connected to them.

The refinement of *Processes* by further *DFItems* is modeled with *refines*-edges. *PointOf Contact*-vertices are used as surrogates for dataflows incident to a refined process in its refinement.

The dataflow metamodel is used to generate an editor for dataflow diagrams [Drü96] with the KOGGE-Generator, described in section 5.2.   ∎

[CEW95] 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 a given EER diagram itself. The set *SchemaGraph* of those TGraphs which describe EER diagrams is the domain of the semantic function

$$graphSpecOf : SchemaGraph \rightarrow GraphSpec$$

which assigns a graph class specification to every instance of *SchemaGraph*. (This function is formally defined using $\mathcal{Z}$.)
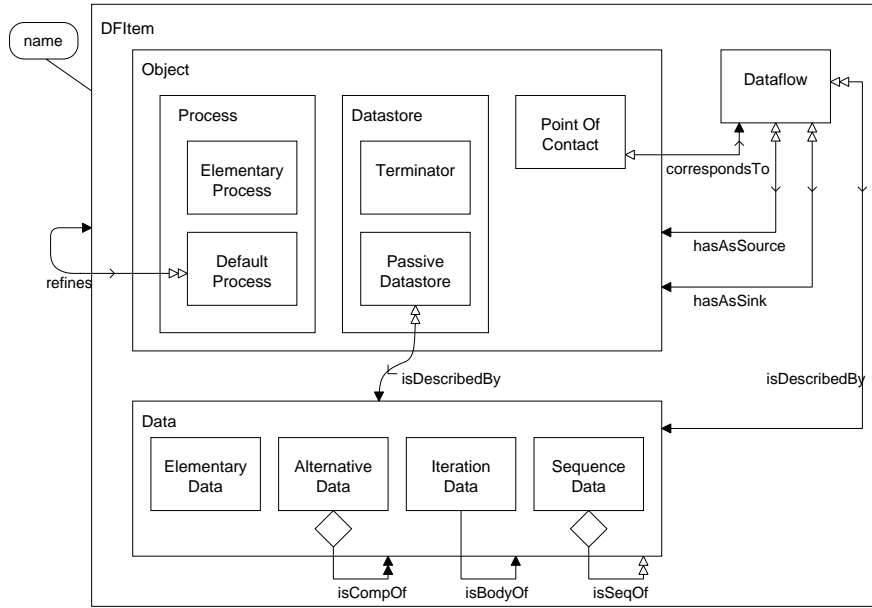
Figure 1: EER diagram for dataflow diagrams

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 fulfil 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. Some examples will be given below.

3.2 GRAL Assertions

EER diagrams only allow to describe the local structure of TGraphs, especially 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 formulized 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**
> $\quad pred_1; \ \ldots; \ pred_k$

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
> ▷ using a library of basic predicates and functions which are efficiently computable.

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 ($\rightharpoonup, \leftharpoonup, \rightleftharpoons$), possibly annotated with an edge type (like in $\rightharpoonup_{writes}$) and followed by a • 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 the 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 have also a $\mathcal{Z}$-compatible semantics.

Example: The graph class *DFD* defined in figure 1 has further properties which are shown as a GRAL assertion in figure 2: Refining of processes by further dataflow diagrams has to be cyclefree (1). 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 (2) 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 (3). Accordingly, the regular descriptions of a data flow incident with a datastore, has to be conformant with the description of the datastore (4).

Balanced dataflow diagrams and their TGraph-representation according to the

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

(1)   isDag($refines$)

(2) $\forall\, p : DefaultProcess \mid \text{indegree}_{refines}(p) \neq 0 \bullet$

$\qquad p(\overset{\leftarrow}{}_{hasAsSource}|\overset{\leftarrow}{}_{hasAsSink}) = p \overset{\leftarrow}{}_{refines} \bullet_{PointOfContact} \overset{\rightarrow}{}_{correspondsTo}$

(3) $\forall\, c : PointOfContact \bullet$

$\qquad c \overset{\rightarrow}{}_{correspondsTo} \overset{\rightarrow}{}_{isDescribedBy} \left(\overset{\leftarrow}{}_{IsCompOf}|\overset{\leftarrow}{}_{isBodyOf}|\overset{\leftarrow}{}_{isSeqOf}\right)^{*}$

$\qquad \overset{\leftarrow}{}_{isDescribedBy} \left(\overset{\leftarrow}{}_{hasAsSink}|\overset{\leftarrow}{}_{hasAsSource}\right)c$

(4) $\forall\, d : Dataflow \bullet$

$\qquad d(\overset{\leftarrow}{}_{hasAsSource}|\overset{\leftarrow}{}_{hasAsSink}) \bullet_{PassiveDatastore} \overset{\rightarrow}{}_{isDescribedBy} \overset{\leftarrow}{}_{IsIn}{}^{*} \overset{\leftarrow}{}_{isDescribedBy}$

Figure 2: GRAL Assertion for Dataflow Diagrams

graph class definition given in figures 1 and 2 are shown in figure 3.
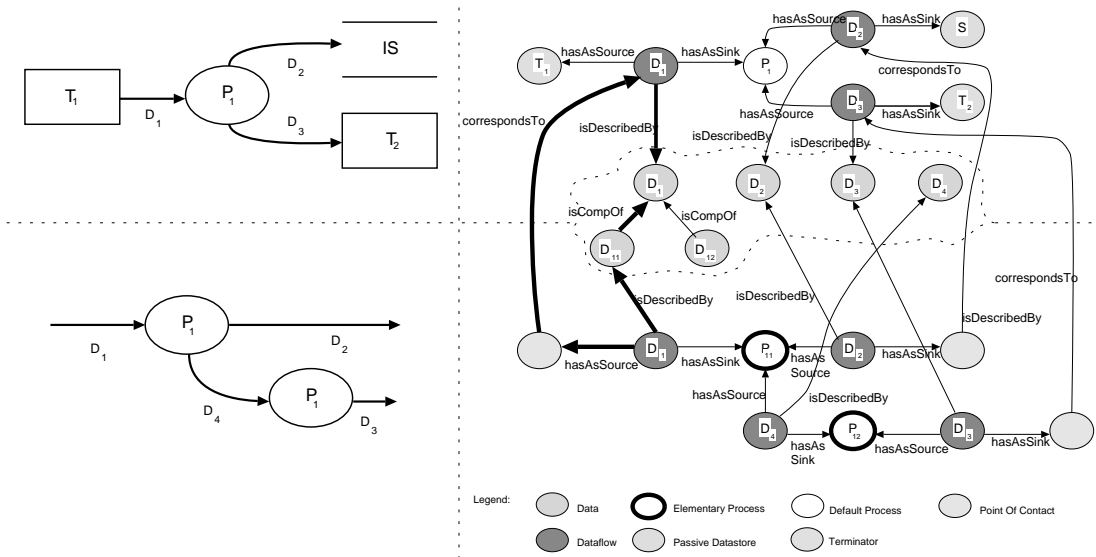

Figure 3: Dataflow Diagram and its TGraph Representation

The emphasized circle in the TGraph representation illustrates constraint 3 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]) gives a set of C++ classes to directly implement TGraphs 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,
> ▷ 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.

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. The GraLab interface is depicted in more detail in appendix A.

5   Applications

The approach described in the previous chapters has been successfully applied in several different applications 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 different 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   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.

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

In [RBP+91] 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.

In [BES95] an EER/GRAL formalization for OMT is given. This formalization is called the *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 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.

Figure 7 in appendix B shows the integrated OMT metamodel of [BES95] including its EER diagram and its GRAL assertion. The latter contains more than 20 consistency constraints.

5.2   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*)[1] 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 editing EER diagrams. A KOGGE tool consists of 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. With such a tool the user or modeler can produce a number of visual documents.

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 can be 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 capturing 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.

---

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

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

The EER specification of KOGGE tool descriptions is shown in figure 8 in appendix C.

5.3   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[2] 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 (i.e. a tour system) that gets all pupils to their destinations while respecting all relevant constraints.

Tour planning is one of the classical application areas of graph theory. Thus, it was decided to follow a graph based approach to software development also in MOTOS. In this process, the first step was to define the TGraph class to be used in the planning process. 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. There are two GRAL-assertions: 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.

---

[2] The MOTOS project is a joint project of AED Süd, Meckenheim, Germany and the Institute for Software Technology, University of Koblenz–Landau, Koblenz, Germany.

In summary, 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 point of view on the MOTOS data structure.

In appendix D figure 9 shows the EER/GRAL definition of the MOTOS graph class, including the different GRAL assertions.

5.4 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.

The GUPRO approach (*Generic Understanding of PROgrams*)[3] [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 description of TGraph classes 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 more 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 com-

---

ponent. 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 will be generated from the programming language grammars, the user defined EER diagram 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.

The coarse-grained model cited above is defined in Figure 10 in appendix E.

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 EER diagrams 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 model as rules. Relative constraints in first order logic are introduced by [BT94]. Cardinality constraints are discussed by [Tha92].

Graph classes for concrete applications can also be specified by graph grammars (see [EK95]). PROGRES [Sch91], [SWZ95] gives a language for specifying graph replacement systems, which can be used for this purpose. PROGRES also includes ER diagrams for specifying simple schemata. Theoretical foundations for

constructive graph class descriptions are laid by [Cou96], who uses monadic second order logic.

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. For storing graphs persistently as external structures graph storage software like GRAS [KSW93], [KSW95] or even data base systems can be used, if the interface is adapted for our approach. Of course, there is 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: For defining 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 is 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 Manfred Kamp, Bernt Kullbach, and Martin Hümmerich.

References

[BES95]   F. Bohlmann, J. Ebert und R. Süttenbach. Ein OMT-Metamodell. Interner Projektbericht 1/96, Universität Koblenz-Landau, Institut für Softwaretechnik, Koblenz, 1995.

[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.

[CEW95]   M. Carstensen, J. Ebert und A. Winter. Entity-Relationship-Diagramme und Graphenklassen. erscheint als Fachbericht Informatik, 1995, Institut für Softwaretechnik, Universität Koblenz-Landau, 1995.

[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 und C. Litauer. Das EMS-Graphenlabor 3.0. Projektbericht, Universität Koblenz-Landau, Institut für Softwaretechnik, Koblenz, 1994.

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

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

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

[EC94]   J. Ebert und M. Carstensen. Ansatz und Architekur von KOGGE. Interner 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 und 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 und 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. Fachbericht Informatik 11/96, 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.

[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 Enity-Relationship Approach, Rome, Italy, November 16-18, 1988*, 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.

[KSW93]   N. Kiesel, A. Schürr, and B. Westfechtel. Design and Evaluation of GRAS, a Graph-Oriented Database System for Engineering Applications. *H.Y. Lee, Th.F. Reid, St. Jarzabek (eds.): CASE '93 Proc. 6th Int Workshop on Computer-Aided Software Engineering*, pages 272–286, 1993.

[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.

[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.

[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.

[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.

[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.

A   GraLab Interface

The following tables show the signatures of the main methods of the GraLab library to manipulated an examine TGraphs, to instantiate type systems and to handle types and attributes.

create and delete vertices and edges		
G_vertex	createVertex()	creates a new vertex
G_edge	createEdge(G_vertex v, G_vertex w)	creates a new edge from v to w
void	deleteVertex(G_vertex v)	deletes vertex v and incident edges
void	deleteVertex(G_vertex e)	deletes edge e
traverse graphs*		
G_forAllVertices(g,v) {...}		process in graph g each vertex v
G_forAllEdges(g,e) {...}		process in graph g each edge e
G_forAllIncidentEdges(g,v,e) {...}		process in graph g each edge e incident to vertex v
G_forAllInEdges(g,v,e) {...}		process in graph g each edge e going into vertex v
G_forAllOutEdges(g,v,e) {...}		process in graph g each edge e going out of vertex v
retrieve start and end vertices		
G_vertex	alpha(G_edge e)	get start vertex of edge e
G_vertex	omega(G_edge e)	get end vertex of edge e
G_vertex	thisV(G_edge e)	get "this" vertex† of edge e
G_vertex	thatV(G_edge e)	get "that" vertex‡ of edge e
relink edges		
void	changeAlpha(G_edge e, G_vertex v)	relink an edge to another start vertex
void	changeOmega(G_edge e, G_vertex v)	relink an edge to another end vertex
void	changeThis(G_edge e, G_vertex v)	relink an edge to another this vertex
void	changeThat(G_edge e, G_vertex v)	relink an edge to another that vertex
change order of incidences		
void	putBefore(G_edge e, G_edge f)	put edge e before edge f in $\Lambda(\text{this}(e))$
void	putAfter(G_edge e, G_edge f)	put edge e after edge f in $\Lambda(\text{this}(e))$
count vertices and edges		
unsigned	vertexCount()	get the number of vertices
unsigned	edgeCount()	get the number of edges
unsigned	degree(G_vertex v)	get the number of edges incident with vertex v
unsigned	inDegree(G_vertex v)	get the number of edges going into vertex v
unsigned	outDegree(G_vertex v)	get the number of edges going out of vertex v
retrieve edges between vertices		
G_edge	edgeFromTo(G_vertex v, G_vertex w)	get edge going from vertex v to vertex w
G_edge	edgeBetween(G_vertex v, G_vertex w)	get edge between vertices v and w

---

* implemented as macros

† the vertex from whose Λ-sequence e was taken

‡ the other vertex of e with respect to this(e)

Figure 4: Methods for Manipulating and Examining Structure

defining new types		
G_type	newType(char *tId, G_attrSchema aSch)	creates a new type with attribute schema aSch*
G_type	getType(char *tId)	returns a type given by an identifier
handling the type hierarchy		
void	setIsA(G_type subType, G_type superType)	make subType a sub type of superType
int	isA(G_type subType, G_type superType)	checks whether subType is a sub type of superType

* All attribute classes must be registered and can be identified by objects of class G_attrSchema.

Figure 5: Methods for Type Systems

defining type and attribute object		
void	setVType(G_vertex v, G_type t, G_attribute *pA)	set type and attribute object of vertex v
void	setEType(G_vertex v, G_type t, G_attribute *pA)	set type and attribute object of edge e
accessing type and attribute object		
G_type	getVType(G_vertex v)	get type of vertex v
G_type	getEType(G_edge e)	get type of edge e
int	isAV(G_vertex v, G_type t)	checks whether the type of v is a subtype of t
int	isAE(G_edge e, G_type t)	checks whether the type of e is a subtype of t
G_attribute *	getPVAttr(G_vertex v)	get attribute object of vertex v
G_attribute *	getPEAttr(G_edge e)	get attribute object of edge e

Figure 6: Methods for Type and Attribute Handling

B   OMT Metamodel

Figure 7 shows the graph class of OMT models, which integrates the object model, the dynamic model and the functional model.

The EER diagram is extended by several GRAL assertions. The first group of predicates deals with the object model.

for G **in** *OMT* **assert**

$(O1)$ $\forall\, q : V_{Qualifier};\ r : V_{Association} \mid q \xleftarrow{}_{isQualified} r \bullet$

         $q \xleftarrow{}_{qualifies} \xleftarrow{}_{hasAttribute} \xleftarrow{}_{relatesTo} r \xrightarrow{}_{relatesTo} \xrightarrow{}_{attachedTo} q\ ;$

$(O2)$ $\forall\, q : V_{Qualifier};\ r : V_{Association} \mid q \xleftarrow{}_{isQualified} r \bullet \mathrm{outdegree}_{relatesTo}(r) = 2\ ;$

$(O3)$ $\forall\, c : V_{Class} \ \bullet \neg \left(c\, (\xrightarrow{}_{isSuperclassIn} \xleftarrow{}_{isSubclassIn})^{+}\, c \right)\ ;$

$(O4)$ $\forall\, o : V_{Object};\ t : V_{ObjectTuple} \mid o \xrightarrow{}_{consistsOf} t \bullet$

         $t \xleftarrow{}_{isTagIn} \xleftarrow{}_{hasAttribute} \xleftarrow{}_{isInstanceOf} o\ ;$

$(O5)$ $\forall\, p : V_{Operation} \mid \mathrm{indegree}_{hasAbstractOperation}(p) > 0 \bullet$

         $\exists\, c : V_{Class} \mid p \xleftarrow{}_{hasAbstractOperation} \xrightarrow{}_{isSuperclassIn} \xleftarrow{}_{isSubclassIn} c\ ;$

$(O6)$ $\forall\, p : V_{Operation};\ c : V_{Class} \mid$

         $p \xleftarrow{}_{hasAbstractOperation} (\xrightarrow{}_{isSuperclassIn} \xleftarrow{}_{isSubclassIn})^{+} c \ \bullet$

         $(\exists\, p_1 : V_{Operation} \bullet p.name = p_1.name$

             $\wedge\, c\, (\xrightarrow{}_{isSubclassIn} \xleftarrow{}_{isSuperclassIn})^{*} \xrightarrow{}_{hasOperation} p_1)\ ;$

$(O7)$ $\forall\, p, p_1 : V_{Operation} \mid$

         $p.name = p_1.name \wedge \mathrm{indegree}_{hasAbstractOperation}(p) > 0 \bullet$

         $\neg\, (p \xleftarrow{}_{hasAbstractOperation} (\xleftarrow{}_{isSubclassIn} \xrightarrow{}_{isSuperclassIn})^{+}$
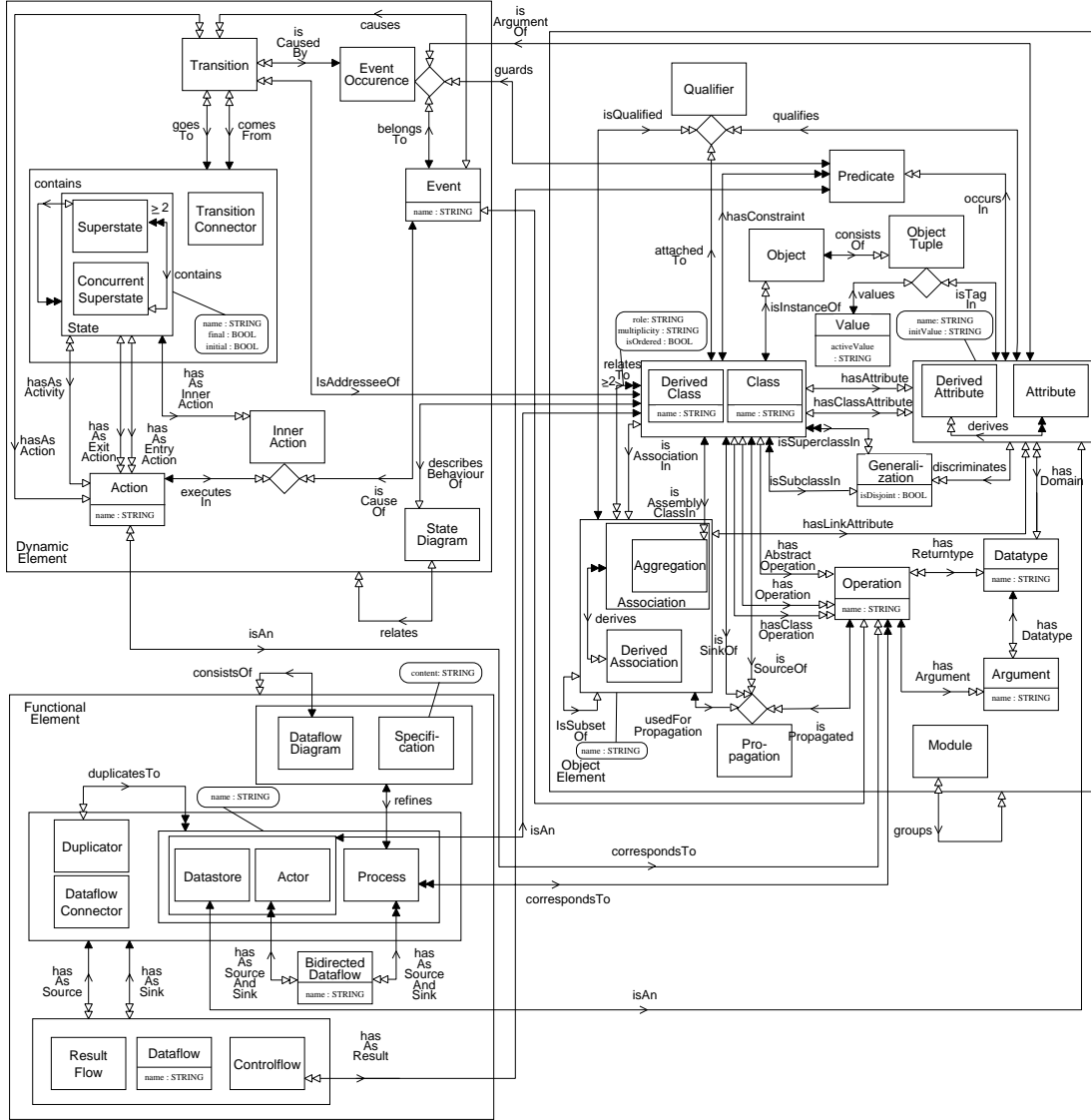
             $\xrightarrow{}_{hasOperation} p_1)\ ;$

Figure 7: Metamodel of OMT

$(O8)$ $\quad \forall\, g : V_{Generalization};\ a : V_{Attribute} \mid g \xleftarrow{}_{discriminates} a \bullet$

$\qquad a \xleftarrow{}_{hasAttribute} \xrightarrow{}_{isSuperclassIn} g\ ;$

$(O9)$ $\quad \forall\, po : V_{Propagation};\ p : V_{Operation};\ r : V_{Association} \mid$

$\qquad r \xrightarrow{}_{usedForPropagation} po \xleftarrow{}_{isPropagated} p \bullet$

$\qquad p \xleftarrow{}_{hasOperation} \xleftarrow{}_{relatesTo} r \xrightarrow{}_{relatesTo} \xrightarrow{}_{isSinkOf} po\ ;$

$(O10)$ $\quad \forall\, po : V_{Propagation};\ c : V_{Class};\ p : V_{Operation} \mid$

$\qquad c \xrightarrow{}_{isSinkOf} po \xleftarrow{}_{isPropagated} p \bullet$

$\qquad \exists\, p_1 : V_{Operation} \bullet p_1 \xleftarrow{}_{hasOperation} c \wedge p.name = p_1.name\ ;$

$(O11)$ $\quad \forall\, a : V_{Attribute};\ pr : V_{Predicate} \mid a \xrightarrow{}_{occursIn} pr \bullet$

$\qquad pr \xleftarrow{}_{hasConstraint} \xrightarrow{}_{hasAttribute} a\ ;$

$(O12)$ $\quad \forall\, c : V_{DerivedClass} \bullet c \xrightarrow{}_{hasAttribute} \subseteq V_{DerivedAttribute}\ ;$

$(O13)$ $\quad \forall\, r : V_{Association};\ r_1 : V_{DerivedAssociation} \mid r \xrightarrow{}_{derives} r_1 \bullet$

$\qquad r_1 (\xrightarrow{}_{relatesTo} \xleftarrow{}_{relatesTo})^* r\ ;$

$(O14)$ $\quad \forall\, r_1, r_2 : V_{Association} \mid r_1 \xrightarrow{}_{relatesTo} r_2 \bullet \exists\, c_1, c_2 : V_{Class} \bullet$

$\qquad c_1 \xleftarrow{}_{relatesTo} r_1 \xrightarrow{}_{relatesTo} c_2 \xleftarrow{}_{relatesTo} r_2 \xrightarrow{}_{relatesTo} c_1\ ;$

The second group pertains to with the dynamic model.

for G **in** OMT **assert**

$(D1)$ $\quad \forall\, s : V_{States} \mid s.final = true \bullet \text{indegree}_{goesTo}(s) \;=\; 0\ ;$

$(D2)$ $\quad \forall\, s : V_{Superstate} \bullet \#\{e : V_{State} \mid e.initial = true \wedge s \rightharpoonup_{contains} e\} = 1\ ;$

$(D3)$ $\quad \forall\, s : V_{Superstate} \bullet \neg\,(s \rightharpoonup^{+}_{contains} s)\ ;$

$(D4)$ $\quad \forall\, s : V_{ConcurrentSuperstate} \bullet \neg\,(s \rightharpoonup^{+}_{contains} s)\ ;$

The third group formulates constraints on the functional model.

for G **in** OMT **assert**

$(F1)$ $\quad \forall\, v, w : V_{Actor} \cup V_{Process};\ \ d : V_{Dataflow} \mid v \leftharpoonup_{hasAsSink} d \rightharpoonup_{hasAsSource} w \bullet$
$\qquad\quad type(v) = Process \ \ \vee\ \ type(w) = Process\ ;$

$(F2)$ $\quad \forall\, c : V_{DataflowConnector} \mid \text{indegree}_{hasAsSource}(c) = 1 \bullet$
$\qquad\quad \text{indegree}_{hasAsSink}(c) \geq 2\ ;$

$(F3)$ $\quad \forall\, c : V_{DataflowConnector} \mid \text{indegree}_{hasAsSink}(c) = 1 \bullet$
$\qquad\quad \text{indegree}_{hasAsSource}(c) \geq 2\ ;$

$(F4)$ $\quad \forall\, c : V_{DataflowConnector} \bullet$
$\qquad\quad \text{indegree}_{hasAsSink}(c) = 1 \vee \text{indegree}_{hasAsSource}(c,) = 1\ ;$

$(F5)$ $\quad \forall\, c : V_{Controlflow} \ \bullet c \rightharpoonup_{hasAsSink} \cup\ c \leftharpoonup_{hasAsSource} \subseteq V_{Process}\ ;$

$(F6)$ $\quad \forall\, r : V_{ResultFlow} \ \bullet r \rightharpoonup_{hasAsSink} \subseteq v_{Datastore}\ ;$

The last group, glues the partial models together, by describing integration constraints.

for G **in** OMT **assert**

$(C1)$ $\quad \forall\, a : V_{Action} \ \bullet a \rightharpoonup_{isAn} \leftharpoonup_{hasOperation} \leftharpoonup_{describesBehaviourOf} \rightharpoonup_{relates} a\ ;$

$(C2)$ $\quad \forall\, e : V_{Event} \mid \text{outdegree}_{causes}(e) > 0 \ \bullet$

$\qquad\quad e \rightharpoonup_{correspondsTo} \leftharpoonup_{hasOperation} \leftharpoonup_{describesBehaviourOf} \rightharpoonup_{relates} e\ ;$

C   KOGGE Model

The class of specification graphs for KOGGE tools is in figure 8.

The syntax of the supported language is modeled in the EER part of the graph (south west part of the diagram). Statecharts (north west) control the tool operations and trigger actions. These actions are described in a procedural way (east). The menus are specified also (south).

Figure 8: Metamodel of KOGGE

D MOTOS Model

Figure 9 shows the graph class of MOTOS. An instance of it models persons involved, the road map, the buses and the tour system.

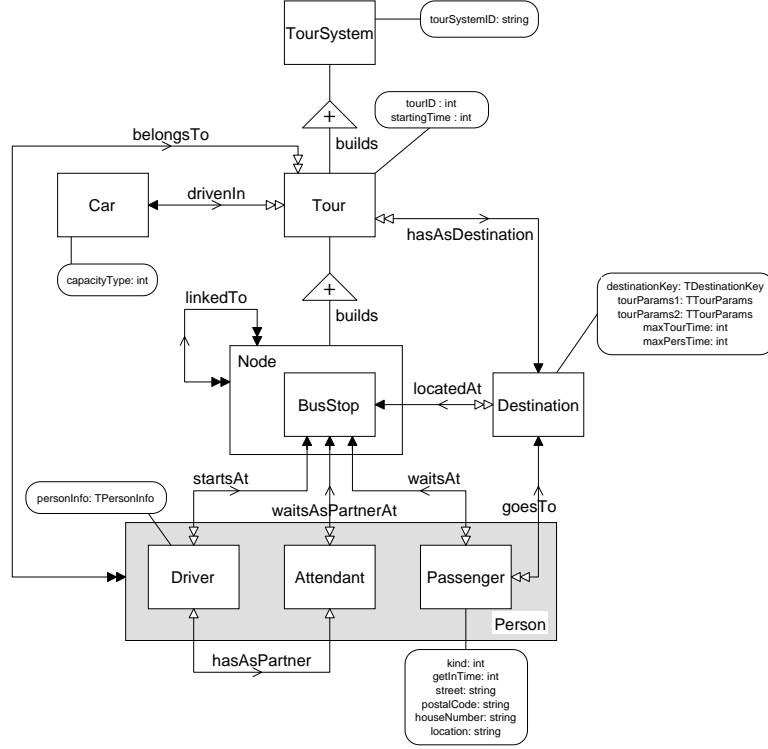Figure 9: Graph Class of MOTOS

Each MOTOS graph has to satisfy the following properties:

for G **in** $MOTOS$ **assert**

$\#V_{Busstop} \geq 1$;

$\#V_{Destination} \geq 1$;

$\#V_{Driver} \geq 1$;
$\#V_{Attendant} \geq 1$;

$\#V_{Car} \geq 1$;

$\forall\, v, w : V_{Node} \mid v \neq w \bullet v.coord \neq w.coord$;

$\forall\, v, w : V_{Destination} \mid v \neq w \bullet v.destinationKey \neq w.destinationKey$;

$\forall\, v, w : V_{Driver} \mid v \neq w \bullet v.personInfo \neq w.personInfo$;
$\forall\, v, w : V_{Attendant} \mid v \neq w \bullet v.personInfo \neq w.personInfo$;
$\forall\, v, w : V_{Passenger} \mid v \neq w \bullet v.personInfo \neq w.personInfo$;

$isConnected(vGraph(V_{Node}))$;
$\forall\, v : V_{Node} \bullet \neg\, (v \rightharpoonup_{linkedTo} v)$;

$\forall\, b : V_{BusStop} \bullet \exists\, p : V_{Person} \bullet p(\rightharpoonup_{startsAt} \mid \rightharpoonup_{waitsAsPartnerAt} \mid \rightharpoonup_{waitsAt})b$;

After completion of the planning algorithm the graph has to satisfy the following additional constraints:

> **for** G **in** $MOTOS_{planned}$ **assert**
> $\#V_{Toursystem} = 1$;
> $\#V_{Tour} \geq 1$;
>
> $\forall\, v, w : V_{Tour} \mid v \neq w \bullet v.tourID \neq w.tourID;$
>
> $\forall\, a : V_{Tour} \bullet$
> $\quad \exists_1\, b : V_{Driver} \bullet d \rightharpoonup_{belongsTo} b \,\wedge$
> $\quad \exists_1\, b : V_{Attendant} \bullet a \rightharpoonup_{belongsTo} b \,\wedge$
> $\quad \exists\, b : V_{Passenger} \bullet p \rightharpoonup_{belongsTo} b$;
> $\forall\, d : V_{Driver} \bullet \exists_1\, a : V_{Attendant} \bullet d \rightharpoonup_{hasAsPartner} b$;
> $\forall\, n_1, n_2 : V_{Node} \mid n_1\, isNextTo_{builds}\, n_2 \bullet n_1 \rightharpoonup_{linkedTo} n_2$;

E   GUPRO Model

The coarse-grained model of the software inventory at Volksfürsorge is given in the following. The edges of types represented by dotted lines cross the boundaries of different partial graph classes, which model the single programming languages. They must be created during integration.

Some objects can be identified by name attributes:

> **for** G **in** $MAKRO$ **assert**
> $\forall\, v, w : \mathrm{V}_{SourceFile} \mid v.name = w.name \bullet v = w$
> $\forall\, v, w : \mathrm{V}_{Program} \mid v.name = w.name \bullet v = w$
> $\forall\, v, w : \mathrm{V}_{Pl1Procedure} \mid v.name = w.name \bullet v = w$
> $\forall\, v, w : \mathrm{V}_{CspStmtGrp} \mid v.name = w.name \bullet v = w$
> $\forall\, v, w : \mathrm{V}_{CspRecord} \mid v.name = w.name \bullet v = w$
> $\forall\, v, w : \mathrm{V}_{CspTable} \mid v.name = w.name \bullet v = w$
> $\forall\, v, w : \mathrm{V}_{CspMap} \mid v.name = w.name \bullet v = w$
> $\forall\, v, w : \mathrm{V}_{File} \mid v.dsName = w.dsName \bullet v = w$
> $\forall\, v, w : \mathrm{V}_{ImsDbd} \mid v.name = w.name \bullet v = w$
> $\forall\, v, w : \mathrm{V}_{Db2Table} \mid v.name = w.name \bullet v = w$

Some objects can be identified by a name attribute and some context:

> **for** G **in** $MAKRO$ **assert**
> $\forall\, v, w : \mathrm{V}_{Segment} \mid v \rightleftharpoons^*_{hasParent} w \wedge v.name = w.name \bullet v = w$
> $\forall\, v, w : \mathrm{V}_{Db2Column} \mid v \rightharpoonup_{isColumnIn}\!\leftharpoonup_{isColumnIn} w \wedge v.name = w.name \bullet v = w$

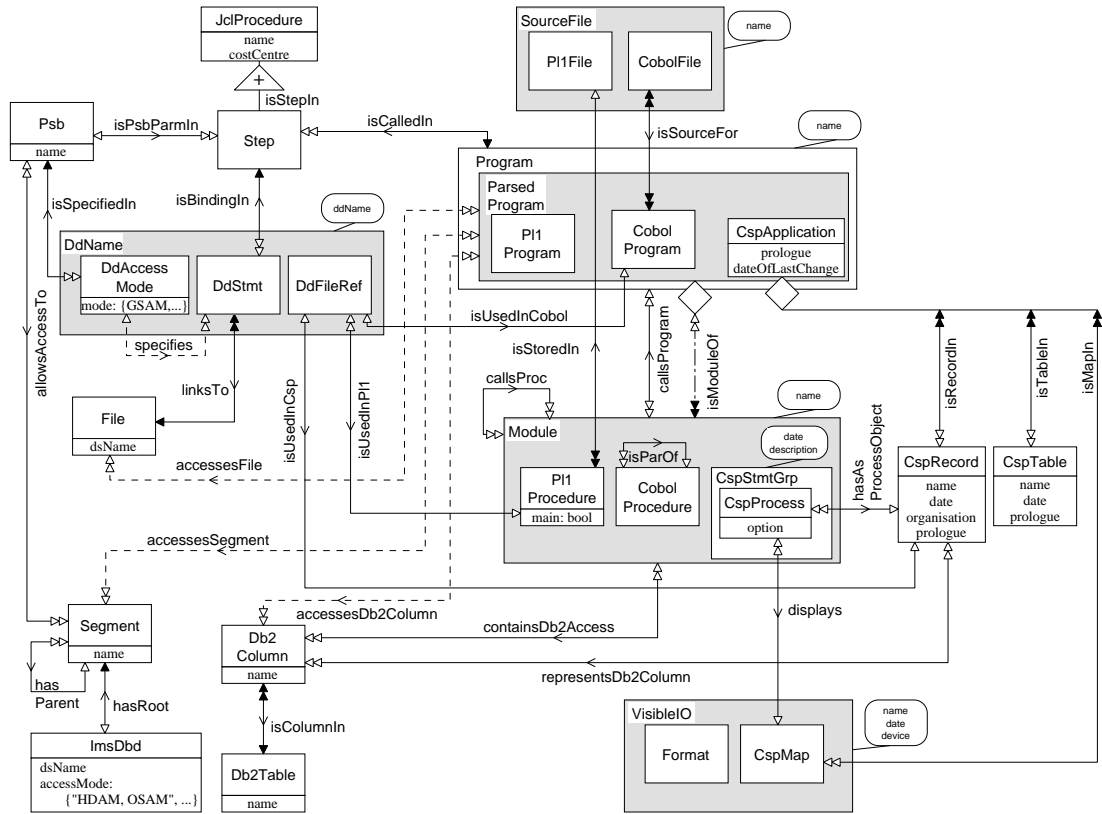The edges (marked by dotted lines in figure 10) connect vertices from different parts:

Figure 10: Model of a heterogeneous software environment

for G **in** $MAKRO_{integrated}$ **assert**

accessesFile:

$\forall\, v : V_{Pl1Program},\, w : V_{File}\, \bullet$

$v \rightharpoonup_{accessesFile} w \Leftrightarrow$

$(\exists\, u : V_{DdFileRef},\, u' : V_{DdStmt}\, \bullet$

$\quad u \rightharpoonup_{isUsedInPl1} \leftharpoonup_{callsProc} * \rightharpoonup_{isModuleOf} v \rightharpoonup_{isCalledIn} \leftharpoonup_{isBindingIn} u'$

$\qquad \rightharpoonup_{linksTo} w \wedge u.ddName = u'.ddName)\,;$

$\forall\, v : V_{CobolProgram},\, w : V_{File}\, \bullet$

$v \rightharpoonup_{accessesFile} w \Leftrightarrow$

$(\exists\, u : V_{DdFileRef},\, u' : V_{DdStmt}\, \bullet$

$\quad u \rightharpoonup_{isUsedInCobol} v \rightharpoonup_{isCalledIn} \leftharpoonup_{isBindingIn} u' \rightharpoonup_{linksTo} w \wedge$

$\quad u.ddName = u'.ddName)\,;$

$\forall\, v : V_{CspApplication},\, w : V_{File}\, \bullet$

$v \rightharpoonup_{accessesFile} w \Leftrightarrow$

$(\exists\, u : V_{DdFileRef},\, u' : V_{DdStmt}\, \bullet$

$\quad u \rightharpoonup_{isUsedInCsp} \rightharpoonup_{isRecordIn} v \rightharpoonup_{isCalledIn} \leftharpoonup_{isBindingIn} u' \rightharpoonup_{linksTo} w \wedge$

$\quad u.ddName = u'.ddName)\,;$

accessesSegment:

$\forall\, v : V_{ParsedProgram},\, w : V_{Segment}\, \bullet$

$v \rightharpoonup_{accessesSegment} w \Leftrightarrow$

$v \rightharpoonup_{isCalledIn} \leftharpoonup_{isPsbParmIn} \rightharpoonup_{allowsAccessTo} w\,;$

accessesDb2Column:

$\forall\, v : \mathrm{V}_{Pl1Program}, w : \mathrm{V}_{Db2Column} \bullet v \rightharpoonup_{accessesDb2Column} w \Leftrightarrow$

$v \leftharpoonup_{isModuleOf} \rightharpoonup_{callsProc}{}^{*} \rightharpoonup_{containsDb2Access} w$;

$\forall\, v : \mathrm{V}_{CobolProgram}, w : \mathrm{V}_{Db2Column} \bullet$

$v \rightharpoonup_{accessesDb2Column} w \Leftrightarrow$

$v \leftharpoonup_{isModuleOf} \leftharpoonup_{isParOf}{}^{*} \rightharpoonup_{containsDb2Access} w$;

$\forall\, v : \mathrm{V}_{CspApplication}, w : \mathrm{V}_{Db2Column} \bullet$

$v \rightharpoonup_{accessesDb2Column} w \Leftrightarrow$

$v((\leftharpoonup_{isModuleOf} \rightharpoonup_{containsDb2Access}) \mid (\leftharpoonup_{isRecordIn} \rightharpoonup_{representsDb2Column}))w$;

specifies:

$\forall\, v : \mathrm{V}_{DdAccessMode}, w : \mathrm{V}_{DdStmt} \bullet$

$v \rightharpoonup_{specifies} w \Leftrightarrow$

$v \rightharpoonup_{isSpecifiedIn} \rightharpoonup_{isPsbParmIn} \leftharpoonup_{isBindingIn} w \wedge v.ddName = w.ddName$

Available Research Reports (since 1994):

1996

11/96 *J. Ebert, A. Winter, P. Dahm, A. Franzke, R. Süttenbach.* Graph Based Modeling and Implementation with EER/GRAL.

10/96 *Angelika Franzke.* Querying Graph Structures with G^2QL.

9/96 *Chandrabose Aravindan.* An abductive framework for negation in disjunctive logic programming.

8/96 *Peter Baumgartner, Ulrich Furbach, Ilkka Niemelä .* Hyper Tableaux.

7/96 *Ilkka Niemelä, Patrik Simons.* Efficient Implementation of the Well-founded and Stable Model Semantics.

6/96 *Ilkka Niemelä .* Implementing Circumscription Using a Tableau Method.

5/96 *Ilkka Niemelä .* A Tableau Calculus for Minimal Model Reasoning.

4/96 *Stefan Brass, Jürgen Dix, Teodor. C. Przymusinski.* Characterizations and Implementation of Static Semantics of Disjunctive Programs.

3/96 *Jürgen Ebert, Manfred Kamp, Andreas Winter.* Generic Support for Understanding Heterogeneous Software.

2/96 *Stefan Brass, Jürgen Dix, Teodor. C. Przymusinski.* A Comparison of Static Semantics and D-WFS.

1/96 *J. Ebert (Hrsg.).* Alternative Konzepte für Sprachen und Rechner, Bad Honnef 1995.

1995

21/95 *J. Dix and U. Furbach.* Logisches Programmieren mit Negation und Disjunktion.

20/95 *L. Priese, H. Wimmel.* On Some Compositional Petri Net Semantics.

19/95 *J. Ebert, G. Engels.* Specification of Object Life Cycle Definitions.

18/95 *J. Dix, D. Gottlob, V. Marek.* Reducing Disjunctive to Non-Disjunctive Semantics by Shift-Operations.

17/95 *P. Baumgartner, J. Dix, U. Furbach, D. Schäfer, F. Stolzenburg.* Deduktion und Logisches Programmieren.

16/95 *Doris Nolte, Lutz Priese.* Abstract Fairness and Semantics.

15/95 *Volker Rehrmann (Hrsg.).* 1. Workshop Farbbildverarbeitung.

14/95 *Frieder Stolzenburg, Bernd Thomas.* Analysing Rule Sets for the Calculation of Banking Fees by a Theorem Prover with Constraints.

13/95 *Frieder Stolzenburg.* Membership-Constraints and Complexity in Logic Programming with Sets.

12/95 *Stefan Brass, Jürgen Dix.* D-WFS: A Confluent Calculus and an Equivalent Characterization..

11/95 *Thomas Marx.* NetCASE — A Petri Net based Method for Database Application Design and Generation.

10/95 *Kurt Lautenbach, Hanno Ridder.* A Completion of the S-invariance Technique by means of Fixed Point Algorithms.

9/95 *Christian Fahrner, Thomas Marx, Stephan Philippi.* Integration of Integrity Constraints into Object-Oriented Database Schema according to ODMG-93.

8/95 *Christoph Steigner, Andreas Weihrauch.* Modelling Timeouts in Protocol Design..

7/95 *Jürgen Ebert, Gottfried Vossen.* I-Serializability: Generalized Correctness for Transaction-Based Environments.

6/95 *P. Baumgartner, S. Brüning.* A Disjunctive Positive Refinement of Model Elimination and its Application to Subsumption Deletion.

5/95 *P. Baumgartner, J. Schumann.* Implementing Restart Model Elimination and Theory Model Elimination on top of SETHEO.

4/95 *Lutz Priese, Jens Klieber, Raimund Lakmann, Volker Rehrmann, Rainer Schian.* Echtzeit-Verkehrszeichenerkennung mit dem Color Structure Code — Ein Projektbericht.

3/95 *Lutz Priese.* A Class of Fully Abstract Semantics for Petri-Nets.

2/95 *P. Baumgartner, R. Hähnle, J. Posegga (Hrsg.).* 4th Workshop on Theorem Proving with Analytic Tableaux and Related Methods — Poster Session and Short Papers.

1/95 *P. Baumgartner, U. Furbach, F. Stolzenburg.* Model Elimination, Logic Programming and Computing Answers.

1994

18/94 *W. Hower, D. Haroud, Z. Ruttkay (Eds.).* Proceedings of the AID'94 workshop W9 on Constraint Processing in Computer-Aided Design.

17/94 *W. Hower.* Constraint satisfaction — algorithms and complexity analysis.

16/94 *S. Brass, J. Dix.* A Disjunctive Semantics Based on Unfolding and Bottom-Up Evaluation.

15/94 *S. Brass, J. Dix.* A Characterization of the Stable Semantics by Partial Evaluation.

14/94 *Michael Möhring.* Grundlagen der Prozeßmodellierung.

13/94 *D. Zöbel.* Program Transformations for Distributed Control Systems.

12/94 *Martin Volk, Michael Jung, Dirk Richarz, Arne Fitschen, Johannes Hubrich, Christian Lieske, Stefan Pieper, Hanno Ridder, Andreas Wagner.* GTU – A workbench for the development of natural language grammars.

11/94 *S. Brass, J. Dix.* A General Approach to Bottom-Up Computation of Disjunctive Semantics.

10/94 *P. Baumgartner, F. Stolzenburg.* Constraint Model Elimination and a PTTP Implementation.

9/94 *K.-E. Großpietsch, R. Hofestädt, C. Steigner (Hrsg.).* Workshop Parallele Datenverabeitung im Verbund von Hochleistungs-Workstations.

8/94 *Baumgartner, Bürckert, Comon, Frisch, Furbach, Murray, Petermann, Stickel (Hrsg.).* Theory Reasoning in Automated Deduction.

7/94 *E. Ntienjem.* A descriptive mode inference for normal logic programs.

6/94 *A. Winter, J. Ebert.* Ein Referenz-Schema zur Organisationsbeschreibung.

5/94 *F. Stolzenburg.* Membership-Constraints and Some Applications.

4/94 *J. Ebert (Hrsg.).* Alternative Konzepte für Sprachen und Rechner – Bad Honnef 1993.

3/94 *J. Ebert, A. Franzke.* A Declarative Approach to Graph Based Modeling.

2/94 *M. Dahr, K. Lautenbach, T. Marx, H. Ridder.* NET CASE: Towards a Petri Net Based Technique for the Development of Expert/Database Systems.

1/94 *U. Furbach.* Theory Reasoning in First Order Calculi.