Graph Rewrite Systems

and

Visual Database Languages

Marc Andries

Graph Rewrite Systems and Visual Database Languages

Proefschrift
ter verkrijging van de graad van Doctor
aan de Rijksuniversiteit te Leiden,
op gezag van de Rector Magnificus Dr. L. Leertouwer,
hoogleraar in de faculteit der Godgeleerdheid,
volgens besluit van het College van Dekanen
te verdedigen op woensdag 14 februari 1996
te klokke 15.15 uur

door

Marc Adolf Annie Andries

geboren te Wilrijk (België) in 1968

Promotiecommissie

Promotores:	Prof. dr. G. Engels
	Prof. dr. J. Paredaens (Universiteit Antwerpen (UIA), België)
Referenten:	Prof. dr. M. Gyssens (Limburgs Universitair Centrum, België)
	Prof. dr. H.-J. Kreowski (Universität Bremen, Duitsland)
Overige leden:	Prof. dr. H.A.G. Wijshoff
	Prof. dr. G. Rozenberg
	Prof. dr. J.N. Kok

Contents

Chapter 1

Introduction

1.1 A Need for Visual Languages

Wherever large amounts of structured data are to be stored for retrieval and manipulation, *database management systems*, also referred to as *information systems*, have found their way over the past few decades. As a consequence, as well as through recent evolutions in telecommunication technology, a growing number of unexperienced and untrained users are confronted with information systems. Unfortunately, the functionality and outlook of the *user interfaces* of these systems quite often seem more geared towards knowledgeable computer experts, rather than towards their intended users.

In response to this problem, much research effort has been put into user interface design in general, and the development of user-friendly information systems interfaces in particular. Within this area, a major line of research (subsuming the topic of this thesis) aims at fully exploiting the two-dimensional nature of the computer screen, by *visualizing* various aspects of the information systems user interface:

1. One aspect is the definition of the structure of the database, leading to *graphical scheme notations*, the most well-known of which is undoubtedly Chen's Entity-Relationship model [Che76].

2. Another important aspect of information systems suitable for visualization are languages for querying and manipulating the database, leading (among others) to database languages based on Shneiderman's *direct manipulation* paradigm [Shn83]. Systems built according to this paradigm allow the user to directly manipulate the objects of interest in the form of a visual representation, as opposed to systems offering access to these objects indirectly, e.g., by means of some textual language. In [Shn83], Shneiderman mentions the following advantages of direct manipulation interfaces:

   (a) Continuous representation of the object of interest,

   (b) Physical actions instead of complex syntax,

   (c) Rapid, incremental, reversible operations whose impact on the object of interest is immediately visible,

   (d) Layered approach to learning that permits usage with minimal knowledge.

3. A final aspect which lends itself most naturally to visualization concerns the results of manipulations of and queries to the database. Research on this topic is part of the broader research area of *information visualization* [LG94]. Research in this area aims at developing methods to present in a comprehensible manner huge and complex structured data sets to information systems users by means of techniques such as three-dimensional rendering.

The research reported on in this thesis concerns the second of the above aspects, namely *visual database query and manipulation languages*. The applicability of the notion of direct manipulation to database manipulation languages was already recognized by Shneiderman in the aforementioned article [Shn83]. Based on a study of various kinds of direct manipulation interfaces, he ascertains that

> *Graphic representations can be especially helpful when there are multiple relationships among objects and when the representation is more compact than the detailed object.*

This statement most clearly applies to database manipulation.

Also, in his introduction to the impressive collection [Gli90a, Gli90b] of influential and representative articles on visual programming environments, Glinert remarks that

> *The computer's ability to represent in a visible manner normally abstract and ephemeral aspects of the computing process such as recursion, concurrency, and the evolution of data structures has had a remarkable and positive impact on both the productivity of programmers and their degree of satisfaction with the working environment.*

One of the articles included in [Gli90b] supports this claim on the basis of real-life experiments. In [GR87], Gerstendörfer and Rohr ascertain that

> *Structural tasks are difficult to comprehend if not presented in pictures or more generally by means of visual aids. (…) Tasks with structural characteristics we find e.g., in all database applications.*

It is therefore not surprising that these insights in human-computer interaction, together with the ongoing evolution of hardware, has triggered the development of a range of formalisms and tools for visual interaction with information systems, including, among many others, [Mar89, ACS90, CM90, LP91, PPT91, Hou92, WMS+92].

1.2 A Need for Formally Defined Visual Languages

Remarkably, many proposals, though visually attractive, lack any kind of formal definition [Kan88, ADD+91, Sch91a, SBOO95]. In other cases, pseudo-formal constructions are presented, which barely deserve to be called formal definitions [GG87, CTYY89].

There is however, a clear and widely recognized need for clearly and unambiguously defining any computer language in general, and visual database languages in particular, since

- it forces the language designer to think clearly about both the major concepts as well as the minor details of the language,

- it enables truthful implementation,
- it guarantees to the user a unique and clearly determined semantics for any sentence in the language he cares to write down, and
- it offers the possibility to formally study properties of the language (without having to implement it). One important such property is a language's expressive power (cf. Section 5.3 of this thesis), a characterization of which in turn allows the comparison of different languages.

Fortunately, numerous proposals of visual database languages are indeed formally defined. Within this range of definitions, two main streams may be distinguished:

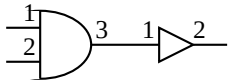
1. definitions using various logics, such as G-Log [PPT95] and GraphLog [CM90] based on first-order logic, the Categorial Graphs model [Haa95] based on modal logic, DOODLE [Cru92] based on F-logic [KL89], and VQL [VAO93] based on Datalog [CGT90].

2. definitions using various extensions of string grammars, such as context-free positional grammars [CTODL95], constraint multiset grammars [Mar94], and picture layout grammars [GR89].

   In such grammars, special (textual) symbols and operators as well as special-purpose attributes are used to indicate (spatial) interrelationships between the symbols used in the grammars productions. For instance, the following expression is a production from a positional grammar for logic circuits [OPT+92]:

   $$\text{Circuit} \rightarrow \text{AND } 3^0\_1 \text{ NOT}$$

   It captures the fact that the start-symbol Circuit of the grammar may be rewritten to an AND-gate, whose third "attaching point" (representing its output) is linked to a NOT-gates first attaching point (representing its input), i.e., the logic circuit

   The superscript "0" indicates that the link concerns the immediately preceding gate. A superscript $n \neq 0$ would refer to the n-th preceding gate.

Definitions from the first category deal almost exclusively with the logical aspects of the considered languages, i.e., their semantics, and include no formalization of syntax, i.e., the correspondence between rules and their visual representation.

The strange thing about the latter category of definitional mechanisms is that string grammars were intended for (and have been successfully applied to) the definition of *textual* languages. In contrast, over 25 years ago, a counterpart for formally defining *graph* languages was introduced in the form of *graph grammars* and *graph rewrite systems* [PR69, Sch70].

A graph grammar basically consists of an "initial" graph and a set of graph rewrite rules. Such a rewrite rule in turn basically consists of two graphs, called its *left-* and *right-hand side*. A rule may be applied to a graph by looking for an isomorphic occurrence of the left-hand side in the graph, and rewriting it into an isomorphic occurrence of the right-hand side. The grammar defines the language

of all graphs that may be obtained by applying an arbitrary sequence of graph rewrite rules to the initial graph. In graph grammars (as well as in textual grammars) a distinction is often made between *terminal* and *non-terminal* symbols. Terminal symbols are the elements of the graphs that are actually part of the defined language, while non-terminal symbols are not allowed to occur in these graphs, but are merely used in intermediate stages of their construction.

A graph rewrite system consists of a structured collection of graph rewrite rules, and can be used to transform one instance of a given class of graphs into another instance of the same class. [1]

Since 1969, the theory of graph grammars has become a well-researched area [GG90]. Mainly in the seventies, some work was already done on applying results obtained in this area to various aspects of database management, such as conceptual modeling and the description of concurrency [EK76, GF79, ADS80, EK80], all in the context of relational databases. Only in the early nineties, this line of research was picked up again [GPVG90a, GPVG90b, Eng90, BM90, Cou91, EF94], this time mainly in the context of object-based models.

1.3 Topic of this Thesis: Defining Visual Database Languages with Graph Rewriting

In this thesis, we investigate how graph rewrite systems may be used to formally define both syntax and semantics of visual database languages. We present two possible ways to do so.

1.3.1 A Query Language Defined Using Graph Rewriting

In a first approach, we start from the following well-known technique [EL85, Kan88, Mar89] to "derive" a *fully graphical query language* from a given database model which includes a graphical representation for database schemes (such as the aforementioned Entity-Relationship model). Intuitively, formulation of a query in such a language primarily consists of composing graphical components from a concrete database scheme into a *pattern*, describing the desired information. Again intuitively, this pattern then has to be *matched* against the database instance to retrieve this information.

As an example, consider Figure 1.1. On one hand, this diagram may be seen as the graphical representation of an (extremely simple) ER scheme, also called an *ER diagram*. In this case, the diagram models a world of customers and cash-cards, as indicated by the rectangular nodes, representing so-called *entity types*. Customers can have cash-cards, as indicated by the diamond-shaped node linked to both rectangles, representing a *relationship type*. Customers are characterized by a name, while cash-cards are characterized by a password (both of which are strings). This is indicated with oval-shaped nodes, representing so-called *attributes* of the entity types.

On the other hand, the diagram may also be read as a query, expressing an interest in the names of customers and the passwords of cash-cards these customers have. Czejdo et al. [CERE87] list (among others) the following advantages of query languages in which queries are composed of graphical components from a database scheme:

[1] Despite the differences outlined above, the terms "graph grammar" and "graph rewrite system" are often used as synonyms. Since in this thesis, both formalisms are used, we make a conscious effort to use the appropriate term in each context.

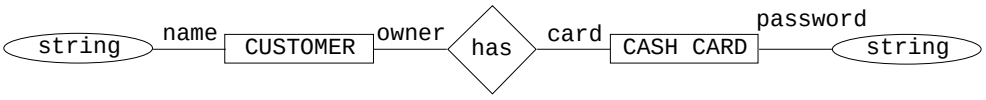


Figure 1.1: A diagram with a double purpose

1. *The query language is two dimensional. Diagrams that depict a view of the database schema are displayed and can be manipulated interactively.*

2. *Query formulation is flexible. A query can be formulated in many different ways since the order in which the diagram manipulating operators are invoked is often immaterial.*

3. *The user always has a convenient frame of reference. The current diagram reflects the current status of query formulation and is always a valid query.*

4. *The approach is applicable to a wide range of semantic data models.*

5. *The intended query can be specified in several different ways (…) The strategy to be used can be selected by the user.*

Based on the *Extended Entity-Relationship* model [EGH+92] (which is, as the name suggests, an extended version of Chen's Entity-Relationship model [Che76]) we present in this thesis the Graph-Oriented Query Language GOQL/EER, in which queries may be expressed graphically using elements from a given EER diagram. The word "diagram" rightfully suggests that GOQL/EER falls within the category of visual query languages using the *diagrammatic representation paradigm*, according to the taxonomy introduced by Batini et al. in their comprehensive survey of visual query systems [BCCL91]. According to this taxonomy, the representation paradigm of a visual query language may be either

tabular : using a visualization of prototypical tables (typically used in the context of the relational database model);

diagrammatic : using a fixed set of symbols, each representing a specific type of concepts, as well as a fixed set of allowed connections, each representing a specific logical relationship type between concepts;

iconic : using icons denoting entities of the real world as well as functions offered by the system, which are combined into a query; or

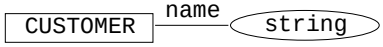
hybrid : using any combination of the other three approaches.

The main topic of Chapter 4 of this thesis is an investigation of how both syntax and semantics of GOQL/EER may be formally defined by means of a graph grammar. As for the syntax of the language, we concentrate on the *abstract* syntax. Defining the abstract syntax of a graph-oriented language boils down to specifying the structural characteristics a graph must satisfy in order to represent a syntactically correct GOQL/EER-query.

The exact correspondence between a graph and the query it represents would be the topic of a study of the *concrete* syntax. The latter basically establishes the concrete visual representation of language elements, stating for instance that an entity should be represented by a rectangle, as well as their layout and spatial relationships, stating for instance that one endpoint of an edge representing a role should lie on the boundary of a rectangle, while the other should coincide with the corner of a diamond (representing the relationship). The topic of concrete syntax of visual languages is in turn closely related to the research area of *visual parsing* [Wit92], which in a sense studies the recognition of concrete syntax elements, and the transition from concrete to abstract syntax. It is currently being investigated to what extent graph grammars may also be used for this aspect of visual language definition [RS95]. Conversely, the research area of *visualization* [LG94] in a sense studies the transition from abstract to concrete syntax.

For defining the languages semantics, we take the following approach. The EER model includes a fully textual query language SQL/EER [HE92]. We exploit the availability of this formally defined language by defining the semantics of GOQL/EER queries by translating them to SQL/EER queries. Intuitively, this means that graph increments are associated to elements of the textual language. For instance, the `CUSTOMER`- and `CASH CARD`-entities in Figure 1.1 can be seen as the "graphical equivalent" of the declaration of variables (say, `c` of type `CUSTOMER` and `cc` of type `CASH_CARD`) in the textual language. Likewise, the `string`-labeled nodes in the figure can be seen as the graphical equivalents of the terms `c.name` and `cc.password`. Finally, the `has`-relationship expresses the formula `c has cc`.

One of the main contributions of this thesis is the observation that the definition of both syntax and semantics (in the way described above) may be integrated seemlessly into one and the same graph grammar specification. In a sense, this idea may be seen to follow naturally from the above mentioned association between components of respectively graphical and textual queries. Indeed, consider once more the graphical increment

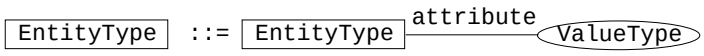


in which (as mentioned above) the `string`-labeled node should be seen as the graphical equivalent of the term `c.name`. In the EBNF grammar defining the syntax of the textual query language SQL/EER, the fact that `c.name` is indeed a correct part of a textual query is captured in the rule

TERM ::= VARIABLE "." ATTRIBUTE

expressing the fact that a variable, followed by a dot, followed by an attribute name, is a syntactically correct term.

Likewise, the fact that the above depicted graphical increment is indeed a correct part of a graphical query may be captured in the *graph rewrite rule*



capturing the fact that given a node labeled with an `EntityType`, a node labeled with a `ValueType` may be linked to it by means of an `attribute`-edge.

Rather than to define the semantics of the graph-oriented query language by formalizing this association between EBNF-rules and graph rewrite rules (a language definition technique known as Pratt's

pair-grammars [Pra71]) we chose to *integrate* this association into the graph grammar specification by extending the latter with *attributes*,[2] thus obtaining an *attributed graph grammar*. This technique of attributing a grammar is also used quite frequently for defining the semantics of textual languages. When the syntax of a textual language is defined using an EBNF grammar, then this grammar may be extended into an attributed one, defining the semantics of the language as follows:

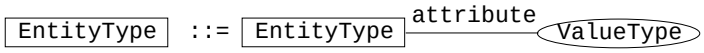
1. attributes (in the formal language sense of the word!) are associated to the non-terminal symbols of the EBNF grammar and

2. attribute derivation rules are associated to the EBNF rules, which compute (in the attributes) an expression in some other formally defined language, which is then considered to define the semantics of the expression in the original textual language.

This technique has been known since long in the context of defining the semantics of textual languages. It was introduced in [Knu68, Knu71] by Knuth under the name of *attribute translation grammars*. In these two papers, Knuth applies the technique to the translation of a toy-language "turingol" to Turing Machines, and to the reduction of λ-calculus expressions to some canonical form. More examples on applications of attribute translation grammars may be found in [Pag81].

The same technique is used in [HE92], in which SQL/EER is formally defined by mapping its statements to corresponding formulas in a previously defined calculus for the EER model [HG88]. For instance, an attribute derivation rule associated to the rule "TERM ::= VARIABLE "." ATTRIBUTE" translates such a term into an expression in the EER calculus.

Likewise, in a graph grammar specification, attributes (once more in the formal language sense) may be associated to rewrite rules. Informally, the graph rewrite rule depicted above then becomes



transfer ValueType.Term := EntityType.Term ''.'' attribute

capturing the fact that, given the Term corresponding to the node labeled with an `Entity-Type`, the Term corresponding to the newly added node labeled with a `ValueType` is obtained by appending (a string-representation of) the `attribute` to the former Term with a dot.

1.3.2 Choosing a Graph Grammar Formalism: PROGRES

From the above ideas clearly follows our need for an expressive graph grammar formalism. At this moment, the most expressive specification formalism based on graph rewrite rules is Schürr's **PRO**grammed **G**raph **RE**writing **S**ystem language [Sch90a, Sch91b] (in brief, PROGRES).

A graph language is specified in PROGRES by means of a two step process:

1. In a *graph scheme*, the types of nodes and edges that may occur in a graph of the considered language are declared. Additionally, node attributes (i.e., values representing non-structural information concerning nodes) are declared and initialized.

[2] Note that the word "attribute" is now used in two different meanings: one in the context of graphs, and the other in the context of the EER-model!

2. A set of graph rewrite rules or *productions*, obeying the type restrictions imposed by the graph scheme, captures those structural integrity constraints not expressible in a graph scheme. Additionally, productions also incorporate attribute computations.

Even though, besides the features mentioned above, the PROGRES language also includes a wide variety of nondeterministic control structures [ZS92], we use mainly the powerful pattern matching and replacement facilities, as well as attribute derivation mechanism offered by its graph rewrite rules, to specify both syntax and semantics of GOQL/EER as described previously.

An additional motivation for choosing the PROGRES formalism was the availability of an integrated set of language-specific tools supporting editing, analyzing, and debugging of specifications [NS90]. The specification of GOQL/EER has been entered using this toolsets syntax-directed editor [Sch90b], which allowed them to be analyzed by the incrementally working type-checker and tested using the integrated interpreter.

1.3.3 From Graphical to Hybrid Languages

However successful visual query languages may appear to be, visual representations also have their limitations. Especially when looking at the ever increasing collection of fully graphical query languages, one gets the impression that some of this research overshoots its mark in the sense that a purely graphical formulation of a query quite often becomes even more complex than its textual equivalent.

As an example, consider the query that retrieves (from some financial information system) the names of those banks that manage only and exactly all accounts with a balance over 1.000.000. Figure 1.2 shows an expression of this query in SQL/EER, while Figure 1.3 shows the same query in the fully graphical query language GRAQULA [SBMW93]. The `(implies)`-box on the left corresponds to the condition that if an account has a balance over a million, the considered bank should manage the account. An `(and)`-box contains the consequent of an implication (in this case, the `manages`-relationship). Likewise, the `(implies)`-box on the right corresponds to the condition that if the bank manages some account, the account's balance should be over a million.

select name
from b **in** BANK
where for all a **in** ACCOUNT
with balance $\geq$ 1.000.000 : (b manages a)
and for all a **in** ACCOUNT **with** b manages a :
(balance $\geq$ 1.000.000)

Figure 1.2: Names of banks that manage exactly all accounts with a balance over 1.000.000 (in SQL/EER)

Similar highly expressive yet visually unattractive fully graphical query languages are introduced in [Miu94, Hou92]. To say the least, one could doubt that both formulating and deciphering such a nested structure of boxes corresponding to logical primitives, is easier than doing the same with its textual equivalent.

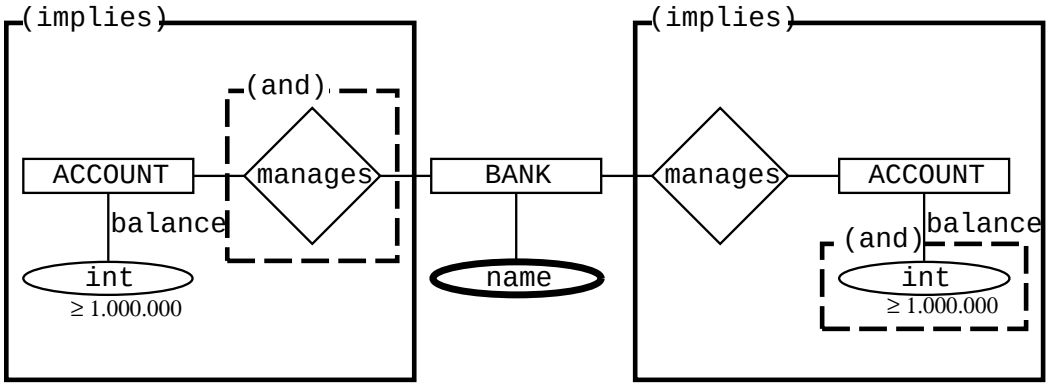


Figure 1.3: The query of Figure 1.2 in GRAQULA

In support of this observation, the aforementioned collection of articles on visual programming [Gli90b] includes an experience report [Gra90] intriguingly entitled "Visual Programming and Visual Languages: Lessons Learned in the Trenches". The fifth lesson in this report is briefly summarized as "Quit While Winning", a statement which is clarified as follows:

> *(…) a system (…) ends up cumbersome and difficult to use (…) when the extension of the visual language begins creating more problems than it solves. (…) Shift to another presentation mode (usually text) as soon as appropriate.*

The second lesson of the report supports this claim, by stating that a

> *(…) visual language should be based on a minimum of icons and constructs.*

Likewise, in his introduction to [Gli90a, Gli90b], Glinert prophesies

> *(…) would it not be more pleasant and productive to work in multiparadigm environments which could support, within a single program, both textual and graphical representations for all sorts of computing objects (…) ?*

In the context of databases, the question then arises whether it is possible to combine the "best of both worlds", that is, to develop a *hybrid* query language that allows those parts of a query that are most clearly specified graphically respectively textually, to be indeed specified graphically respectively textually.[3] In a sense, Zloof's Query-By-Example [Zlo77], which is commonly considered to be one of the first attempts at "two-dimensional" query languages, already offers facilities along this line. Indeed, while join conditions and selections are entered "graphically" (that is, in table skeletons), complex conditions involving e.g., aggregate functions, have to be entered in plain text in a so-called "condition box". Similar facilities are offered in prototype interfaces for semantic database models, like SNAP's "node restriction" [BH86]. In other proposals, like [Kun92, KM89], tools are presented

[3] The term "hybrid" was inspired by hybrid editors, where the user can freely choose between a syntax-directed and a free style of editing [ELN+92].

which give the user a (limited) choice between graphical and textual specification of operations, limited in the sense that graphics and text may not be mixed within the same operation.

We answer the above question positively by merging GOQL/EER and SQL/EER into a Hybrid Query Language for the EER-model, in brief HQL/EER. This language has both GOQL/EER and SQL/EER as sublanguages, but it also allows the expression of queries by means of a mixture of graphical elements from GOQL/EER and textual elements from SQL/EER.

1.3.4 Database Manipulation Defined as Graph-Rewriting

In a second approach towards the definition of visual database languages by means of graph rewriting, we shift our attention towards database manipulation languages. On one hand, when we introduced GOQL/EER, we already mentioned an intuitive way to interpret GOQL/EER queries, namely by looking upon them as *patterns* that are to be *matched* against (a graph-representation of) the database instance. On the other hand, when recalling the notion of a graph grammar, we mentioned that pattern-matching is precisely the mechanism underlying the application of graph rewrite rules. An intuitively natural idea would therefore be to use graph rewriting as database manipulation paradigm.

However, SQL's **update**-command illustrates the fact that a database manipulation generically consists of a query together with the specification of some modification to be performed on the outcome of the query [ACPB95]. Consequently, if we wish to use graph rewrite rules for the specification of database manipulations, we have to attribute a particular semantics to graph rewrite rule application, allowing rules to be applied to various parts of a graph at the same time, rather than to a single subgraph. More precisely, we need the possibility to apply a rule *exhaustively*, which means that it is to be applied to every possible matching of its left-hand side in the graph representation of the database instance [And94].

As an example, part of the pattern of Figure 1.1 can be used to graphically express the database update that the password of all cash-cards should be set to the name of their owner (assuming for the sake of simplicity that a card has at most one owner).

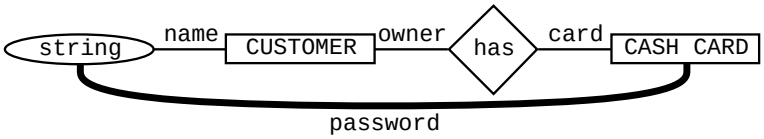


Figure 1.4: A graph rewrite rule expressing a database update

Using the terminology introduced formerly, the part of this diagram drawn in thin lines corresponds to the left-hand side of the rewrite rule. On applying this rule to some graph representing a database instance, this left-hand side matches the names of all customers having a cash-card. The diagram as a whole (i.e., including the thick line) corresponds to the right-hand side of the rewrite rule. On applying the rule to a database instance, the password of each cash-card matching the left-hand side is set to the name of the customer occurring in that same matching (i.e., the customer having that cash-card).

Conversely, suppose we wish to revoke ownership for those cash-cards whose password equals precisely the name of their owner. This manipulation may be performed using the graph rewrite rule

depicted in Figure 1.5.

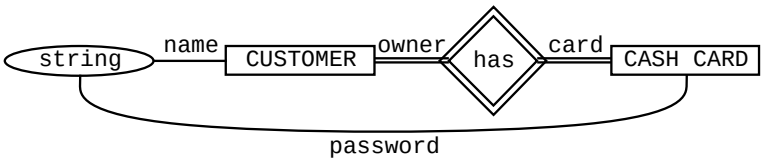


Figure 1.5: Another graph rewrite rule expressing a database update

This time, the diagram as a whole corresponds to the left-hand side of the rewrite rule. On applying this rule to some graph representing a database instance, this left-hand side matches the names of all customers having a cash-card the password of which equals their name. The diagram excluding the parts indicated with double lines (i.e., the `has`-relationship and its two incident edges) corresponds to the right-hand side of the rewrite rule. On applying the rule to a database instance, these are the elements to be removed from the database.

Since in general, it is impossible to express any conceivable database manipulation by means of a single rewrite rule, we need the ability to structure a number of rules into a program. Such a program is commonly called a *graph rewrite system*. For instance, applying the rules depicted in the Figures 1.4 and 1.5 in that order would remove (among others) *all* `has`-relationships from the database.

We incorporate the above ideas in a third language introduced in this thesis, called the Graph-Oriented Object Database language GOOD/ER. For reasons of succinctness, we do not present this language in terms of the EER model, but rather use (a slightly modified version of) the original ER model as a framework. In summary, the GOOD/ER language allows the expression of database manipulations as graph rewrite systems. Note the contrast with the use of graph rewrite rules in the definition of GOQL/EER: whereas the semantics of GOOD/ER programs is defined *as* graph rewriting, both syntax and semantics of GOQL/EER queries are defined *using* graph grammar productions. Besides, whereas the graph rewriting mechanism used for defining GOQL/EER is incorporated within the definition of PROGRES, for the definition of GOOD/ER we define our own graph rewriting mechanism.

Characterizing GOOD/ER's Expressive Power

Besides formally defining GOOD/ER, we also formally study the expressive power of the language, in support of our claim made earlier on that the possibility of such a formal study is precisely one of the main reasons why one should formally define computer languages. This study is furthermore facilitated by the fact that the semantics definition of GOOD/ER is "self-contained", in the sense that the definitions incorporate a description of the way in which GOOD/ER programs are to be applied.[4]

Studying the expressive power of GOOD/ER boils down to answering the question: which category of manipulations can be performed using the GOOD/ER language? To answer this question, we use a so-called *completeness*-criterion, a well known technique in the area of database languages. A completeness-criterion provides a necessary and sufficient condition which a pair of database instances

[4] As opposed to the semantics of GOQL/EER, defined by means of a translation to some other language.

must satisfy in order for one instance to be transformable by means of the language into the other instance. This condition is necessarily independent of the considered language, and may therefore be used to compare the *relative* expressive power of different languages.

The criterion used to characterize GOOD/ER's expressive power is an adaptation of the criterion called *BP-completeness* [CH80], originally introduced in the context of the relational database model, to object-based database models.

1.4 Organization of this Thesis

In Chapter 2, we recall those aspects of the Extended Entity-Relationship model to be used in the definition of graph-oriented languages further on in the thesis. After an informal presentation (Section 2.1) we recall the formal definitions of the EER data model (Section 2.2), including the definitions of EER schemes and instances. In Section 2.3, we (informally) recall the textual query language SQL/EER.

In Chapter 3, we recall the aspects of the PROGRES graph rewriting formalism needed further on in this thesis. After some considerations on formal graph representation, we present the various aspects which constitute the PROGRES specification of a graph language. As a running example, we present a PROGRES specification for EER diagrams, i.e., graphical representations of EER schemes.

In Chapter 4, we introduce and define both the Graph-Oriented and Hybrid Query Languages for the EER model (abbreviated respectively GOQL/EER and HQL/EER). Following an example-based presentation of GOQL/EER (Section 4.1), we formally define this language by means of a PROGRES specification (Section 4.2). In Section 4.3, we then introduce HQL/EER.

In Chapter 5, we introduce the Graph-Oriented Object Database language GOOD/ER based on a slightly modified version of the original ER model, presented in Section 5.1. A formal definition of GOOD/ER is presented in Section 5.2. In Section 5.3, we then characterize GOOD/ER's expressive power (i.e., the set of transformations expressible by the language) in terms of the BP-completeness criterion.

In Chapter 6, we contrast the languages studied in Chapters 4 and 5, and discuss the outcome of our study on the applicability of graph rewrite systems to the definition of visual database languages. We also present some open issues for future research.

In Appendix A, we summarize some notational conventions. Appendix B presents the full syntax of SQL/EER by means of an EBNF-grammar, while Appendix C includes the full PROGRES specification of GOQL/EER.

Chapter 2

The Extended Entity-Relationship Model

In this chapter we recall some aspects of the database formalism we will use as a vehicle for the definition of visual languages in the sequel of this thesis. For this purpose, we chose one of the numerous extensions made over the past two decades to Chen's original Entity Relationship model [Che76], namely the *Extended Entity-Relationship* model [EGH+92], hereafter referred to as the *EER model*.

The chapter is organized as follows. In Section 2.1, we informally present the EER data model, including data type signatures, schemes, and instances. All concepts are introduced by means of an elaborate example. In Section 2.2, we recall the formal definitions of the EER data model [Hoh93, Gog94]. Finally, in Section 2.3, we (informally) recall a query language for the EER model, called SQL/EER [HE90, HE92].

2.1 Informal Introduction to the EER Model

We start with an informal sketch of the main concepts of the EER data model. As universe of discourse (to be used as a running example throughout the remainder of this thesis), we use a network of automatic teller machines. The following description was adopted and adapted from [RBP+91].

> *A computerized banking network includes both human cashiers and automatic teller machines (ATMs) to be shared by a consortium of banks. Each bank maintains its own accounts. Cashier stations are owned by individual banks. Human cashiers, employed by banks, enter account and transaction data. An automatic teller machine accepts a cash card from a customer, and carries out the required (remote) transaction on the account associated to the cash card.*

The structural characteristics of data relevant to an application area such as the one described above, may be expressed by means of an EER *scheme*. The formal (i.e., mathematical) definition of EER schemes is presented in Section 2.2. Although such a formal definition is a prerequisite for unambiguously capturing the precise syntax and semantics of schemes, a mathematical structure is surely not a very suitable tool to work with in, e.g., the process of deriving a scheme from a given informal requirements specification, like the one presented above for ATMs. Therefore, the EER model (like many other object models) offers the possibility to *visually* represent a scheme, by means of an *EER*

diagram. Figure 2.1 shows a (partial) EER diagram to model our running example. We now briefly discuss the various components of an EER scheme that may be distinguished in this diagram.

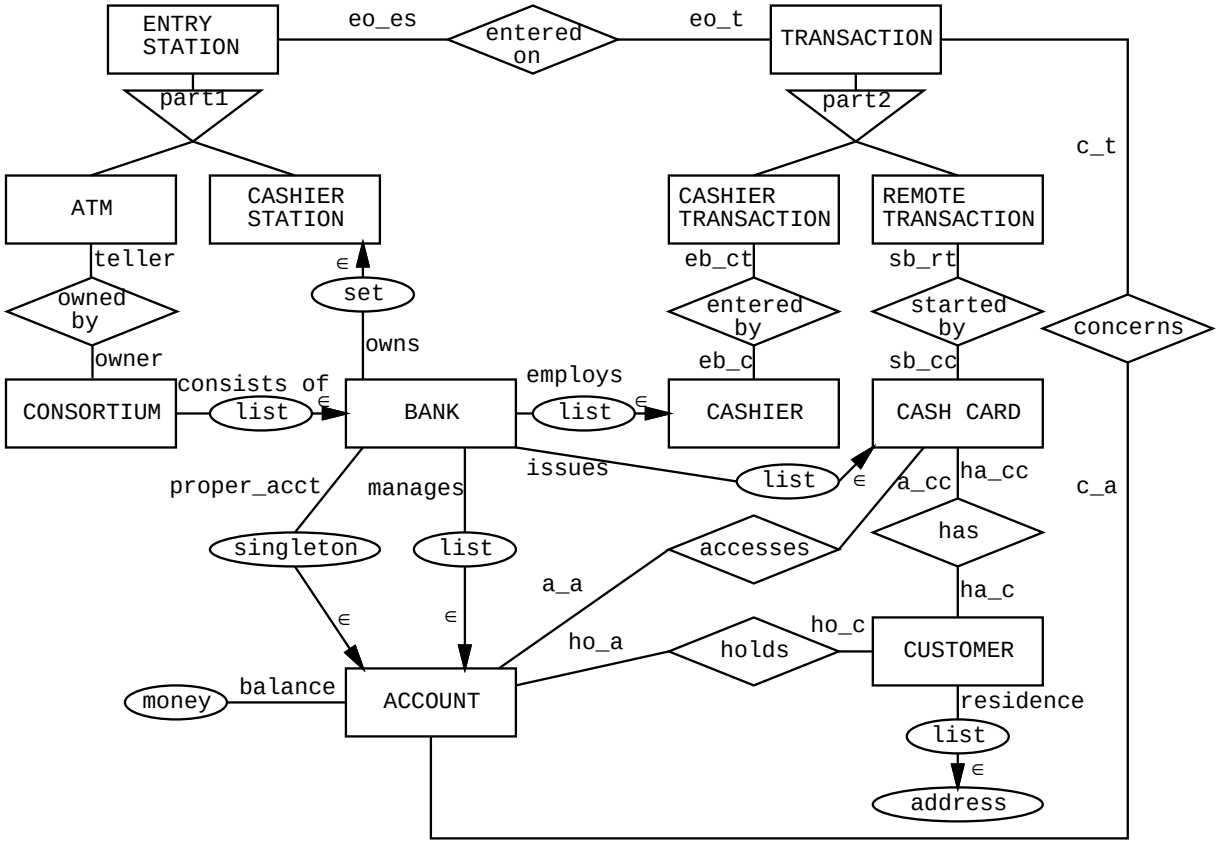


Figure 2.1: An EER diagram, modeling a network of automatic teller machines

The basic building block of an EER scheme is the *entity type*. An entity type provides the name for a collection of real world things (called *entities*) that share certain characteristics. E.g., in the world of ATMs, banks play an important part, hence the considered EER scheme includes the entity type `BANK`. In an EER diagram, an entity type is drawn using a rectangular box.

Possible real world associations between entities of given entity types are modeled using *relationship types*. E.g., an `ATM` is `owned by` a `CONSORTIUM`. The entity types `ATM` respectively `CONSORTIUM` are said to play the *role* of respectively `teller` and `owner` in the relationship type `owned by`. Visually, a diamond-shaped node is used to denote a relationship type. An undirected edge linking a diamond representing a relationship type with a rectangle representing an entity type (like the edge labeled `owner`) denotes a role. We wish to stress the fact that, although the names of relationship types often contain verbs, a relationship denotes a *static* fact recorded in the database. Although our example only contains binary relationship types, any non-zero number of entity types may participate in a relationship type.

Besides relationships, one can also establish *component*-links between entities. Relationships and components are the EER equivalents of modeling mechanisms which in many object-oriented model-

ing techniques are referred to as *associations* respectively *aggregations*. While aggregation is used to express the fact that entities of one type are a property of entities of some other type, an association is used to model a more general correspondence between entities of two or more types.

In our example EER scheme, a `CONSORTIUM` `consists of` a number of `BANK`s, ordered in a list. In the diagram, this is indicated using an extra node labeled `list`, linked to the node labeled `CONSORTIUM`, respectively `BANK`, by means of an edge labeled `consists of`, respectively $\in$. Components are either singletons, lists, sets or bags (i.e., multisets).

Both entities and relationships are characterized by means of *attributes*. Attributes are either single values or collections of values (i.e., sets, lists or bags) describing properties of the pertaining entities or relationships. In Figure 2.1, we can see that `ACCOUNT`s have a `balance`, which is of type `money`, while `CUSTOMER`s have a list of `address`es as their `residence`.

Attributes are indicated in an EER diagram by an undirected edge, labeled with the attribute name, linking the rectangle corresponding to the entity (or relationship) type to an oval labeled with the domain of the attribute (in case of a single-valued attribute) or the keyword `list`, `set` or `bag` (in case of a complex attribute).

Entity type	Attribute	Data type
ENTRY STATION	location	address
TRANSACTION	entry_time amount	time money
ATM	cash on hand dispensed	money money
CASHIER	name	string
BANK	name location	string address
CONSORTIUM	name	string
CUSTOMER	name residence	string list(address)
CASH CARD	password serial number limit	string int money
ACCOUNT	blocked balance	bool money

Table 2.1: Attributes for the EER diagram for the automatic teller machine example

The complete collection of attributes of our running example EER scheme is shown in Table 2.1 in a tabular format. Imagine that all of the attributes in this table would be represented graphically in the diagram of Figure 2.1. The resulting diagram would obviously become quite incomprehensible. Therefore we opted for a “hybrid” representation of the EER scheme, by presenting the attributes in a tabular format, and all other parts of the scheme in the diagram (cf. our motivation of the use of hybrid representations in Chapter 1).

Data types, including operations and predicates applicable to their data values, are defined in a *data type signature*. E.g., the data type signature underlying our example EER scheme includes a definition for the data type `money` (used e.g., as the domain of the attribute `dispensed` of the entity type `ATM`). This definition states that all rational numbers with at most two decimals are allowed as values. To money values, the function `compute interest` is applicable. This function takes an integer value as input as interest rate, and returns a new money value, including the interest. Predicates applicable to money values include typical mathematical comparison such as $<_{\texttt{money}}$, $\leq_{\texttt{money}}$,.... Note that these predicates are subscripted with the name of the type in order to distinguish them from similar predicates applicable to e.g., integer values.

Finally, by means of the notion of *type constructions*, the EER model also incorporates *inheritance*, another well-known from many object-oriented modeling technique. It allows the specification of new entity types as special cases of existing ones. The EER model uses a particular terminology in this context. A type construction takes a number of *input* entity types, and redistributes some of the entities of these types over a number of *output* types. In common object-oriented terminology, input types are called superclasses, while output types are called subclasses. In our example, the types of `ATM`s and `CASHIER STATION`s are "constructed" from the type of `ENTRY STATION`s by means of the type construction `part1`.

Output types of a type construction inherit all properties (i.e., attributes, components and roles in relationships) from the input types of that construction. In our example, this means that both `CASHIER TRANSACTION`s and `REMOTE TRANSACTION`s (which are specializations of `TRANSACTION`) concern `ACCOUNT`s.

2.2 Formal Definition of the EER Model

In this section, we formally define the EER model, following the definitions of [Hoh93].

The EER model consists essentially of two levels. The top level consists of a formalism for modeling database schemes and instances. Below is a level that allows the specification of arbitrary data types to be used as attribute domains in the specification of EER schemes. The presence of this lower level solves the problem of e.g., traditional relational database management systems which offer only a fixed set of predefined data types (such as string, money, date,...) over which relations can be defined.

A collection of data types is declared in a *data type signature*, which, besides the names of a number of *sorts*, also provides the signatures of *operations* and *predicates* defined on these sorts. Formally:

Definition 2.1 (Data type Signature) *A* data type signature DT *is a sixtuple* $(SORT_{DT}, OPER_{DT}, PRED_{DT}, \mathrm{source}, \mathrm{dest}, \mathrm{arg})$ *where*

- $SORT_{DT}, OPER_{DT}$ *and* $PRED_{DT}$ *are finite sets of names of respectively* sorts, operations, *and* predicates.
- $\mathrm{source}, \mathrm{dest}$ *and* arg *are functions with respective signatures* $\mathrm{source} : OPER_{DT} \rightarrow SORT^*_{DT}$, $\mathrm{dest} : OPER_{DT} \rightarrow SORT_{DT}$, *and* $\mathrm{arg} : PRED_{DT} \rightarrow SORT^+_{DT}$.

□

Notation 2.2 *If ω is an operation name with $\mathrm{source}(\omega) =< D_1, \ldots, D_n >$ and $\mathrm{dest}(\omega) = D$, then we write $\omega : D_1 \times \ldots \times D_n \rightarrow D$ and call it the* operation signature *of ω.*

If π is a predicate name with $\mathrm{arg}(\pi) =< D_1, \ldots, D_n >$, then we write $\pi : D_1 \times \ldots \times D_n$ and call it the predicate signature *of π.*

As an illustration, we present part of the data type signature underlying our example EER scheme. The set $SORT_{DT}$ contains the sort names `address`, `bool`, `int`, `money`, `string` and `time`. As mentioned in Section 2.1, the set $OPER_{DT}$ contains the operation name `compute interest`, with the functions source and dest defined on it as follows:

$$\mathrm{source}(\texttt{compute interest}) = (\texttt{money},\texttt{int})$$

$$\mathrm{dest}(\texttt{compute interest}) = \texttt{money}$$

The set $PRED_{DT}$ contains among others the predicate name "$\leq_{\texttt{money}}$" with the function arg defined on it as follows:

$$\mathrm{arg}(\leq_{\texttt{money}}) = (\texttt{money},\texttt{money})$$

Hence `compute interest` has operation signature

$$\texttt{compute interest} : \texttt{money} \times \texttt{int} \rightarrow \texttt{money}$$

while "$\leq_{\texttt{money}}$" has predicate signature

$$\leq_{\texttt{money}} : \texttt{money} \times \texttt{money}$$

Actual (sets of) values are associated to the sorts of a data type signature by means of an *interpretation*. At the same time, an interpretation assigns a function to each operation and a relation to each predicate, respecting their respective signatures.

Definition 2.3 (Interpretation of a Data type Signature) *An* interpretation of a data type signature *DT is a three-tuple $\mu[DT] = (\mu[SORT_{DT}], \mu[OPER_{DT}], \mu[PRED_{DT}])$ of functions where*

- *$\mu[SORT_{DT}]$ assigns a (possibly infinite) set of* values *to each sort name in $SORT_{DT}$;*
- *$\mu[OPER_{DT}]$ assigns to each operation name in OPER with signature $\omega : d_1 \times \ldots \times d_n \rightarrow d$ a function $\mu[OPER_{DT}](\omega) : \mu[SORT_{DT}](d_1) \times \ldots \times \mu[SORT_{DT}](d_n) \rightarrow \mu[SORT_{DT}](d)$;*
- *$\mu[PRED_{DT}]$ assigns to each predicate name in PRED with signature $\pi : d_1 \times \ldots \times d_n$ a relation such that $\mu[PRED_{DT}](\pi) \subseteq \mu[SORT_{DT}](d_1) \times \ldots \times \mu[SORT_{DT}](d_n)$.*

□

In the interpretation of the data type signature underlying our running example EER scheme, the function $\mu[SORT_{DT}]$ assigns to the sort name `money` the set of all rational numbers with at most two decimals. The function $\mu[PRED_{DT}]$ assigns to the predicate name "$\leq_{\texttt{money}}$" the binary relation, containing those pairs (m, m') of money values for which m is indeed less than or equal to m'.

The above definitions already allow one to specify arbitrary data types. Note in particular that the actual structure of the data is totally *encapsulated* and is only accessible through the use of the provided operations and predicates. Looking back at the attributes of the scheme for our example application of ATMs in Table 2.1, we see that the domain of the `residence`-attribute of the entity type `CUSTOMER` is in fact explicitly specified as a *list* of addresses. Hence one should also be able to specify some of the structure of attributes *outside* the data type signature.

As the same type constructors used for complex attributes (i.e., list, set, and bag) are also used for components, it should not only be possible to explicitly specify complex structured values, but the model should also offer the possibility to build such a structure on top of entities. Therefore, the following definition will be stated in more general terms than just the context of data types. More precisely, we define *sort expressions* based on some unspecified set of abstract symbols:

Definition 2.4 (Sort expressions) *Let S be a set of symbols with a mapping $\mu[S]$ which maps each element of S to a finite set. The set* $\mathrm{expr}(S)$ *of* sort expressions *over S is defined recursively as follows:*

1. $S \subset \mathrm{expr}(S)$*;*
2. *if* $s \in \mathrm{expr}(S)$*, then* $\{\mathrm{set}(s), \mathrm{bag}(s), \mathrm{list}(s)\} \subset \mathrm{expr}(S)$*;*
3. *if* $s_1, \ldots, s_n \in \mathrm{expr}(S)(n > 1)$*, then* $\mathrm{prod}(s_1, \ldots, s_n) \in \mathrm{expr}(S)$.

$\mu[S]$ induces a mapping $\mu[\mathrm{expr}(S)]$ on $\mathrm{expr}(S)$ as follows:

1. $\mu[\mathrm{expr}(S)](s) := \mu[S](s)$ *for* $s \in S$*;*
2. $\mu[\mathrm{expr}(S)](\mathrm{set}(s)) := \mathcal{F}(\mu[\mathrm{expr}(S)](s))$*;*
3. $\mu[\mathrm{expr}(S)](\mathrm{bag}(s)) := \mathcal{B}(\mu[\mathrm{expr}(S)](s))$*;*
4. $\mu[\mathrm{expr}(S)](\mathrm{list}(s)) := (\mu[\mathrm{expr}(S)](s))^*$*;*
5. $\mu[\mathrm{expr}(S)](\mathrm{prod}(s_1, \ldots, s_n)) := \mu[\mathrm{expr}(S)](s_1) \times \ldots \times \mu[\mathrm{expr}(S)](s_n)$.

□

The reason for defining sort expressions in terms of abstract symbols is that the collection of types (i.e., entity types, relationship types, data types,...) on which actual sort expressions can be based, depends on whether they are used in a query or in the definition of an EER scheme. E.g., an interpretation $\mu[DT]$ of a data type signature DT induces an interpretation $\mu[\mathrm{expr}(\mathit{SORT}_{DT})]$ on the set of sort expressions over DT, where SORT_{DT} then plays the role of the set of symbols S.

Besides, in Section 2.3, we introduce a query language for EER databases in which the outcome of a query may be an arbitrarily complex type built using entity types, relationship types, data types, and type constructors. Therefore, we define *sort expressions* to any level of depth, rather than to just one level as required for the modeling of EER schemes.

As an example, consider the query "*Give the serial numbers and limits of all cash cards*". The answer to this query is a subset of the interpretation of the sort expression $\mathrm{prod}(\mathrm{int}, \mathrm{money})$. Another example is the domain of the `residence`-attribute of entity type `CUSTOMER`, being `list(address)`.

Since the above definition allows sort expressions to be nested to any depth, more "exotic" expressions such as `set(list(time, money))`, which could be used to represent the attributes of the entity type `TRANSACTION` in a tabular format, are also allowed.

Definition 2.4 concludes the formalization of the first layer of the EER data model, namely that of data types. We now turn our attention to EER schemes and instances. As mentioned in the introduction to this section, an EER scheme is defined over a fixed data type signature. It is formalized as a number of sets containing the names of each of the basic building blocks such as entity types, attributes etc. Connections between these building blocks are formalized by means of functions, satisfying a number of constraints to be explained following the definition.

Definition 2.5 (EER Scheme) *Let DT be a data type signature. An* EER scheme $\mathcal{S}_{DT}$ *over DT consists of*

- *six disjoint finite sets* ***E-TYPE***, ***R-TYPE***, ***ROLE***, ***ATTR***, ***COMP***, ***CONS***
- *seven functions with the following signatures:*

$$\begin{aligned}
\text{participants} &: \textbf{\textit{R-TYPE}} \to \textbf{\textit{E-TYPE}}^{+} \\
\text{relship} &: \textbf{\textit{ROLE}} \to \textbf{\textit{R-TYPE}} \\
\text{entity} &: \textbf{\textit{ROLE}} \to \textbf{\textit{E-TYPE}} \\
\text{owner} &: \textbf{\textit{ATTR}} \to \mathcal{F}(\textbf{\textit{E-TYPE}} \cup \textbf{\textit{R-TYPE}}) \\
&\quad\ \textbf{\textit{COMP}} \to \mathcal{F}(\textbf{\textit{E-TYPE}}) \\
\text{domain} &: \textbf{\textit{ATTR}} \to \{D' \mid \exists D \in \textbf{\textit{SORT}}_{\textbf{\textit{DT}}} : D' \in \{D, \text{set}(D), \text{bag}(D), \text{list}(D)\}\} \\
&\quad\ \textbf{\textit{COMP}} \to \{E' \mid \exists E \in \textbf{\textit{E-TYPE}} : E' \in \{E, \text{set}(E), \text{bag}(E), \text{list}(E)\}\} \\
\text{input}, \text{output} &: \textbf{\textit{CONS}} \to \mathcal{F}(\textbf{\textit{E-TYPE}}) - \emptyset
\end{aligned}$$

satisfying the following constraints:

1. *For each $R \in$* ***R-TYPE*** *with* $\text{participants}(R) =< E_1, \ldots, E_m >$*, there must be m different $P_i \in$* ***ROLE*** $(1 \leq i \leq m)$ *with* $\text{relship}(P_i) = R$ *and* $\text{entity}(P_i) = E_i$*.*
2. *For $T_1, T_2 \in$* ***CONS*** *holds that if $T_1 \neq T_2$, then* $\text{output}(T_1) \cap \text{output}(T_2) = \emptyset$*.*
3. *Let $E \in$* ***E-TYPE****. Then (E, E) should not be in the transitive closure of the relation* $\{(I, O) \mid \exists T \in$ ***CONS***$, I \in \text{input}(T), O \in \text{output}(T)\}$*.*

□

The fact that relship respectively entity are functions that map a role to one single relationship type respectively entity type, implies that role names must be unique within an EER scheme.

Likewise, each attribute or component is assigned a unique domain by the corresponding function. In contrast, the function owner maps each attribute (respectively component) to a *set* of entity and relationship types (respectively entity types). Consequently, entities and relationships (respectively entities) can have attributes (respectively components) with the same name, on the condition that they

all have the same domain. For instance, in the example EER scheme (cf. Table 2.1), both BANKs, CASHIERs, CONSORTIUMs and CUSTOMERs have names, all of which are strings.

By constraint 1, a unique role exists for each participation of an entity type in a relationship type (cf. Notation 2.6 below).

Constraint 2 prevents entity types from being constructed in two different ways. Constraint 3 prevents an entity type from being part of its own construction.

As an illustration, we present part of the EER scheme corresponding to the EER diagram in Figure 2.1.

$$
\begin{aligned}
\texttt{E-TYPE} &= \{\texttt{BANK, CASHIER, ACCOUNT,} \ldots\} \\
\texttt{R-TYPE} &= \{\texttt{accesses, holds, has,} \ldots\} \\
\texttt{ROLE} &= \{\texttt{teller, owner,} \ldots\} \\
\texttt{ATTR} &= \{\texttt{balance, location, amount, name,} \ldots\} \\
\texttt{COMP} &= \{\texttt{manages, issues,} \ldots\} \\
\texttt{CONS} &= \{\texttt{part1, part2}\}
\end{aligned}
$$

$$
\begin{aligned}
\text{participants}(\texttt{owned by}) &= (\texttt{ATM, CONSORTIUM}) \\
\text{relship}(\texttt{teller}) &= \texttt{owned by} \\
\text{entity}(\texttt{teller}) &= \texttt{ATM} \\
\text{owner}(\texttt{balance}) &= \{\texttt{ACCOUNT}\} \\
\text{domain}(\texttt{balance}) &= \texttt{money} \\
\text{owner}(\texttt{residence}) &= \{\texttt{CUSTOMER}\} \\
\text{domain}(\texttt{residence}) &= \texttt{list(address)} \\
\text{owner}(\texttt{employs}) &= \{\texttt{BANK}\} \\
\text{domain}(\texttt{employs}) &= \texttt{list(CASHIER)} \\
\text{owner}(\texttt{name}) &= \{\texttt{BANK,CASHIER,CUSTOMER,CASHIER}\} \\
\text{domain}(\texttt{name}) &= \texttt{string} \\
\text{input}(\texttt{part1}) &= \{\texttt{ENTRY STATION}\} \\
\text{output}(\texttt{part1}) &= \{\texttt{ATM, CASHIER STATION}\}
\end{aligned}
$$

Notation 2.6 *For $R \in$ R-TYPE with $\text{participants}(R) =< E_1, \ldots, E_m >$, and $P_i \in$ ROLE$(1 \leq i \leq m)$ with $\text{relship}(P_i) = R$ and $\text{entity}(P_i) = E_i$, we denote $R(P_1 : E_1, \ldots, P_m : E_m) \in$ R-TYPE and $P_i : R \to E_i \in$ ROLE.*

For $A \in$ ATTR with $\text{owner}(A) \ni S$ and $\text{domain}(A) = D$ we denote $A : S \to D$ and likewise, for $C \in$ COMP with $\text{owner}(C) \ni E$ and $\text{domain}(C) = E'$ we denote $C : E \to E'$.

For $T \in$ CONS with $\text{input}(T) = \{I_1, \ldots, I_n\}$ and $\text{output}(T) = \{O_1, \ldots, O_m\}$ we denote $T(I_1, \ldots, I_n; O_1, \ldots, O_m) \in$ CONS.

Finally, an EER instance consists of a number of functions, assigning among others actual entities to each entity type.

Definition 2.7 (Universe of Entities) *Let $\mathcal{S}$ be an EER scheme. $\mathcal{E} = \bigcup_{E \in \text{E-TYPE}} \mathcal{E}_E$ is a countably infinite set, called the* universe of entities*. It includes for each entity type E in $\mathcal{S}$ a countably infinite set $\mathcal{E}_E$ of entities of type E.*

For different entity types E and E', $\mathcal{E}_E \cap \mathcal{E}_{E'} = \emptyset$.
□

Definition 2.8 (EER Instance) *Let $\mathcal{S}_{DT}$ be an EER scheme over a data type signature DT. An* EER instance $\mathcal{I}$ *over $\mathcal{S}_{DT}$ consists of the following six functions:*

- μ[*E-TYPE*], *which maps each entity type $E \in$ E-TYPE to a finite subset of $\mathcal{E}_E$;*
- μ[*R-TYPE*], *which maps each relationship type $R(P_1 : E_1, \ldots, P_m : E_m) \in$ R-TYPE to a finite set of tuples of entities called* relationships, *such that μ[R-TYPE]$(R) \subseteq \mu$[E-TYPE]$(E_1) \times \ldots \times \mu$[E-TYPE]$(E_m)$;*
- μ[*ROLE*], *which maps each role $P_i : R \to E_i$ to the function μ[ROLE]$(P_i) : \mu$[R-TYPE] $\to \mu$[E-TYPE] satisfying μ[ROLE]$(P_i)(r) = e_i (1 \leq i \leq m)$ for each $r = (e_1, \ldots, e_m) \in \mu$[R-TYPE]$(r)$;*
- μ[*ATTR*], *which maps each attribute $A : E \to D'$ respectively $A : R \to D'$ to a function μ[ATTR]$(A) : \mu$[E-TYPE]$(E) \to \mu[\exp($SORT$_{DT})](D')$ respectively μ[ATTR]$(A) : \mu$[R-TYPE]$(R) \to \mu[\exp($SORT$_{DT})](D')$;*
- μ[*COMP*], *which maps each component $C : E \to E'$ to a function μ[COMP]$(C) : \mu$[E-TYPE]$(E) \to \mu[\exp($E-TYPE$)](E')$:*
- μ[*CONS*], *which maps each construction $T(I_1, \ldots, I_n; O_1, \ldots, O_m)$ to a function μ[CONS]$(T) : \cup_{j=1}^{m} \mu$[E-TYPE]$(O_j) \to \cup_{k=1}^{n} \mu$[E-TYPE]$(I_k)$.* □

As an illustration, we present part of an EER instance over our running example EER scheme.

$$
\begin{array}{lcl}
\mu[\text{E-TYPE}](\texttt{ATM}) & = & \{e_1, e_2, e_3\} \\
\mu[\text{E-TYPE}](\texttt{CONSORTIUM}) & = & \{e_4\} \\
\mu[\text{E-TYPE}](\texttt{BANK}) & = & \{e_5, e_6, e_7\} \\
\mu[\text{E-TYPE}](\texttt{ENTRY STATION}) & = & \{e_8, e_9, e_{10}\} \\
\mu[\text{R-TYPE}](\texttt{owned by}) & = & \{(e_1, e_4), (e_2, e_4)\} \\
\mu[\text{ROLE}](\texttt{owner})(e_1, e_4) & = & e_1 \\
\mu[\text{ROLE}](\texttt{teller})(e_1, e_4) & = & e_4 \\
\mu[\text{ATTR}](\texttt{dispensed})(e_1) & = & 1234.50 \\
\mu[\text{COMP}](\texttt{consists of})(e_4) & = & (e_5, e_6) \\
\mu[\text{CONS}](\texttt{part1})(e_1) & = & e_8 \\
\mu[\text{CONS}](\texttt{part1})(e_2) & = & e_9 \\
\mu[\text{CONS}](\texttt{part1})(e_3) & = & e_{10}
\end{array}
$$

This instance includes a consortium owning two ATMs. From a third ATM, which apparently has no owner, an amount of 1234.50 has been dispensed. The consortium consists of two banks. A third bank is not part of any consortium. As required by the construction `part1`, all three ATMs are constructed from an entry station.

2.3 A Textual Query Language : SQL/EER

As the original Entity-Relationship model [Che76] was primarily intended for database design purposes, it consisted mainly of a formalism for describing conceptual database schemes. An ER scheme resulting from a database design process is therefore traditionally translated into e.g., a relational database scheme. Consequently, queries on the actual database have to be formulated in terms of a model (namely the relational model) quite different from that in which the database was originally specified (namely the ER model).

Remarkably enough, Chen remarks in [Che76, Section 3.4]:

> *The semantics of information retrieval requests become very clear if the requests are based on the entity-relationship model of data.*

In the same section, Chen also introduces (albeit by means of a single example) a query language in which queries may be expressed *directly* in terms of the ER formalism, i.e., by referring to entity and relationship types of a given ER scheme, rather than to tables and columns of its relational equivalent. Perhaps inspired by this remark, several ER-specific query languages, such as ERROL [MR83] and CLEAR [MP80] (to name just a few from the "early days" of ER research) were developed.

The Extended ER model presented in Section 2.2 eventually became the core of an integrated DB design environment called CADDY [EHH+89]. Even in a database *design* environment, a query language has its importance e.g., as the basis for a tool which allows browsing a prototype. This observation motivated the development of a query language for the EER model, named SQL/EER [HE90]. Two of the major characteristics of this language are that

- it directly supports all concepts of the EER model, such as attributes of relationships, components and type constructions, and takes into account features well known from nowadays query languages (exceeding those of relational SQL [CAE+76]), such as arithmetic, aggregate functions, output nesting and subqueries as variable domains;
- its syntax as well as its semantics are formally defined.

The context free syntax is defined in the usual way, by means of an Extended Backus Naur Form grammar. To describe context sensitive conditions formally, the EBNF grammar is extended to an attribute grammar by adding attributes and attribute evaluation rules. Definition of the languages semantics is realized using a special case of the *operational* approach to language specification, namely by means of *translation*. An extension of the attribute grammar is used to formally map an SQL/EER query into an equivalent expression of the EER *calculus* [HG88]. This calculus is in term described using the *denotational* approach to language specification : the semantics of each calculus expression is described as an associated input/output function.

As they are of no relevance to the contribution of this thesis, we do not discuss the EER calculus, neither do we elaborate on the attribute mechanism used for mapping SQL/EER into the calculus. The interested reader is referred to [HG88, HE91, HE90]. Besides an informal introduction to most (but not all) of SQL/EER's characteristics, we limit ourselves in the remainder of this section to those parts of the EBNF grammar which describe the context free syntax of SQL/EER (also collected in Appendix B).

Analogous to relational SQL, SQL/EER uses the **select-from-where** clause. This is captured in the following EBNF grammar rule.

SFW-TERM ::= **select** TERMLIST
from DECLLIST
[**where** FORMULA]

As a first example, consider the SQL/EER query of example 2.9 (over the scheme of Figure 2.1). It retrieves the `serial number` of all `CASH CARD`s with the trivial `password` "password" and a (credit) `limit` less than or equal to 100.000.

Example 2.9

select cc.serial_number
from cc **in** CASH_CARD
where cc.password = "password" **and** cc.limit $\leq_{\texttt{money}}$ *100.000*

In the result of a query, duplicates are not eliminated (the reason for which will become clear when we discuss aggregate functions in Example 2.12), hence the above query returns a bag of integers.

In this query, the variable "cc" is declared. It ranges over the set of currently stored `CASH CARD`s.

DECL ::= VARIABLE **in** ENTITYTYPE

The variable cc can be used to build terms like "cc.password" and "cc.limit", designating the `password` and `limit` of the `CASH CARD` "cc", respectively.

TERM ::= VARIABLE
| TERM '.' ATTRIBUTE

The formula "cc.limit $\leq_{\texttt{money}}$ 100.000" uses the predicate "$\leq_{\texttt{money}}$", defined for the `money` data type.

FORMULA ::= TERM DATAPRED TERM

Besides entity types and relationship types, any multi-valued term can also be used as range in a declaration.

RANGE ::= TERM

For instance, in the SQL/EER query of Example 2.10, the variable a1 is bound to the finite list of addresses being the residence of person p1.

Example 2.10

select c1.name
from a1 **in** p1.residence, c1 **in** CUSTOMER, a2 **in** p2.residence, c2 **in** CUSTOMER
where a1 = a2 **and** p2.name = "`John`"

This query retrieves the names of all CUSTOMERs who share one of their residences with a CUSTOMER called "John". Note that the result of an SQL/EER is a bag. This means that the same name may appear several times in the answer to this query. By placing the reserved word **distinct** in front of the term list in the **select**-clause, a set of distinct names is computed.

Note that since we do not require the customers c1 and c2 to be different, the answer to this query will also include all customers named John themselves.

The next example shows the use of relationship types as predicates in SQL/EER. Suppose we want to know the names of those CUSTOMERs who have a CASH CARD that accesses ACCOUNTs which they also hold. Example 2.11 shows the corresponding SQL/EER query.

Example 2.11

select c.name
from ac **in** ACCOUNT, cc **in** CASH CARD, cu **in** CUSTOMER
where cu holds ac **and** cc accesses ac **and** cu has cc

Relationship types can be used as predicate names in formulas. In the case of relationships with more than two participating entity types, prefix notation is used instead of infix.

FORMULA ::= PARTICIPANT RELSHIPTYPE PARTICIPANT
| RELSHIPTYPE '(' PARTLIST ')'
PARTICIPANT ::= TERM
PARTLIST ::= PARTICIPANT [',' PARTLIST]

The query of example 2.12 returns the names of those CUSTOMERs holding an ACCOUNT for which the following holds: if the balance of their ACCOUNT is raised with a five percent interest, then the new balance becomes higher than the average of all balances of all ACCOUNTs with a positive balance.

Example 2.12

select cu.name
from ac1 **in** ACCOUNT, cu **in** CUSTOMER
where cu holds ac1 **and**
compute_interest(ac1.balance,5) $\geq_{\texttt{money}}$ **avg** (**select** ac2.balance
from ac2 **in** ACCOUNT)
where ac2 $\geq_{\texttt{money}}$ 0)

This example first of all illustrates the use of data operations. An application of the function compute interest (cf. Section 2.1) may be used as a term. Second, the example shows the use of subqueries in the **where**-clause of an SQL/EER query (used to retrieve the bag of balances of all ACCOUNTs). In general, a **select-from-where** block may be used like any other (bag-valued) term. Third, the example shows how SQL/EER incorporates aggregate functions as known from relational SQL. Aggregate functions, such as **avg** (standing for "average"), **min** (standing for "minimum") and **sum** may be applied to bags of values on which addition and mathematical comparison are defined (such as integer and money).

TERM ::= DATAOPNS '(' TERMLIST ')'
| '(' SFW-TERM ')'
| AGGROPNS '(' TERM ')'

Example 2.13 illustrates the handling of lists, as well as the occurrence of multiple terms in the **select**-clause. The query of this example enumerates the `name`s of `bank`s which are part of the `CONSORTIUM` `name`d "*Banks United*".

Example 2.13

select i, co.consists_of[i].name
from i **in ind**(co.consists_of), co **in** CONSORTIUM
where co.name = '*Banks United*'

The (integer) variable i ranges over the set of all indices occurring in the list `co.consists_of`, containing the `bank`s which are part of the `CONSORTIUM` `name`d "*Banks United*" (assuming for the sake of simplicity that banks have a uniquely identifying name). This set of indices is retrieved using the built-in function **ind**. A particular element of a list is specified using rectangular braces "[" and "]".

TERM ::= TERM '[' INTEGER ']'
| ind '(' TERM ')'

Since this query has two terms in its **select**-clause, it returns a bag of two-tuples, consisting of an integer value and a string. In general, if a query has n terms in its **select**-clause, it returns a bag of n-tuples. This gives us two special cases, namely those where n is either zero or one:

- If a query has only one term in its **select**-clause, then it formally returns a bag of one-tuples, but one-tuples are considered equal to their single element.
- If a query has no terms in its **select**-clause, then it formally returns a bag of tuples of length zero. Hence there are two possibilities:
  1. If the **from**- and **where**-clauses of the query "return" an empty result (for instance, if we look for `CASH CARD`s whose `limit` is both strictly positive and negative), then the query returns the empty bag.
  2. Otherwise, the query returns the bag consisting of the (empty) tuple of length zero.

In the (rather theoretical) case of an SQL/EER query with an empty **from**-clause, an empty bag is also returned.

Suppose we want to retrieve the `name`s of those `CASHIER`s who only enter `CASHIER TRANSACTION`s of more than 100.000, or `CASHIER TRANSACTION`s concerning an `ACCOUNT` with a `balance` over 100.000. In Example 2.14, this query is formulated in SQL/EER.

Example 2.14

select ca.name
from ca **in** CASHIER
where for all ct **in** CASHIER_TRANSACTION : (ct entered_by ca) **implies**
((ct.amount $\geq_{\texttt{money}}$ *100.000*) **or**
(**exists** ac **in** ACCOUNT : (ct concerns ac **and**
ac.balance $\geq_{\texttt{money}}$ *100.000*))
)

This example illustrates the use of inheritance in queries. Both the relationship `concerns` and the attribute `amount` are defined on the entity type `TRANSACTION`, but since `CASHIER TRANSACTION` is a subtype of `TRANSACTION`, `concerns` and `amount` also apply to `CASHIER TRANSACTION`s.

Next, this example also illustrates how SQL/EER supports quantifiers and logical connectives such as "**or**" and "**implies**".

FORMULA ::= '(' FORMULA **implies** FORMULA ')'
| '(' FORMULA **or** FORMULA ')'
| '(' FORMULA **and** FORMULA ')'
| **not** '(' FORMULA ')'
| **exists** DECLLIST ':' FORMULA
| **forall** DECLLIST ':' FORMULA

A final example (Ex.2.15) illustrates the use of subqueries in the **select**- and **from**-clauses of an SQL/EER query. A subquery in the **select**-clause is used to structure the output of a query. The example query returns, for each `password` of some `CASH CARD` (retrieved using a subquery in the **from**-clause), the `serial number`s of the `CASH CARD`s having this `password`.

Example 2.15

select pwd, (**select** cc.serial_number
from cc **in** CASH_CARD
where cc.password = pwd))
from pwd **in distinct** (**select** cc.password
from cc **in** CASH_CARD)

This query returns a bag of two-tuples, the first component of which is a string, while the second is in turn a bag of integers.

Chapter 3

An Introduction to PROGRES

3.1 On Formal Graph Representations

In this thesis, we present *graph-based* definitions for both the EER data model (in this chapter) and a query language for this model (see Chapter 4). In a graph-based definition of a graphical formalism (such as that of EER diagrams), each "instance" of the formalism is formally represented by an attributed,[1] directed labeled graph.

As an example, consider the part of the EER diagram of Figure 2.1 shown in Figure 3.1. Figure 3.2 shows a representation of this partial EER diagram as a formal graph. Table 3.1 shows the attributes of this graph.

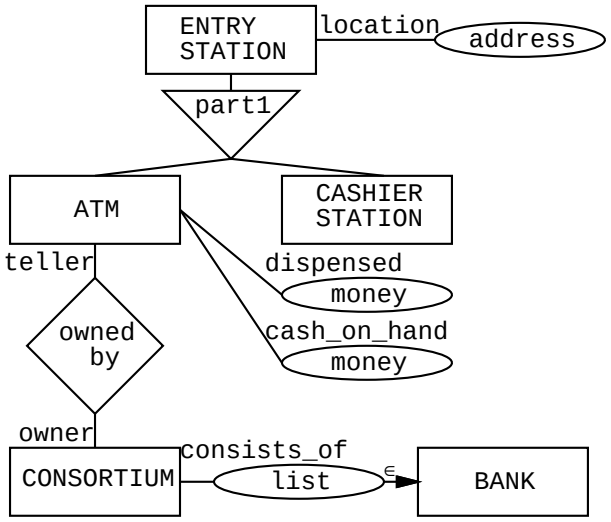


Figure 3.1: Part of the EER diagram of Figure 3.1

[1] Note that the word "attribute" is now used in two different meanings : one in the context of graphs, and the other in the context of the EER-model.

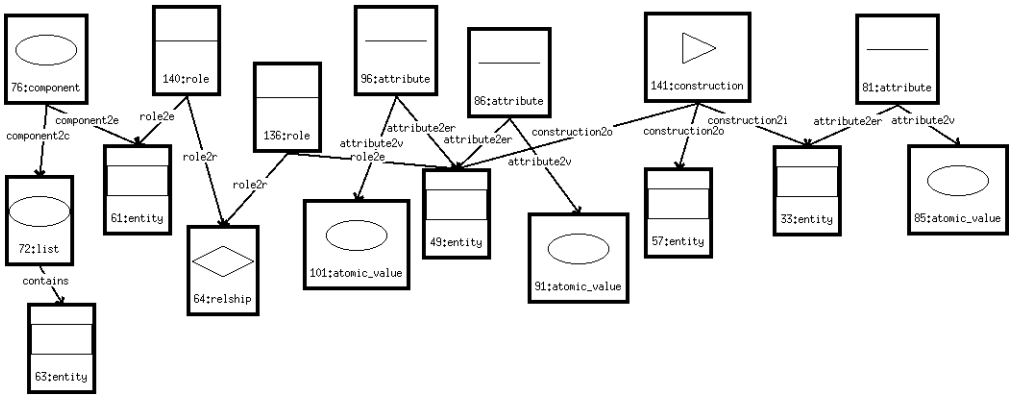


Figure 3.2: Graph representation of the EER diagram of Figure 3.1

Id.	Label	Att.Name	Attribute Value
76	component	Name	consists of
140	role	Name	owner
136	role	Name	teller
96	attribute	Name	dispensed
86	attribute	Name	cash on hand
141	construction	Name	part1
81	attribute	Name	location
72	list	Name	
63	entity	Name	BANK
61	entity	Name	CONSORTIUM
64	relship	Name Arity	owned by 2
101	atomic_value	Name	money
49	entity	Name	ATM
91	atomic_value	Name	money
57	entity	Name	CASHIER STATION
33	entity	Name	ENTRY STATION
85	atomic_value	Name	address

Table 3.1: Attributes of some nodes from Figure 3.2

In this formal graph representation, each element of the EER diagram (such as a role, an entity type or a data type) is represented by a node, shown as a black rectangle in Figure 3.2. Within each node are shown an icon (which has only a decorative purpose), a unique node-identifier and a *label*. Every node has a `Name`-attribute, which contains the name of the corresponding element in the EER diagram. For instance, the node with identifier 33 corresponds to the entity type `ENTRY STATION` in the EER diagram, while the node with identifier 81 corresponds to the `location` attribute (*not* the attribute's domain !) of this entity type. In addition, nodes representing relationship types have an `Arity`-attribute. Since `owned by` is a binary relationship type, the `Arity`-attribute of the corresponding node has the value 2.

The interrelationships between the various elements are represented in the formal graph representation by means of directed labeled edges. For instance, an edge labeled `attribute2er` links the node corresponding to the `location` attribute to the node corresponding to the `ENTRY STATION` entity type.

Note that all labels used in the graph are independent of any particular EER scheme: all EER scheme dependent information is stored in node attributes.

3.2 PROGRES specifications

From the above ideas clearly follows our need for an expressive *graph model*, i.e., a formalism that allows the specification of a family of graphs. To this end, we chose to use the graph rewriting formalism PROGRES [Sch89, Sch91b], a very high level operational specification language based on **PRO**grammed **G**raph **RE**writing **S**ystems.

A major motivation for choosing the PROGRES formalism was that the specifications presented in this thesis could consequently be made using the PROGRES-*system* [NS90]. Concretely, the specifications were entered using this systems syntax-directed editor [Sch90b], which allowed them to be analyzed by the system's incrementally working type-checker and executed by the system's integrated interpreter. For instance, the graph depicted in Figure 3.2 was generated using *EDGE* [New91], a generic graph browser which has been incorporated into the PROGRES system.

In this chapter we present an informal overview of those features of the PROGRES language to be used in the remainder of this thesis, by means of an example specification for the set of all syntactically correct EER diagrams. Basically, the PROGRES language allows the specification of the static structure of a family of directed, node-attributed, and labeled graphs, together with a collection of operations on graphs in this family. Recalling Section 1.2, a PROGRES specification is therefore a *graph rewrite system* (as the languages name rightfully suggests). Demonstrating the fact that graph rewrite systems in general, and PROGRES specifications in particular, may be used to formally specify visual database languages is the main contribution of this thesis. This chapter therefore offers an introduction specifically to the way in which we propose to use PROGRES to formally specify visual (database) languages (and not to the PROGRES formalism in general).

A PROGRES specification of a graph grammar consists basically of two parts. In the *graph scheme*, the types of nodes and edges as well as node attributes, which may occur in a graph, are declared. For instance, the graph scheme from the specification of EER diagrams contains node type declarations

stating that there may be nodes labeled `relship`, `construction` and `entity`,[2] all of which have a `Name` attribute. Additionally, the scheme contains edge type declarations stating that edges labeled either `construction2i` or `construction2o` may go from nodes labeled `construction` to nodes labeled `entity`.

Although the collection of declarations in a graph scheme already defines a family of graphs, not all possible structural integrity constraints one would like to impose on such a family can be expressed in such a scheme. For instance, it is impossible to express the acyclicity constraint on entity type constructions using only graph scheme declarations. Such a constraint may be captured in the second part of a PROGRES specification, being a set of graph rewrite rules or *productions*, obeying the type restrictions imposed by the graph scheme. In our example, such a production specifies that, given two nodes labeled `entity`, a new node labeled `construction` may be added, linked to the two given nodes with edges labeled respectively `construction2i` and `construction2o`, on the condition that this addition would not violate the acyclicity constraint. How exactly such a constraint is modeled in a PROGRES production is explained further on in this chapter.

Together, a graph scheme and a set of productions constitute an *operational* specification of a graph grammar: a syntactically correct graph in the language defined by this grammar is yielded by *applying* a sequence of productions to an initial empty graph. In the remainder of this chapter, we elaborate on respectively graph schemes (Section 3.2.1), productions (Section 3.2.2), and the notion of applying production sequences (Section 3.2.3).

3.2.1 Graph Schemes

The basis of a PROGRES specification is a graph scheme. The following components of a graph scheme are distinguished:

- *type declarations*: these are used to introduce labels for nodes and edges in the considered collection of graphs, as well as to initialize node attributes;
- *class declarations*: these denote coercions of node types with common properties by means of multiple inheritance, hence they play the role of second order types. Class declarations include attribute declarations.

The section `GraphScheme` of the specification `EER` (as shown in Figure 3.3) contains all type and class declarations of the specification. Note that sections are merely a syntactical way of structuring a specification, and have no semantical meaning whatsoever.

The root of the node class hierarchy of the example specification is `NODE`. Essentially, the only purpose of this class is the definition of the string-valued attribute `Name`, in which all names from the EER scheme corresponding to the EER diagram (such as names of entity types, attributes,…) may be stored. Since these names are "proper" to a node, as opposed to attributes which are *derived* from other information present in the graph (see further on), the attribute `Name` is declared as being an `intrinsic` one. In the same declaration, the attribute is initialized with the empty string.

The next two node classes in the specification do not correspond directly to some component of EER diagrams, but must be seen as coercions of other node classes that share some common properties.

[2] Note that "node label" and "node type" are used as synonyms.

```
section GraphScheme
   node class NODE
      intrinsic
         Name : string := "";
   end;
   node class ENT_REL is a NODE end;
   node class ENTITY is a ENT_REL end;
   node type entity : ENTITY end;
   node class CONSTRUCTION is a NODE end;
   node type construction : CONSTRUCTION end;
   edge type construction2i : CONSTRUCTION -> ENTITY [1:n];
   edge type construction2o : CONSTRUCTION -> ENTITY [1:n];
   node class RELSHIP is a ENT_REL
      derived
         Arity : integer;
   end;
   node type relship : RELSHIP
      redef derived
         Arity = card ( self.<-role2r- );
   end;
   node class ROLE is a NODE end;
   node type role : ROLE end;
   edge type role2e : ROLE -> ENTITY [1:1];
   edge type role2r : ROLE [1:n] -> RELSHIP [1:1];
   node class VALUE is a NODE end;
   node class ATOMIC_VALUE is a VALUE end;
   node type atomic_value : ATOMIC_VALUE end;
   node class COMPLEX_VALUE is a VALUE end;
   edge type contains : COMPLEX_VALUE -> NODE;
   node class SET is a COMPLEX_VALUE end;
   node type _set : SET end;
   node class SINGLETON is a COMPLEX_VALUE end;
   node type singleton : SINGLETON end;
   node class MVALUE is a COMPLEX_VALUE end;
   node class BAG is a MVALUE end;
   node type bag : BAG end;
   node class LIST is a MVALUE end;
   node type list : LIST end;
   node class ATTRIBUTE is a NODE end;
   node type attribute : ATTRIBUTE end;
   edge type attribute2er : ATTRIBUTE -> ENT_REL [1:1];
   edge type attribute2v : ATTRIBUTE -> VALUE [1:1];
   node class COMPONENT is a NODE end;
   node type component : COMPONENT end;
   edge type component2e : COMPONENT -> ENTITY [1:1];
   edge type component2c : COMPONENT -> COMPLEX_VALUE [1:1];
end;
```

Figure 3.3: A PROGRES specification for EER diagrams, graph scheme

For example, the node class ENT_REL is the common superclass of the classes ENTITY and RELSHIP since entities and relationships share the property of being able to have attributes (in the EER sense of the word) defined on them.

The node class ENTITY (which is a subclass of ENT_REL) has a single type entity. Likewise, the node class CONSTRUCTION has a single type construction. The reason for declaring these types is that actual nodes have to belong to a unique type, and cannot belong directly to a class. For simplicity however, in the sequel we will often refer to "nodes of a certain class" rather than to " nodes of a type of a certain class".

By means of edges of type construction2i (respectively construction2o) constructions are linked to their input entity types (respectively their output entity types). In addition to specifying the label, source class and target class of edges, an edge type declaration may also put *cardinality constraints* on edges of that type. For example, the expression [1:n] in the edge type declarations construction2i and construction2o ensures that no construction is ever created which has no input and/or no output types. Other allowed cardinality constraints are [0:1], [1:1] and [0:n] (with the latter indicating the "absence" of a cardinality constraint).

In the node class RELSHIP and its single node type relship we demonstrate the second kind of attributes, namely derived attributes (as opposed to intrinsic ones). Derived attributes are those attributes whose value may be computed by means of a derivation rule. In the declaration of the node class RELSHIP, a single derived attribute Arity is declared, whose value will at any time equal the number of entity types that play a role in the considered relationship type. This derivation rule is expressed in the declaration of the node type relship. The expression "card (self.<-role2r-)" counts (card stands for "cardinality") the number of edges of type role2r entering (hence the arrowhead to the left of the edge type name) self, that is, the considered relationship type.

A third kind of attributes (not exemplified in the EER specification) are so-called *meta*-attributes. These are attributes of *node types*, rather than nodes. In the PROGRES specification which is the main theme of Chapter 4, meta-attributes always take a *node-type* (or a set of node-types) as value. For instance, given the declaration "meta a_meta_attribute : type in NODE", the attribute a_meta_attribute could take any type in the graph scheme of Figure 3.3 as value. The need for this kind of attributes will be extensively motivated in Chapter 4.

The edge type role2r (as well as its counterpart role2e) are declared following the declaration of the node class ROLE and its single node type role. Comparable to the situation with constructions, edges of type role2r (respectively role2e) link a role to its corresponding relationship type (respectively the entity type that plays the role in that relationship type). Note the cardinality constraints : to each role corresponds exactly one relationship type as well as exactly one entity type, while for each relationship type, at least one role must be defined (cf. Definition 2.5).

The node class VALUE is the common superclass of the node classes ATOMIC_VALUE and COMPLEX_VALUE. Edges of type contains link a node of class COMPLEX_VALUE to its "component type". COMPLEX_VALUE has in turn three direct subclasses, being SET, SINGLETON and MVALUE.

The node class SET has a single type _set.[3] Nodes of class SINGLETON are used for modeling "simple" components, i.e., components which are not lists, bags or (general) sets.

[3]The reason for the underscore in _set is technical: "set" is a reserved word in the PROGRES language!

The node class MVALUE is the common superclass of the node classes BAG and LIST, which are the data types in which an element may occur **m**ultiple times, hence the name MVALUE.

Finally, the node classes ATTRIBUTE and COMPONENT as well as their corresponding edge types are declared similarly to for instance the node class CONSTRUCTION. Note how the declaration of the edge type attribute2v motivates the declaration of the node class VALUE.

3.2.2 Productions

We now turn our attention to the Productions section of the specification. Productions specify how graphs are constructed/manipulated/modified by substituting an occurrence in this graph of the *left-hand side* of the production (which is itself an ''extended" graph (see further on)) by a copy of the *right-hand side* of the production (which is just a graph). As can be seen in the production Add_ER as depicted below, left- and right-hand side of the graph part of a production are separated with the sign "::=". The production Add_ER has an empty left-hand side, and is therefore applicable to any graph. From the right-hand side, it follows that the production adds a single node of type ERtype, which is a type-valued *input parameter*.

```
production Add_ER
   ( ERname : string ; ERtype : type in ENT_REL ; out ER : ENT_REL ) =



   ::=

                 ┌──────────┐
                 │1' :ERtype│
                 └──────────┘


   transfer 1'.Name := ERname;
   return ER := 1';
end;
```

From the fact that a class may have an arbitrary number of types, it follows that nodes which are *created* by a production (i.e., nodes that are in the productions right-hand side but not in its left-hand side) must be labeled with a node type and not with a node class.

In general, productions are *parametrized* by

- atomic values : for instance, the first parameter of the production Add_ER (called ERname) is of type string;
- node types : for instance, the second parameter of the production Add_ER (called ERtype) is a type of class ENT_REL.
- nodes : for instance, the third parameter of the production Add_ER (called ER) is a node of class ENT_REL.

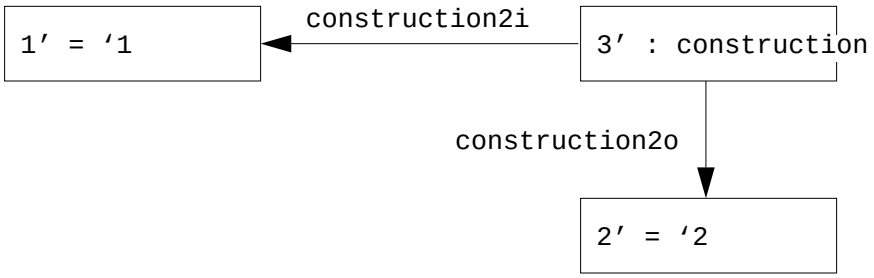
The first two parameters of the production Add_ER are input parameters, while the third parameter is an output parameter (indicated with the keyword out). The output parameter is set in the return-clause of the production.

Besides manipulating nodes and edges of the graph, productions also affect node attributes. Attribute-computations are performed in the **transfer**-clause. In the production Add_ER, the Name attribute of the newly added node is assigned the input parameter ERname. Note how a number *followed* by a single *quote* (e.g., 1') is used to refer to a node in the right-hand side of a production. Likewise numbers *preceded* by a single *backquote* refer to nodes in the left-hand side of a production.

Both kinds of quoted numbers are used in the production Add_Construction, which also illustrates two additional features of productions.

```
path Is_Constructed_From : ENTITY -> ENTITY =
   (<-construction2o- & -construction2i->) +
end;

production Add_Construction
   ( E1 : ENTITY ; E2 : ENTITY ; CName : string ; out C : CONSTRUCTION ) =
```


```
   ::=
```




```
   condition card ( ‘2.<-construction2o- ) = 0;
   transfer 3’.Name := CName;
   return C := 3’;
end;
```

Add_Construction allows the addition of a construction (in the EER sense of the word) with a given input and output entity type. In determining an isomorphic occurrence of the left-hand side, three phases are distinguished:

1. First, the nodes ‘1 and ‘2 are given as input parameters, which is indicated with the expressions “‘1=E1” and ”‘2=E2“ labeling the rectangles. Another possibility for labeling rectangles in the left-hand side of a production is by means of expressions of the form “‘3:a_node_type”, which means that a node of the type a_node_type should be *sought for* in the graph.

2. Next, an *application condition* is checked. This condition is specified in the `condition`-clause, and is stated in terms of structural and attribute properties of (the isomorphic occurrence of) the left-hand side. In the production `Add_Construction`, the application condition ensures that the node `'2` has no incoming edges of type `construction2o`. Therefore, we first compute the set of all such edges by means of the path-expression `'2.<-construction2o-` (the arrowhead "<" indicates incoming edges, whereas an expression of the form `-edge_type->` would indicate outgoing edges). Next, we compute the cardinality of this set of edges using the function `card`, and check if this cardinality is indeed equal to zero. Note how this condition corresponds to item 2 in Definition 2.5 of EER schemes.

3. Finally, the *path* `Is_Constructed_From` is used to enforce condition 3 in Definition 2.5 of EER schemes, which prevents the specification of "circular" constructions. Intuitively, paths are "virtual edges": they are declared in terms of edges and other paths, and are computed "on demand", rather than stored explicitly in a graph (as is the case for "ordinary" edges).

   A path `Is_Constructed_From` (specified prior to the production `Add_Construction`) exists between two nodes of class `ENTITY` (as shown in the signature `ENTITY -> ENTITY`) if there exists a non-empty (indicated by the sign "+") path of incoming edges of type `construction2o`, alternating (indicated by the sign "&") with outgoing edges of type `construction2i`. The cross over the double arrow in the left-hand side of the production indicates that such a path should *not* be present between the two input nodes.

Path expressions serve the same purpose in the productions `Add_Input` and `Add_Output`, which allow the addition of an extra input respectively output entity type to a construction. It would have been possible to give a production that would allow the addition of a construction, taking as input all input and output entity types at once. However, since the number of input and output entity types is arbitrary, this would have implied the use of *set valued* input parameters, a PROGRES feature not used in the sequel of this thesis, and hence not used in this specification either.

The production `Add_Input` only allows the addition of a given entity type as input to a given construction, if the entity type is not yet (directly or indirectly) in the output of the construction. This condition is checked using the path `E_Is_Output_of_C`. Starting from a node of class `CONSTRUCTION`, this path consists of an outgoing edge of type `construction2o`, followed by a sequence of incoming edges of type `construction2i` alternating with outgoing edges of type `construction2o`. The fact that the latter sequence may be of length zero, is indicated by the symbol "*". (as opposed to the symbol "+" denoting a non-empty sequence). A violation of this application condition would result in a cycle in the graph of entity type constructions, which is clearly forbidden by Definition 2.5 of EER schemes.

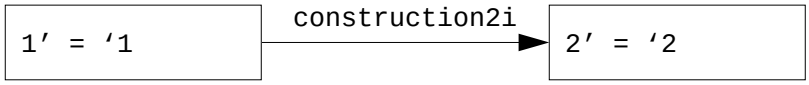
The four remaining productions allow the addition of attributes, components and roles to the graph:

- The production `Add_Atomic_Attribute` allows the addition of an atomic attribute to an entity or relationship type. Both the name of the attribute as well as the name of the domain of the attribute are passed to the production as strings in respectively the input parameters `AttName` and `DomName`.

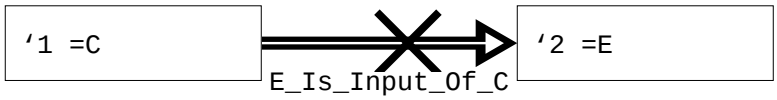
- The production `Add_Complex_Attribute` allows the addition of a `set`-, `bag`- or `list`-valued attribute to an entity or relationship type. Since `singleton` is also a type of class

```
path E_Is_Output_Of_C : CONSTRUCTION -> ENTITY =
   -construction2o-> & (<-construction2i- & -construction2o->) *
end;

production Add_Input ( C : CONSTRUCTION ; E : ENTITY ) =
```

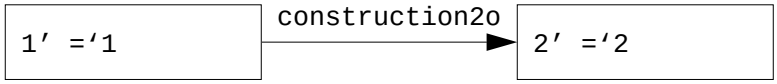

```
   ::=
```




```
end;

path E_Is_Input_Of_C : CONSTRUCTION -> ENTITY =
   -construction2i-> & (<-construction2o- & -construction2i->) *
end;

production Add_Output ( C : CONSTRUCTION ; E : ENTITY ) =
```




```
   ::=
```




```
   condition card ( ‘2.<-construction2o- ) = 0;
end;
```

`COMPLEX_VALUE` and since "singleton" attributes have to be added using `Add_Atomic_Attribute`, we have to exclude the possibility of calling this production with input parameter `Atype` equal to `singleton`, by means of the `condition`-clause.

- The production `Add_Component` allows the addition of a component relationship between two entity types already in the graph.

  By default, all nodes in the left-hand side of a production should be mapped to different nodes in the graph. However, in the production `Add_Component`, it should be allowed to map the nodes with identifiers '1 and '2 to the same node, since an entity and one a component of this entity may well have the same type. This is specified in the `folding`-clause of the production as "`folding` $\{`1, `2\}$".

- The production `Add_Role` may be used to add a given entity type as participant in a given relationship type.

A final feature of productions (not illustrated in the specification for EER diagrams) is the `embedding`-clause, which is used to affect entire sets of edges. The only case of an `embedding`-clause used elsewhere in this thesis, has the form "`embedding redirect` <-`edge_type`- `from '1 to 2'`". The result of such a clause is that *all* edges of type `edge_type` entering the node `'1` in the isomorphic occurrence of the left-hand side are simultaneously redirected to node `2'`.

By default, edges whose type is not mentioned in the `embedding`-clause, and which are adjacent to a node that is both part of the left- and right-hand side, are not affected by the application of a production.

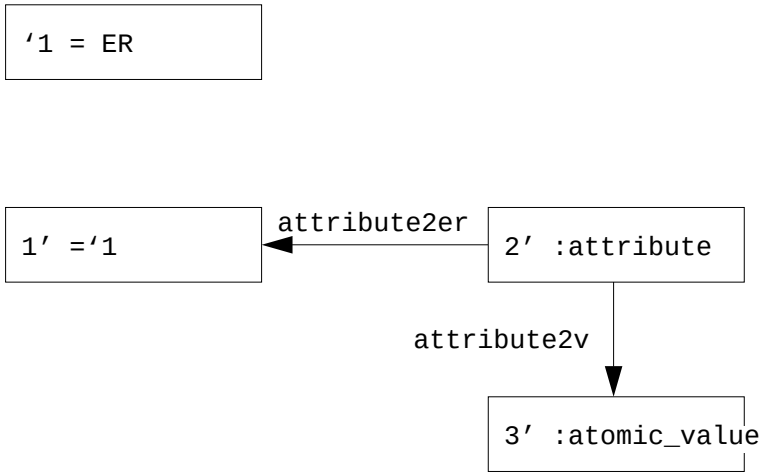
3.2.3 Transactions

We finally turn our attention to the `Transactions` section of the EER specification. In a sense, a PROGRES transaction is a program, the basic steps of which are *calls* to productions and/or other transactions. Although the PROGRES language offers a wide variety of programming language constructs (such as loops and conditional statements) to be used in the specification of transactions, in the remainder of this thesis we only use one simple kind of statement. This statement has the form

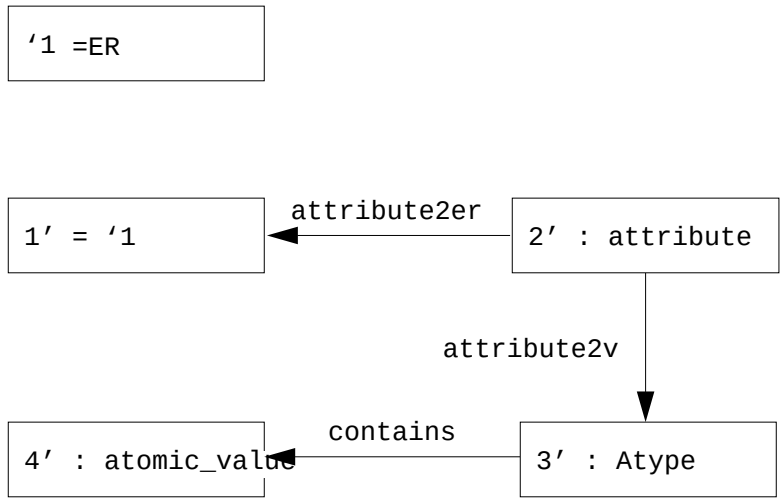
```
use v1 :  type1;
    ...;
    vN : typeN
do  production_call1 (in_p1,...,in_pP, out out_p1,...,out out_pQ)
    & ...
    & production_callR (in_p1,...,in_pS, out out_p1,...,out out_pT)
end;
```

In the part preceding the keyword "`do`", local variables are declared. Their type is either a node class, or an atomic value type. The keyword "`do`" is followed by a sequence (indicated with the symbols "`&`") of production calls. Naturally, productions must be called with the appropriate number and type of parameters. Output parameters must be preceded by the keyword "`out`" and must be variables.

```
production Add_Atomic_Attribute
   ( ER : ENT_REL ; AttName : string ; DomName : string ) =
```

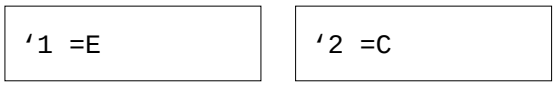



```
   transfer 2’.Name := AttName;
            3’.Name := DomName;
end;

production Add_Complex_Attribute
   ( ER : ENT_REL ; AttName : string ; Atype : type in COMPLEX_VALUE ;
      DomName : string )                                                    =
```

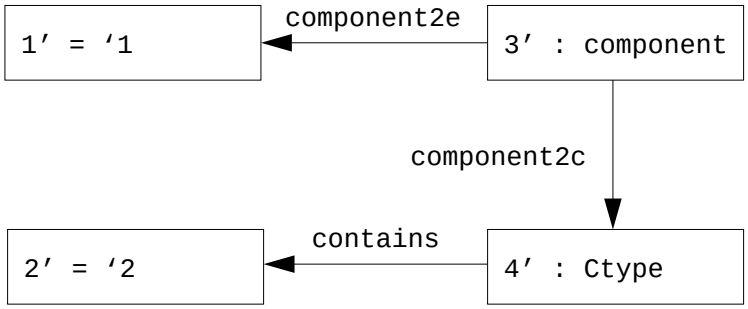



```
   condition not (Atype = singleton);
   transfer 2’.Name := AttName;
            4’.Name := DomName;
end;
```

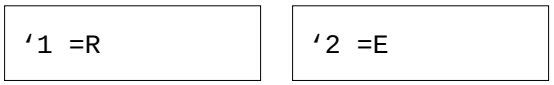
```
production Add_Component
   ( E : ENTITY ; CName : string ; Ctype : type in COMPLEX_VALUE ;
     C : ENTITY )
=
```



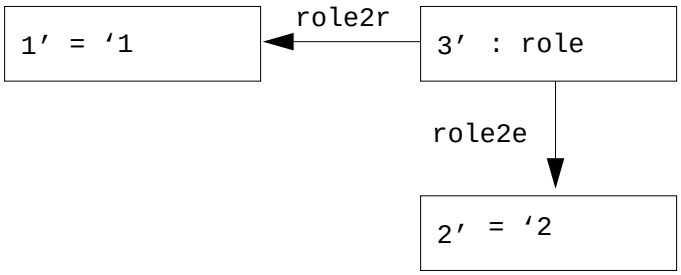

```
   ::=
```




```
   folding { ‘1, ‘2 };
   transfer 3’.Name := CName;
end;

production Add_Role ( R : RELSHIP ; RoleName : string ; E : ENTITY ) =
```




```
   ::=
```




```
   transfer 3’.Name := RoleName;
end;
```

```
section Transactions
   transaction MAIN =
      use e1, e2, e3, e4, e5, r : ENT_REL;
          c : CONSTRUCTION
      do
           Add_ER ( "ENTRY STATION", entity, out e1 )
         & Add_ER ( "ATM", entity, out e2 )
         & Add_ER ( "CASHIER STATION", entity, out e3 )
         & Add_ER ( "CONSORTIUM", entity, out e4 )
         & Add_ER ( "BANK", entity, out e5 )
         & Add_ER ( "owned by", relship, out r )
         & Add_Role ( (r : RELSHIP), "teller", (e2 : ENTITY) )
         & Add_Role ( (r : RELSHIP), "owner", (e4 : ENTITY) )
         & Add_Construction ( (e1 : ENTITY), (e2 : ENTITY), "part1", out c )
         & Add_Output ( c, (e3 : ENTITY) )
         & Add_Component ( (e4 : ENTITY), "consists of", list, (e5 : ENTITY) )
         & Add_Atomic_Attribute ( (e1 : ENTITY), "location", "address" )
         & Add_Atomic_Attribute ( (e2 : ENTITY), "cash on hand", "money" )
         & Add_Atomic_Attribute ( (e2 : ENTITY), "dispensed", "money" )
      end
   end;
end;
```

The transaction `MAIN` in the EER specification creates a graph corresponding to the EER diagram of Figure 2.1. For instance, the production `Add_ER` is first called with the constant string `"ENTRY STATION"` and the type `entity` as input parameters, and the variable `e1` as output parameter. Since this production returns a node of class `ENT_REL`, we have to explicitly *cast* `e1` to class `ENTITY` if we want to pass it as input to for instance the production `Add_Construction`, which indeed expects a node of class `ENTITY` as first input parameter.

Figure 3.2 shows the graph resulting from the execution of the `MAIN` transaction by the PROGRES system.

Chapter 4

A Query Language Defined Using Graph-Rewriting

In this chapter, we introduce and define the Graph-Oriented and Hybrid Query Languages for the EER model (abbreviated respectively GOQL/EER and HQL/EER). Following an example-based introduction to GOQL/EER (Section 4.1), we formally define this language by means of a PROGRES specification (Section 4.2). In Section 4.3, we then introduce HQL/EER.
This chapter is based on [AE94, AE97].

4.1 GOQL/EER: A Graph-Oriented Query Language for the EER Model

In this Section, we introduce the Graph-Oriented Query Language for the EER model (GOQL/EER) by means of examples. A query in GOQL/EER basically consists of graphical symbols used in EER-diagrams. For instance, variables for a customer and a cash card may be declared by drawing two (rectangular) nodes labeled respectively `CUSTOMER` and `CASH CARD`. Note that these rectangles now represent entities rather than entity *types*, as in EER-diagrams. The structural constraints applying to the construction of EER schemes, apply to graphical queries as well. That is,

- an entity playing a role in a relationship is represented as a rectangle connected to a diamond by means of an undirected edge;
- an attribute of an entity (respectively a relationship) is represented as an oval connected to a rectangle (respectively a diamond) by means of an undirected edge;
- membership of a complex attribute is represented by means of two ovals connected by means of a directed edge from the attribute to its element;
- a component of an entity is represented as an oval, connected to the rectangle representing the entity by means of an undirected edge, and connected to the rectangle representing the component by means of a directed edge. If the component is a list, a set or a bag, then multiple entities may be connected to the oval by means of directed edges.

In summary, a composition of subgraphs of an EER diagram may be used to describe configurations of entities, relationships and values in which one is interested. For instance, the condition that we are only interested in pairs of a customer and a cash card such that the customer has the cash card, is indicated by drawing a (diamond shaped) node labeled `has` with (appropriately labeled) edges to both other nodes (see Figure 4.1).

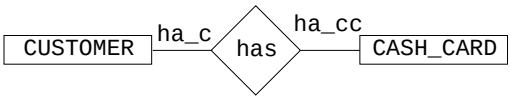


Figure 4.1: A sample partial GOQL/EER query

Reconsidering the graph pattern of Figure 1.1 in the Introduction (which in includes the one from Figure 4.1, expressing an interest in the names of customers and the passwords of cash-cards these customers have, one may note how such a pattern may be seen to correspond to the **from**- and **where**-clause of an SQL/EER query. In such a textual query, the **select**-clause is then used to indicate those elements of the query that actually have to be retrieved from the database. As EER diagrams offer no natural means of indicating such a "selection" on a query pattern, we have to introduce some additional notation. For the sake of clarity, we choose to draw selected nodes using *bold* lines, but any other way to distinguish one kind of nodes from the rest would do. As an example, suppose we want to retrieve the names of all customers registered in our database, together with the passwords of the cash cards they have. Then Figure 4.2 shows a possible expression of this query in SQL/EER, while Figure 4.3 shows a possible expression of this query in GOQL/EER.

select c.name, cc.password
from c **in** CUSTOMER, cc **in** CASH_CARD
where c has cc

Figure 4.2: Names of customers and passwords of the cash cards which they have (SQL/EER version)

Note how the graphical representation of this query is obtained by simply selecting two of the nodes in the graph pattern of Figure 1.1. Indeed, in the introduction to this chapter, we described how this graph could be read as a description of precisely the information we need for this query. The only thing that was still missing from the graph pattern of Figure 1.1 in order to match the given query precisely, is an indication of what information should be returned as a result of the query, i.e., the names and passwords. This is indicated in Figure 4.3 by drawing the `string`-labeled nodes representing this information in bold.

In the graphical query depicted in Figure 4.3, the underlying pattern equals a subgraph of an EER diagram. As soon as a query involves for instance multiple entities of the same type, a simple subgraph of an EER diagram is no longer sufficient for graphically expressing the query. Therefore, we allow the pattern underlying a graphical query to be constructed by "joining" multiple subgraphs of an EER diagram by identifying some of their nodes. Two nodes in different subgraphs may be identified if either

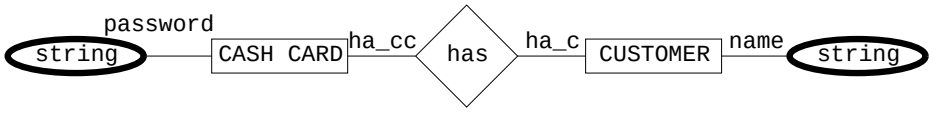


Figure 4.3: Names and passwords of customers and the cash cards which they have (GOQL/EER version)

1. they have the same label; or
2. they are both entities, and the type of one entity is a subtype of the other entity's type. In this case, the "identified" node is labeled with the subtype.

As an example, suppose we wish to retrieve the names of those pairs of banks that employ one and the same cashier. Then Figure 4.4 shows a possible expression of this query in SQL/EER, while Figure 4.5 shows a possible expression of this query in GOQL/EER. Note that since we do not require that ba1 differs from ba2, this query will at least return all identical pairs of banks.

select ba1.name, ba2.name
from ba1 **in** BANK, ba2 **in** BANK, ca **in** ba1.employs
where ca **in** ba2.employs

Figure 4.4: Names of banks employing one and the same cashier (SQL/EER version)

The graph of Figure 4.5 may be seen to consist of two copies of one and the same subgraph of Figure 2.1 (each consisting of four nodes, labeled respectively `string`, `BANK`, `list`, and `CASHIER`, linked by three edges) which have been *joined* on the node labeled `CASHIER` to express sharing of this entity. Since nothing prevents us from matching the two `BANK`-nodes to one and the same bank, this query, just like its textual equivalent, will at least return all identical pairs of banks.

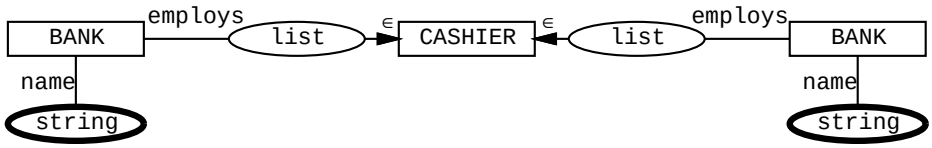


Figure 4.5: Names of banks employing one and the same cashier (GOQL/EER version)

The query depicted in both Figure 4.4 and Figure 4.5 also illustrates what constitutes in our view, a major advantage of graphical over textual expression of queries. Consider namely the SQL/EER statement depicted in Figure 4.6, which is semantically equivalent to that of Figure 4.4, and hence to the GOQL/EER query of Figure 4.5. While the SQL/EER query of Figure 4.4 uses the **in**-predicate to express the "crucial" condition in the query (namely the sharing of a cashier by two banks), the query in Figure 4.6 introduces an extra variable and uses the equality predicate. This possibility to express

one and the same concept (in this case, sharing) in a variety of ways, is often mentioned as a major cause of confusion for users of a language. Looking at Figure 4.5, the reader will notice that in a graph-oriented language such as GOQL/EER, the notion of sharing is expressed in a most straightforward manner, namely by one and the same node representing the shared item.

select ba1.name, ba2.name
from ba1 **in** BANK, ba2 **in** BANK, ca1 **in** ba1.employs,
ca2 **in** ba2.employs
where ca1 = ca2

Figure 4.6: Names of banks employing one and the same cashier (SQL/EER version II)

An additional example illustrates in an even more convincing way the advantage of graphical over textual formulation of certain aspects of a query, especially when it comes to interrelationships between the elements of interest. Suppose we wish to retrieve the names of those customers who hold an account, and also have a cash card which accesses that same account. In the SQL/EER formulation (Figure 4.7), three variables have to be introduced, each of which occurs twice in the **where**-clause. It requires a conscious effort while studying this query, to detect the interrelationships between the elements of interest.

select c.name
from c **in** CUSTOMER, cc **in** CASH_CARD, a **in** ACCOUNT
where c has cc **and** c holds a **and** cc accesses a

Figure 4.7: Names of customers who hold an account and have a cash card which accesses that account (SQL/EER version)

In contrast, in the GOQL/EER formulation of this query (Figure 4.8), these interrelationships may be directly derived from the picture, since all conditions imposed on a single element, are visualized in the element's direct vicinity. For instance, concentrating on the `CUSTOMER`-entity in the picture, one can immediately see that we wish to retrieve the name of `CUSTOMER`s participating in a `has`- and a `holds`-relationship. In contrast, to deduce this information from the SQL/EER version of this query, one has to search for *four* different occurrences of the variable `c`, once in the **select**-clause, once in the **from**-clause, and twice in the **where**-clause. Likewise, the conditions imposed on the `CASH CARD` respectively the `ACCOUNT` entity may be found by concentrating on the respective corresponding nodes.

As mentioned earlier on in this section, a query in GOQL/EER should consist basically of graphical symbols used in EER diagrams. By simply composing these symbols into patterns, already a large variety of conditions may be expressed graphically, as shown in the above examples. As motivated in Section 1.3.3, it is most certainly not our intention to introduce a query language in which *every* possible condition may be expressed graphically. Just for the sake of illustrating how even more kinds of conditions may be incorporated seemlessly into our language, we now add some minor features to the language as discussed so far:

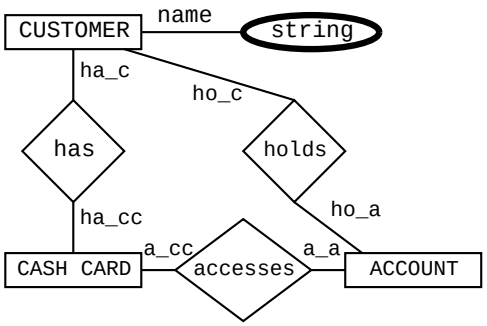


Figure 4.8: Names of customers who hold an account and have a cash card which accesses that account (GOQL/EER version)

- the condition that an attribute should have a particular value is indicated by writing this value next to the node representing the attribute;
- the condition that an element should occur at a specific place in a list is indicated by writing the index in square brackets next to the arrow linking the list to the element.

As an example, suppose we wish to know the name of the second bank of the "General Banking" consortium. Figure 4.9 shows an expression of this query in SQL/EER.

select b.name
from b **in** BANK, c **in** CONSORTIUM
where c.consists_of[2] = b **and** c.name = "General Banking"

Figure 4.9: The second bank of the General Banking consortium (SQL/EER version)

In the graphical formulation of this query (see Figure 4.10), the required consortium name is added to the leftmost `string`-node, while the required index is added to the $\in$-labeled edge.

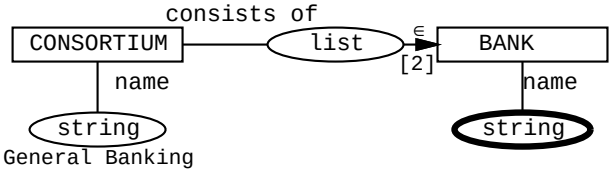


Figure 4.10: The second bank of the General Banking consortium (GOQL/EER version)

With the language introduced so far, we are only able to graphically express queries corresponding to "flat" SQL/EER queries, i.e. queries not involving subqueries. However, reconsidering the semantics of SQL/EER (sub)queries, it appears that EER diagrams offer a natural representation for subqueries ! Indeed, since subqueries return a bag (of e.g. entities or values), we may as well use the

graphical convention used in EER-diagrams for depicting a bag, namely an oval with `bag` inscribed in it. The graphical equivalent of the subquery is put *inside* the oval, indicating the fact that this subquery defines the bag. Furthermore, there are basically two ways of connecting such a "subquerybag" to the remainder of the graphical query:

1. Nodes of the graph pattern underlying a (sub)query may be identified (according to the "identification-rules" given above) with nodes of the graph pattern underlying any (direct or indirect) subquery, analogous to the fact that variables declared in a certain SQL/EER query may be used in subqueries.

2. A node may be linked to a subquerybag by means of a directed $\in$-labeled edge from the subquerybag to the node, indicating that the node corresponds to a variable ranging over the result of the subquery. In this case, only one node may be selected in the graph pattern underlying the superquery, and the type of this node should either be identical to or, in case of an entity type, it should be a supertype of that of the target of the $\in$-labeled edge.

Let us have a look at how these ideas are used in some example queries. Figure 4.11 presents a textual and Figure 4.12 a graphical version of the query which retrieves, for each address of a bank recorded in the database, the bag of names of banks that have this address as their location.

```
select ad, ( select  b.name
             from    b in BANK
             where b.location = ad )
from  ad in ( select ba.location
              from  ba in BANK )
```

Figure 4.11: For each address of a bank recorded in the database, the bag of names of banks that have this address as their location (SQL/EER version)

Although Figure 4.12 at first glance does not look much like an EER diagram, a closer look reveals that it is indeed still composed of nothing but graphical primitives also present in EER diagrams. The large oval in the bottom right corner of the picture corresponds to the subquery in the **from**-clause of the SQL/EER query. Indeed, the query depicted inside this oval selects the `location`-attribute (indicated with the bold `address`-oval) of every bank in the database. The $\in$-labeled directed edge starting at the border of this large oval denotes the fact that the address depicted in the top left corner of the picture (which, as indicated superficially in the picture, corresponds to the variable `ad` in the SQL/EER-query) ranges over the result of this subquery.

The large oval in the top right corner of the picture corresponds to the subquery in the **select**-clause of the SQL/EER-query. The fact that the node labeled `string` is drawn in bold, indicates that this subquery selects a bag of names. The `location`-labeled edge connecting the node labeled `BANK` (which corresponds to the variable `b` in the SQL/EER query) to the address depicted in the top left corner of the picture, expresses precisely the condition "`b.location = ad`" in the SQL/EER-query.

In the outermost SQL/EER query, the address ad as well as the bag of names is selected. Likewise, the `address`-node, corresponding to the variable `ad` as well as the `bag`-node in the top right

corner, corresponding to this bag of names, are selected in the GOQL/EER query, which is indicated by drawing them in bold. Remember that the fact that the `string`-node in the top right corner as well as the `address`-node in the bottom right corner are also drawn in bold, indicates that these nodes are selected in the *subqueries* and not in the outer query (which is impossible anyway, since they are not within the scope of the outer query).

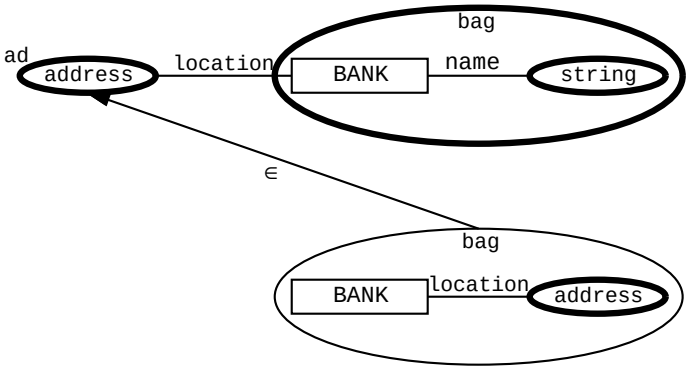


Figure 4.12: For each address of a bank recorded in the database, the names of banks located at this address (GOQL/EER version)

For clarity, the table depicted in Figure 4.13 visualizes once more the correspondence between elements of the textual and graphical representation of the above query.

A final example query illustrates nesting of subqueries to more than one level. The query depicted textually in Figure 4.14 and graphically in Figure 4.15 returns the name of each consortium, together with a bag of pairs, one for each bank the consortium consists of. Each such pair consists of the bank's name together with the bag of names of cashiers the bank employs.

The GOQL/EER version of this query is structured as follows:

- In the outermost query, the `name` of the `CONSORTIUM` is selected, as well as the bag depicted by means of the largest oval in the picture.
- In this bag, the `BANK` ranges over the `consists_of` attribute of the `CONSORTIUM`. The `name` of the `BANK` is selected, as well as the bag depicted by means of the second largest oval in the picture.
- In that bag, the `CASHIER` ranges over the `employs` attribute of the `BANK`. Only the `name` of the `CASHIER` is selected in this subquery.

4.2 Formal Specification of GOQL/EER

In Section 4.1, we introduced the ideas and concepts behind GOQL/EER by means of examples. In this section, we formally define syntax and semantics of GOQL/EER in graph-theoretical terms. Our

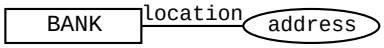
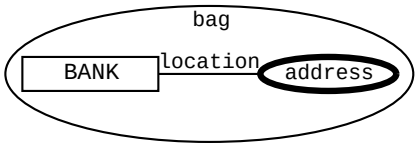
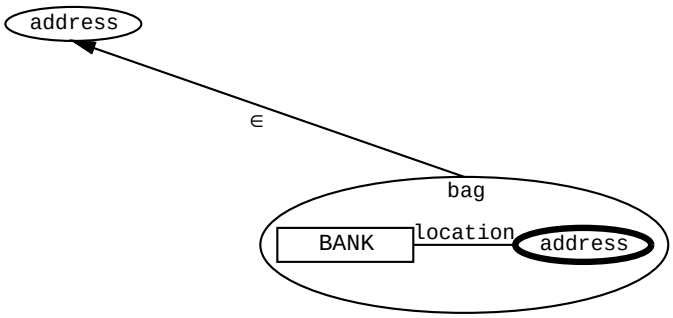
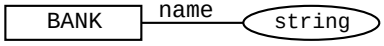
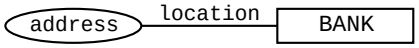
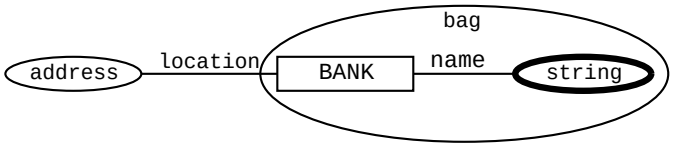
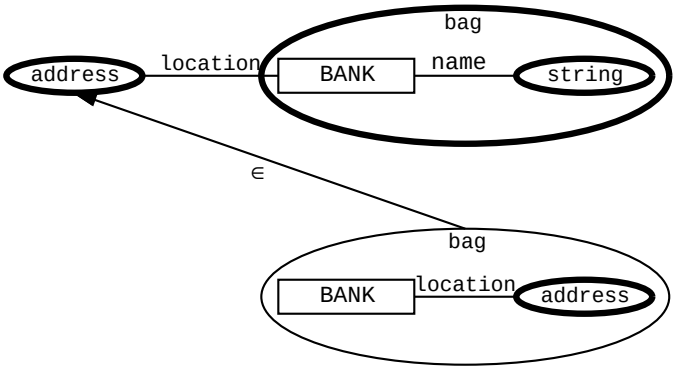
SQL/EER fragments	GOQL/EER fragments
ba ba in BANK	
ba.location	
select ba.location from ba in BANK	
ad in (select ba.location from ba in BANK)	
b b in BANK	
b.name	
b.location = ad	
select b.name from b in BANK where b.location = ad	
select ad, (select b.name from b in BANK where b.location = ad) from ad in (select ba.location from ba in BANK)	

Figure 4.13: Correspondence between fragments of graphical and textual queries

```
select co.name, ( select b.name, ( select ca.name
                                   from  ca in b.employs )
                  from  b in co.consists_of )
from  co in CONSORTIUM
```

Figure 4.14: The name of each consortium, together with the bag consisting of a pair for each bank the consortium consists of, with the bank's name together with the bag of names of cashiers the bank employs (SQL/EER version)

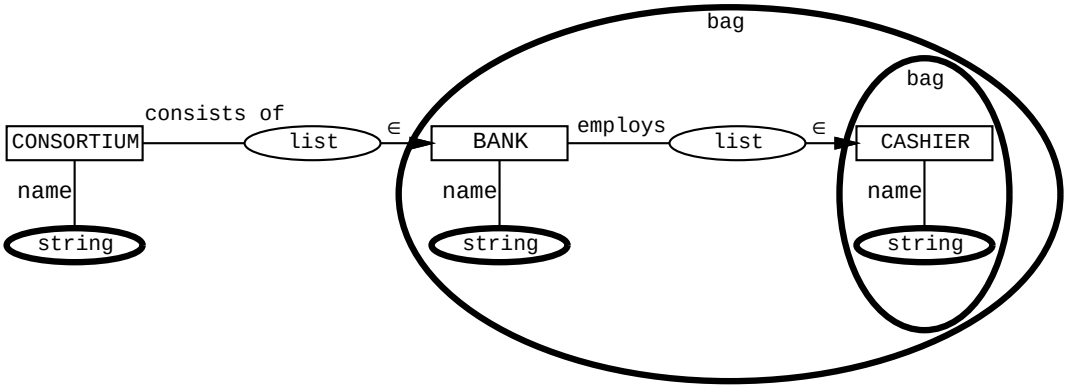


Figure 4.15: The name of each consortium, together with the bag consisting of a pair for each bank the consortium consists of, with the bank's name together with the bag of names of cashiers the bank employs (GOQL/EER version)

need for an expressive graph model motivates our choice of the graph grammar formalism PROGRES, an informal overview of which was presented in Chapter 3.

The PROGRES specification is obtained in a three step process. In a first step (see Section 4.2.1) we show how GOQL/EER queries may be represented by means of *labeled, attributed graphs*. In a second step (see Section 4.2.2), the class of all graphs that correspond to GOQL/EER queries is defined by means of a PROGRES specification. In other words, this specification captures the *syntactic* structure of GOQL/EER. The specification obtained in the second step is then extended in a third step (see Section 4.2.3) to define also the *semantics* of GOQL/EER. This is done by introducing additional node attributes and attribute derivation rules. These rules translate the GOQL/EER query into an SQL/EER query, defining the semantics of the graph-oriented query.

Looking from a different perspective, the PROGRES specification for GOQL/EER may be seen to consist of the following four parts (cf. Figure 4.16):

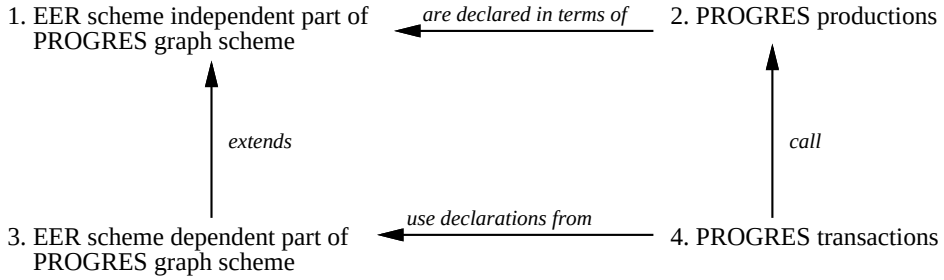


Figure 4.16: Structure of the PROGRES specification for GOQL/EER

1. an EER scheme *independent* part of a graph scheme, which for instance formalizes the fact that there are nodes for representing entities and nodes for representing values;

2. a set of productions, specified in terms of the EER scheme independent part of the graph scheme. For instance, some production formalizes the fact that given an entity, a value of a given type may be linked to it, indicating that the value represents one of the entities attributes.

Together, these two parts actually define the language of all syntactically correct GOQL/EER-graphs. A graph is part of this language if it may be obtained by applying any correct (in the PROGRES sense) sequence of productions to an initial empty graph. What exactly constitutes a "correct" sequence of productions is the topic of Section 4.2.4.

The fact that PROGRES is an *operational* specification language, plus the presence of an interpreter in the PROGRES environment inspired us to incorporate a scheme *dependent* part in the PROGRES specification, mainly for the purpose of testing the specification. This scheme dependent part of the specification consists of

3. an EER scheme dependent part of the graph scheme, extending the EER scheme independent part with additional node classes and node types. Among others, this part of the graph scheme formalizes the fact that there are entities of type `BANK`, and values of type `address`.

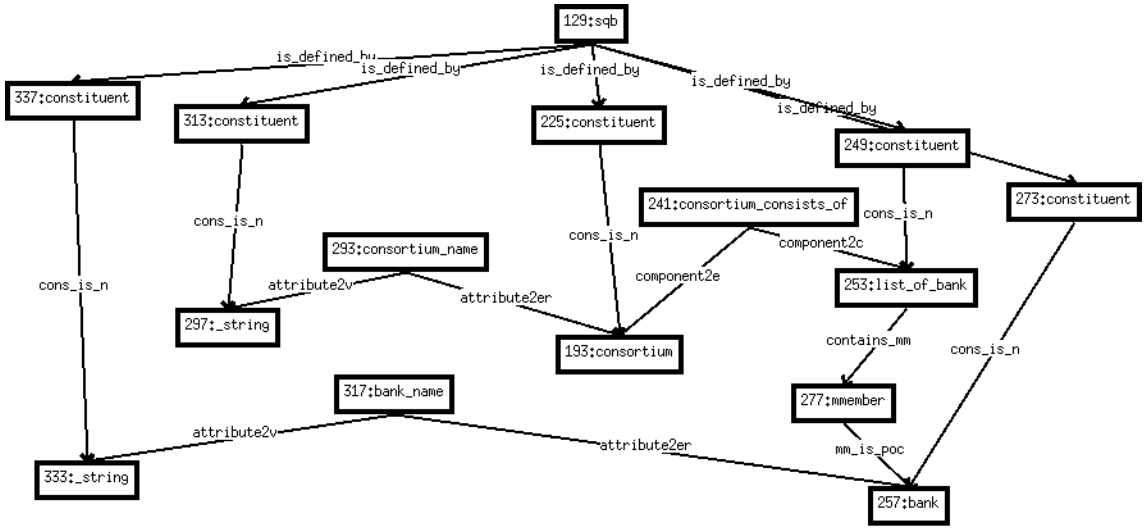


Figure 4.17: Graph representation of the graphical query of Figure 4.10

4. some transactions, i.e., sequences of production calls. Each transaction creates the formal graph representation of a particular GOQL/EER query over the EER scheme of Figure 2.1.

4.2.1 A Graph Model for GOQL/EER

Formalization of GOQL/EER is based on the representation of graphical queries as labeled, attributed graphs. In informal terms, the figures of Section 4.1 correspond to what a user would see on the screen of a tool supporting GOQL/EER, while the graphs correspond to the internal representation of such queries. These graphs serve a double purpose:

- On one hand, such a graph represents the (abstract) syntactical structure of a query. Node labels in the graph correspond either to EER scheme elements (such as entity types, relationship types or roles) or to "query elements" (such as queries or subqueries). Node attributes[1] are used for the storage of non-structural information which is part of the query, such as concrete atomic values (used in predicates such as "`b.name = 'General Banking'`") and list indices (used in terms such as "`co.consists_of[2]`").

- On the other hand, such a graph also incorporates the semantics of the corresponding query: extra attributes are used to store (SQL-)declarations, formulas and terms corresponding to nodes. A unique node in the graph corresponds to the query itself, and has associated to it (in one of its attributes) a complete SQL/EER-query, whose semantics is said to define the semantics of the graphical query.

[1] Note once more that the word "attribute" is used in two different meanings: one in the context of graphs, and the other in the context of the EER-model.

As an example, consider Figure 4.17, which shows the graph corresponding to the graphical query of Figure 4.10. Some of the attributes of nodes in the graph of Figure 4.17 are shown in Table 4.1. On one hand, the attributes Output (indicating whether a node has been selected for output), Value (containing the actual value of a node corresponding to an atomic value), and Index (containing a position in a list) are part of the representation of the query's syntax in the graph. On the other hand, the attributes SFW-Term, Term Declaration, and Formula (all containing SQL/EER-statements) are part of the representation of the query's semantics in the graph.

Id.	Label	Att.Name	Attribute Value
129	sqb	SFW-Term	(select co.consortium_consists_of[2].bank_name from co in consortium where co.consortium_name = 'General Banking')
249	constituent	Term	co.consortium_consists_of
273	constituent	Term	co.consortium_consists_of[2]
337	constituent	Term Output	co.consortium_consists_of[2].bank_name TRUE
313	constituent	Term Output Formula	co.consortium_name FALSE co.consortium_name = 'General Banking'
297	_string	Value	'General Banking'
225	constituent	Declaration Term	co in consortium co
277	mmember	Index	2

Table 4.1: Attributes of some nodes of the graph in Figure 4.17

As already mentioned, a major motivation for our choice to work with the PROGRES formalism was that the specification of GOQL/EER could consequently be made using the PROGRES-system. Among others, this allowed us to execute the specification[2] using the systems integrated interpreter. The graph shown in Figure 4.17 is the result of such an interpretation and was generated using the PROGRES system.

Within each node of the graph, its (unique) node identifier and its node type are depicted. This graph clearly illustrates the different aspects of the correspondence between a GOQL/EER query, depicted using the conventions introduced in Section 4.1, and its representation as a formal attributed labeled graph:

- The nodes in the formal graph representation (and their types) are obtained from the GOQL/EER query as follows:
  - All EER scheme dependent names occurring in the graphical query occur as node types in the graph representation, either

[2]The precise meaning of "executing a PROGRES specification" is elaborated on in Section 4.2.4.

  - directly, as is the case with the `bank` and `consortium` nodes in Figure 4.17;
  - prefixed with an underscore in case the name coincides with a reserved word of the PROGRES language, as is the case with the `_string` node;
  - prefixed or affixed with "related" names to ensure unique naming of PROGRES types. This is illustrated with the `name`-attributes of the entity types `CONSORTIUM` and `BANK`. Since these attributes have to be mapped onto different and hence uniquely named PROGRES types, they have to be prefixed with the name of the entity (or relationship) type to which they apply, resulting in the PROGRES types `bank_name` respectively `consortium_name`.
    Likewise, the `consists_of` attribute of the consortium is represented with a node of type `consortium_consists_of`.
    Because of the same uniqueness requirement, the `list`-labeled node in Figure 4.10 is represented in the formal graph with the `list_of_bank`-node, since this node indeed represents a list of banks.

- The single EER scheme independent name which may occur in a GOQL/EER query is "$\in$". Remember that $\in$-labeled edges are used to link a list, set or bag to its elements. For representing such edges in a formal graph, a distinction is made between the following two cases:
  1. if the source of the edge is a set, the $\in$-labeled edge is simply represented with an edge labeled `cont` (rather than $\in$, since PROGRES does not allow special symbols as type names).
  2. if the source of the edge is a list or bag, it means that the target of the edge (which is a value or entity) may occur multiple times in this list or bag, and hence may be connected to this list or bag through multiple edges. In PROGRES however, it is impossible to have multiple edges with the same label between two given nodes. Hence an additional type of nodes is introduced, namely `mmember`, which stands for multi-`member`. This is illustrated with `bank` being an `mmember` of `list_of_bank`. The introduction of these `mmember`-nodes has the additional advantages that they offer a convenient place to store index-values, in case the query specifies at what particular place in the list or bag the element should occur.
- The query and its subqueries are represented by nodes of type `sqb`, which stands for subquerybag. The nodes "constituting" the query are linked to the node corresponding to this query by means of a sequence of an `is_defined_by`-edge, a `constituent`-node and a `cons_is_n`-edge. The reason for this auxiliary `constituent`-node is explained further on in this section.

- Besides edges of types `cont`, `is_defined_by` and `cons_is_n` (whose purposed is explained above), eight more types of edges are used to link the various nodes of a formal graph representation together:
  - A node of type `mmember` is linked to a list or bag by means of an incoming `contains_mm`-edge, and to the element of the list or bag by means of an outgoing `mm_is_poc`-edge (where `poc` stands for "part of complex value".

  - A node corresponding to a role is linked to a relationship by means of a `role2r`-edge and to an entity by means of a `role2e`-edge.
  - A node corresponding to an attribute is linked to a relationship or entity by means of an `attribute2er`-edge and to a value by means of an `attribute2v`-edge
  - A node corresponding to a component is linked to the owning entity by means of a `component2e`-edge and to the component entity by means of a `component2c`-edge.

- In Table 4.1, two major categories of attributes may be distinguished, which correspond to the double purpose of the formal graph representation as outlined above:

  1. The attributes `Value`, `Index` and `Output` contain non-structural information which is also present in the GOQL/EER query itself. The use of the `Value` and `Index` attributes has been explained previously, while the boolean `Output` attribute is true for those nodes that correspond to an element of the GOQL/EER query drawn in bold.
  2. The attributes `SFW-Term`, `Term`, `Declaration` and `Formula` are used in defining the semantics of the query. In our example, Table 4.1 formalizes the correspondence between nodes of the graph in Figure 4.17 (and hence graphical elements of the GOQL/EER query depicted in Figure 4.10) with elements of the query's textual equivalent depicted in Figure 4.9. For instance, the node in Figure 4.17 with identifier 253 (labeled `list_of_bank`) on one hand corresponds to the `list`-labeled node in the graphical query of Figure 4.9. On the other hand, the `Term`-attribute of the `constituent`-node through which it is linked to the `sqb`-labeled node (i.e., the node with identifier 249) shows that it also corresponds to the term `co.consists_of` in the textual query of Figure 4.9.[3]

In the following two sections, we discuss how the graph model outlined above, is formally captured in a PROGRES specification. The full specification may be found in Appendix C.

4.2.2 The Syntax of GOQL/EER

We now define the syntax of GOQL/EER by means of a PROGRES specification. As outlined in the introduction to this section, such a specification consists of a graph scheme and a set of productions. We first discuss the graph scheme in detail.

Graph Scheme of the GOQL/EER specification

The node class `QUERY_ELEM` is the superclass of those node classes that actually constitute an element of a query, such as an entity, relationship or value. The node class `PART_OF_COMPLEX` is the common superclass of `ENTITY` and `ATOMIC_VALUE`. From Definition 2.5 of EER schemes, we know that it are precisely entities and atomic values that may be part of sets, lists or bags, respectively in the case of components and (complex) attributes, hence the name of the node class.

[3] The need for the prefix `consortium_` in this term is explained above.

```
node class QUERY_ELEM end;
node class PART_OF_COMPLEX is a QUERY_ELEM end;
node class ENT_REL is a QUERY_ELEM end;
node class ENTITY is a ENT_REL, PART_OF_COMPLEX end;
node class RELSHIP is a ENT_REL end;
node class ROLE
 meta rel : type in RELSHIP [1:1];
      ent : type in ENTITY [1:n];
end;
edge type role2e : ROLE -> ENTITY [1:1];
edge type role2r : ROLE -> RELSHIP [1:1];
```

Similarly, the node class ENT_REL is the common superclass of ENTITY and RELSHIP, which share the property of being able to have attributes. Consequently, the node class ENTITY is a direct subclass of both PART_OF_COMPLEX and ENT_REL. In Figure 4.17, the nodes labeled bank and consortium are nodes of this class.

The following four declarations in the graph scheme concern roles and relationships. A role2r-edge links a ROLE to the (unique, hence the "[1:1]" cardinality constraint) RELSHIP to which it belongs, while a role2e-edge links a ROLE to the (unique) ENTITY that plays the role.

To understand why roles (as well as attributes and components, see further on) are modeled by means of nodes rather than edges, consider for instance the production that allows the addition of a new role to a given (existing) relationship (for instance, a relationship of type owned by).[4] This production expects the role-"type" (for instance, teller) as input. However, in PROGRES it is impossible to pass an edge *type* as input to a production (since there is no such thing as an "edge class hierarchy"). Hence one has to model roles by means of nodes, such that the role type can be passed to a production by means of the corresponding node type.

Unfortunately, the modeling of roles by means of nodes rather than edges, introduces an additional problem. Suppose we would have modeled the role teller by means of an edge type with owned by as source and ATM as target. Then it would be a type error to try and add a teller role to for instance a concerns relationship. However, since the edge types role2r and role2e are now independent of the EER scheme, this error may no longer be recognized by the type system, so we have to introduce explicit checks for this situation. This is done using the meta attributes ent and rel declared for node class ROLE.

As explained in Section 3, a meta attribute applies to a *node type*, and may have (an)other node type(s) as value. For any node type of class ROLE, the meta attribute rel is supposed to refer to the node type corresponding to the relationship type to which the role belongs. The meta attribute ent is supposed to refer to the *set* of all node types corresponding to the entity types that can play the given role in the given relationship. To this end, the declaration of ent is qualified with "[1:n]" which makes it a set-valued attribute.

In general, the need for meta attributes in this and other parts of the specification, comes from the fact that not all "type-related" characteristics of GOQL/EER (such as the relation between relationship types and entity types playing their roles) may be modeled in and enforced by the PROGRES type system.

[4] See the production Add_Role further on in this section.

The precise usage of these (and other) meta attributes is explained in more detail when the relevant productions are discussed.

```
node class VALUE is a QUERY_ELEM end;
node class ATOMIC_VALUE is a VALUE, PART_OF_COMPLEX
 intrinsic Value : string;
end;
node class COMPLEX_VALUE is a VALUE
 derived Elem_Type : type in QUERY_ELEM;
end;
node class SET_VALUE is a COMPLEX_VALUE
 intrinsic Singleton : boolean := false;
end;
edge type cont : SET_VALUE -> PART_OF_COMPLEX;
node class MVALUE is a COMPLEX_VALUE end;
node class BAG_VALUE is a MVALUE end;
node class LIST_VALUE is a MVALUE end;
node class MMEMBER
 intrinsic Index : integer := 0;
end;
node type mmember : MMEMBER end;
edge type contains_mm : MVALUE -> MMEMBER;
edge type mm_is_poc : MMEMBER -> PART_OF_COMPLEX [1:1];
```

The class VALUE is a direct subclass of QUERY_ELEM, and has the classes ATOMIC_VALUE and COMPLEX_VALUE as direct descendants. Nodes of class ATOMIC VALUE have an attribute Value in which the "actual value" is stored. For simplicity, we assume all values are stored as texts (which is the PROGRES type for "lengthy" strings). In Figure 4.17, the two nodes labeled _string are nodes of class ATOMIC VALUE.

SET_VALUE is one of COMPLEX_VALUEs direct subclasses. It's attribute Singleton is set to true in the case of a singleton component (such as the proper_account of a BANK). Edges of type cont link a set to its elements, which must be of class PART_OF_COMPLEX. As subclass of COMPLEX-_VALUE, SET_VALUE (as well as BAG_VALUE and LIST_VALUE) inherits the attribute Elem_Type, which serves a similar purpose as the attributes of class ROLE. Indeed, since edges of type cont are used both for linking a CASHIER_STATION to a set of CASHIER_STATIONs, as well as for linking an address to a set of addresses, the PROGRES type system cannot prevent the linking of an address to a set of CASHIER_STATIONs. Hence the need for the attribute Elem_Type, which may be used to check if the type of a complex value, and the type of a value to be added to it are compatible.

The other direct subclass of COMPLEX_VALUE is MVALUE, whose name stands for Multi-VALUE. This name is explained by its two subclasses, namely LIST_VALUE and BAG_VALUE: one and the same value or entity may be an element of a list or bag multiple times. As mentioned in Section 4.2.1, this fact motivates the introduction of auxiliary nodes, both for representing membership of multi-values and for storing index values. Their node class is called MMEMBER, and nodes of this class have an integer Index attribute. Edges of type contains_mm link a multi-value to its multi-members, while edges of type mm_is_poc link a multi-member to the element, the membership of which it represents. As this must be a unique element (of class PART_OF_COMPLEX, as in the case of sets) we

impose the cardinality constraint [1:1].

The usage of the classes MVALUE and MMEMBER and the edge types contains_mm and mm_is_poc is exemplified in Figure 4.17 with the bank as a member of the list_of_banks.

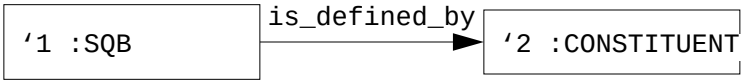
```
node class DERIVED_SQL end;
node class SQB is a BAG_VALUE, DERIVED_SQL
 redef derived Elem_Type =
  ((self.=OutCons=>:CONSTITUENT[1:1]).-cons_is_n->:QUERY_ELEM[1:1]).type;
end;
node type sqb : SQB end;
node class CONSTITUENT
 intrinsic Output : boolean := false;
end;
node type constituent : CONSTITUENT end;
edge type is_defined_by : SQB [1:1] -> CONSTITUENT;
edge type cons_is_n : CONSTITUENT -> QUERY_ELEM [1:1];
node class SQB_CONS is a CONSTITUENT, DERIVED_SQL end;
node type sqb_cons : SQB_CONS end;
```

Nodes of class CONSTITUENT (together with edges of type is_defined_by and cons_is_n) are used to link a subquerybag to node that constitute part of its definition. In Figure 4.17, one can see that five nodes constitute the query, namely the two _strings, the bank, the consortium and the list_of_banks.

The Output-attribute of a CONSTITUENT is set to true if the node reachable by means of the outgoing cons_is_n-edge is selected in the query corresponding to the node reachable by means of the outgoing is_defined_by edge (cf. the attributes of constituent 337 in Table 4.1). Since nodes may be constituents of more than one (sub)querybag (in case of subqueries), selection for output has to be indicated on the constituent-nodes, rather than on the query elements themselves.

The class SQB (which stands for SubQueryBag) is used to represent (sub)queries. Since a query returns a bag, SQB is declared a subclass of BAG_VALUE. The derivation rule for the attribute Elem_Type inherited from BAG_VALUE, uses the path OutCons. A path of type OutCons exists between a node of class SQB and any of its CONSTITUENTs whose Output-attribute is true.

```
path OutCons : SQB -> CONSTITUENT =
   ‘1 => ‘2 in
```




```
   condition ‘2.Output;
end;
```

Figure 4.18: Declaration of the path OutCons

There are two possibilities:

1. A unique constituent is selected for output. In that case, the constraint that the path expression `=OutCons=>` should return exactly one constituent (indicated with `CONSTITUENT [1:1]`) is satisfied. From Section 4.1, we know that it are exactly such subquerybags that may be used as a range for a variable with a type compatible to the unique selected constituent. The attribute `Elem_Type` can then be used to check the latter condition, as we will see when discussing the production which is used to let a variable range over a bag.

2. More than one constituent is selected. In that case, the constraint that `=OutCons=>` should return exactly one constituent is not satisfied, hence the evaluation of this derivation rule fails, and the attribute `Elem_Type` is undefined. Consequently the subquerybag cannot be used as a variable domain.

The fact that a subquery is itself a constituent of its superquery explains why `SQB` is declared a subclass of `QUERY_ELEM`. However, in this case, nodes of class `SQB_CONS` are used to link a subquery to its superquery, rather than nodes of class `CONSTITUENT`. The need for this special class is explained by the fact that SQL/EER-terms associated to such constituents, and to the `SQB`-nodes representing the queries themselves (see Section 4.2.3 on the semantics of GOQL/EER) are derived automatically, rather than computed in productions. This also explains the need for (and the name of) the class `DERIVED_SQL`, which is the common superclass of `SQB` and `SQB_CONS`.

In Section 4.2.3, it is shown how in an attribute of nodes of class `SQB`, information (that is, declarations, terms and formulas) is "collected" from its constituents, which is combined into an SQL/EER query.

```
node class ATTRIBUTE
 meta entrel : type in ENT_REL [1:n];
      val : type in VALUE [1:1];
end;
edge type attribute2er : ATTRIBUTE -> ENT_REL [1:1];
edge type attribute2v : ATTRIBUTE -> VALUE [1:1];
node class COMPONENT
 meta cent : type in ENTITY [1:n];
      comp : type in COMPLEX_VALUE [1:1];
end;
edge type component2e : COMPONENT -> ENTITY [1:1];
edge type component2c : COMPONENT -> COMPLEX_VALUE [1:1];
```

The usage of node class `ATTRIBUTE` as well as of the edge types `attribute2er` and `attribute2v` is exemplified in Figure 4.17 with the names of respectively the `bank` and the `consortium`. The meta attributes of `ATTRIBUTE` serve an identical purpose as those of `ROLE`.

The node class `COMPONENT` and the two edge types `component2e` and `component2c` are used for modeling components, in the same way as explained above for roles. Their usage is exemplified in Figure 4.17 with the `consortium` consisting of a `list_of_bank`s.

Besides the node classes and edge types discussed above, the graph scheme also contains four node types `sqb`, `sqb_cons`, `mmember` and `constituent` of respectively the classes `SQB`, `SQB_CONS`, `MMEMBER` and `CONSTITUENT`. The need for these node types is explained when we discuss the productions of the specification, which is what we are about to do right now.

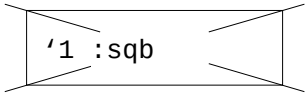
Productions of the GOQL/EER specification

The collection of declarations in the graph scheme described above, is in itself insufficient to completely describe the syntax of graphical queries. Indeed, the part of the specification given so far, for instance, does not yet *use* the various meta-attributes declared in the graph scheme to enforce type correctness. Hence in general, we still need to specify in more detail which configurations of nodes and edges are allowed. As PROGRES is an *operational* specification language, this is done by means of a set of productions. GOQL/EER graphs are those graphs that are obtained as a result of the application of any correct sequence of these productions to an initial empty graph.

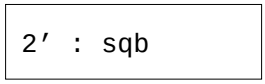
The complete specification includes fifteen productions, most of which are discussed below.

First, reconsider Figure 4.17, depicting the graph representation of the graphical query of Figure 4.10. Construction of this (and of any other) graph representing a GOQL/EER query starts with the creation of an `sqb`-labeled node, representing the query. This is done by means of the production `Add_first_SQB`.

```
production Add_first_SQB ( out NewS : SQB ) =
```




```
::=
```




```
   return NewS := 2';
end;
```

The left-hand side of this production consists of a single crossed node of type `sqb`. A cross over a node or edge indicates negation. In this case, the left-hand side of this production matches the graph to which the production is applied, if it contains no nodes of type `sqb`. The fact that the specification contains no productions that allow the deletion of nodes of this type, implies that this production *may be applied at most once* to any graph. If not, it would be possible to create a query featuring various unrelated (sub)queries.

Together with the fact that the left-hand side of all other productions in the specification contains at least one "non-negated" node, this implies that this production *must be applied exactly once* to any graph, namely at the start of any sequence of production applications (i.e., when the graph is still empty).

All other nodes of type `sqb` must be added using the production `Add_SQB`. This production takes an existing `sqb`-node `S` as input, and links a new `sqb`-node to it by means of an `sqb_cons`-node. Both new nodes are returned as output parameters.

```
production Add_SQB ( S : SQB ; out NewS : SQB ; out SC : SQB_CONS ) =

        ‘1 =S

   ::=

        1’ =‘1
                  is_defined_by
        3’ :sqb_cons        cons_is_n        2’ :sqb

   return NewS := 2’;
          SC := 3’;
end;
```

The fact that `Add_first_SQB` and `Add_SQB` are the only two productions that allow the addition of `sqb`-nodes, ensures that these nodes are always arranged in a tree-structure, which corresponds precisely to the allowed relations between super- and subqueries in SQL/EER. This is illustrated in Figure 4.19, which shows the tree of `sqb`-nodes in the formal graph representation of some GOQL/EER query which has two subqueries, one of which has in turn a single subquery.

Looking back at the graph of Figure 4.17, we could now add the required entities to the query, i.e., the `bank` and the `consortium`. This is done using the production `Add_ER`, which adds a new entity (or relationship, hence the use of the class `ENT_REL`) to a given (sub)querybag by means of a constituent. Both new nodes are returned as output parameters, so they can be used by other productions. For instance, selection for output of the newly created entity must be indicated on its corresponding constituent, hence the need for returning also the newly created constituent.

The following observation can be made on the graph-part of this production. From Chapter 3, we know that nodes which are created by a production (i.e., nodes that are in the productions right-hand side but not in its left-hand side) must be labeled with a node type. In general, there are two possibilities for labeling a node in the the right-hand side of a production with a node type, both of which are illustrated in `Add_ER`:

1. A node type may be provided as input to the production. This is the case for the node with identifier 3’ in `Add_ER`, which is labeled with the input type `ERtype`. On calling this production, a concrete EER scheme dependent type (such as `bank` or `consortium`) must be provided.

2. A concrete node type from the graph scheme may be used. This is the case for the node with

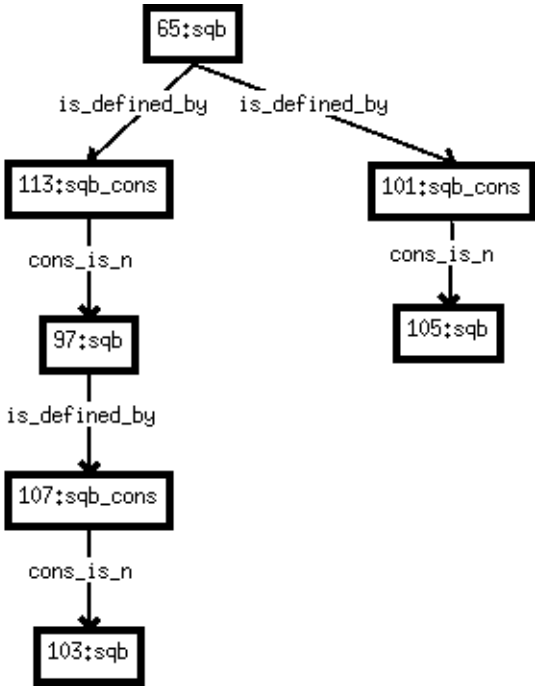
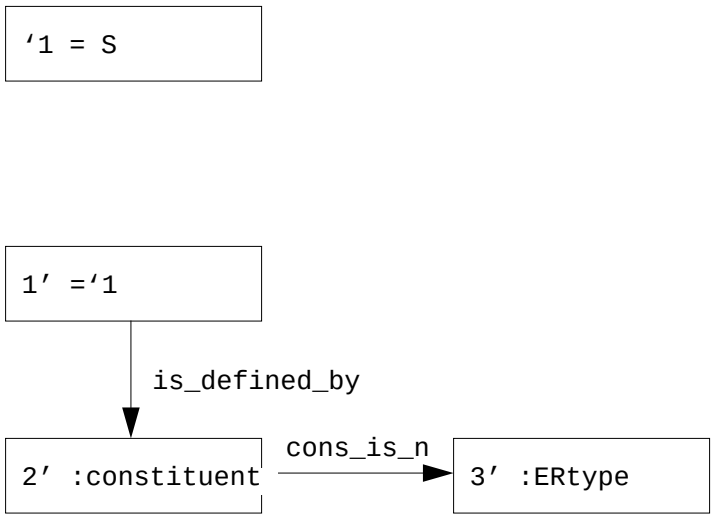


Figure 4.19: Arrangement of `sqb`-nodes in the formal graph representation of a GOQL/EER query

```
production Add_ER
   ( S : SQB ; ERtype : type in ENT_REL ;
      out E : ENT_REL ; out C : CONSTITUENT ) =
```

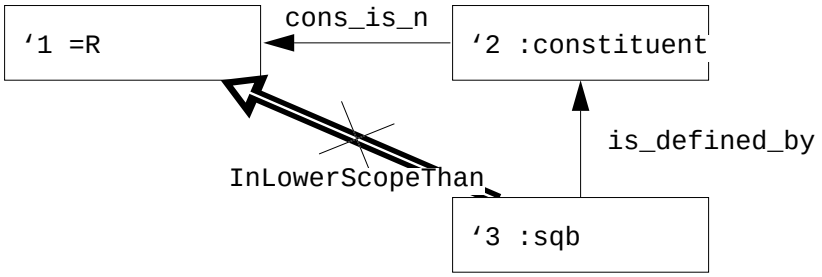



```
   return E := 3’;
          C := 2’;
end;
```

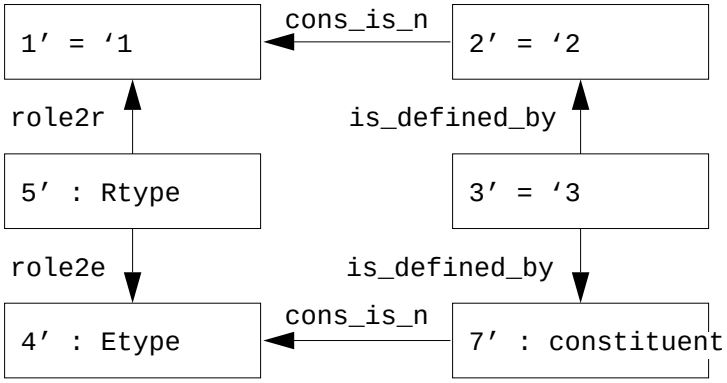
identifier 2' in Add_ER, labeled with the type constituent, which is not EER scheme dependent. This explains why this type had to be added to the graph scheme presented above. The same holds for the types mmember, sqb and sqb_cons.

Given a relationship (created using Add_ER), an entity may be added to it as a role, using the production Add_Role. This production takes the following input:

```
production Add_Role
   ( R : RELSHIP ; Etype : type in ENTITY ; Rtype : type in ROLE ;
      out E : ENTITY ; out C : CONSTITUENT ) =
```




```
   ::=
```




```
   condition Rtype.rel = R.type;
             Etype in Rtype.ent;
   return E := 4';
          C := 7';
end;
```

- The only node that is passed to the production is the relationship R to which a role must be added. The (sub)query to which the newly created entity must be linked as a constituent is not passed as input (as was the case with Add_ER) but may be computed in the left-hand side of the production itself. In general, a relationship (or entity or value) may be a constituent of any non-zero number of (sub)queries. This is the case when relationships declared in different subqueries, have

been identified to express an equality condition in the query at hand. However, among these (sub)queries, one must be a (direct or indirect) superquery of all the others (except itself). If this were not the case, this would mean that the relationship is used in two incomparable scopes, which is a clear violation of scoping rules. The entity that has to be created by this production, *must* be linked as a constituent to this "highest" subquery, since this entity must be known in all scopes in which the relationship in which it plays a role is known.

In the left-hand side of `Add_Role`, the `sqb`-node corresponding to this "highest" (sub)query of which R is a constituent, is determined as the single `sqb`-node of which R is a constituent, but for which there is no `InLowerScopeThan`-path leading to R.

```
path InLowerScopeThan : SQB -> QUERY_ELEM =
   (<-cons_is_n- & instance of SQB_CONS & <-is_defined_by-) + &
    -is_defined_by-> & -cons_is_n->
end;
```

Indeed, since starting from the given `sqb`-node, this path leads to an `sqb`-node which is a superquery of the given node (by means of the expression `(<-cons_is_n- & instance of SQB_CONS & <-is_defined_by-) +`) and of which R is a constituent (obtained by the expression `-is_defined_by-> & -cons_is_n->`). Clearly, if such a path would exist, then the considered `sqb`-node did not correspond to this "highest" query. This is exemplified in Figure 4.20, in which the situation is depicted where R is a constituent of both a query and a direct subquery. It may be verified that there is an `InLowerScopeThan`-path from the bottom `sqb`-node to R, but not from the top `sqb`-node to R.

- The remaining two input parameters to the production are the node types of respectively the entity and the role to be added to R. In the condition-clause, it is checked whether these types and R are all "mutually compatible". First it is checked whether the type of R (obtained using the built-in function `type`) precisely matches the meta attribute `rel` of the given role-type (cf. the discussion of roles in Section 4.2.1). Next it is checked if the given entity type is an element of the meta attribute `ent` of the given role-type. Remember that the `ent` attribute is indeed set-valued, and contains precisely all types, nodes of which may play the considered role in the given relationship R.

  In our running example, this allows for instance the participation of an `ATM` in an `entered on` relationship, since `ATM` is a subclass of `ENTRY_STATION`, which in the scheme of our running example is declared as the entity type corresponding to the `es_eo` role of `entered on`.

Construction of the graph of Figure 4.17 could be continued with the addition of the `name` attributes (in the EER sense of the word) to the `bank` and `consortium` entities. This is done using the production `Add_Attribute`, which works completely analogous to `Add_Role`.

Likewise, the `consists_of` component of the `consortium` is added to the graph of Figure 4.17 using the production `Add_Component`, which in turn works completely analogous to `Add_Attribute`, and is therefore omitted here (but may be found in Appendix C).

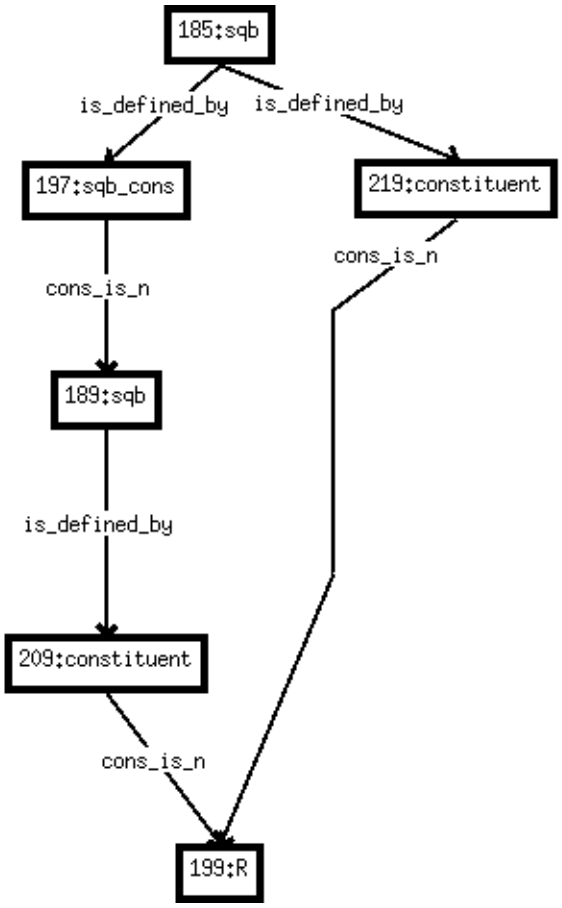
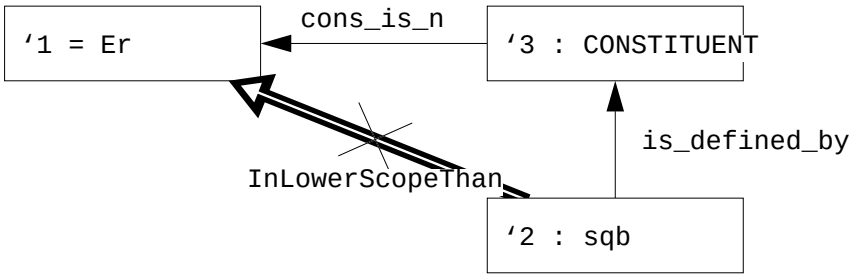
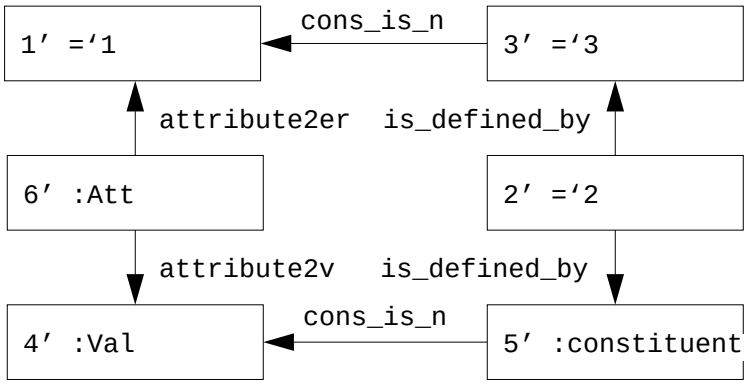


Figure 4.20: A relationship in different scopes

```
production Add_Attribute
   ( Er : ENT_REL ; Att : type in ATTRIBUTE ; Val : type in VALUE ;
      out v : VALUE ; out C : CONSTITUENT )
=
```



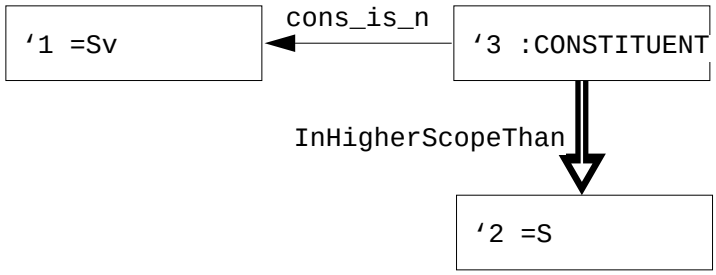

```
   ::=
```



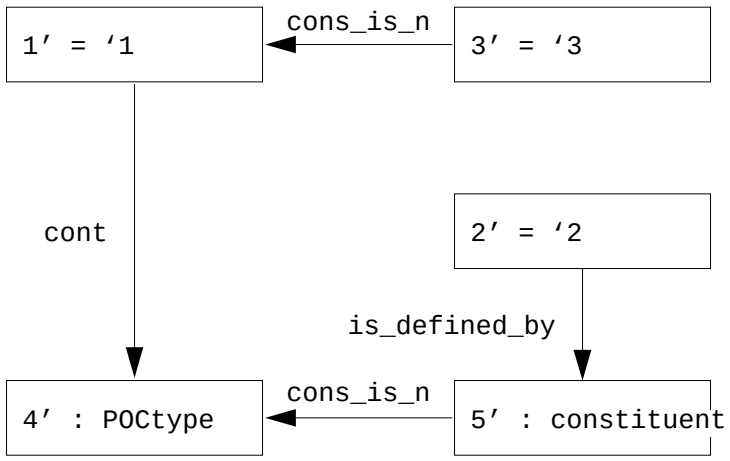

```
   condition Er.type in Att.entrel;
             Att.val = Val;
   return v := 4’;
          C := 5’;
end;
```

The following three productions of the specification allow the addition of elements to complex values such as sets or bags. First, the production Add_to_Set allows the addition of an element (that is, a node of class PART_OF_COMPLEX) to a set.

```
production Add_to_Set
   ( Sv : SET_VALUE ; S : SQB ; POCtype : type in PART_OF_COMPLEX ;
      VarName : string ; out POC : PART_OF_COMPLEX ; out C : CONSTITUENT ) =
```




```
   ::=
```




```
   condition POCtype in Sv.Elem_Type;
             ‘1.Singleton => (card ( ‘1.-cont-> ) = 0);
   return POC := 4’;
          C := 5’;
end;
```

In this production, the (sub)query to which the newly created element must be linked as a constituent cannot be computed by the production itself, as was the case with Add_Role, but must be passed to the production as input. The reason for this lies in the following basic difference between on one hand the act of adding an entity as role to a relationship, or a value as attribute to an entity, and on the other hand the act of letting a value or entity range over a set, list or bag:

- The connection between a relationship and the entity that plays one of its roles (or between an

entity and a value that is one of its attributes,…), is essentially one to one. Hence any reference made to a certain role of a certain relationship, be it in the query where this relationship is declared, or in any subquery of this query, refers to the same entity.

Consequently, the entity created by the production `Add_Role` should be linked as a constituent to the "highest" query of which the given relationship is a constituent, so it can be used in any subquery of this query.

- The connection between a set and an entity ranging over this set (or between a bag and a value ranging over this bag,…) is clearly one to many. Hence the semantics of the query changes, depending on which (sub)query this entity is linked to as a constituent.

  For instance, the graphical queries of Figures 4.21 and 4.22 both return the names of all banks, together with the locations of the cashier stations they own. In Figure 4.21, the `CASHIER STATION` entity as well as the `set` belong to the same query. In terms of the graph model, this is the situation that would occur if the production `Add_to_Set` would associate an element of a set to the "highest" query of which the set is a constituent. Since this query is "flat", it returns a bag of pairs, each consisting of a `string` and an `address`.

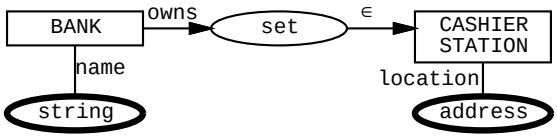


Figure 4.21: Banks and their cashier stations, Version I

In Figure 4.22, the `CASHIER STATION` entity is part of a subquery of the query in which the `set` occurs. Hence this query returns a bag of pairs, each consisting of a `string` and a *bag* of addresses. In terms of the graph model, this situation can only be obtained if it is possible to tell the production `Add_to_Set` explicitly to which subquery the element should be linked as a constituent.

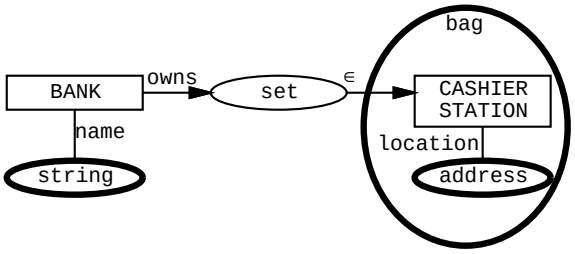


Figure 4.22: Banks and their cashier stations, Version II

The set `Sv`, the subquery `S` and the element type `POCtype` passed to the production `Add_to_Set` still have to obey some additional constraints:

- The subquery S must either be equal to some (sub)query of which Sv is a constituent, or it must be a (direct or indirect) subquery of such a query. In the left-hand side of Add_to_Set, this condition is expressed by looking for a CONSTITUENT-node corresponding to Sv, such that there exists an InHigherScopeThan-path from this node to S.

```
path InHigherScopeThan : CONSTITUENT -> SQB =
 <-is_defined_by- &
 (-is_defined_by-> & instance of SQB_CONS & -cons_is_n-> & instance of SQB)*
end;
```

  This path goes from a CONSTITUENT to the SQB it is linked to (by following an incoming is_defined_by-edge), and then follows zero or more alternating outgoing is_defined_by- and cons_is_n-edges, thereby going from the SQB to any SQB-node that corresponds to one of its (direct or indirect) subqueries. For instance, in Figure 4.20, a path of type InHigherScope-Than exists between the lowest of the two constituent-nodes and the top sqb-node.

- The element type POCtype must equal the Elem_Type meta attribute of (the node type of) the set Sv (cf. the condition-clause).

- If the set Sv is a Singleton, then it should not yet contain any elements (which is expressed using the condition that the cardinality of the set of its outgoing cont-edges should be zero).

Two productions offer the possibility of adding an element to a bag or list (i.e., a node of class MVALUE). The production Add_to_Mvalue differs from Add_to_Set only in the absence of the condition on the Singleton-attribute, which is not relevant to multi-values, hence the production is not shown here. Note that the multi-value may be itself a subquerybag, in which case the derivation rule for the Elem_Type-attribute of class SQB is triggered.

In the case of the GOQL/EER-query of Figure 4.10, we need the possibility to add an element at a *specific position* in a multi-value. This can be done using the production Add_Indexed_to_Mvalue.

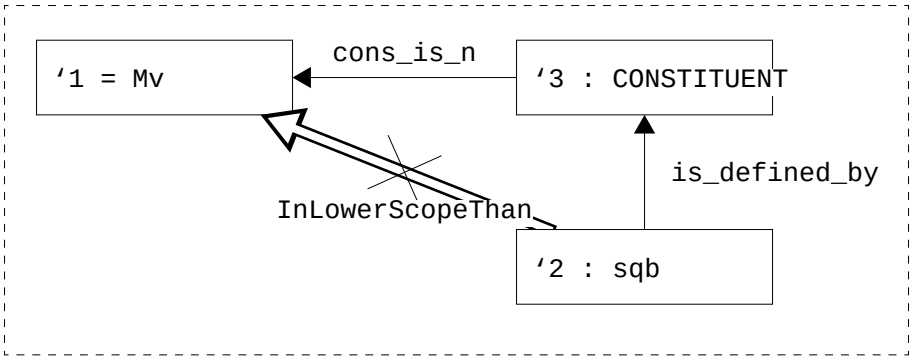
The graph part of this production is analogous to that of Add_Role. The condition-clause contains the usual comparison between the element-type of the multi-value and the component type passed to the production. In the transfer-clause, the Index passed to the production is stored in the Index-attribute of the newly created mmember-node.

In order to complete the graph-representation in Figure 4.17 of the query in Figure 4.10, we still have to add two more "non-graphical" features besides the list-index. First, we have to be able to select the string representing the name of the BANK for output. In terms of the graph model, this means setting its Output attribute to true. This can be done using the Select production.

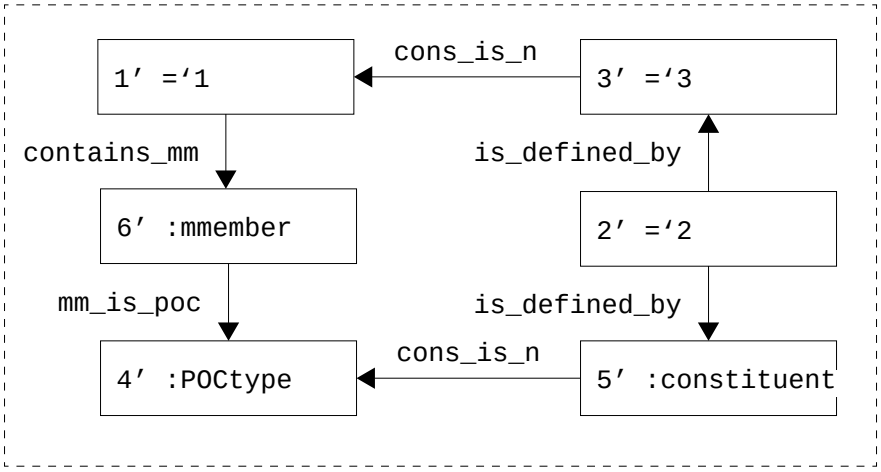
The condition-clause of this production takes care of a problem related to entities or values ranging over a subquery. If a subquery is used as a range, then exactly one of the subquery's constituents should be selected for output. In other words, the situation depicted in Figure 4.23 is not allowed. The subquery on the left returns a bag of pairs (each consisting of an address and a string), so there is no way to label the value ranging over this subquery.[5]

[5] Unless we would allow "(address, string)" as node label, which would contradict our intention to use only elements from the representation of EER diagrams.

```
production Add_Indexed_to_Mvalue
   ( Mv : MVALUE ; POCtype : type in PART_OF_COMPLEX ; Ind : integer ;
      out P : PART_OF_COMPLEX ; out C : CONSTITUENT )                    =
```




```
   ::=
```




```
   condition POCtype in ‘1.Elem_Type;
   return P := 4’;
          C := 5’;
end;
```

```
production Select ( C : CONSTITUENT ) =
      ┌ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┐
        ┌ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┐
      │ │ '1 =C               │
        └ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┘
      └ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┘
   ::=
      ┌ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┐
        ┌ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┐
      │ │ 1' = '1             │
        └ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┘
      └ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┘
   condition card ( '1.<-is_defined_by-.-contains_mm-> ) = 0;
   transfer 1'.Output := true;
end;
```

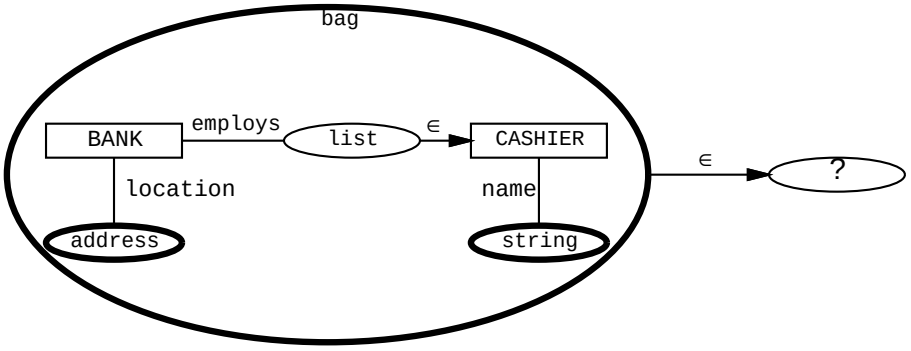
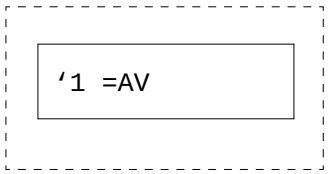


Figure 4.23: Multiple selections in a subquery serving as variable range

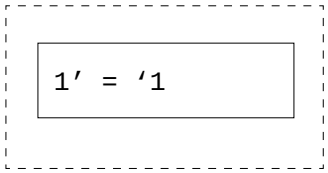
In terms of the graph model, this means that a constituent of some subquery may only be selected for output if it satisfies the condition that nothing ranges over the subquery, in other words, that there are no targets of outgoing `contains_mm`-edges.

In order to complete the graph-representation in Figure 4.17 we still have to assign the atomic value "General Banking" to the `string` representing the `name` of the `CONSORTIUM`. We therefore use the production `Assign_Value` which allows the assignment of an actual `Value` to the `Value`-attribute of a given `ATOMIC_VALUE` node `AV`.

```
production Assign_Value ( AV : ATOMIC_VALUE ; Val : string ) =
```




```
   ::=
```




```
 transfer
  1'.Value := Val;
end;
```

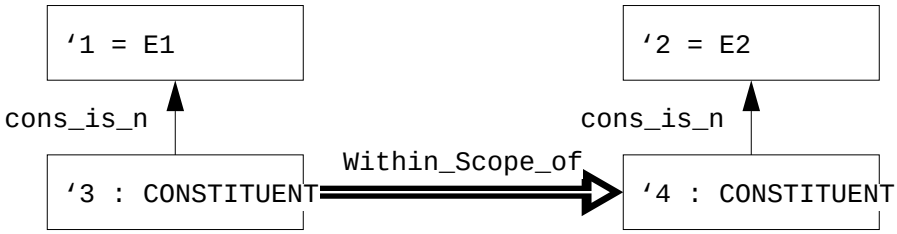
A final category of productions allows the merging of various kinds of query elements, in order to express equality conditions. The specification contains five such "merging" productions, namely for merging entities, relationships, atomic, set- and multi-values. In summary, all query elements can be merged, with the exception of subqueries, since it would be rather difficult to visualize merged subqueries.

As all five merging productions are quite similar, we only discuss the one for merging entities. The production `Merge_Entities` accepts as input two entities, which must obey two conditions.

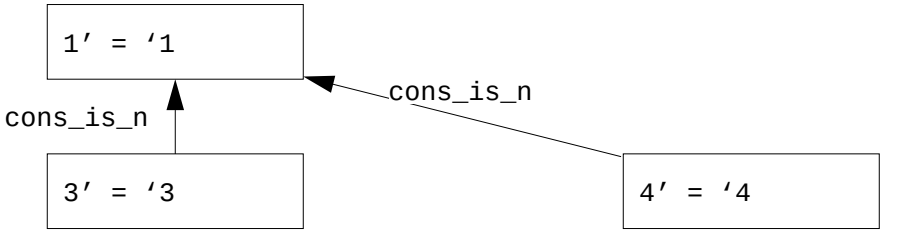
1. The `condition`-clause checks if their types are equal. This prevents the merging of e.g., an `ATM` with a `CASHIER`, which would obviously be an undesirable merge. Note however that this condition also prevents the merging of e.g., an `ATM` with an `ENTRY STATION`, even though these types are "compatible". The reason why we do not *have* to allow the merging of entities with compatible types is that all productions that allow the addition of an entity to the graph (i.e., `Add_ER, Add_to_Set, Add_to_MValue, Add_Indexed_to_MValue` and `Add_Role`) already allow the addition of entities of a subtype of the "expected" type, so if two entities have to be merged at some point, they can always be created with equal types.

2. The left-hand side of the production checks if no scoping rules are violated. More precisely, two entities E1 and E2 may only be merged (that is, compared) if either

```
production Merge_Entities ( E1, E2 : ENTITY ) =
```




```
   ::=
```




```
   condition E1.type = E2.type;
   embedding redirect <-role2e- from ‘2 to 1’;
             redirect <-mm_is_poc- from ‘2 to 1’;
             redirect <-cons_is_n- from ‘2 to 1’;
             redirect <-attribute2er- from ‘2 to 1’;
             redirect <-component2e- from ‘2 to 1’;
end;
```

- there exists a (sub)query of which both are constituents; or
- there exist (sub)queries S1 and S2 such that E1 is a constituent of S1 and E2 is a constituent of S2, and such that S1 is a direct or indirect subquery of S2. This situation is illustrated in Figure 4.24.

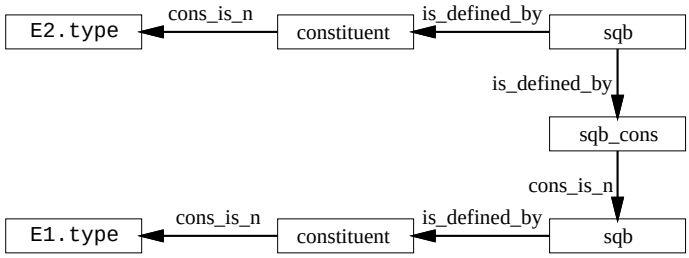


Figure 4.24: "Comparable" entities

Both possibilities are checked together, by looking for `CONSTITUENT`-nodes linked to the input entities, such that a `Within_Scope_of`-path exists between them.

```
path Within_Scope_of : CONSTITUENT -> CONSTITUENT =
   <-is_defined_by- & (<-cons_is_n- & <-is_defined_by-) * & -is_defined_by->
end;
```

By comparing the body of the `Within_Scope_of`-path to the graph of Figure 4.24, it may easily be verified that this path indeed leads from a constituent of some (sub)query to a constituent of a superquery of that query.

The effect of the production `Merge_Entities` is that the second input entity is removed (as shown in the right-hand side) and that all edges involving the removed entity, are redirected to the remaining entity. It may be verified in the graph scheme of the specification that the edge types listed in the `redirect`-clause of this production are indeed the types of all edges that could possibly involve a node of class `ENTITY`.

The difference between the production `Merge_Entities` and the productions for merging relationships, atomic, set- and multi-values lies mainly in the set of edge-types in the `redirect`-clause. The latter productions redirect, in the case of

relationships : incoming edges of types `role2r`, `attribute2er` and `cons_is_n`

atomic values : incoming edges of types `cont`, `mm_is_poc`, `attribute2v` and `cons_is_n`

set values : incoming edges of types `component2c`, `attribute2v` and `cons_is_n` and outgoing edges of type `cont`

bag values : incoming edges of types `component2c`, `attribute2v` and `cons_is_n` and outgoing edges of type `contains_mm`

In addition, the `condition`-clause of the production for merging multi-values uses the expression `not ('1.type = sqb)` to prevent its application to (sub)querybags, as motivated earlier on.

This concludes the discussion of those parts of the productions concerning the syntax of GOQL/EER queries. In the following section, most productions discussed above, as well as the graph scheme are extended (with among others additional attributes and attribute transfers) to incorporate also the semantics of GOQL/EER (that is, its translation to SQL/EER).

4.2.3 The Semantics of GOQL/EER

We now define the semantics of GOQL/EER queries in terms of the (formally defined) semantics of SQL/EER [HE92]. This mainly involves extending the PROGRES specification presented so far, with additional attribute declarations and corresponding attribute derivation rules, which translate the graphical query into an SQL/EER query. This SQL/EER query is then said to define the semantics of the GOQL/EER query.

As remarked in Section 2.3, in which we recalled SQL/EER, the basic building blocks of SQL/EER queries are (variable) declarations, formulas and terms. In Figure 4.13, it was already shown informally how fragments of graphical and textual queries may be seen to correspond, for instance:

- a node labeled with the entity type `BANK` corresponds both to the declaration of a variable (say `ba`) of this type, as well as to any occurrence of such a variable, as (part of) a term
- an `address`-node linked to this `BANK` entity by means of a `location`-edge, corresponds to the term `ba.location`
- if the `address`-node is also selected for output, then the entire graph corresponds to the query "`select ba.location from ba in bank`"
- another `address`-node linked to an oval surrounding this graph, corresponds to a variable, say `ad`, that ranges over the result of this (sub)query
- the fact that this same `address`-node is also linked to another `BANK`-node (corresponding to the variable `b`) by means of a `location`-edge, corresponds to the formula `b.location = ad`.

In summary, certain nodes as well as certain subgraphs of the graphical query correspond to terms. For certain nodes, this term is simply a variable, in which case the node also corresponds to this variable's declaration. Subgraphs may correspond to complete **select-from-where**-statements. In addition, certain "configurations" correspond to formulas. In the remainder of this section, we discuss how this correspondence is formalized in the PROGRES specification for GOQL/EER.

First of all, let us consider the part of the above introduced PROGRES graph scheme which is to be extended with (parts of) declarations to capture the semantics of GOQL/EER.

In the specification of node class `CONSTITUENT`, attributes of type `text` are declared for storing `Formula`s, `Declaration`s and `Term`s that correspond to the (unique) `QUERY_ELEM`ent linked to the `CONSTITUENT` (by means of a `cons_is_n`-edge). The reason for storing this information in the `CONSTITUENT` rather than in the `QUERY_ELEM`ent itself, is that this information really depends

```
section FixedGraphScheme
 node class CONSTITUENT
  intrinsic
   Formula : text := Text ( "true" );
   Declaration : text := EmptyText;
   Term : text := EmptyText;
   Output : boolean := false;
 end;
 node class DERIVED_SQL is a QUERY_ELEM
  derived SFW_Term : text = EmptyText;
 end;
 node class SQB_CONS is a CONSTITUENT, DERIVED_SQL
  redef derived SFW_Term = self.(-cons_is_n-> : SQB [1:1]).SFW_Term;
 end;
 node class SQB is a BAG_VALUE, DERIVED_SQL
  redef derived
   Elem_Type =
    ((self.=OutCons=>:CONSTITUENT[1:1]).-cons_is_n->:QUERY_ELEM[1:1]).type;
   SFW_Term =
    Concat(
     Concat(
      Concat (
       Text ( "( select " ),
       concom ( concom ( EmptyText, all self.=OutCons=>.Term ),
                concom ( EmptyText, all self.=OutSQB_Cons=>.SFW_Term ) ) ),
      Concat(
       Text ( " from " ),
       concom ( EmptyText, all self.-is_defined_by->.Declaration ) ) ),
     Concat(
      Text ( " where " ),
      conand ( Text ( "true" ), all self.-is_defined_by->.Formula ) ) )
    && " )" ;
 end;
end;
```

on the "context" in which the QUERY_ELEMent is seen, that is, the unique (sub)query to which the CONSTITUENT is linked (by means of an is_defined_by-edge). For instance, an ENTITY-node linked to both a query and one of its subqueries, may have one variable as Term in the context of the query, and another variable in the context of the subquery.

The reason for using the imported data type text (implemented in MODULA2) is that the built-in PROGRES data type string only supports strings of very limited length.

```
from LongStrings import

   types
      text;

   functions
      EmptyText      : -> text,
      Text           : ( string) -> text,
      &&             : ( text, string) -> text,
      ==             : ( text, string) -> boolean,
      Concat         : ( text, text) -> text;

end;
```

The following operations concern texts:

EmptyText	: returns an empty text
Text	: converts a string to a text
&&	: concatenates a text and a string into a text
==	: compares a text with a string
Concat	: concatenates two texts into a text

The default Formula is "true", while the default Terms and Declarations are the empty text. All three attributes are declared as intrinsic attributes since they have to be computed in the transfer-clause of various productions, among others because they may depend on "externally provided" information, such as variable names.

Terms corresponding to subqueries, however, may be derived by means of a derivation rule. This explains the name of the class DERIVED_SQL, whose sole purpose is the declaration of the attribute SFW_Term, which stands for Select-From-Where_Term, since terms corresponding to subqueries are always **select-from-where**-statements.

The node class DERIVED_SQL has two subclasses SQB and SQB_CONS, each of which has its own derivation rule for the SFW_Term-attribute.

The value of the SFW_Term-attribute of a node of class SQB_CONS (which is used to link a subquery to its superquery) simply equals the value of the SFW_Term-attribute of the (unique) node of class SQB that may be reached by following the outgoing cons_is_n-edge.

Probably the single most important expression in the entire specification of GOQL/EER (as far as semantics is concerned) is the derivation rule for the attribute SFW_Term as given in the declaration of the node class SQB. By means of this rule, information gathered from all over the graph is combined into an SQL/EER query. Basically, the "cascade" of applications of the Concat-function in this derivation rule results in a select-from-where-statement (in brackets, since it is also used for subqueries), the three parts of which are computed as follows:

- The `select`-clause consists of a comma-separated list of terms selected for output. Comma-separated lists are obtained using the function `concom`. This function takes two `text`s as arguments. If one of the arguments is empty, then the other argument is returned, otherwise a `text` is returned which consists of both arguments, separated by a comma.

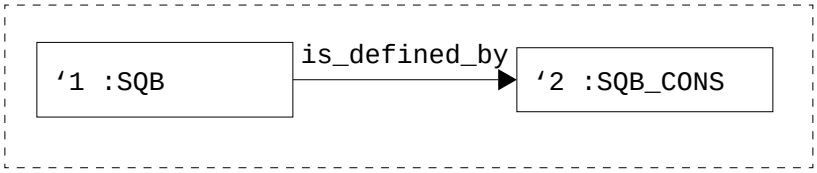
```
function concom : ( S1 : text ; S2 : text ) -> text =
   [ S1 == "" :: S2
   | S2 == "" :: S1
   | Concat ( S1 && ", ", S2 ) ]
end;
function conand : ( S1 : text ; S2 : text ) -> text =
   [ S1 == "true" :: S2
   | S2 == "true" :: S1
   | Concat ( S1 && " and ", S2 ) ]
end;
```

The `select`-clause consists first of all of the `Term`-attributes of those `CONSTITUENT`s that are not of type `sqb_cons`, and that have been selected for output. These `CONSTITUENT`s are retrieved using the path expression `OutCons`, used earlier on for determining the value of the `Elem_Type` attribute. The reason why we do not have to use an extended version of this path, excluding nodes of type `sqb_cons` is that the `Term`-attribute of the later kind of nodes is empty anyway. The set of all these attributes is passed to the function `concom` as a whole, using the `all`-operator. This operator specifically allows the second argument of a binary, associative and commutative[6] function to be replaced with a *set* of values. The result of applying a function f to a value v and a set of values v_1 through v_n is then defined as

$$f(v, \{v_1, \ldots, v_n\}) = f(\ldots f(f(v, v_1), v_2), \ldots, v_n)$$

The `select`-clause furthermore consists of the `SFW_Term`-attributes of those `SQB_CONS`-nodes that have been selected for output. These nodes are retrieved using the path `OutSQB_Cons`.

```
path OutSQB_Cons : SQB -> SQB_CONS =
   '1 => '2 in
```




```
   condition '2.Output;
end;
```

[6] Note that when it comes to computing the `select`-clause of an SQL/EER query, `concom` is indeed commutative, since in GOQL/EER there is no way of indicating the order in which things should be selected, so it doesn't matter in what order the corresponding terms appear in the `select`-clause.

- The `from`-clause consists of a comma-separated list of all `Declaration`-attributes of `CONSTITUENT`-nodes reachable from the considered `SQB`-node by means of outgoing `is_defined_by`-edges.
- The `where`-clause consists of an "and"-separated list of all `Formula`-attributes of `CONSTITUENT`-nodes reachable from the considered `SQB`-node by means of outgoing `is_defined_by`-edges. The function `conand` is used to generate this "and"-separated list.

We now discuss how the intrinsic attributes of `CONSTITUENT`-nodes, that is, `Term`, `Declaration` and `Formula`, are computed in the productions of the specification. If we look at the affected attributes, the fifteen productions of the specification can be divided into four categories:

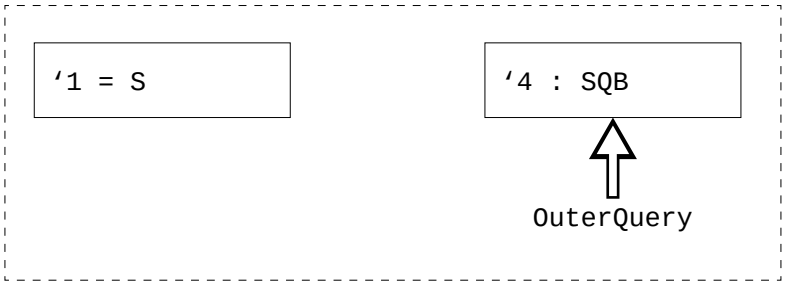
1. `Add_first_SQB` and `Add_SQB` do not affect these attributes, since they simply do not involve `CONSTITUENT`s (on the exception of nodes of class `SQB_CONS`, whose intrinsic attributes are not used).
2. `Add_Role`, `Add_Attribute`, `Add_Component` and `Add_Indexed_to_Mvalue` all add a single constituent to a given (sub)query. The `Term`-attribute of this newly created constituent is computed using the `Term`-attribute of other given nodes, hence its `Declaration`-attribute is not affected.
3. `Add_ER`, `Add_to_Set` and `Add_to_Mvalue` also add a single constituent to a given (sub)query, which however corresponds to a new variable in SQL/EER. Hence this variable is assigned to the `Term`-attribute of this newly created constituent, while the `Declaration`-attribute is set accordingly.
4. `Assign_Value` as well as the five productions for merging `QUERY_ELE`Ments express formulas. Hence each of them affects only the `Formula`-attribute of some constituent.

We now discuss the productions from the last three categories in detail. For a complete overview of all productions, we refer to Appendix C.

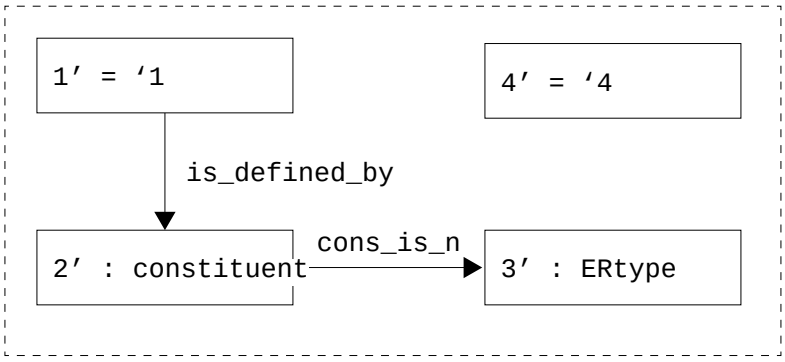
Adding an entity or relationship (i.e., a node of class `ENT_REL`) of a certain type (in the EER sense of the word) to the graphical representation of a GOQL/EER query by means of the production `Add_ER` corresponds to the declaration of a variable of this type in SQL/EER. Hence the production `Add_ER`, part of which was presented in Section 4.2.2, needs an additional input parameter `VarName` for the name of this variable. This variable is assigned to the `Term`-attribute of the `CONSTITUENT` by means of which the new entity or relationship is linked to the given subquery. The `Declaration`-attribute of the `CONSTITUENT` is set to the concatenation of the variable name, the keyword "in" and the type of the new entity or relationship (converted to a string using the function `string`). As an example, the node with identifier 225 in Figure 4.17 has "co" as `Term`-attribute and "co in consortium" as `Declaration`-attribute.

In order to avoid name clashes, we require that no variable name is used in two different declarations. The `condition`-clause enforces this by comparing `VarName` to the `Term`-attributes of all `CONSTITUENT`s found in the graph. To obtain the set of all these `CONSTITUENT`s, we have to start from the outermost query. This query is determined in the left-hand side of the production, by looking for the `SQB`-node that satisfies the restriction `OuterQuery`. This restriction simply checks if the

```
production Add_ER
 ( S : SQB ; VarName : string ; ERtype : type in ENT_REL ;
   out E : ENT_REL ; out C : CONSTITUENT )                                    =
```




```
        ::=
```




```
 folding { ‘1, ‘4 };
 condition not (Text(VarName) in ‘4.=recCons=>.Term);
 transfer 2’.Term := Text ( VarName );
          2’.Declaration := Text(VarName) && " in " && string(ERtype);
 return E := 3’;
        C := 2’;
end;
```

given SQB-node has no incoming cons_is_n-edges. In the folding-clause of the production, it is indicated that this SQB-node may be identical to the input parameter S. To this SQB-node, we then apply the path expression recCons, which recursively looks for all Constituents in the graph.

```
restriction OuterQuery : SQB =
   not with <-cons_is_n-
end;
```

```
path recCons : SQB -> CONSTITUENT =
   -is_defined_by-> & (-cons_is_n-> & instance of SQB & -is_defined_by->) *
end;
```

The productions Add_to_Set (not shown here) and Add_to_Mvalue differ from Add_ER only in the computation of the Declaration-attribute of the new CONSTITUENT. Add_to_Set uses as range the Term-attribute of the CONSTITUENT (whose choice was explained in Section 4.2.2) linked to the given. Depending on whether or not the multi-value given as input to Add_to_Mvalue is a (sub)querybag or not, the latter production uses the SFW_Term-attribute or the Term-attribute for this purpose.

Adding an attribute to an entity or relationship with the production Add_Attribute corresponds to stating a term in SQL/EER, namely the concatenation of the Term of the given entity or relationship, a dot and the string representation of the given attribute type. As an example, the node with identifier 313 in Figure 4.17 has "co.consortium_name" as Term-attribute.[7]

Add_Component and Add_Role are completely analogous to Add_Attribute (and hence not shown here). The production Add_Indexed_to_Mvalue concatenates the Term of the given multi-value with the given index enclosed in square braces. As an example, the node with identifier 273 in Figure 4.17 has "co.consortium_consists_of[2]" as Term-attribute.

Assigning a concrete value to an ATOMIC_VALUE with Assign_Value corresponds to expressing the condition (or formula) that the term corresponding to this ATOMIC_VALUE should equal the concrete value. Hence the production Assign_Value appends this formula to the Formula-attribute of some constituent linked to the ATOMIC_VALUE, using the function conand. As an example, the node with identifier 313 in Figure 4.17 has "co.consortium_name = 'General Banking' " as Formula-attribute.

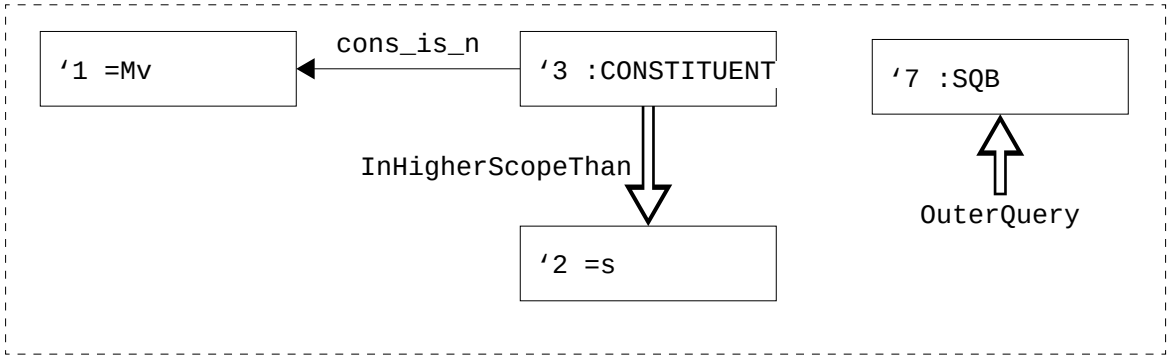
It does not matter which constituent is chosen by this production, since if there exists more than one constituent linked to an ATOMIC_VALUE, then this must be the result of (an) application(s) of the production Merge_Atomic_Values. Applying this production to two ATOMIC_VALUEs corresponds to expressing the condition (or formula) that the terms corresponding to these ATOMIC_VALUEs should be equal. Hence the production Merge_Atomic_Values appends this formula to the Formula-attribute of some constituent linked to the remaining ATOMIC_VALUE, using the function conand.[8]

The other four productions for merging query elements are totally analogous to Merge_Atomic_Values.

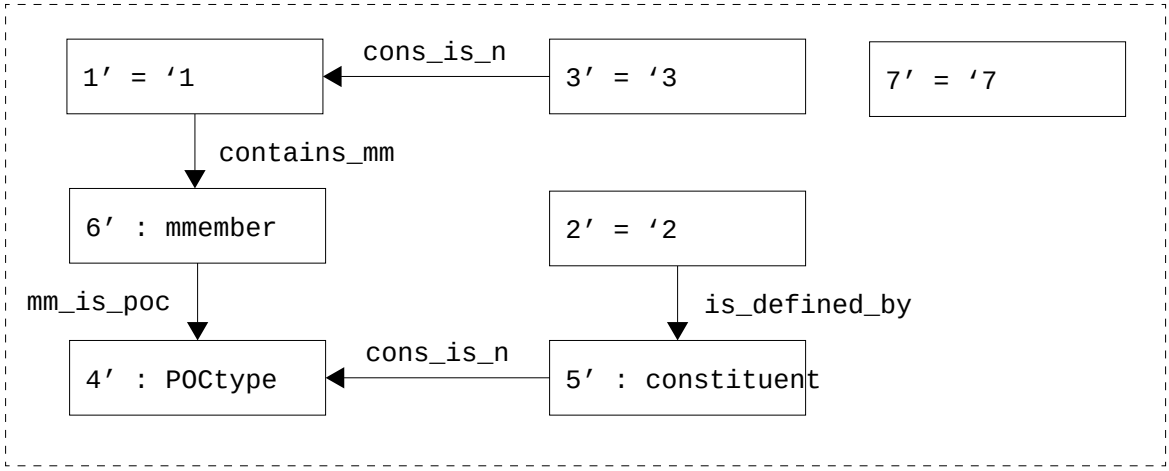
[7] The need for prepending the entity type name consortium to the attribute name name is explained in Section 4.2.1.

[8] The choice of the latter constituent was discussed in Section 4.2.2.

```
production Add_to_Mvalue
 ( Mv : MVALUE ; s : SQB ; POCtype : type in PART_OF_COMPLEX ;
   VarName : string ; out c : PART_OF_COMPLEX ; out C : CONSTITUENT )
=
```

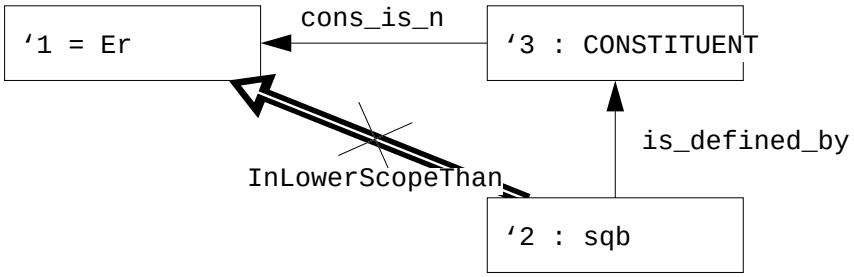


::=

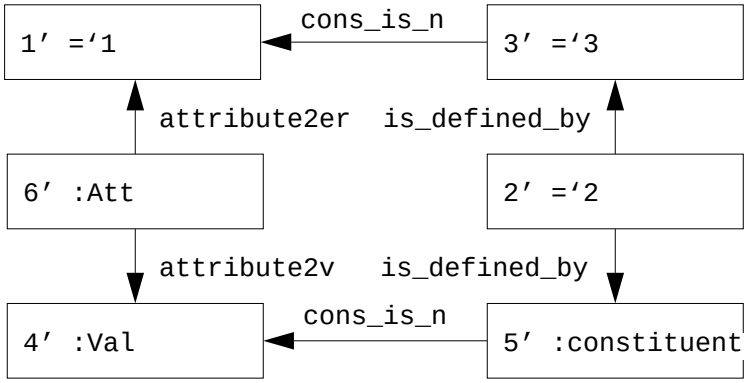



```
   folding { ‘2, ‘7 };
   condition POCtype in ‘1.Elem_Type;
             not (Text ( VarName ) in ‘7.=recCons=>.Term);
   transfer 5’.Term := Text ( VarName );
            5’.Declaration := Concat ( Text ( VarName & " in " ),
                      [ ‘3.type = sqb_cons :: (‘3 : SQB_CONS).SFW_Term
                      | ‘3.Term ]                                      );
   return c := 4’;
          C := 5’;
end;
```

```
production Add_Attribute
   ( Er : ENT_REL ; Att : type in ATTRIBUTE ; Val : type in VALUE ;
      out v : VALUE ; out C : CONSTITUENT )
=
```

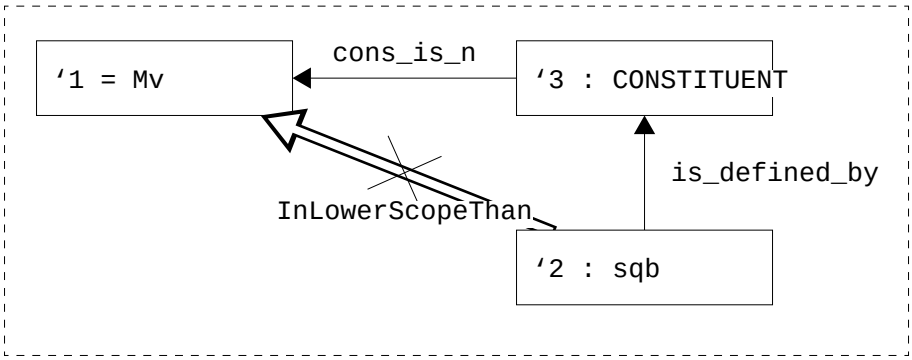



```
   ::=
```

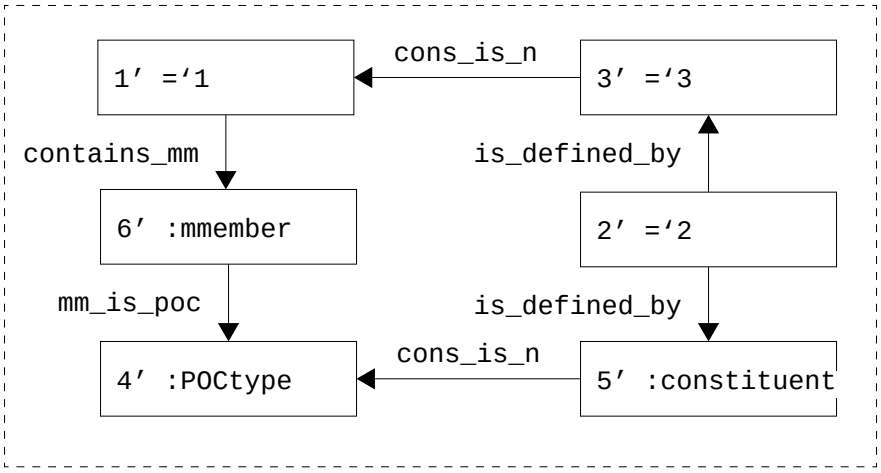



```
   condition Er.type in Att.entrel;
             Att.val = Val;
   transfer 5’.Term := ‘3.Term && "." && string ( Att );
   return v := 4’;
          C := 5’;
end;
```

```
production Add_Indexed_to_Mvalue
   ( Mv : MVALUE ; POCtype : type in PART_OF_COMPLEX ; Ind : integer ;
      out P : PART_OF_COMPLEX ; out C : CONSTITUENT )                    =
```

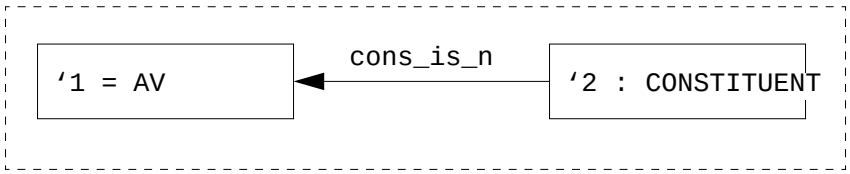



```
   ::=
```

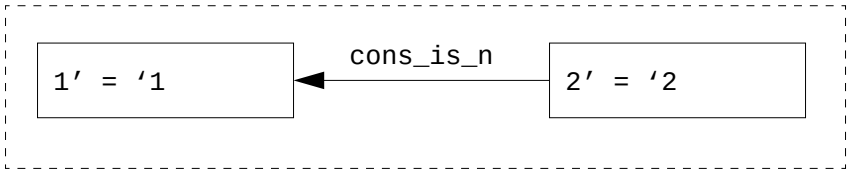



```
   condition POCtype in ‘1.Elem_Type;
   transfer 6’.Index := Ind;
            5’.Term := ‘3.Term && "[" && string ( Ind ) && "]";
   return P := 4’;
          C := 5’;
end;
```

```
production Assign_Value( AV : ATOMIC_VALUE ; Val : string) =
```




```
::=
```




```
  transfer
    2’.Formula := conand
      ( ‘2.Formula, Concat ( ‘2.Term && " = ", Text ( Val ) ) );
    1’.Value := Val;

end;
```

4.2.4 Executing the Specification

As mentioned in the introduction to this chapter, the choice of an *executable* graph grammar specification language, allows the execution of the specification using the PROGRES systems integrated interpreter. We now elaborate on how the specification presented so far is extended with an EER scheme dependent part in order to allow this execution.

This extension consists of two parts. First, the graph scheme of the specification is extended with additional node classes and types, modeling the various elements of the given EER scheme. Figure 4.25 shows part of the PROGRES graph scheme corresponding to the EER scheme depicted in Figure 2.1.

The extra node classes are introduced to cope with inheritance relationships between entity types, present in the EER scheme. On one hand it is not possible to specify inheritance relationships between node types, so we have to use a class for each entity type in the EER scheme. More precisely, if entity type A is an input type of some type construction, while entity type B is an output type of this type construction, then the graph scheme must contain a declaration `node class B is a A`. On the other hand, actual nodes have to belong to a type, so for each class we have to declare a type of each of these classes. The other node types are obtained from the EER scheme as follows:

- For each n-ary relationship type, a node type of class `RELSHIP` together with n node types of class `ROLE` are added. E.g., for the relationship type `entered_on`, we add the node type `entered_on` (of class `RELSHIP`) as well as node types `eo_es` and `eo_t` (of class `ROLE`). The `rel` meta attribute of the latter two node types is set to `entered_on`. Note that the `ent` meta attribute of `eo_es` is set to the node *class* `ENTRY_STATION` rather than to the node *type* `entry_station`. Since the name of a node class stands for the set of all its types, this means that nodes of the types `atm` and `cashier_station` may also play the `eo_es`-role in relation-

```
section VariableGraphScheme
   section NodeClasses
      node class ENTRY_STATION is a ENTITY end;
      node class ATM is a ENTRY_STATION end;
      node class CASHIER_STATION is a ENTRY_STATION end;
      node class CONSORTIUM is a ENTITY end;
      node class BANK is a ENTITY end;
      node class ACCOUNT is a ENTITY end;
      node class GENERIC_TRANSACTION is a ENTITY end;
      node class CASHIER_TRANSACTION is a GENERIC_TRANSACTION end;
      node class REMOTE_TRANSACTION is a GENERIC_TRANSACTION end;
      node class CASHIER is a ENTITY end;
      node class CASH_CARD is a ENTITY end;
      node class CUSTOMER is a ENTITY end;
   end;
   section NodeTypes
      node type entry_station : ENTRY_STATION end;
      node type atm : ATM end;
      node type cashier_station : CASHIER_STATION end;
      node type set_of_cashier_station : SET_VALUE
         redef derived Elem_Type = CASHIER_STATION;
      end;
      node type bank : BANK end;
      node type list_of_bank : LIST_VALUE
         redef derived Elem_Type = BANK;
      end;
      node type account : ACCOUNT end;
      node type account_s : SET_VALUE
         redef intrinsic Singleton := true;
         redef derived Elem_Type = ACCOUNT;
      end;
      node type entered_on : RELSHIP end;
      node type eo_es : ROLE
         redef meta rel := entered_on ;
                    ent := ENTRY_STATION ;
      end;
      node type eo_t : ROLE
         redef meta rel := entered_on ;
                    ent := TRANSACTION ;
      end;
      node type money : ATOMIC_VALUE end;
      node type address : ATOMIC_VALUE end;
      node type list_of_address : LIST_VALUE
         redef derived Elem_Type = address;
      end;
      node type entry_station_location : ATTRIBUTE
         redef meta entrel := ENTRY_STATION ;
                    val := address ;
      end;
      node type consortium_consists_of : COMPONENT
         redef meta cent := CONSORTIUM ;
                    comp := list_of_bank ;
      end;
   end;
end;
```

Figure 4.25: EER scheme dependent part of the PROGRES graph scheme for GOQL/EER

ships of type entered_on (cf. the condition-clause of production Add_Role).

- For each set, list, bag or singleton of entities occurring as a component, a corresponding node type is added, and its Elem_Type meta attribute is initialized. In the case of a singleton, a node type of class SET_VALUE is added, and the Singleton-attribute is set to true. For instance, since the owns-component of a BANK is a set of CASHIER_STATIONs, we add the node type set_of_cashier_station, and initialize its Elem_Type-attribute to CASHIER_STATION.
- For each set, list or bag of atomic values occurring as attribute domain, a corresponding node type is added. E.g., since the residence of a customer is a list of addresses, we add the node type list_of_address of class LIST_VALUE, and initialize its Elem_Type-attribute to address.
- For each atomic value type occurring as attribute domain or as element type of a complex value, a corresponding node type of class ATOMIC_VALUE is added. E.g., since the location of an entry_station is an address, we add a node type address.
- For each attribute, a corresponding node type of class ATTRIBUTE is added. E.g., we add the node type entry_station_location, and initialize its entrel meta attribute to ENTRY_STATION and its val meta attribute to address.
- For each component, a corresponding node type of class COMPONENT is added. E.g., we add the node type consortium_consists_of, and initialize its cent meta attribute to CONSORTIUM and its comp meta attribute to list_of_bank.

Looking back at Figure 4.16, this EER dependent part of the graph scheme can be used to *call* productions, generating the graph representation of GOQL/EER queries (including their translation to SQL/EER).

```
transaction MAIN =
 use s : SQB;
     e1 : ENT_REL;
     e2 : PART_OF_COMPLEX;
     c1, c2, c3, c4, c5, c6 : CONSTITUENT;
     av1, av2 : VALUE;
     cv1 : COMPLEX_VALUE
 do
     Add_first_SQB ( out s )
   & Add_ER ( s, "co", consortium, out e1, out c1 )
   & Add_Component
     ( (e1 : ENTITY), consortium_consists_of, list_of_bank, out cv1, out c2 )
   & Add_Indexed_to_Mvalue ( (cv1 : MVALUE), bank, 2, out e2, out c3 )
   & Add_Attribute ( e1, consortium_name, _string, out av1, out c4 )
   & Assign_Value ( (av1 : ATOMIC_VALUE), "General Banking" )
   & Add_Attribute ( (e2 : ENTITY), bank_name, _string, out av2, out c6 )
   & Select ( c6 )
 end
end;
```

As an example, the transaction MAIN creates the graph of Figure 4.17. As mentioned earlier, the creation of the graph representation of a GOQL/EER query always starts with a single application

of Add_first_SQB. Next, a consortium is added, with (SQL/EER) variable name "co". To this consortium, a consortium_consists_of component is added, which is of type list_of_-bank. In the application of this production, the variable e1 (referring to the consortium) has to be *cast* to the node class ENTITY, since this is what Add_Component expects, whereas Add_ER returns nodes of class ENT_REL. Next, a bank is added in position 2 to the list of banks. The consortium then gets a consortium_name attribute of type _string, which is assigned the value "General Banking". Likewise, the bank gets a bank_name attribute of type _string, which is selected for output.

At this point we come back to a statement made at the very beginning of Section 4.2, namely that a graph is part of the language of graph representations of GOQL/EER queries, if it may be obtained by applying any *correct* sequence of productions to an initial empty graph. A careful examination of the productions of the specification reveals that indeed *any* sequence of production calls that is correctly specified and successfully executable, results in the graph representation of some GOQL/EER query.

As an example, note that even the transaction which consists of merely an application of the production Add_first_SQB corresponds to the empty GOQL/EER query, whose semantics is defined by the "trivial" SQL/EER query "**select from where** true".

A "correctly specified" production is production which is called with sufficient parameters of the correct type, where variables used as input-parameters must be initialized by some previous production application, and cast to the correct class where necessary. A "successfully executable" production is a production which is called in such a way that its pre-conditions (as expressed in its header, its condition-clause and its left-hand side) are not violated.

Consequently, the notion of correctness depends solely on the semantics of PROGRES, and not on the semantics of GOQL/EER.

4.3 A Hybrid Query Language: HQL/EER

In the language GOQL/EER as introduced in Section 4.1, we consciously restricted the set of language constructs (such as declarations, atomic formulas,…) which may be represented graphically to the set of symbols used for graphically representing EER schemes. The fact that consequently, concepts such as negation and aggregate functions cannot be expressed in a GOQL/EER query, implies that the expressive power of this language is strictly less than that of SQL/EER. One possible way to go in order to give GOQL/EER the full expressive power of SQL/EER would be to invent graphical counterparts for all language constructs of SQL/EER. As argued in Chapter 1, however, graphical expressions in the resulting language would most probably be equally hard to understand and use as their counterparts in the textual language.

The other way to go is to *merge* the textual and the graphical language into a *hybrid* language, which allows the use of both textual and graphical elements in the formulation of one and the same query. Or, in other words, given the fact that GOQL/EER offers graphical counterparts for only those language elements of SQL/EER which (in our view) are worth representing graphically, such a hybrid language allows the graphical formulation of those parts of a query that are more easily formulated graphically than textually.

Concretely, in this section we discuss the Hybrid Query Language for the Extended Entity Relation-

ship model (in brief, HQL/EER). Syntactically, a query in HQL/EER consists first of all of a (possibly empty) GOQL/EER query, possibly involving subqueries. To this query and each of its subqueries, (possibly incomplete) SQL/EER queries may be associated. In a (partial) SQL/EER query associated to a graphical (sub)query, whenever a term of a certain type is expected, an *identifier* of a node of that type in the graphical query (or its super-queries) may be used. These identifiers are the mechanism by means of which the textual parts of a hybrid query are linked to the graphical part.

The semantics of a hybrid query is obtained by combining (in a sense to be made precise) the SQL/EER-query corresponding to the graphical part of the hybrid query (obtained by means of the specification discussed in Section 4.2) with the textual parts into a full SQL/EER-query.

4.3.1 Examples of Hybrid Queries

We clarify the ideas presented above by providing hybrid "versions" for most of the SQL/EER queries used in the recapitulation of SQL/EER in Section 2.3. First, reconsider Example 2.9, retrieving the `serial number`s of all `CASH CARD`s with the trivial `password` "password" and a (credit) `limit` less than or equal to 100.000. Since the latter condition involves an inequality, this query cannot be represented in GOQL/EER. However, Figure 4.26 shows a *hybrid* representation of this query in HQL/EER.

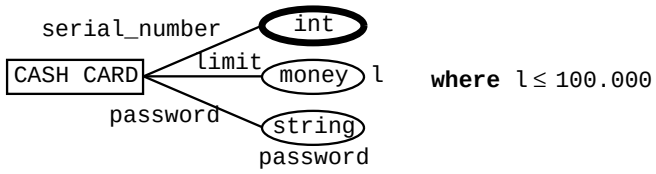


Figure 4.26: Serial numbers of cash cards with trivial password and limit under 100.000 (HQL/EER version)

The graphical part of this hybrid query already indicates that we want to retrieve the `serial number` of all `CASH CARD`s with the trivial `password` "password". To the `money`-node, representing the `CASH CARD`s `limit`, the identifier l has been associated. This identifier is used in the textual part of the hybrid query to express the additional condition that only those `CASH CARD`s should be considered whose `limit` is less than or equal to 100.000.

Note that the graphical part of this hybrid query, including the `money`-node, is indeed a syntactically correct GOQL/EER query. Looking back at the formal specification of GOQL/EER presented in Section 4.2, it may easily be seen that nothing prevents the addition of "obsolete" nodes such as the `money`-node of the example, using in this case the production `Add_Attribute`. If such a node is neither selected for output, nor merged with some other node to express an equality condition, then it simply will not influence the semantics of the (GOQL/EER) query.

Figure 4.27 shows an HQL/EER version of the query of Example 2.13, which enumerates the `name`s of `bank`s which are part of the `CONSORTIUM` `named` "*Banks United*". The identifier cco, associated to the `consists_of` attribute of the `CONSORTIUM`, is used both in the **select**-clause and

in the from-clause of the textual part of the query. Note that the BANK-node serves no actual purpose, but has been added merely for clarity.

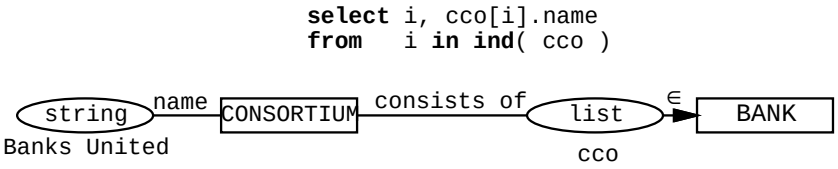


Figure 4.27: Enumeration of the names of banks which are part of the consortium named "*Banks United*" (HQL/EER version)

In the foregoing examples, we put as many elements of the queries in the graphical part as possible. In many cases, however, there exists a wide range of possibilities to express one and the same query in HQL/EER, since any element of the query that can be expressed graphically, can also be represented textually. We illustrate this fact by providing four hybrid versions of the textual query of Example 2.10. This query retrieves the names of all CUSTOMERs who share one of their residences with a CUSTOMER called "John". The SQL/EER query of Example 2.10 is itself already an HQL/EER query, since any textual query is itself a hybrid query with an empty graphical part.

As shown in Figure 4.28, this same query may easily be represented in GOQL/EER, and hence in HQL/EER, since any graphical query is itself a hybrid query with an empty textual part!

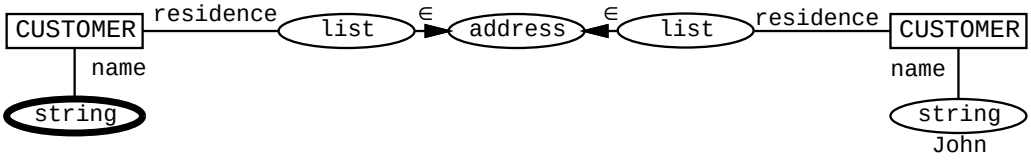


Figure 4.28: Names of customers sharing an address with John (HQL/EER version I, or GOQL/EER version)

Figure 4.29 shows a "real" hybrid version of still the same query. In comparison to the GOQL/EER version depicted in Figure 4.28, the selection has now been moved to the textual part, and is connected to the graphical part by means of the identifier c1 corresponding to one of the CUSTOMERs.

Finally, Figure 4.30 shows another hybrid version of the same query. In this version, also the condition that the other CUSTOMER should be named John has been moved to the textual part, which is now also connected to the graphical part by means of the identifier c2 corresponding to this other CUSTOMER.

We now turn our attention towards queries involving subqueries. Consider the following variation on the SQL/EER query depicted in Figure 4.11. Suppose we wish to retrieve for each address of a BANK recorded in the database, the *number* of BANKs located at this address (rather than the names of all these BANKs). Figure 4.31 shows an SQL/EER version of this query.

Comparing Figures 4.12 (depicting the GOQL/EER version of the original query) and 4.32 (depicting the HQL/EER version of the modified query), we see that they only differ in the subquery depicted

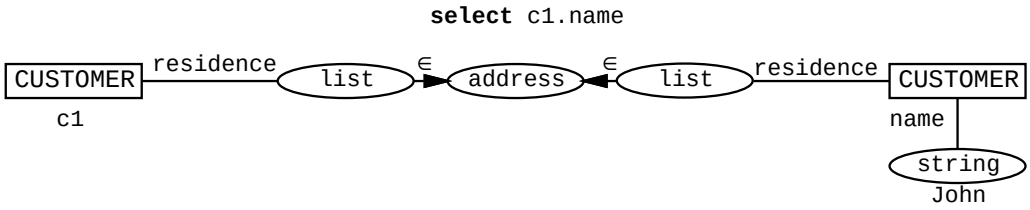


Figure 4.29: Names of customers sharing an address with John (HQL/EER version II)

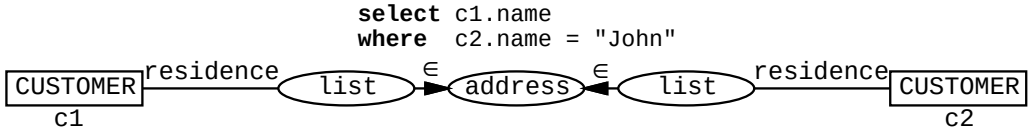


Figure 4.30: Names of customers sharing an address with John (HQL/EER version III)

```
select ad, cnt ( select  b
                 from   b in BANK
                 where b.location = ad )
from  ad in ( select ba.location
              from  ba in BANK )
```

Figure 4.31: For each address of a bank recorded in the database, the number of banks located at this address (SQL/EER version)

in the top right corner (which in both cases corresponds to the subquery in the **select**-clause of the corresponding SQL/EER query). In Figure 4.32, this subquery simply computes the bag of all BANKs in the database located at the address ad. Selection of the cardinality of this bag is done in the textual part of the hybrid query.

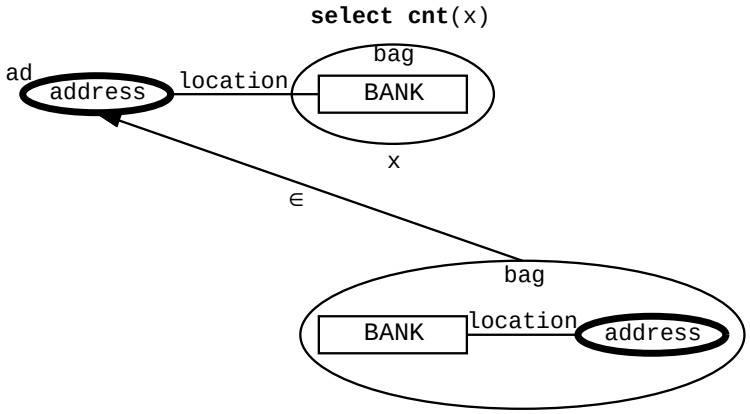


Figure 4.32: For each address of a bank recorded in the database, the number of banks located at this address (HQL/EER version)

In the previous example, a textual part was only associated to the outermost query. As an example of a textual part associated to a subquery, we present a hybrid version of the SQL/EER query from Example 2.12. This query returns the names of those CUSTOMERs holding an ACCOUNT for which the following holds: if the balance of the ACCOUNT is raised with a five percent interest, then the new balance becomes higher than the average of all balances of all ACCOUNTs with a positive balance. The elements of this query that cannot be represented graphically are the inequality $\geq_{\texttt{money}}$, the aggregate function **avg** and the data operation compute interest.

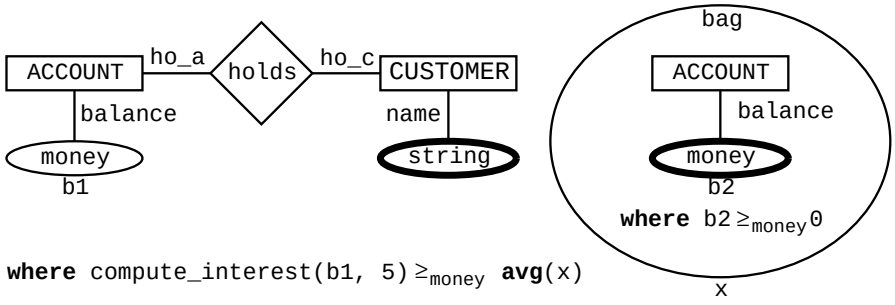


Figure 4.33: A hybrid query with text associated to a subquery

Figure 4.33 shows a hybrid version of this query. The "hybrid subquery" depicted on the right corresponds precisely to the subquery in the SQL/EER version: the declaration of the ACCOUNT and the selection of its balance are formulated graphically, while the condition that the balance should be positive is formulated textually. The textual and graphical part are linked using the identifier b2.

In the graphical part on the left, a CUSTOMER and an ACCOUNT are declared. The condition that the CUSTOMER must hold the ACCOUNT is expressed, and the CUSTOMERs name is selected. Additionally, the ACCOUNTs balance is also depicted graphically, and the identifier b1 is associated to it. This identifier, as well as the identifier x associated to the subquery are then used to express the remaining condition that if the balance of the ACCOUNT is raised with a five percent interest, then the new balance becomes higher than the average of all balances of all ACCOUNTs with a positive balance.

Note that in the graphical representation of this query, the subquery is "disconnected" from the graphical part of the outer query, a peculiarity which perfectly fits the PROGRES specification presented previously, just like the addition of "obsolete" nodes.

4.3.2 On the Semantics of Hybrid Queries

Remember that an HQL/EER query consists of a GOQL/EER query, with a SQL/EER query associated to some of its subqueries. In an SQL/EER query associated to a graphical (sub)query, whenever a term of a certain type is expected, an identifier of a node of that type in the graphical query (or its superqueries) may be used.

The PROGRES specification presented in Section 4.2 associates SQL/EER terms, declarations and formulas to nodes in the formal graph representation of the GOQL/EER query. The combination of this information with the textual parts of the hybrid query into a complete SQL/EER-query, defines the semantics of the hybrid query.

We illustrate how all these bits and pieces of SQL/EER queries can be combined into one complete textual query, using the HQL/EER query of Figure 4.33. Figure 4.34 shows the graph representation of the graphical part of this query, while Table 4.2 shows the attributes of the relevant nodes of this graph. Composing the SQL/EER query corresponding to this hybrid query starts with the subquery. The textual part associated to this subquery is "**where** b2 $\geq_{\texttt{money}}$ 0". The identifier b2 refers to the money-node in the graphical part of the subquery.

In the formal graph representation, this money-node is represented by the node with identifier 339 (of type money). The term "ac.account_balance" corresponding to this node is stored in the constituent with identifier 355. In the general case, finding this term is not as straightforward as in this example, since an element of a query (i.e., a node of class QUERY_ELEM) may be linked to *several* sqb-nodes by means of constituent-nodes. The following question then naturally arises: given a query element linked to several constituents (by means of cons_is_n-edges), whose Term-attribute should we use in replacement of an identifier referring to this query element?

In Section 4.2.2 (in the discussion of the input parameters of the production Add_Role), it was already noted that among all the (sub)queries of which a query element is a constituent, one is a (direct or indirect) superquery of all the others. If this were not the case, this would mean that the query element is used in two incomparable scopes, which is a clear violation of traditional scoping rules.

By the above reasoning, the constituent to be used is the one connecting the query element to the (sub)query which is a (direct or indirect) superquery of all the other subqueries of which the query element is a constituent. Indeed, the Term associated to this constituent is "known" in all scopes in which the query element is used. In addition, it has the same *semantics* in all these scopes. Indeed, the only way in which a query element can become a constituent of more than one

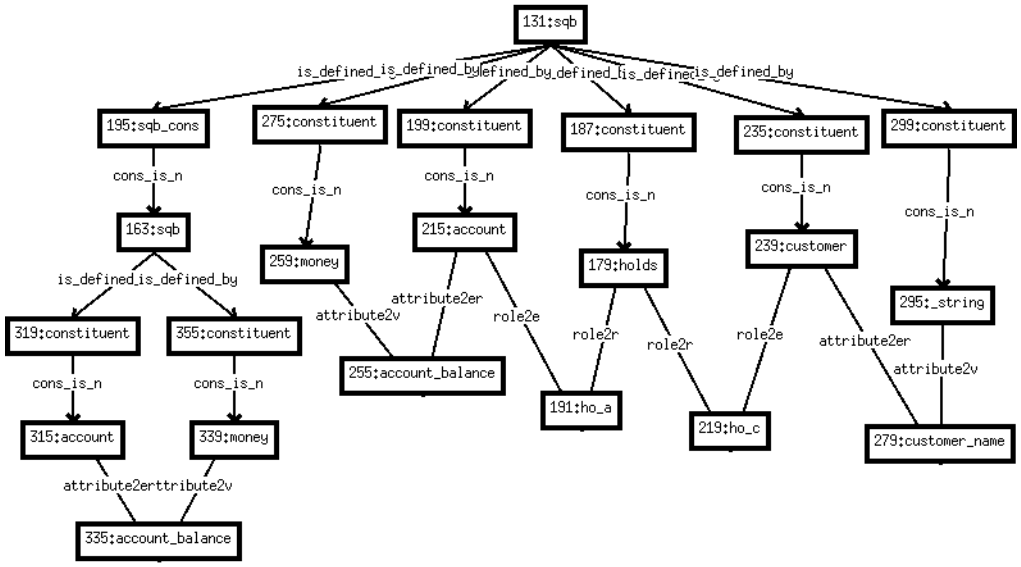


Figure 4.34: Graph representation of the graphical part of the hybrid query of Figure 4.33

Id.	Label	Att.Name	Attribute Value
131	sqb	SFW-Term	(select ho.ho_c.customer_name from ho in holds where true)
199	constituent	Term	ho.ho_a
275	constituent	Term	ho.ho_a.account_balance
187	constituent	Declaration Term	ho in holds ho
235	constituent	Term	ho.ho_c
299	constituent	Term Output	ho.ho_c.customer_name TRUE
163	sqb	SFW-Term	(select ac.account_balance from ac in account where true)
319	constituent	Declaration Term	ac in account ac
355	constituent	Term Output	ac.account_balance TRUE

Table 4.2: Attributes of some nodes of the graph in Figure 4.34

(sub)query, is by the application of the appropriate "merging" production. In the `transfer`-clause of these merging productions, the "semantics" of the node resulting from this merging is defined as the formula expressing the equality of the terms corresponding to the two merged nodes.

Returning to our example, the term "ac.account_balance" is substituted for the identifier b2 in the textual part of the subquery, resulting in the condition "**where** ac.account_balance $\geq_{\texttt{money}}$ 0". This condition is added (using **and**) to the **where**-clause of the SQL/EER-query stored in the SFW-Term attribute of the `sqb`-node corresponding to the subquery (that is, the node with identifier 163). This results in the (sub)query

select ac.account_balance
from ac **in** ACCOUNT
where ac.account_balance $\geq_{\texttt{money}}$ 0

The same procedure is now applied to the outer query. The textual part of this query is the condition "**where** compute_interest(b1, 5) $\geq_{\texttt{money}}$ **avg**(x)". The identifier b1 refers to the `money`-node in the outer query. The term "ho.ho_a.account_balance" corresponding to this node is stored in the `constituent`-node with identifier 275. The identifier x refers to the subquery, hence it is replaced with the SQL/EER query corresponding to this subquery, as computed above. All this results in the condition

where compute_interest(ho.ho_a.account_balance,5) $\geq_{\texttt{money}}$
avg (**select** ac.account_balance
from ac **in** ACCOUNT)
where ac.account_balance $\geq_{\texttt{money}}$ 0)

This condition is appended to the **where**-clause of the SQL/EER corresponding to the outer query, which may be found in the SFW-Term attribute of the `sqb`-node with identifier 131. This results in the SQL/EER query of Example 2.12.

4.3.3 Towards a Formal Definition of HQL/EER

So far we have illustrated by means of examples both the syntax (i.e., the representation as a formal graph) and the semantics (i.e., the translation to SQL/EER) of HQL/EER queries. To conclude this section, we present these ideas on a somewhat higher level of abstraction.

Syntactically, a query in HQL/EER consists of a GOQL/EER query, together with an SQL/EER query associated to this query and each of its subqueries. We assume without loss of generality that at least the "trivial" SQL/EER query "**select from where** true" is associated to any graphical (sub)query. In an SQL/EER query associated to a graphical (sub)query, "identifiers" of nodes in that graphical query (or its super-queries) may be used wherever a term is expected. The node referred to by the identifier must have the correct type (which equals its label, cf. the correspondence between the pictorial representation of a GOQL/EER query and its formal graph representation as discussed in Section 4.2.1).

The semantics of an HQL/EER query is defined by translating it to SQL/EER, using the following algorithm:

1. The formal graph representation G (including the node attributes) of the graphical part of the hybrid query is determined, according to the PROGRES specification discussed in Section 4.2.

2. The formal graph G obtained as a result of the foregoing step includes a tree T of `sqb`-nodes (cf. Figure 4.19), corresponding to the subqueries of the graphical query, with the root corresponding to the query itself. Each node q in G which is of class `QUERY_ELEM` is linked to one or more nodes of class `SQB` by means of a node of class `CONSTITUENT` (and edges of type `is_defined_by` and `cons_is_n`). Among all the `SQB`-nodes to which q is linked, one is an ancestor (in T) of all the others. Let us denote with $a(q)$ the `CONSTITUENT`-node which links q to the latter `SQB`-node.

   The SQL/EER query corresponding to the given hybrid query is computed by traversing T in postorder (that is, a node of the tree is visited *after* its children), applying the following two operations to each node n with associated SQL/EER query $s(n)$:

   (a) Each identifier i occurring in $s(n)$, referring to some query element q of n, is replaced as follows:

      i. If q is of type `sqb`, then i is replaced by the `SFW_Term`-attribute of q (which may have been "recomputed" by an earlier application of step 2b of this algorithm).

      ii. If q is not of type `sqb`, then i is replaced by the `Term`-attribute of the `constituent`-node $a(q)$.

   (b) The **select-from-where**-statement resulting from step 2a is "merged" with the `SFW_Term`-attribute of n by joining the **select**- respectively the **from**-clauses with a comma, and the **where**-clauses with an **and**. In subsequent iterations of this algorithm, the resulting "identifier-free" SQL/EER-query is used in replacement of the `SFW_Term`-attribute of n.

   This iteration eventually results in an "identifier-free" replacement of the `SFW_Term`-attribute of the root node of T, which is considered to define the semantics of the original hybrid query.

We conclude this Chapter with the observation that since

- each HQL/EER query can (by definition) be translated into an equivalent SQL/EER query, and
- each SQL/EER query is (by definition) itself an HQL/EER query

HQL/EER and SQL/EER have indeed precisely the same expressive power.

Chapter 5

Database manipulation defined as graph-rewriting

In Section 1.2, we informally described a graph rewrite rule as a pair of graphs called its left- and right-hand side. Basically, the left-hand side describes a configuration to be matched to some part of the graph to which the rule is to be applied, while the right-hand side (or more precise, the way in which the right-hand side differs from the left-hand side) describes the desired modification to be performed on the chosen matching of the left-hand side.

If we now consider database manipulations (as opposed to database queries, the topic of Chapter 4) on a very general level, we see that a database manipulation generically consists of a query together with the specification of some modification to be performed on the outcome of the query. When in addition, we look upon a database instance as a network or graph of objects or data items, the question arises quite naturally whether graph rewriting is a suitable database manipulation paradigm.

In this chapter, we answer this question positively by introducing the *Graph-Oriented Object Database language* GOOD/ER in which database manipulations may indeed be expressed as rewritings of a database graph. GOOD/ER is based on a slightly modified version of the original ER model, formally defined in Section 5.1.[1] A formal definition of GOOD/ER is presented in Section 5.2. In Section 5.3, we characterize GOOD/ER's expressive power (i.e., the set of transformations expressible in the language) in terms of a completeness criterion.

Section 5.2 of this chapter is based on [AGP$^+$92, PVdBA$^+$92], while Section 5.3 is based on [AP92, AP96].

5.1 The Entity-Relationship Model

In this section, we briefly discuss a restricted version of the EER model as discussed in Chapter 2. We call this version "the" Entity-Relationship Model, even though it does not exactly match the ER model as introduced by Chen in [Che76].[2]

[1] As noted in Chapter 1, the reason for presenting this language in terms of this ER model rather than the EER model is merely one of succinctness: it would be perfectly possible to define a GOOD/EER language, but this would only enlarge and complicate all definitions and results, and not add anything to the point we wish to make in this chapter.

[2] Most notably, we do not consider cardinality constraints on roles.

As an illustration, Figure 5.1 shows an ER version of part of the EER diagram shown in Figure 2.1. Table 5.1 shows the attributes of this ER diagram. The ER version differs from the EER version mainly by the absence of inheritance (i.e., type constructions), components and complex attributes:

- Inheritance has been eliminated by "replacing" a superclass and its subclasses either by one of the subclasses (e.g., `ENTRY STATION` and its subclass `CASHIER STATION` were replaced by its subclass `ATM`) or by the superclass (e.g., the subclasses `CASHIER TRANSACTION` and `REMOTE TRANSACTION` of `TRANSACTION` have been removed).
- Components have either been replaced by relationships (e.g., `manages`), or have simply been removed (e.g., `proper_acct`).
- Complex attributes as well as relationship attributes have been removed all together (e.g., `residence`).

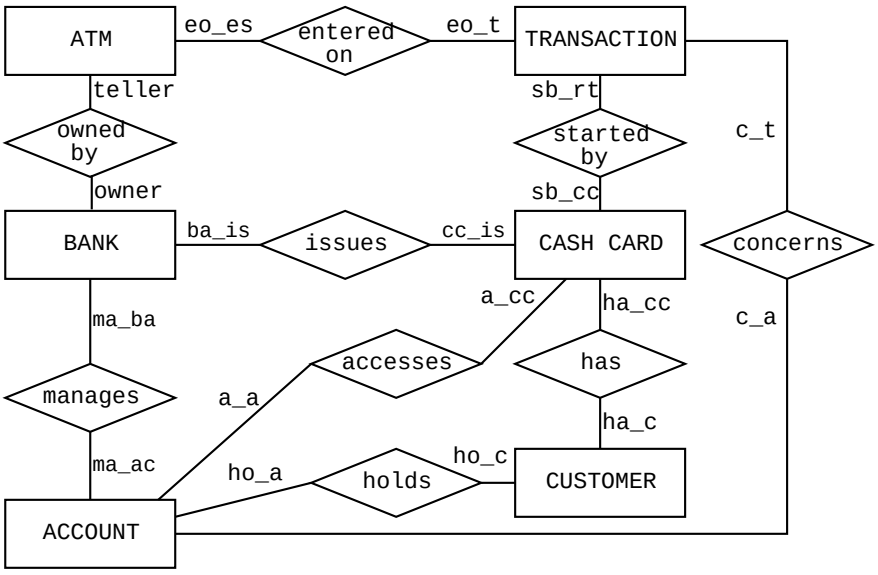


Figure 5.1: An ER diagram, modeling a network of automated teller machines

The formal definition of ER schemes is just a "restriction" of Definition 2.5 of EER schemes to those elements still present in ER schemes, i.e., entity types, relationship types, roles, attributes and data types:

Definition 5.1 (ER Scheme) *An* ER scheme $\mathcal{S}$ *consists of*

- *five disjoint finite sets E-TYPE, R-TYPE, ROLE, ATTR and D-TYPE;*
- *five functions with signatures:*

$$\text{participants} \quad : \quad \textit{R-TYPE} \rightarrow \textit{E-TYPE}^+$$

Entity type	Attribute	Data type
TRANSACTION	entry_time	time
	amount	money
ATM	cash on hand	money
	location	address
	dispensed	money
BANK	name	string
	location	address
CUSTOMER	name	string
	residence	address
CASH CARD	password	string
	serial number	int
	limit	money
ACCOUNT	blocked	bool
	balance	money

Table 5.1: Attributes for the ER diagram for the automated teller machine example

$$\begin{aligned} \text{relship} &: \textit{ROLE} \to \textit{R-TYPE} \\ \text{entity} &: \textit{ROLE} \to \textit{E-TYPE} \\ \text{owner} &: \textit{ATTR} \to \mathcal{F}(\textit{E-TYPE}) \\ \text{domain} &: \textit{ATTR} \to \textit{D-TYPE} \end{aligned}$$

such that $\forall R \in$ *R-TYPE with* $\text{participants}(R) =< E_1, \ldots, E_m >$ *it holds that* $\forall 1 \leq i \leq m, \exists P_i \in$ *ROLE* : $\text{relship}(P_i) = R \wedge \text{entity}(P_i) = E_i$.

□

As in the case of EER instances (cf. Definition 2.8), for the definition of ER instances we need a universe of entities (cf. Definition 2.7). Additionally, since complex values are no longer allowed as attribute value, there is no longer a need for data type signatures. In replacement, we assume that for each data type in a given ER scheme, a (countably infinite) set of values of that type is given. Formally:

Definition 5.2 *Given an ER scheme $\mathcal{S}$, the function μ[D-TYPE] assigns to each data type $D \in$ D-TYPE a countably infinite set of* values*, such that different members of D-TYPE are mapped to disjoint sets. For any $D \in$ D-TYPE, μ[D-TYPE](D) includes a* typed null value $\perp_D$.
□

Null values are used as *default attribute values* by the GOOD/ER operation that allows the addition of new entities to an ER instance.

The following shorthand notation for relationships and attributes is a simplification of the one for EER schemes:

Notation 5.3 *For $R \in$ R-TYPE with* $\text{participants}(R) =< E_1, \ldots, E_m >$*, and $P_i \in$ ROLE$(1 \leq i \leq m)$ with* $\text{relship}(P_i) = R$ *and* $\text{entity}(P_i) = E_i$*, we denote $R(P_1 : E_1, \ldots, P_m : E_m) \in$ R-TYPE and $P_i : R \to E_i \in$ ROLE.*

For $A \in$ ATTR with $\text{owner}(A) \ni E$ *and* $\text{domain}(A) = D$ *we denote $A : E \to D$.*

Apart from the absence of inheritance, components and complex attributes, ER instances differ in one more significant aspect from EER instances, namely in the way relationships are formalized. In the EER model, relationships are just n-tuples of entities, hence they do not contain any information about roles (apart from the fact that the role which a given entity plays, may be derived from the entity's position in the n-tuple). As a consequence, one and the same relationship can belong to multiple relationship types. Consider for instance the (E)ER diagram depicted in Figure 5.2. Let us first consider this diagram as an EER diagram. Let e_1 be an entity of type `E1`, and let e_2 be an entity of type `E2`. According to Definition 2.8, the pair (e_1, e_2) may be an element of both $\mu[\text{R-TYPE}](\texttt{R1})$ and $\mu[\text{R-TYPE}](\texttt{R2})$.

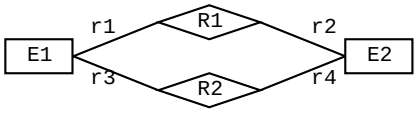


Figure 5.2: An (E)ER diagram with multiple relationship types between entity types

Further on in this chapter, however, we want to be able to look upon ER instances (to be defined shortly) as *graphs*. If an ER instance contains two different relationships, then this graph should contain two clearly distinct nodes, one for each relationship. If, however, one and the same n-tuple of entities may belong to an arbitrary number of relationship types (as illustrated above), then this is not the case. Hence there is a need to incorporate information about roles *explicitly* into the relationships. Note that this does not mean that relationships are "objectified" in the ER variant used in this chapter. Even with role names incorporated into them, relationships still do not have an identity of their own. It is for instance not possible to have two or more relationships of the same type in which the same entities play the same roles.

All of the above is formalized in the following definition of ER instances:

Definition 5.4 (ER Instance) *An* ER instance *$\mathcal{I}$ over an ER scheme $\mathcal{S}$ consists of the following four functions:*

- *μ[E-TYPE], which maps each entity type $E \in$ E-TYPE to a finite subset of $\mathcal{E}_E$;*
- *μ[R-TYPE], which maps each relationship type $R(P_1 : E_1, \ldots, P_m : E_m) \in$ R-TYPE to a finite set of* relationships*, such that $\mu[\text{R-TYPE}](R) \subseteq \{P_1\} \times \mu[\text{E-TYPE}](E_1) \times \ldots \times \{P_m\} \times \mu[\text{E-TYPE}](E_m)$;*
- *μ[ROLE], which maps each role $P_i : R \to E_i$ to a function $\mu[\text{ROLE}](P_i) : \mu[\text{R-TYPE}]\ (R) \to \mu[\text{E-TYPE}](E)$ such that for each $r = (P_1, e_1, \ldots, P_m, e_m) \in \mu[\text{R-TYPE}](R)$ and for each $1 \leq i \leq m$, $\mu[\text{ROLE}](P_i)(r) = e_i$;*

- μ[*ATTR*], *which maps each attribute* $A \in$ *ATTR to a function* μ[*ATTR*]$(A) : \bigcup_{E \in \mathrm{owner}(A)} \mu$[*E-TYPE*]$(E) \rightarrow \mu$[*D-TYPE*]$(\mathrm{domain}(A))$.

□

We introduce the following alternative notation for relationships:

Notation 5.5 *A relationship* $(P_1, e_1, \ldots, P_m, e_m) \in \mu$[*R-TYPE*]$(R)$ *for some relationship type* $R(P_1 : E_1, \ldots, P_m : E_m) \in$ *R-TYPE is also denoted* $(P_1 : e_1, \ldots, P_m : e_m)$

As an illustration, we present part of an ER instance over the example ER scheme.

Example 5.6

$$
\begin{array}{lcl}
\mu[\text{E-TYPE}](\texttt{ATM}) & = & \{e_1, e_2\} \\
\mu[\text{E-TYPE}](\texttt{BANK}) & = & \{e_3, e_4\} \\
\mu[\text{R-TYPE}](\texttt{owned by}) & = & \{(\texttt{teller} : e_1, \texttt{owner} : e_3), \\
 & & (\texttt{teller} : e_2, \texttt{owner} : e_4)\} \\
\mu[\text{ROLE}](\texttt{teller})((\texttt{teller} : e_1, \texttt{owner} : e_3)) & = & e_1 \\
\mu[\text{ROLE}](\texttt{owner})((\texttt{teller} : e_1, \texttt{owner} : e_3)) & = & e_3 \\
\mu[\text{ROLE}](\texttt{teller})((\texttt{teller} : e_2, \texttt{owner} : e_4)) & = & e_2 \\
\mu[\text{ROLE}](\texttt{owner})((\texttt{teller} : e_2, \texttt{owner} : e_4)) & = & e_4 \\
\mu[\text{ATTR}](\texttt{dispensed})(e_1) & = & 1234.50
\end{array}
$$

In this instance, two ATMs are present, each of which is owned by a different bank. From one of these ATMs, an amount of 1234.50 has been dispensed.

5.2 GOOD/ER: a Graph-Oriented Object Database language

In this Section, we present and formally define the Graph-Oriented Object Database language for the ER model, in brief, GOOD/ER. GOOD/ER is a *database programming language*.

A program in GOOD/ER consists of a sequence of operations, to be applied to an ER instance. A GOOD/ER operation in turn consists basically of two parts:

1. a *pattern*, describing the configurations of entities, relationships and values to which the operation should be applied. Note the close correspondence to GOQL/EER queries, which also include a pattern, describing configurations, parts of which should be returned as the result of the query.

2. a *manipulation* to be performed on all parts of the database instance that *match* the pattern of the operation. Such a manipulation consists of the addition or the deletion of certain entities, relationships, values, attributes, and/or roles.

Even though both operations in GOOD/ER and queries in GOQL/EER have a pattern as their basic constituent, the notion of pattern matching in GOQL/EER is merely an intuitive one. Indeed, in Section 4.2 the actual semantics of GOQL/EER queries is defined by means of a translation to the textual query language SQL/EER. In other words, the actual *evaluation* of queries is not treated in the definition of GOQL/EER, since this is taken care of in the definition of SQL/EER.

In GOOD/ER however, we incorporate this notion of pattern matching *explicitly* in the definition of the language. Concretely, patterns are formalized as attributed, directed, node- and edge-labeled graphs, which, when executing the operation of which they are part, are to be matched (in the formal graph-theoretical sense of the word) against the database instance. This in turn shows the need for a "graph-oriented look" on ER instances.

Before formalizing the latter, however, we first formally define the kind of graphs used in formalizing both this graph-oriented look on ER instances as well as (ER) patterns. Whereas the kind of graphs used in formalizing GOQL/EER queries was defined in an *operational* manner, namely by means of a PROGRES specification, the graphs used in this chapter for formalizing both ER instance graphs as well as the syntax of GOOD/ER operations are defined *declaratively*, by means of the following definition:

Definition 5.7 (Graph) *Let NL be a countably infinite set of* node labels*, EL a countably infinite set of* edge labels*, and AV a countably infinite set of* attribute values*. A* graph *over (NL,EL,AV) is a four-tuple* (V, W, λ, π)*, where* V *is the set of* nodes*,* $W \subset V \times EL \times V$ *the set of* edges*,* $\lambda : V \to NL$ *the* node labeling function*, and* $\pi : V \nrightarrow AV$ *the (partial)* node attribution function*. The labeling function* λ *is extended to* W *as* $\lambda((v_1, \alpha, v_2)) := \alpha$*.*
□

Notation 5.8 *The components of a given graph* G *are denoted respectively* V_G*,* W_G*,* λ_G *and* π_G*.*

Given this definition of graphs, we now capture, by means of the definition of *ER instance graphs*, how an ER instance may be looked upon as a graph. Briefly, entities, relationships and values play the part of nodes, while attributes and roles correspond to edges between the appropriate nodes.

Definition 5.9 (ER Instance Graph) *Given an ER instance* $\mathcal{I}$ *over an ER scheme* $\mathcal{S}$*, the* instance graph *corresponding to* $\mathcal{I}$ *is the graph* (V, W, λ, π) *over* $(\textit{E-TYPE} \cup \textit{R-TYPE} \cup \textit{D-TYPE}, \textit{ROLE} \cup \textit{ATTR}, \bigcup_{d \in \textit{D-TYPE}} \mu[\textit{D-TYPE}](d))$ *with*

- $$V = \bigcup_{E \in \textit{E-TYPE}} \mu[\textit{E-TYPE}](E) \cup \bigcup_{R \in \textit{R-TYPE}} \mu[\textit{R-TYPE}](R) \cup \bigcup_{A \in \textit{ATTR}} \mathrm{rng}(\mu[\textit{ATTR}](A))$$
- λ *is defined on* V *as*
  - $e \in \mu[\textit{E-TYPE}](E) \Rightarrow \lambda(e) = E$
  - $r \in \mu[\textit{R-TYPE}](R) \Rightarrow \lambda(r) = R$
  - $d \in \mathrm{rng}(\mu[\textit{ATTR}](a)) \subset \mu[\textit{D-TYPE}](D) \Rightarrow \lambda(d) = D$

- $W = \{(r, P, e) \mid \mu[\textit{ROLE}](P)(r) = e\} \cup \{(e, A, d) \mid \mu[\textit{ATTR}](A)(e) = d\}$
- π *is the identity on* $\bigcup_{A \in \textit{ATTR}} \mathrm{rng}(\mu[\textit{ATTR}](A))$*, and is undefined elsewhere.*

□

The attribution function π is in a sense obsolete in this definition (being merely the identity on the values of the ER instance), but will turn out to be useful in the subsequent definition of embeddings of patterns, in which unattributed nodes labeled with a data type represent an "unknown" (or "don't-care") value of that type.

Because of the one-to-one correspondence between ER instances and ER instance graphs, in the remainder of this chapter we treat both terms as synonyms (e.g., in phrases such as "the nodes of an instance" or "the entities in a graph").

As an example, Figure 5.3 shows the ER instance graph corresponding to the ER instance of Example 5.6. In this graphical representation of ER instance graphs, the same conventions are used as in (E)ER diagrams, i.e., nodes corresponding to entities are shown as rectangles, nodes corresponding to relationships are shown as diamonds, and nodes corresponding to values are shown as ovals. In the case of the latter kind of nodes, the value itself is shown near the node, thereby "visualizing" the attribution function π. In Figure 5.3, this is illustrated with the `money`-labeled oval, and the actual value 1234.50. For clarity, null values are omitted both from this ER instance graph as well as from most other instance graphs used in the remainder of this chapter.

To show the correspondence between this figure and Example 5.6, the "identifiers" used in Example 5.6 are shown near the entities they correspond to.

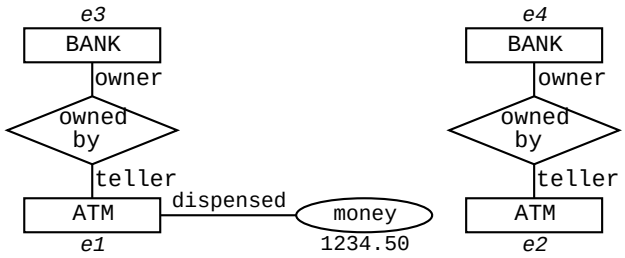


Figure 5.3: An ER instance graph corresponding to the ER instance of Example 5.6

5.2.1 Patterns and Embeddings

As mentioned previously, the basic component of an operation in GOOD/ER is a *pattern*, describing the configuration(s) in the instance (graph) to which the operation is to be applied. A pattern is basically a graph, with an "abstract" set of nodes (as opposed to the nodes of an ER instance graph, which are the "actual" entities, relationships and values in the corresponding ER instance) and labeled in accordance with a given ER scheme:

Definition 5.10 (ER Pattern) *A* pattern *over an ER scheme* $\mathcal{S}$ *is a graph* (V, W, λ, π) *over* $(\textit{E-TYPE} \cup \textit{R-TYPE} \cup \textit{D-TYPE}, \textit{ROLE} \cup \textit{ATTR}, \bigcup_{D \in \textit{D-TYPE}} \mu[\textit{D-TYPE}](D))$ *such that* $\forall (n, \alpha, m) \in W$:

1. $\alpha \in$ *ROLE* $\Rightarrow (\lambda(n) = \text{relship}(\alpha) \wedge \lambda(m) = \text{entity}(\alpha))$
2. $\alpha \in$ *ATTR* $\Rightarrow (\lambda(n) \in \text{owner}(\alpha) \wedge \lambda(m) = \text{domain}(\alpha))$

□

The correspondence between the pattern of an operation and the parts of an ER instance graph which it matches, is formalized by the notion of an *embedding* of a pattern in an instance graph. Such an embedding is a mapping from the nodes of the pattern to the nodes of the instance graph, preserving typing (i.e., labels) and structure of the pattern.

We define embeddings in a somewhat more general context, namely for graphs in general, since further on, we also need the possibility to embed one instance into another.

Definition 5.11 (Embedding) *An* embedding *of a graph $\mathcal{G}$ in a graph $\mathcal{H}$ is a total function $m : V_\mathcal{G} \to V_\mathcal{H}$ that satisfies*

1. $\forall v \in V_\mathcal{G} : \lambda_\mathcal{H}(m(v)) = \lambda_\mathcal{G}(v)$
2. $\forall v \in V_\mathcal{G} : \pi_\mathcal{G}(v)$ *is defined* $\Rightarrow \pi_\mathcal{H}(m(v)) = \pi_\mathcal{G}(v)$
3. $\forall (n, \alpha, m) \in E_\mathcal{G} : (m(n), \alpha, m(m)) \in E_\mathcal{H}$

An embedding is extended to $E_\mathcal{G}$ as $m((n, \alpha, m)) := (m(n), \alpha, m(m))$.

□

As an illustration, Figure 5.4 shows an ER pattern over the ER scheme depicted in Figure 5.1. It may be easily verified that this ER pattern may be embedded exactly once in the ER instance graph depicted in Figure 5.3, namely by mapping the ATM-node, the BANK-node and the money-node in the pattern to respectively the ATM-node with identifier *e1*, the BANK-node with identifier *e3* and the single money-node in the instance graph.

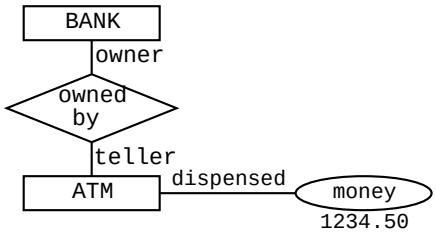


Figure 5.4: An ER pattern (I)

The ER pattern shown in Figure 5.5 on the other hand, may be embedded twice in the ER instance graph depicted in Figure 5.3. The first embedding maps the ATM-node and the BANK-node in the pattern to respectively the ATM-node with identifier *e1* and the BANK-node with identifier *e3* in the instance graph. The second embedding maps the ATM-node and the BANK-node in the pattern to respectively the ATM-node with identifier *e2* and the BANK-node with identifier *e4* in the instance graph.

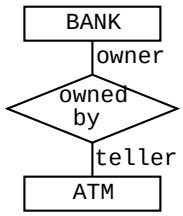


Figure 5.5: An ER pattern (II)

Note that the latter embedding may be *extended* to the embedding of the pattern depicted in Figure 5.4. Extending embeddings is a key notion in the definition of the GOOD/ER operations. To formalize this notion, we first have to define *subgraphs*:

Definition 5.12 (Subgraph) *A graph $G = (V, W, \lambda, \pi)$ is a* subgraph *of a graph $G' = (V', W', \lambda', \pi')$ if $V \subseteq V'$, $W \subseteq W'$, $\lambda = \lambda'|_V$ and $\pi = \pi'|_V$.*
□

Since both ER instance graphs and ER patterns are graphs, we use the terms *subinstance* and *subpattern* as synonyms for subgraph, wherever appropriate.

Definition 5.13 (Embedding Extension) *Let $\mathcal{I}$ be an ER instance graph, and $\mathcal{P}$ and $\mathcal{P}'$ be ER patterns such that $\mathcal{P}$ is a subpattern of $\mathcal{P}'$. Let m and m' be embeddings of respectively $\mathcal{P}$ and $\mathcal{P}'$ in $\mathcal{I}$. Then m' is an* extension *of m if $m'|_{V_\mathcal{P}} = m$.*
□

As a third example of embeddings, Figure 5.6 shows a pattern that may be embedded *four* times in the ER instance graph depicted in Figure 5.3. This fact follows readily from the observation that an embedding is not necessarily an injective mapping (as was the case with the embeddings of the patterns in Figures 5.4 and 5.5). Concretely, an embedding of this pattern in the considered instance may either

- map the leftmost ATM-node to the ATM-node with identifier *e1* and the rightmost ATM-node to the ATM-node with identifier *e2*;
- map the leftmost ATM-node to the ATM-node with identifier *e2* and the rightmost ATM-node to the ATM-node with identifier *e1*;
- map both ATM-nodes to the ATM-node with identifier *e1*;
- map both ATM-nodes to the ATM-node with identifier *e2*.

The effect of these four embeddings on the remaining four nodes of the pattern follows straightforwardly from their effect on the ATM-nodes.

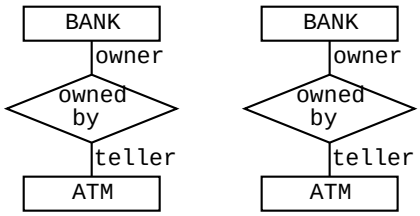


Figure 5.6: An ER pattern (III)

5.2.2 Basic Operations

We now illustrate and formally define the six basic operations offered by GOOD/ER. Three of these operations allow the *addition* of certain elements to an ER instance, one allows the *update* of attributes, while the remaining two allow the *deletion* of certain elements from an ER instance.

By means of an *entity addition*, new entities may be added to an ER instance, with given attribute values and relationships with entities already present in the ER instance. As an example, suppose we wish to represent in the database the fact that all customers living in Antwerp get a new cash card with an initial limit of 5000, and the customers name as default password. This is done using the entity addition depicted in Figure 5.7.

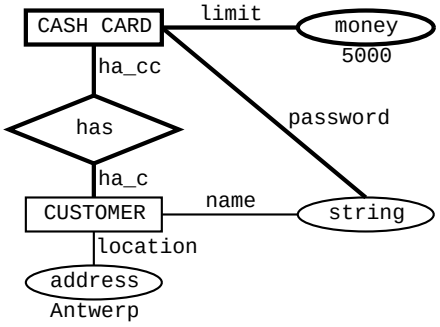


Figure 5.7: An entity addition

Just as any other GOOD/ER operation, an entity addition consists basically of two parts: a pattern, and a part indicating the operation to be performed, in this case, the elements to be added to the ER instance. In the example entity addition of Figure 5.7, the pattern consists of all nodes and edges in thin lines. This pattern matches all customers living in Antwerp, together with their name and location.

The operation part of this (and any) entity addition consists of the parts drawn in bold lines. It indicates that, for each `CUSTOMER`-entity matching the pattern, a `CASH CARD`-entity should be added to the considered ER instance, linked by means of a `has`-relationship to the `CUSTOMER`-entity. In addition, the `password`-attribute of any newly added `CASH CARD`-entity should be set to the `CUSTOMER`s `name`, while its `limit`-attribute should be set to the `money` value 5000.

Note that the latter money value should not be included in the pattern, since this would make the effect of the entity addition depend on the presence or absence of this money value (as attribute-value

of some other entity or relationship) in the considered ER instance. Indeed, since if the money value would be part of the pattern, then on application of the operation, no entities would be added if the money value 5000 was absent from the considered ER instance. This would clearly be an undesirable semantics for this operation, since its effect should solely depend on the presence (or absence) in the ER instance of customers living in Antwerp.

We are now ready for the formal definition of entity additions:

Definition 5.14 (Entity Addition (syntax)) *Given an ER scheme $\mathcal{S}$, an* entity addition *over $\mathcal{S}$ is denoted syntactically as ENTADD$[\mathcal{P}, E, VA, AT, RE, RO]$ with the following input parameters:*

- *an ER pattern $\mathcal{P} = (V, W, \lambda, \pi)$ over $\mathcal{S}$*
- *$E \in$ E-TYPE*
- *VA $= \{(A_1, v_1), \ldots, (A_m, v_m)\} \subset$ ATTR $\times V$ such that $\forall 1 \leq i \leq m : \mathrm{domain}(A_i) = \lambda(v_i) \wedge \mathrm{owner}(A_i) \ni E$*
- *AT $= \{(A_{m+1}, v_{m+1}), \ldots, (A_{m+l}, v_{m+l})\} \subset$ ATTR $\times \bigcup_{D \in \text{D-TYPE}} \mu[\text{D-TYPE}](D)$ such that $\forall 1 \leq i \leq l, \exists D_i \in$ D-TYPE $: v_{m+i} \in \mu[\text{D-TYPE}](D_i) \wedge \mathrm{domain}(A_{m+i}) = D_i \wedge \mathrm{owner}(A_{m+i}) \ni E$*
- *RE $=< R_1, \ldots, R_n > \in$ R-TYPE$^+$ such that $\forall 1 \leq i \leq n : E \in \mathrm{participants}(R_i)$*
- *RO $= \{(< (P_1^1, e_1^1), \ldots, (P_{s_1}^1, e_{s_1}^1) >, P_{t_1}^1), \ldots, (< (P_1^n, e_1^n), \ldots, (P_{s_n}^n, e_{s_n}^n) >, P_{t_n}^n)\}$ $\in \mathcal{F}((\text{ROLE} \times V)^+ \times \text{ROLE})$ such that $\forall 1 \leq i \leq n, \mathrm{relship}^{-1}(R_i) = \{P_1^i, \ldots, P_{s_i}^i, P_{t_i}^i\}$ and $\mathrm{participants}(R_i) =< \lambda(e_1^i), \ldots, \lambda(e_{s_i}^i) >$ with E inserted at the position corresponding to role $P_{t_i}^i$.*

□

In terms of the notation of Definition 5.14, Figure 5.7 shows the entity addition *ENTADD*$[\mathcal{P}, E, \mathrm{VA}, \mathrm{AT}, \mathrm{RE}, \mathrm{RO}]$ with

- the pattern $\mathcal{P} = (V, W, \lambda, \pi)$ consisting of
  - $V = \{e_1, v_1, v_2\}$
  - $W = \{(e_1, \text{name}, v_1), (e_1, \text{location}, v_2)\}$
  - λ is defined by the following table:

node	label
e_1	CUSTOMER
v_1	string
v_2	address

  - π maps v_2 to the string "Antwerp"
- the entity type E equal to CASH CARD, being the type of the entities to be added
- the set VA of (attribute, value)-pairs with values taken from $\mathcal{P}$ equal to $\{(\text{password}, v_1)\}$
- the set AT of (attribute, value)-pairs with values which are not part of $\mathcal{P}$ equal to $\{(\text{limit}, 5000)\}$

- the list RE of types of relationships in which the newly created entities participate equal to $<$ has $>$
- the set RO containing both the roles played by the newly created entities in the relationships of RE, as well as the entities in the pattern that play the remaining roles, equal to $\{(<(\text{ha\_c}, e_1)>, \text{ha\_cc})\}$.

Definition 5.15 (Entity Addition (semantics)) *Let ENTADD$[\mathcal{P}, E, VA, AT, RE, RO]$ be an entity addition over an ER scheme $\mathcal{S}$ as in Definition 5.14.*

Let $\mathcal{P}'$ be the ER pattern (V', W', λ', π') where

- *V' equals V extended with $1+n+l$ new nodes*[3] *$e, r_1, \ldots, r_n, v'_{m+1}, \ldots, v'_{m+l}$, labeled by λ' with respectively $E, R_1, \ldots, R_n, d_1, \ldots, d_l$. On V, λ' equals λ.*
- *W' equals W extended with role edges from the nodes $r_1, \ldots, r_n$ to the nodes $e, e^1_1, \ldots, e^n_{s_n}$, as well as attribute edges from e to the nodes $v_1, \ldots, v_m, v'_{m+1}, \ldots, v'_{m+l}$.*
- *π' attributes $v'_{m+1}, \ldots, v'_{m+l}$ with respectively $v_{m+1}, \ldots, v_{m+l}$ and equals π on V.*

The result of applying the entity addition ENTADD$[\mathcal{P}, E, VA, AT, RE, RO]$ to an ER instance $\mathcal{I}$ over $\mathcal{S}$ is defined as one possible outcome of the following algorithm:

$\mathcal{I}' := \mathcal{I}$;
for each *embedding m of $\mathcal{P}$ in $\mathcal{I}$* **do**
  if not exists *an embedding m' of $\mathcal{P}'$ in $\mathcal{I}$ which extends m*
  then *add to $V_{\mathcal{I}'}$ a new entity $et \in \mathcal{E}_E - V_{\mathcal{I}'}$ of type E*
    the values $v_{m+1}, \ldots, v_{m+l}$
    the relationships $(P^1_1 : e^1_1, \ldots, P^1_{t_1} : et, \ldots, P^1_{s_1} : e^1_{s_1})$ through
      $(P^n_1 : e^n_1, \ldots, P^n_{t_n} : et, \ldots, P^n_{s_n} : e^n_{s_n})$
  add to $E_{\mathcal{I}'}$ the edges (et, a_1, v_1) through (et, a_{m+l}, v_{m+l})
    the role-edges of the newly added relationships
    for each $a \in ATTR$ such that $E \in \text{owner}(a)$ and $a \notin \{a_1, \ldots, a_{m+l}\}$,
      add $(et, a, \perp_{\text{domain}(a)})$
return $(\mathcal{I}')$

□

The ER pattern $\mathcal{P}' = (V', W', \lambda', \pi')$ from Definition 5.15 is obtained by extending the pattern $\mathcal{P}$ with nodes and edges for the parts to be added by the entity addition. In the case of the example, $\mathcal{P}'$ consists of

- $V' = \{e_1, e_2, r_1, v_1, v_2, v_3\}$
- $W' = \{(e_1, \text{name}, v_1), (e_1, \text{location}, v_2), (r_1, \text{ha\_c}, e_1), (r_1, \text{ha\_cc}, e_2), (e_2, \text{password}, v_1), (e_2, \text{limit}, v_3)\}$

[3] That is, nodes not already present in V.

- λ' is defined by the following table:

node	label
e_1	CUSTOMER
e_2	CASH CARD
r_1	has
v_1	string
v_2	address
v_3	money

- π' is defined by the following table:

node	attribute
v_2	Antwerp
v_3	5000

The definition of the semantics of entity additions may best be explained by applying the example entity addition of Figure 5.7 to the sample ER instance shown in Figure 5.8.

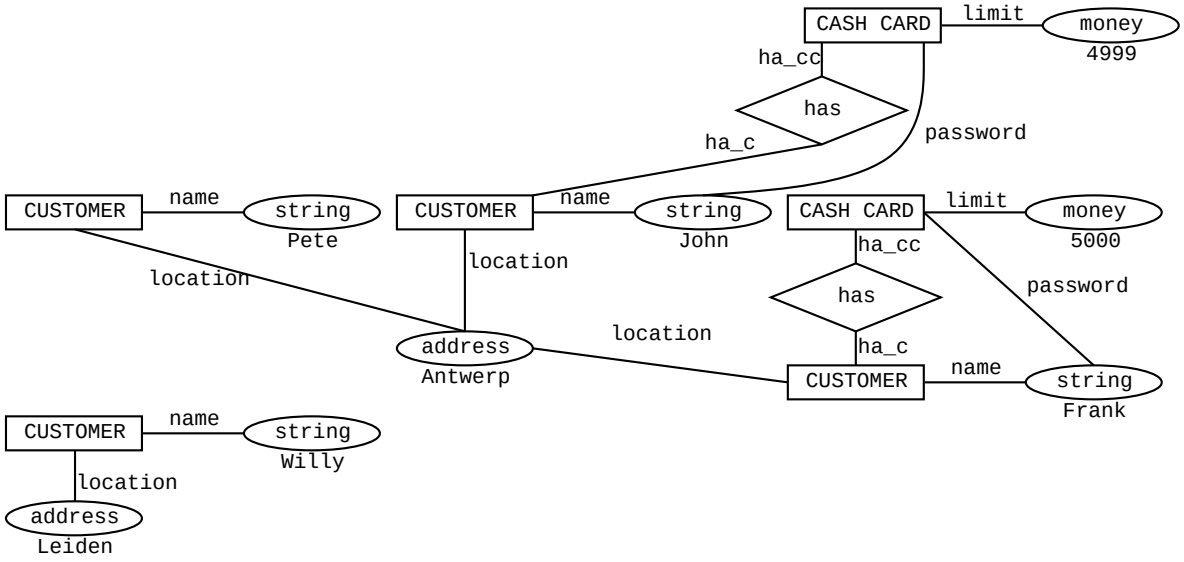


Figure 5.8: Input instance to the example entity addition of Figure 5.7

It may be verified that the pattern of the entity addition of Figure 5.7 has three embeddings in this ER instance, namely for the customers named Pete, John and Frank, since they all live in Antwerp. The pattern doesn't match the customer named Willy, since he lives in Leiden.

These three embeddings are treated by the algorithm defining the semantics of an entity addition in some arbitrary order. Since this algorithm uses the set of embeddings of the pattern in the input instance, and the condition on the extendibility of the pattern of the operation is also checked on the input instance (and not on the intermediate result $\mathcal{I}'$), the algorithm is clearly independent of the chosen order. Suppose that on applying the considered sample entity addition, the algorithm starts with the embedding corresponding to customer Frank. Then the condition on embedding extensions in the algorithm will fail, since Frank already has a cash card with his name as password, and a limit of 5000. Next, the algorithm might consider the embedding corresponding to customer John. Since John has a

cash card with his name as password, but with a limit of 4999, he is assigned a new cash card. Likewise, Pete is assigned a new cash card, since he did not have one in the input instance.

In summary, an application of the entity addition of Figure 5.7 to the input instance of Figure 5.8 (the resulting instance of which is shown in Figure 5.9) adds (among others) two new entities of type CASH CARD to the instance. The fact that these entities are taken "at random" from the universe of entities $\mathcal{E}$ explains the phrase "…one *possible* outcome of the following algorithm…". Indeed, since the algorithm does not specify *which* entities are taken from $\mathcal{E}$, the outcome of this algorithm is a priori non-deterministic.

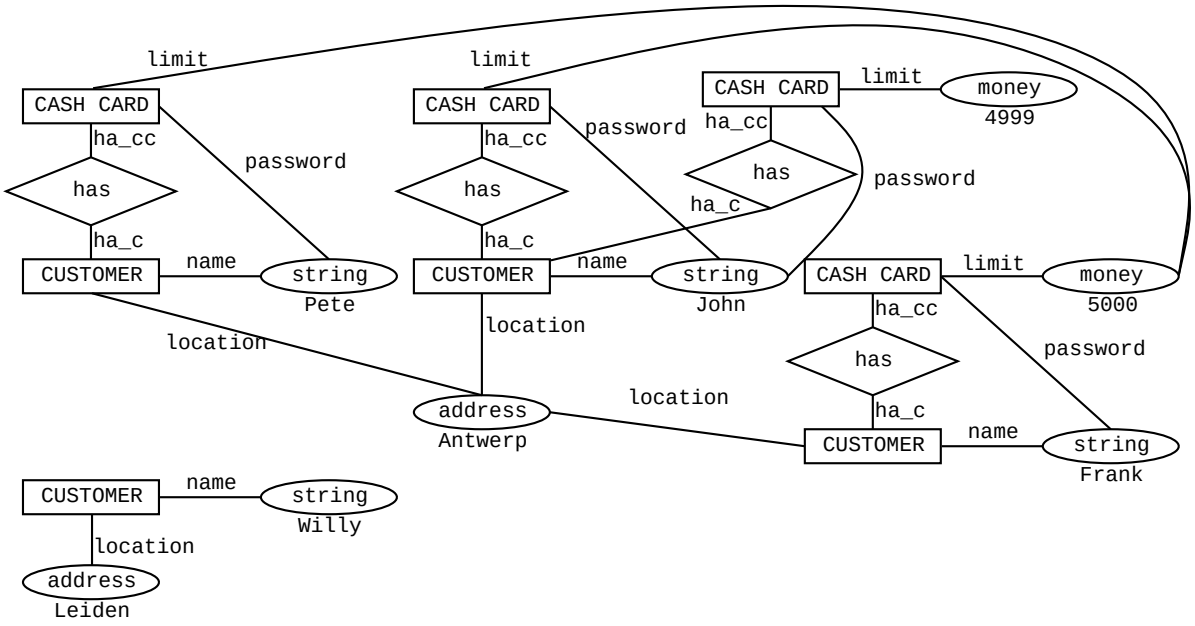


Figure 5.9: Output instance of the example entity addition

However, two possible outcomes of the algorithm are strongly related. On one hand, they both include the input instance. On the other hand, they indeed only differ in the "identity" of the newly added entities. This similarity between possible outcomes of an entity addition may be formally captured as follows. First, the intuitive notion of ER instances differing only in the identity of their entities is formalized by means of the notion of *isomorphism*:

Definition 5.16 (Isomorphism) *Two graphs $\mathcal{I}$ and $\mathcal{I}'$ are* isomorphic*, if $\mathcal{I}$ can be embedded injectively in $\mathcal{I}'$ and vice versa. An injective embedding of a graph into an isomorphic graph is called an* isomorphism.

□

Next, the similarity between two possible outcomes of an entity addition may be formalized using the notion of *$\mathcal{I}$-isomorphism*. As opposed to general isomorphism, $\mathcal{I}$-isomorphism captures the fact that two ER instances are identical on some common subinstance $\mathcal{I}$, and differ elsewhere only in the identity of their entities:

Definition 5.17 ($\mathcal{I}$-isomorphism) *Let $\mathcal{I}$ be a graph. Two graphs $\mathcal{J}$ and $\mathcal{J}'$ are $\mathcal{I}$-isomorphic if there exists an isomorphism from $\mathcal{J}$ to $\mathcal{J}'$ that is the identity on $\mathcal{I} \cap \mathcal{J}$, and whose inverse is the identity on $\mathcal{I} \cap \mathcal{J}'$.*
□

Given this notion, the following lemma may be easily verified:

Lemma 5.18 *Any two outcomes of the application of an entity addition to some instance $\mathcal{I}$ are $\mathcal{I}$-isomorphic. Conversely, if an instance $\mathcal{I}_1$ is $\mathcal{I}$-isomorphic to an instance $\mathcal{I}_2$ which is the outcome of the application of an entity addition to some instance $\mathcal{I}$, then $\mathcal{I}_1$ itself is also a possible outcome of the application of that entity addition to $\mathcal{I}$.*■

A special case of isomorphisms, namely isomorphisms of an instance onto itself, may be used to gain some insight into the set of embeddings of a pattern in an instance.

Definition 5.19 (Automorphism) *Let $\mathcal{I}$ be a graph. An* automorphism *of $\mathcal{I}$ is an isomorphism from $\mathcal{I}$ onto itself. $Aut(\mathcal{I})$ is the group of all automorphisms of $\mathcal{I}$.*
□

Using the notion of automorphisms, the following lemma may be easily seen to hold:

Lemma 5.20 *The composition of an embedding of a graph $\mathcal{I}'$ in a graph $\mathcal{I}$ and an automorphism of $\mathcal{I}$, is itself an embedding of $\mathcal{I}'$ in $\mathcal{I}$.*

Proof The proof follows readily from the fact that isomorphisms, and hence automorphisms, are themselves embeddings, and the fact that the composition of two embeddings is also an embedding. ■

This lemma will turn out to be of crucial importance to the study of the expressive power of the GOOD/ER language in Section 5.3.

Besides adding relationships involving newly added entities, it should also be possible to simply add relationships involving only entities already present in an instance. Hence we define the following GOOD/ER operation:

Definition 5.21 (Relationship Addition) *Given an ER scheme $\mathcal{S}$, a* relationship addition *over $\mathcal{S}$ takes as input*

- *an ER pattern $\mathcal{P} = (V, W, \lambda, \pi)$ over $\mathcal{S}$*
- *$R \in$ R-TYPE*
- *RO $=< e_1, \ldots, e_q > \in V^+$ such that $R(P_1 : \lambda(e_1), \ldots, P_q : \lambda(e_q)) \in$ R-TYPE*

Let $\mathcal{P}'$ be the ER pattern (V', W', λ', π) where

- *V' equals V extended with a new node r labeled R by λ'. On V, λ' equals λ.*

- *W' equals W extended with role-edges from r to the nodes in RO.*

The result of applying the relationship addition RELADD$[\mathcal{P}, R, RO]$ to an ER instance $\mathcal{I}$ over $\mathcal{S}$ is defined as the outcome of the following algorithm:

$\mathcal{I}' := \mathcal{I}$;
for each *embedding m of $\mathcal{P}$ in $\mathcal{I}$* **do**
$V_{\mathcal{I}'} = V_{\mathcal{I}'} \cup \{(r_1 : m(e_1), \ldots, r_q : m(e_q))\}$
$W_{\mathcal{I}'} = W_{\mathcal{I}'} \cup \{$*the role-edges of the newly added relationship*$\}$
return $(\mathcal{I}')$

□

Figure 5.10 shows an example relationship addition over the scheme of Figure 5.1. The result of this operation applied to some ER instance is that, if some customer has a cash card and holds an account, the cash card is "granted access" to the account.

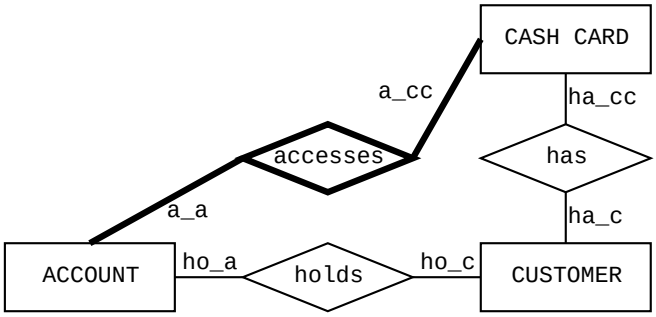


Figure 5.10: An example relationship addition

In terms of the notation of Definition 5.21, Figure 5.10 shows the relationship addition *RELADD*$[\mathcal{P}, R, RO]$ with

- the pattern $\mathcal{P} = (V, W, \lambda, \pi)$ consisting of
  - $V = \{e_1, e_2, e_3, r_1, r_2\}$
  - $W = \{(r_1, \text{ha\_cc}, e_1), (r_1, \text{ha\_c}, e_2), (r_2, \text{ho\_c}, e_2), (r_2, \text{ho\_a}, e_3)\}$
  - λ is defined by the following table:

node	label
e_1	CASH CARD
e_2	CUSTOMER
e_3	ACCOUNT
r_1	has
r_2	holds

  - π is undefined on V.
- the relationship type R equal to `accesses`, being the type of the relationships to be added

- the list RO containing the entities in the pattern that are to play the roles of the newly added relationships, equal to $< e_1, e_3 >$.

The pattern $\mathcal{P}'$ is obtained in a similar way as with the example entity addition.

The semantics of the operation is that for each embedding of the pattern in the given ER instance, mapping the CASH CARD and ACCOUNT nodes in the pattern to respectively entities e' and e'' in the instance, the accesses-relationship $(\text{a\_cc}, e', \text{a\_a}, e'')$ is added. Note that since this relationship is uniquely determined by the combination of the roles and the entities in the input instance, there is no non-determinism involved in the algorithm defining the semantics of a relationship addition, as opposed to the definition of entity additions. In addition, there is no need for a check on the extensibility of the considered embedding, since adding a relationship to an instance in which it is already present, simply has no effect.

We now turn our attention towards attributes. Since attributes are always defined (even if they are set to the null value $\perp_d$) it makes no sense to consider either attribute deletions or additions. The only operation on attributes that does make sense is a modification or *update* of the value of some attribute:

Definition 5.22 (Attribute Update) *Given an ER scheme $\mathcal{S}$, an* attribute update *over $\mathcal{S}$ takes as input*

- *an ER pattern $\mathcal{P} = (V, W, \lambda, \pi)$ over $\mathcal{S}$*
- *$A \in$ ATTR*
- *$e \in V$ such that $\text{owner}(A) \ni \lambda(e)$*
- *one of the following:*
  1. *$v \in V$ such that $\text{domain}(A) = \lambda(v)$ and $\pi(v)$ is defined*
  2. *$v \in \mu[$D-TYPE$](\text{domain}(A))$*

If $v \in V$, then the result of applying the attribute update ATTUPD$[\mathcal{P}, A, e, v]$ to an ER instance $\mathcal{I}$ over $\mathcal{S}$ is defined as the outcome of the following algorithm:

$\mathcal{I}' := \mathcal{I}$;
for each *embedding m of $\mathcal{P}$ in $\mathcal{I}$* **do**
  $W_{\mathcal{I}'} = W_{\mathcal{I}'} - \{(m(e), A, x)\}$ *(for some $x \in V_{\mathcal{I}}$)*
  $W_{\mathcal{I}'} = W_{\mathcal{I}'} \cup \{(m(e), A, m(v))\}$
$V_{\mathcal{I}'} = V_{\mathcal{I}'} - \{v \mid \lambda_{\mathcal{I}}(v) \in$ D-TYPE$\wedge\ \nexists e \in V_{\mathcal{I}'}, A' \in$ ATTR $: (e, A', v) \in W_{\mathcal{I}'}\}$
return $(\mathcal{I}')$

If $v \notin V$, then the result of applying the attribute update ATTUPD$[\mathcal{P}, A, e, v]$ to the instance $\mathcal{I}$ is defined as the outcome of the following algorithm:

$\mathcal{I}' := \mathcal{I}$;
for each *embedding m of $\mathcal{P}$ in $\mathcal{I}$* **do**
  $W_{\mathcal{I}'} = W_{\mathcal{I}'} - \{(m(e), A, x)\}$ *(for some $x \in V_{\mathcal{I}}$)*
  $W_{\mathcal{I}'} = W_{\mathcal{I}'} \cup \{(m(e), A, v)\}$

$V_{\mathcal{I}'} = V_{\mathcal{I}'} \cup \{v\}$
$V_{\mathcal{I}'} = V_{\mathcal{I}'} - \{v \mid \lambda_{\mathcal{I}}(v) \in$ ***D-TYPE*** $\wedge\ \nexists e \in V_{\mathcal{I}'}, A' \in$ ***ATTR*** $: (e, A', v) \in W_{\mathcal{I}'}\}$
return $(\mathcal{I}')$

□

By the one but last statement of each algorithm, isolated values (i.e., values that no longer occur as attribute of some entity) are deleted, since otherwise, the resulting graph of the algorithm would no longer correspond to an ER instance.

In a sense, an attribute addition may also be used both to "delete" an attribute, by supplying $\perp_d$ as the new value, or to "add" an attribute, if the old value happened to be equal to $\perp_d$.

In the above definition, a distinction is made between two kinds of attribute updates: either the attribute value is taken from the pattern, or it is provided separately. We illustrate both kinds by means of an example.

Figure 5.11 shows an attribute update in which the attribute value is taken from the pattern. More precisely, this operation sets to zero the limit of all cash cards that access an account with a zero balance.

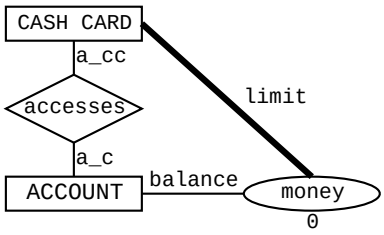


Figure 5.11: An example attribute update (I)

In terms of the notation of Definition 5.22, Figure 5.11 shows the attribute update ***ATTUPD***$[\mathcal{P}, A, e, v]$ with

- the pattern $\mathcal{P} = (V, W, \lambda, \pi)$ consisting of
  - $V = \{e_1, e_2, v_1, r_1\}$
  - $W = \{(r_1, \text{a\_cc}, e_1), (r_1, \text{a\_a}, e_2), (e_2, \text{balance}, v_1)\}$
  - λ is defined by the following table:

node	label
e_1	CASH CARD
e_2	ACCOUNT
r_1	accesses
v_1	money

  - π maps v_1 to 0.
- the attribute type A equal to `limit`, being the type of the attributes to be updated

- the entity e equal to e_1, being the entity whose attribute is to be updated
- the value v equal to the node v_1, being the new value of the updated attributes.

This example may also be used to illustrate the need for the requirement that π must be defined on the node v taken from the pattern as the new value of the updated attribute. Indeed, suppose π were undefined on the money-node in the pattern of the update addition in Figure 5.11. Now consider the ER instance graph depicted in Figure 5.12. Then the pattern would clearly have two embeddings in the instance of Figure 5.12. The outcome of the (first) algorithm in Definition 5.22 would now depend on the *order* in which these embeddings were treated! Indeed, if the embedding mapping the money-node in the pattern to the value 1234 is treated first, then in the instance resulting from the algorithm, the cash card gets the limit 5678, and vice versa. The only way to avoid this clearly undesirable semantics is to require that the target node of a newly added attribute edge is a uniquely specified value in the pattern.

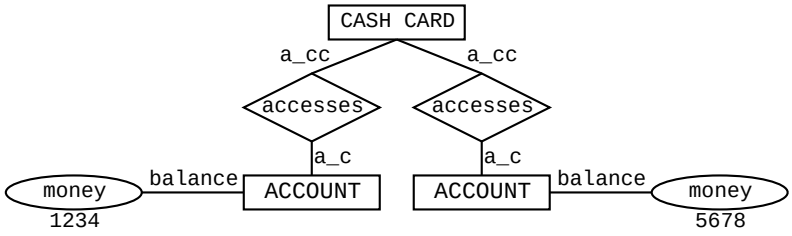


Figure 5.12: An ER instance graph

Figure 5.13 shows an attribute update in which the attribute value is provided separately. More precisely, this operation blocks all accounts with a zero balance.

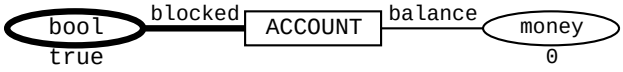


Figure 5.13: An example attribute update (II)

In terms of the notation of Definition 5.22, Figure 5.13 shows the attribute update *ATTUPD*$[\mathcal{P}, A, e, v]$ with

- the pattern $\mathcal{P} = (V, W, \lambda, \pi)$ consisting of
  - $V = \{e_1, v_1\}$
  - $W = \{(e_1, \text{balance}, v_1)\}$
  - λ is defined by the following table:

node	label
e_1	ACCOUNT
v_1	money

  - π maps v_1 to 0.

- the attribute type A equal to `blocked`, being the type of the attributes to be updated
- the entity e equal to e_1, being the entity whose attribute is to be updated
- the value v equal to "true", being the new value of the updated attributes.

The result of applying this attribute update to any ER instance is that the `blocked`-attribute of all `ACCOUNT`s with a zero `balance` is set to "true", regardless of the value of the `blocked`-attribute in the input instance.

Previously, we have introduced GOOD/ER operations that allow the addition of entities and/or relationships. For both operations, we now introduce a "deleting counterpart", i.e., operations that allow the deletion of either entities or relationships.

The first operation, called *entity deletion*, allows the deletion of entities, together with all their attributes and the relationships in which they participate:

Definition 5.23 (Entity Deletion) *Given an ER scheme $\mathcal{S}$, an* entity deletion *over $\mathcal{S}$ takes as input*

- *an ER pattern $\mathcal{P} = (V, W, \lambda, \pi)$ over $\mathcal{S}$*
- *$e \in V$ such that $\lambda(e) \in$ E-TYPE*

The result of applying the entity deletion ENTDEL$[\mathcal{P}, e]$ to an ER instance $\mathcal{I}$ over $\mathcal{S}$ is defined as the outcome of the following algorithm:

$\mathcal{I}' := \mathcal{I}$;
for each *embedding m of $\mathcal{P}$ in $\mathcal{I}$* **do**
  $V_{\mathcal{I}'} = V_{\mathcal{I}'} - \{m(e)\} - \{$*all relationships involving* $e\}$
  $\mathcal{E}_{\lambda(e)} := \mathcal{E}_{\lambda(e)} - \{m(e)\}$;
  $W_{\mathcal{I}'} = W_{\mathcal{I}'} \mid_{V_{\mathcal{I}'}}$
$V_{\mathcal{I}'} = V_{\mathcal{I}'} - \{v \mid \lambda_{\mathcal{I}}(v) \in$ D-TYPE$\wedge\ \nexists e' \in V_{\mathcal{I}'}, A' \in$ ATTR$: (e', A', v) \in W_{\mathcal{I}'}\}$
return $(\mathcal{I}')$

□

By removing a deleted entity from the universe of entities, the tricky situation is avoided where one and the same entity is deleted from an instance, and subsequently reinserted by an entity addition (or abstraction, see furtheron). This is motivated by the fact that the actual identity of entities is of no concern when basic GOOD/ER operations are executed. Hence one cannot and should not be able to make any assumptions about the identity of a newly added entity.

Figure 5.14 shows an example entity deletion. The pattern of this operation consists of the entire picture, whereas the part (the image of which under embeddings is) to be deleted (in this case, the `CASH CARD`-entity) is indicated with double lines. This operation removes all cash cards of the customer named "John Smith", as well as all `has`-, `started by`-, `accesses`- and `issues`-relationships in which these `CASH CARD`-entities are involved. The algorithm of Definition 5.23 takes care of the

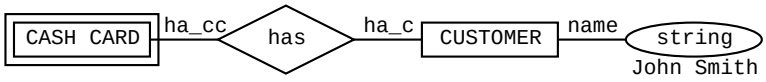


Figure 5.14: An example entity deletion

deletion of all attribute and role edges entering or leaving the deleted entities, by simply restricting the edge set of the ER instance graph to those edges involving nodes which are still in the graph after the removal of the entities and the relationships in which they participate.

The second kind of deleting operations allows the deletion of relationships:

Definition 5.24 (Relationship Deletion) *Given an ER scheme $\mathcal{S}$, a* relationship deletion *over $\mathcal{S}$ takes as input*

- *an ER pattern $\mathcal{P} = (V, W, \lambda, \pi)$ over $\mathcal{S}$*
- *$r \in V$ such that $\lambda(r) \in$ R-TYPE*

The result of applying the relationship deletion RELDEL$[\mathcal{P}, r]$ to an ER instance $\mathcal{I}$ over $\mathcal{S}$ is defined as the outcome of the following algorithm:

$\mathcal{I}' := \mathcal{I}$;
for each *embedding m of $\mathcal{P}$ in $\mathcal{I}$* **do**
    $V_{\mathcal{I}'} = V_{\mathcal{I}'} - \{m(r)\}$
    $W_{\mathcal{I}'} = W_{\mathcal{I}'} \mid_{V_{\mathcal{I}'}}$
return $(\mathcal{I}')$

□

Figure 5.15 shows an example relationship deletion. This operation expresses the fact that no cash cards should be allowed to access blocked accounts. The operation therefore looks for embeddings of the pattern consisting of a `CASH CARD` which `accesses` an `ACCOUNT` whose `blocked`-attribute is set to "true", and deletes the `accesses`-relationship.

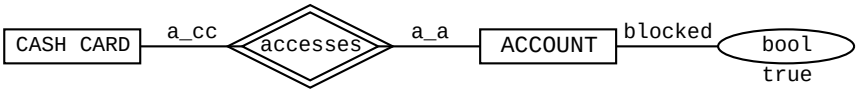


Figure 5.15: An example relationship deletion

A major characteristic of all GOOD/ER operations is that, by the mechanism of pattern matching, operations can only use information *stored explicitly* in the ER instance. Examples of such explicitly stored information are

- the (non)-existence of entities or relationships of a certain type;

- the presence/absence of certain values;
- a value (not) being an attribute of a certain entity;
- an entity (not) playing a certain role in a relationship.

Examples of information *not* accessible through pattern matching are

- specific properties of values (e.g., an integer being positive);
- the identity of entities (e.g., the presence of a particular entity e).

In Section 5.3, we formally prove that the GOOD/ER language is capable of expressing *all possible* transformations using only explicitly stored information (in a sense to be made precise). With the five GOOD/ER operations introduced so far, however, the above claim does not hold. Concretely, using these five operations, it is for instance not possible to *group* entities that are indistinguishable on the basis of explicitly stored information (of a certain type) concerning them. We clarify this kind of manipulation (i.e., the grouping of such "indistinguishable" entities) by means of an example.

Consider the (partial) ER instance graph depicted in Figure 5.16. It shows a number of cash cards (the identifiers *e1* through *e4* are used further on to distinguish the various nodes), accessing a number of accounts. If we restrict our view on cash cards to the information of type `accesses` concerning them (i.e., we don't take into account their attributes, or any other relationships in which they might participate), then we can say that the two leftmost cash cards are indistinguishable, since they both access both accounts shown in the ER instance graph, whereas the third cash card accesses only one of them, and the fourth cash card doesn't access any account.

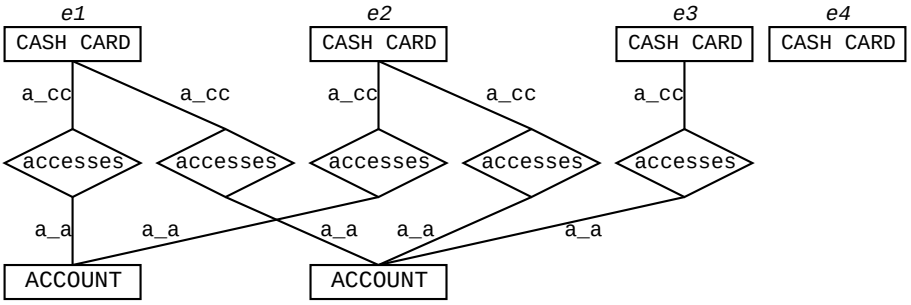


Figure 5.16: The input instance for the example entity abstraction

Hence an operation that *groups* (or, to be precise, partitions the set of) cash cards based on this similarity, should have the effect shown in the ER instance graph of Figure 5.17.

We implicitly assume that the ER scheme includes an entity type `CC-SET` (used for representing sets of cash cards, hence its name) as well as a relationship type `contains`, used for linking `CC-SET`s to the `CASH CARD`s contained in them. The resulting instance of the grouping operation contains three additional entities, each representing a set of cash cards, the elements of which are indistinguishable on the basis of the `accesses`-relationships they participate in.

In [VdBP91], it was shown that this operation cannot be performed using only the five GOOD/ER operations introduced so far. Hence we define the following additional operation:

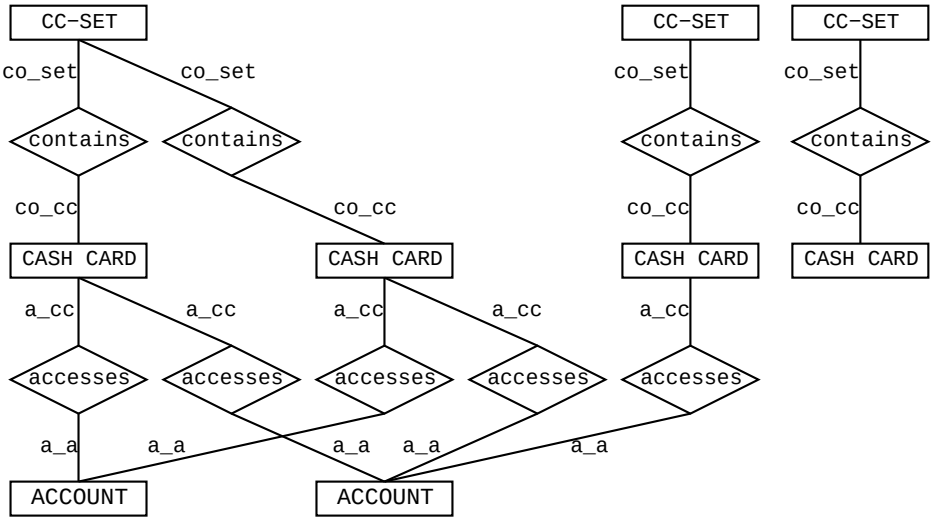


Figure 5.17: The output instance for the example entity abstraction

Definition 5.25 (Entity Abstraction (syntax)) *Given an ER scheme $\mathcal{S}$, an* entity abstraction *over $\mathcal{S}$ is denoted syntactically as ENTABS$[\mathcal{P}, e, E, R, RE]$ with the following input parameters:*

- *an ER pattern $\mathcal{P} = (V, W, \lambda, \pi)$ over $\mathcal{S}$*
- *$e \in V$ such that $\lambda(e) \in$ E-TYPE*
- *$E \in$ E-TYPE*
- *$R \in$ R-TYPE such that $\text{participants}(R) =< E, \lambda(e) >$, with roles $P_1 : R \to E$ and $P_2 : R \to \lambda(e)$*
- *RE $\in$ R-TYPE such that $\text{participants}(RE) =< P_3, P_4 >$, with $P_3 :$ RE $\to \lambda(e)$*[4]

Figure 5.18 shows the entity abstraction which, when applied to the ER instance shown in Figure 5.16, results in the ER instance shown in Figure 5.17.

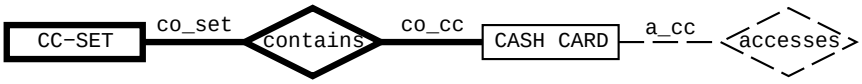


Figure 5.18: An example entity abstraction

In terms of the notation of Definition 5.25, Figure 5.18 shows the entity abstraction *ENTABS*$[\mathcal{P}, e, E, R, RE]$ with

[4]The reason for considering entity abstraction only over *binary* relationships is merely one of succinctness: it is perfectly possible to define abstraction over arbitrary n-ary relationships, but as Section 5.3 shows, this would not add to the expressive power of the GOOD/ER language.

- the pattern $\mathcal{P} = (V, W, \lambda, \pi)$ consisting of
  - $V = \{e_1\}$
  - $W = \emptyset$
  - λ maps e_1 to `CASH CARD`
  - π is undefined on V.
- the entity e equal to e_1
- the entity type E equal to `CC-SET`, being the type of the entities that (will) represent sets of cash cards
- the relationship type R equal to `contains`, being the type of the relationships used to link the sets to their elements
- the relationship type *RE* equal to `accesses`, indicating which relationship type should be considered for partitioning the cash cards

In Figure 5.18, one can see that the pattern and the "addition part" of an entity abstraction are denoted as usual (i.e., using plain lines for the pattern, and bold lines for the addition part). The role and the relationship type used for partitioning the cash cards are indicated with dotted lines.

Definition 5.26 (Entity Abstraction (semantics)) *Let ENTABS$[\mathcal{P}, e, E, R, RE]$ be an entity abstraction over an ER scheme $\mathcal{S}$ as defined in Definition 5.25, and let $\mathcal{I}$ be an ER instance over $\mathcal{S}$.*

Let S be the set $\{m(e) \mid m : \mathcal{P} \to \mathcal{I}$ is an embedding$\}$, and let Σ be the partition of S, defined by the equivalence relation

$$p \equiv q \Leftrightarrow (\forall t \in V_{\mathcal{I}} : (P_3, p, P_4, t) \in V_{\mathcal{I}} \Leftrightarrow (P_3, q, P_4, t) \in V_{\mathcal{I}})$$

Then the result of applying the entity abstraction ENTABS$[\mathcal{P}, e, E, R, RE]$ to $\mathcal{I}$ is defined as one possible outcome of the following algorithm:

$\mathcal{I}' := \mathcal{I}$;
for each $T \in \Sigma$ **do**
  if not exists $v_1 \in V_{\mathcal{I}} : \lambda_{\mathcal{I}}(v_1) = E \wedge \{v_2 \mid (P_1, v_1, P_2, v_2) \in V_{\mathcal{I}}\} = T$
  then *add to $V_{\mathcal{I}'}$ a new entity $et \in \mathcal{E}_E - V_{\mathcal{I}'}$ of type E*
    the relationships (P_1, v_1, P_2, v_2) $(\forall v_2 \in T)$
    add to $W_{\mathcal{I}'}$ the role-edges of the newly added relationships
    for each $A \in$ ATTR such that $E \in \mathrm{owner}(A)$, add $(e, A, \perp_{\mathrm{domain}(A)})$
return $(\mathcal{I}')$

□

The set S of embeddings of the pattern $\mathcal{P}'$ in the ER instance graph of Figure 5.16 obviously contains four elements, since the single node of $\mathcal{P}'$ may be mapped to each of the four CASH CARD-nodes in the graph. The partition Σ of S equals $\{\{m1, m2\}, \{m3\}, \{m4\}\}$, as explained above. Given this partition, the algorithm of Definition 5.26 then works similarly to that of Definition 5.15 of entity additions, with the single difference that whereas the algorithm for entity addition operates on the images of the pattern under the embeddings, the algorithm for entity abstraction operates on the elements of the partition Σ. In summary, the result of applying the entity abstraction of Figure 5.18 to the ER instance shown in Figure 5.16 is shown in Figure 5.17.

5.2.3 GOOD/ER Programs

In order to obtain a *language* based on the six basic GOOD/ER operations, we still have to introduce *programming primitives* that allow the grouping of basic operations into programs. In [AP91, GPVdBVG94], a programming language is discussed, consisting of fourteen programming constructs (including typical ones such as conditional statements and while-loops) which may be used to group into programs basic graph rewriting operations (strongly reminiscent to those introduced in this section). Twelve of these programming constructs are actually "macros", defined in terms of two programming primitives, allowing the *sequencing* respectively the *grouping* of basic operations. As defining programming constructs in terms of each other has little to do with graph rewriting (and hence with the topic of this thesis), we restrict ourselves in this section to introducing these two programming primitives.

The simplest programming construct allows the sequencing of any number of basic GOOD/ER operations or method applications (to be defined further on):

Definition 5.27 (GOOD/ER program) *A* GOOD/ER program *is a sequence of basic GOOD/ER operations (i.e., entity addition, relationship addition, attribute update, entity deletion, relationship deletion and entity abstraction) or method applications. The result of the application of a GOOD/ER program to an instance is obtained by sequentially applying the operations constituting the program to the instance, in the order given by the sequence.*
□

Figure 5.19 shows a GOOD/ER program consisting of five basic GOOD/ER operations. The five operations are separated by dotted lines in the figure, and the sequence should be read from top to bottom. Under the assumption that its input instance does not contain any entities of type CC-SET, this program removes from its input instance all CASH CARDs that do not access any ACCOUNT.

By means of the first three operations of the program, all the CASH CARDs that are to be removed, are grouped by linking them to an entity of type CC-SET. First we create this entity, by means of an entity addition with an empty pattern. An entity addition with an empty pattern creates exactly one entity of the given type, unless the input instance already contains entities of that type. By means of the second operation, a relationship addition, we link *all* CASH CARDs to the newly created entity, by means of relationships of type contains. The CC-SET-entity may now be considered to represent the set of all CASH CARDs in the instance. By means of the third operation, a relationship deletion, we remove from this set the CASH CARDs that *do* access one or more ACCOUNTs. Finally we delete

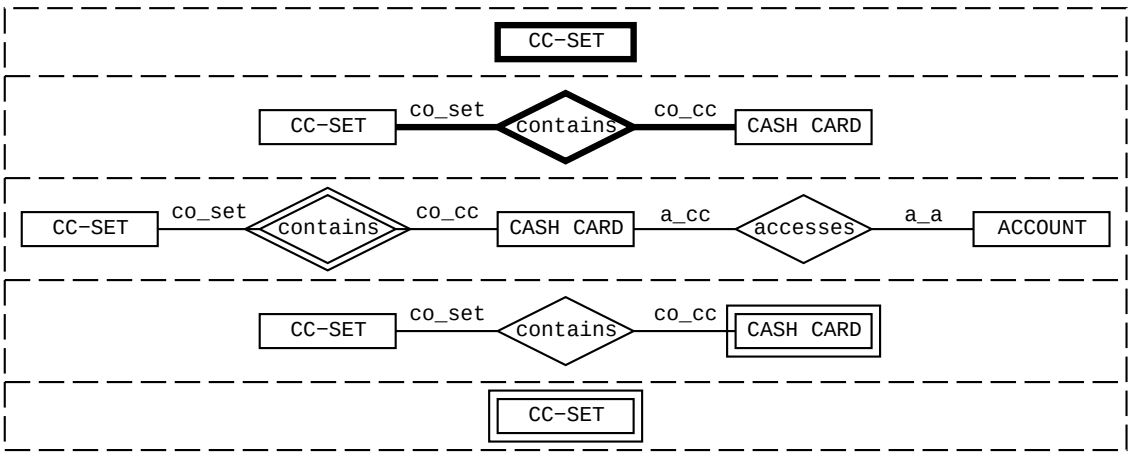


Figure 5.19: An example GOOD/ER program

the CASH CARDs that are still in the set, as well as the set (i.e., the CC-SET-entity) itself by means of two entity deletions.

Note how in this program, deletion is used to express negation, which cannot be expressed with a single ordinary pattern.

The second programming construct we introduce allows to *name* and *parametrize* a sequence of GOOD/ER operations. Such a sequence is called a GOOD/ER *method*, according to the terminology of object-orientation. If supplied with the required actual parameters (in the form of an ER pattern) the sequence may be applied by means of the method's name.

Formally, let $\mathcal{L}$ be a countably infinite universe of *parameter names*.

Definition 5.28 (GOOD/ER method (syntax)) *Let $\mathcal{S}$ be an ER scheme. Syntactically, a* GOOD/ER method $\mathcal{M}$ *over $\mathcal{S}$ is a pair $(S_\mathcal{M}, B_\mathcal{M})$ consisting of the method's* signature $S_\mathcal{M}$ *and the method's* body $B_\mathcal{M}$*:*

- *The signature is a three-tuple $S_\mathcal{M} = (r_\mathcal{M}, L_\mathcal{M}, p_\mathcal{M})$ with*
  - $r_\mathcal{M} \in$ *E-TYPE the method's* receiver type*;*
  - $L_\mathcal{M} \subset \mathcal{L}$ *a finite set containing names for the method's parameters;*
  - $p_\mathcal{M} : L_\mathcal{M} \rightarrow$ *E-TYPE$\cup$D-TYPE a function indicating the types of the method's parameters;*
- *The body is a list $B_\mathcal{M} =< O_1, \ldots, O_n >$ in which $\forall 1 \leq i \leq n$, O_i is either*
  - *an ordinary basic GOOD/ER operation or a method application (see further on); or*
  - *a basic GOOD/ER operation or a method application with as additional parameters*
    - *a set $s^i_\mathcal{M}$ which is either empty or contains a single entity of type $r_\mathcal{M}$ in $\mathcal{P}'$ (where $\mathcal{P}'$ is the operation's pattern), distinguishing the receiver;*

* $p^i_{\mathcal{M}} : L_{\mathcal{M}} \nrightarrow V_{\mathcal{P}}$ *satisfying* $\forall l \in L_{\mathcal{M}} : \lambda_{\mathcal{P}}(p^i_{\mathcal{M}}(l)) = p_{\mathcal{M}}(l)$*, distinguishing the formal parameters.*

□

Suppose it frequently occurs that a bank blocks all accounts of a given customer which are managed by that bank, while at the same time setting the passwords of all cash cards owned by that customer and which access those accounts to some string only known to the bank's system administration. This complex database manipulation may be accomplished with the GOOD/ER method `block`, the signature of which is depicted graphically in Figure 5.20, and the body of which is shown in Figure 5.21.

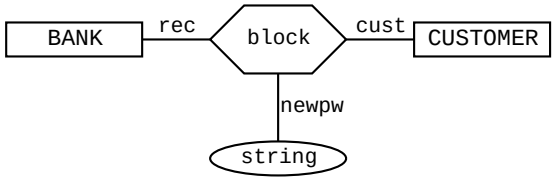


Figure 5.20: An example GOOD/ER method signature

The method's name is shown in the hexagon, which is linked to nodes labeled with the receiver and parameter types by means of edges labeled with respectively `rec` and the parameter names. In terms of the notation of Definition 5.28, Figure 5.20 shows the method signature $S_{\texttt{block}} = (r_{\texttt{block}}, L_{\texttt{block}}, p_{\texttt{block}})$ where

- $r_{\texttt{block}}$, the method's receiver type, equals `BANK`;
- $L_{\texttt{block}}$, the set containing names for the method's parameters, equals $\{\texttt{newpw}, \texttt{cust}\}$
- $p_{\texttt{block}}$ the function indicating the types of the method's parameters, maps `newpw` to the data type `string` and maps `cust` to the entity type `CUSTOMER`.

The body consists of a sequence $< O_1, O_2 >$ of two attribute updates, both with additional parameters.

The first attribute update blocks the accounts managed by the given bank and held by the given customer. The formal parameters are distinguished graphically using the same conventions as in the graphical depiction of the method's signature, that is, by means of a hexagon linked to these various parameters. In terms of the notation of Definition 5.28, the set $s^1_{\texttt{block}}$ contains the entity of type `BANK`. The partial function $p^1_{\texttt{block}}$ maps the parameter name `cust` to the entity of type `CUSTOMER`, and is undefined on the parameter name `newpw` (since the latter parameter is irrelevant to this first operation).

The second attribute update sets the password of the cash cards issued by the given bank and owned by the given customer to the given string. In terms of the notation of Definition 5.28, the set $s^2_{\texttt{block}}$ contains the entity of type `BANK`. The function $p^2_{\texttt{block}}$ is total, and maps the parameter name `cust` to the entity of type `CUSTOMER`, and the parameter name `newpw` to the value of type `string`.

Given what methods look like syntactically, we can now define how they can be applied and what the semantics of such an application is, in a way similar to the application of the basic GOOD/ER operations. Informally, the semantics of the method call is that the operations in the body of the method are

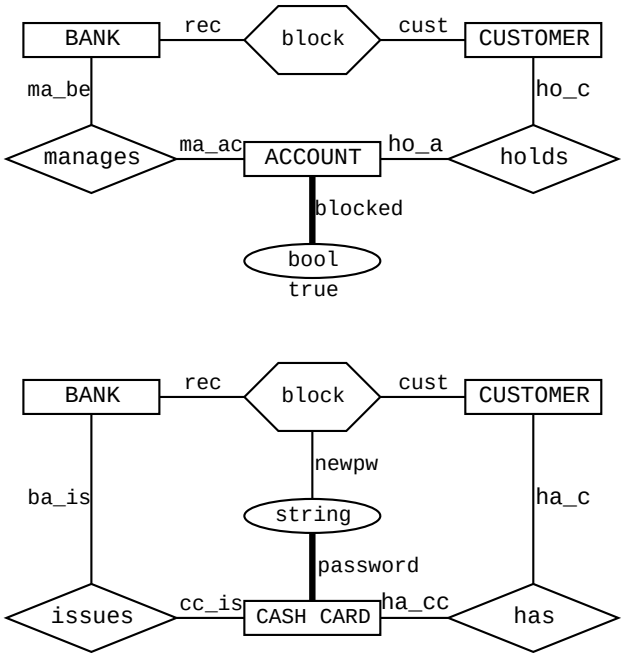


Figure 5.21: An example GOOD/ER method body

applied consecutively, but with the matchings of their patterns restricted according to their additional parameters indicating the actual receiver entity and parameters. Formally:

Definition 5.29 (GOOD/ER method (semantics)) *Let $\mathcal{M} = (S_\mathcal{M}, B_\mathcal{M})$ be a method over an ER scheme $\mathcal{S}$ as defined in Definition 5.28, with signature $S_\mathcal{M} = (r_\mathcal{M}, L_\mathcal{M}, p_\mathcal{M})$ and body $B_\mathcal{M} =< O_1, \ldots, O_n >$. Let $\mathcal{I}$ be an ER instance and $\mathcal{P}$ an ER pattern over $\mathcal{S}$.*

Let e be an entity in $V_\mathcal{P}$ of type $r_\mathcal{M}$, called the receiver entity.

Let $g : L_\mathcal{M} \to V_\mathcal{P}$ be a function that identifies actual parameters *in the pattern $\mathcal{P}$, satisfying $\forall l \in L_\mathcal{M} : \lambda_\mathcal{P}(g(l)) = p_\mathcal{M}(l)$.*

Assume that $\bar{\mathcal{M}} \in$ E-TYPE, rec $\in$ R-TYPE, rec^1 : rec $\to \bar{\mathcal{M}}$, rec^2 : rec $\to r_\mathcal{M} \in$ ROLE and $\forall l \in L_\mathcal{M}$:

- *if $p_\mathcal{M}(l) \in$ E-TYPE, then let $R_l \in$ R-TYPE with $\mathrm{participants}(R_l) =< \bar{\mathcal{M}}, p_\mathcal{M}(l) >$ and roles $r_l^1 : R_l \to \bar{\mathcal{M}}$ and $r_l^2 : R_l \to p_\mathcal{M}(l)$;*
- *if $p_\mathcal{M}(l) \in$ D-TYPE, then let $A_l : \bar{\mathcal{M}} \to p_\mathcal{M}(l) \in$ ATTR.*

Then the result of applying the method $\mathcal{M}[\mathcal{P}, e, g]$ to the instance $\mathcal{I}$ is defined as one possible outcome of the following sequence of GOOD/ER operations:

1. *First, the entity addition ENTADD$[\mathcal{P}, \bar{\mathcal{M}}, \{(A_l, g(l)) \mid l \in L_\mathcal{M}, p_\mathcal{M}(l) \in$ D-TYPE$\}, \emptyset, \{R_l \mid l \in L_\mathcal{M}, p_\mathcal{M}(l) \in$ E-TYPE$\} \cup \{$rec$\}, \{(< (r_l^2, g(l)) >, r_l^1) \mid l \in L_\mathcal{M}\} \cup \{(< (rec^2, e) >,$ rec$^1)\}]$, marking the embeddings of the pattern of the method application.*

2. *Next, for each* $1 \leq i \leq n$*:*

   - *If* O_i *is an ordinary basic GOOD/ER operation or method application, then apply this operation after having augmented its pattern with a single entity of type* $\bar{\mathcal{M}}$*;*
   - *Otherwise, apply* O_i *after having augmented its pattern with*
     - *a single entity* e' *of type* $\bar{\mathcal{M}}$*;*
     - *the relationships* $\{(r_l^1 : e', r_l^2 : p^i_{\mathcal{M}}(l)) \mid l \in L_{\mathcal{M}}, p^i_{\mathcal{M}}(l)$ *defined,* $p_{\mathcal{M}}(l) \in$ *E-TYPE*$\} \cup$ $\{(rec^1 : e', rec^2 : e_i) \mid s^i_{\mathcal{M}} = \{e_i\}\}$ *together with the roles corresponding to these relationships;*
     - *the attributes* $\{(A_l, p^i_{\mathcal{M}}(l)) \mid l \in L_{\mathcal{M}}, p^i_{\mathcal{M}}(l)$ *defined,* $p_{\mathcal{M}}(l) \in$ *D-TYPE*$\}$.

3. *Finally, the entity deletion* $ENTDEL[\mathcal{P}', \{e''\}]$ *where* $\mathcal{P}'$ *is a pattern consisting of a single entity* e'' *of type* $\bar{\mathcal{M}}$*, removing the markings introduced by the first operation of this sequence.*

□

Suppose an employee of the "General Savings" bank wishes to block both the accounts and cash cards of customer John Smith, using the password "secret". He can do this using the method application depicted in Figure 5.22.

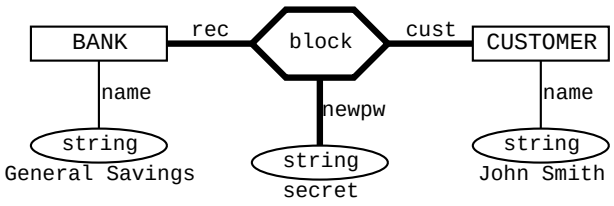


Figure 5.22: An example GOOD/ER method application

Graphically, a method application is represented by a boldface hexagon, linked to the actual parameters in the pattern by means of edges labeled with the parameter's names, as well as an edge labeled `rec` to the actual receiver entity in the pattern.

In terms of the notation of Definition 5.29, Figure 5.22 shows the method application `block`$[\mathcal{P}, e, g]$ where

- the pattern of the method application depicted in Figure 5.22 consists of the nodes and edges drawn in plain lines;
- the receiver entity e is the entity of type `BANK`;
- the function g identifying the actual parameters maps the parameter name `cust` to the entity of type `CUSTOMER`, and the parameter name `newpw` to the value of type `string`.

Figure 5.23 shows the actual sequence of basic GOOD/ER operations to be executed as a result of the method application depicted in Figure 5.22. The initial entity addition marks the embeddings of the pattern of the method application by means of entities of type block. These entities are linked by means of relationships to the parameters which are entities, and have as attributes the parameters which are values. These entities are then used in replacement of the hexagons in the operations from the body of the method. The final operation removes them from the ER instance.

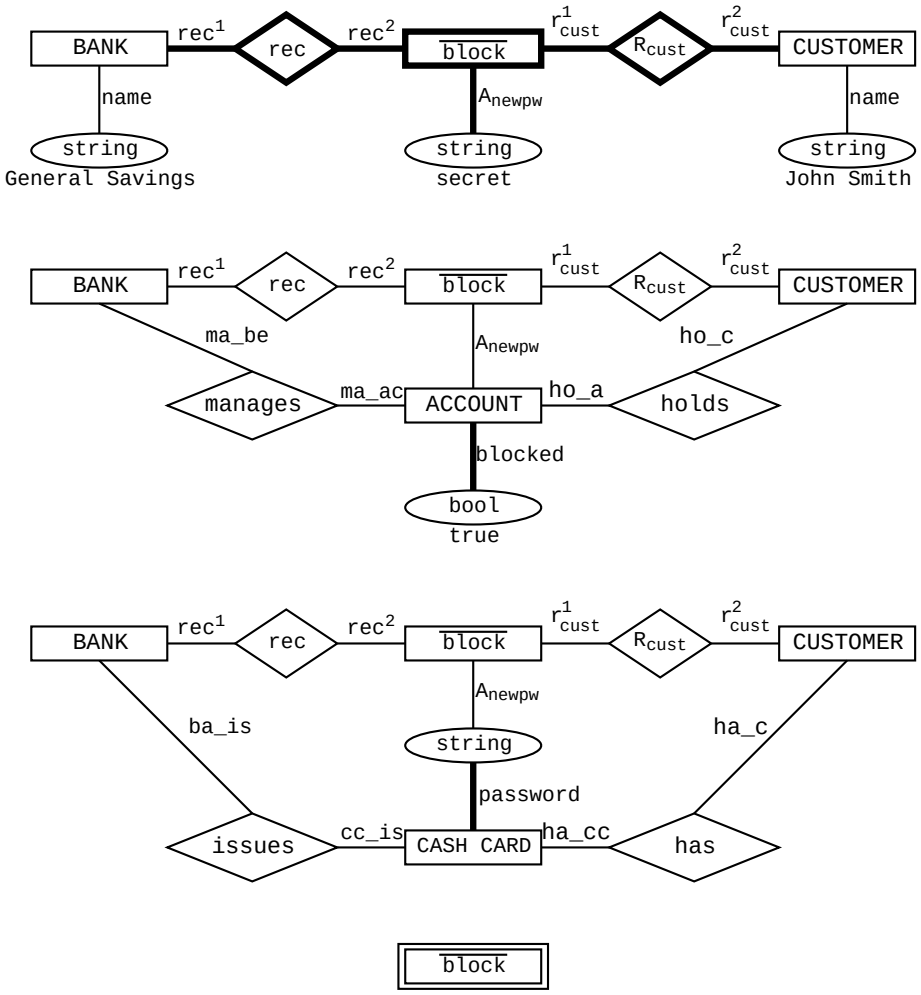


Figure 5.23: An example GOOD/ER method

A single feature from Definition 5.29 is not illustrated in this example, namely that of operations in the body that do not use parameters. In this case, Definition 5.29 nevertheless enforces the addition of an isolated entity (of type block in the case of the example) to their pattern. This ensures that nothing happens if there are no embeddings of the pattern of the method application, since in this case, the initial entity addition has no effect.

What the example does illustrate is how GOOD/ER methods offer the possibility to group and parametrize a sequence of basic GOOD/ER operations for purposes of reuse and encapsulation. But

besides these advantages, the method mechanism also adds to the expressive power of the GOOD/ER language, since they introduces *recursion* into the language.

As an illustration, we conclude this section by modeling a version of the well-known *transitive closure* problem (which is clearly not expressible by means of a simple fixed-length sequence of basic GOOD/ER operations) using a GOOD/ER method. Suppose our example ER scheme includes a relationship type for modeling child-relationships between customers (cf. Figure 5.24).

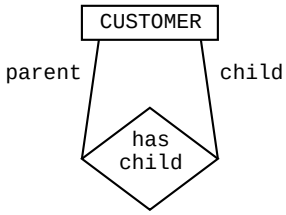


Figure 5.24: A relationship type for modeling child-relationships between customers

Suppose we wish to explicitly represent *descendancy*-relationships between customers: one customer is a descendant of another customer, if he is a child of that customer, or a child of a child of that customer, and so on. In other words, the descendants-relationships represent the transitive closure of the child-relationships. This operation can be performed using the method TC (for Transitive Closure), the signature and body of which are depicted in respectively Figures 5.25 and 5.26.

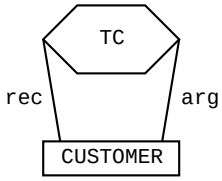


Figure 5.25: Signature of the GOOD/ER method expressing transitive closure

The method TC takes two customers as input, one as receiver and the other as argument.

It adds a `descendant`-relationship between the receiver and argument, and then calls itself on the receiver and any child of the argument.

The program depicted in Figure 5.27 first "doubles" the `childs`-relationships with `descendant`-relationships, after which it initiates the actual computation of the transitive closure by calling the method TC on any customer with any child of a descendant of that customer as argument. Note that this method loops forever in case the input instance contains cycles of `has child`-relationships. This could be taken care of in the body of the method, by checking for the presence of a `has ancestor`-relationship between the receiver and the argument, prior to recursively calling the method.

In [Eng90, Sch91a], another manipulation language was introduced for the EER model, called the Visual Action Language. The definition of this language consists of three levels:

1. *basic* actions affect single elements (such as an entity or relationship) of an EER instance. They operate on a graph-representation of a database instance, which includes among others an en-

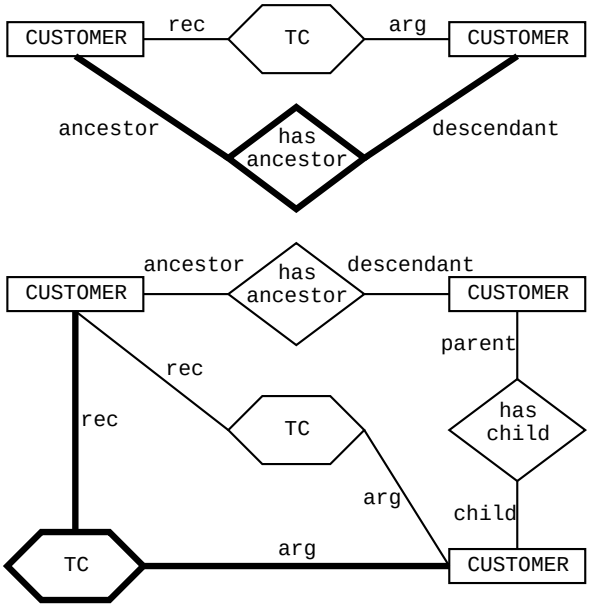


Figure 5.26: Body of the GOOD/ER method expressing transitive closure

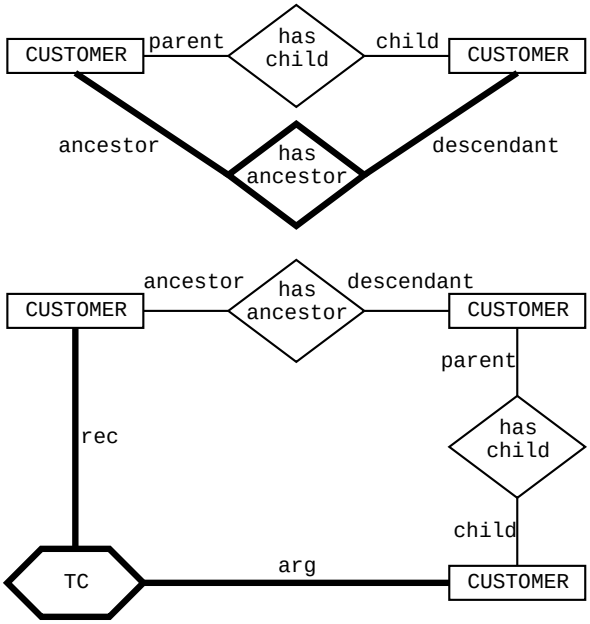


Figure 5.27: Program including an application of the GOOD/ER method expressing transitive closure

coding of the database scheme. Basic actions may be derived automatically from the database scheme;

2. *elementary* actions consist of several basic actions, but respect the integrity constraints from the database scheme. Such actions may therefore also be derived automatically from the database scheme;

3. *complex* actions consist of several elementary actions, and are specified by the user. In order to indicate the database elements to which the action should be applied, pre-computed queries are used, represented by a special graphical symbol.

The following basic differences between the Visual Action Language and GOOD/ER may be derived from this:

- the three-level definition of the actions in the Visual Action Language, as opposed to the two-level definition (i.e., rules and programs) of graph rewrite systems in GOOD/ER;
- the automatic derivation of elementary actions from the scheme;
- the use of pre-computed queries as opposed to GOOD/ER's patterns;
- the use of an encoding of the database scheme in the representation of the database instance.

5.3 On the Expressive Power of GOOD/ER

In Section 5.2, the introduction of the entity abstraction (Definitions 5.25 and 5.26) operation was motivated by the observation that a certain kind of operations (namely the grouping of entities based on common properties) cannot be expressed as a sequence of the other basic GOOD/ER operations. This raises the question whether, given the GOOD/ER language consisting of all six basic operations plus the mechanism of sequencing, there are still other kinds of operations that cannot be expressed. Or, to put things more positive: which category of operations *can* be expressed using the GOOD/ER language as introduced in Section 5.2?

More generally, to demonstrate the viability of any newly proposed (and formally defined) language, its expressive power should be compared to that of other languages. A possible approach towards such a comparison of languages is the use of so-called *completeness*-criteria, a technique well known in the area of database languages. Indeed, already in [Cod72], Codd proposed to call a language for the relational database model *complete* if its expressive power could be shown equivalent to that of some "standard" query language, like the relational calculus.

However, in [Ban78] Bancilhon argued that a completeness-criterion, in order to be sufficiently meaningful, should be *language independent*. In [Ban78] respectively in [Par78] Bancilhon and Paredaens therefore (independently) introduced a similar criterion, stating when a query language for relational databases deserves to be called complete.The criterion says that, in order to be complete, a query language should express exactly the transformations from a database (i.e., a set of relations) D to a relation R that satisfy the following two conditions. First, no new values may be added by the transformation. Second, each domain permutation that maps D to itself, must also map R to itself. A

transformation satisfying these conditions is called a *generic* transformation. These conditions can be summarized by saying that every *automorphism* of D must be an automorphism of R, or yet in other words, every query in the considered language should *commute* with every automorphism of the input database D.

It is also shown in the aforementioned articles that the relational calculus [Ban78] and algebra [Par78] indeed express exactly these transformations, thereby providing an a posteriori justification for Codds choice of the relational calculus as a reference language for testing completeness for relational query languages.

But what does the above criterion *intuitively* signify? The presence of a (non-trivial) automorphism for some database D can be interpreted as follows: for every value in D, there exists another value which can "take its place" in the database. Indeed, when each value in D is substituted by its image under the automorphism, D itself is obtained. Consequently, a value and its image under some automorphism are indistinguishable on the basis of those relationships between them which are stored explicitly in D. The criterion states that if such a resemblance exists in the input database of a query, it should still exist in the resulting relation. Violating this is only possible by manipulating values through more than just their relationships with other values (as stored explicitly in the relations of the database), in other words by *interpreting* them. Consequently, this criterion is really very natural and unrestrictive, since it merely prohibits interpreting values, or in other words, to perform calculations on them.

Apart from its theoretical importance, the validity of the criterion for a certain language can also have a practical usage if it is possible to readily check it for two given instances, since this is equivalent to the existence of a transformation between them in the language under consideration.

In [CH80], the above criterion was named *BP-completeness* (after its inventors). Briefly, a language is BP-complete if it can express exactly all generic transformations. Since its introduction, the notion of genericity has been used frequently in the context of other database models. Naturally, if we want to generalize it to other formalisms besides the relational one, we first have to appropriately define concepts such as "transformation" and "automorphism". An example may be found in [GPVG89], in which BP-completeness is adapted and applied successfully to the nested relational database model.

The recent shift of attention of the database research community from the relational model to object-oriented models raised the question if the BP-completeness criterion could also be applied to the assessment of the expressive power of object-oriented query languages. It soon became clear, however, that one particular characteristic of such languages would considerably complicate such an application. First, remember from the above discussion that a first condition to be satisfied by a transformation expressed in a language satisfying the BP-completeness criterion, is that it may add no new values to the input database. In object-oriented query languages, however, it is common practice to incorporate the result of a query in the database, by creating (a) new object(s) for it.

So how does this influence the second condition of the BP-completeness criterion, namely that transformations should commute with the automorphisms of the input database? On one hand, automorphisms may still be looked upon as permutations of the basic elements of the database (in this case, the objects), that preserve the structure of the instance (in this case, the relationships represented explicitly in the instance). In the course of a transformation, however, new objects may be created, while others may be removed. Consequently, we can no longer impose an inclusion relationship on the sets of automorphisms of the input- and output-instance of the transformation. The most natural translation

of such a relationship to the context of automorphism groups for object-base instances would therefore be to require the existence of a *mapping* between the respective automorphism groups of two instances, with the additional constraint that an automorphism and its image under the given mapping should coincide on the objects still in common to the input- and output-instance. This correspondence is crucial for the understanding of the remainder of this section: database transformations for value-based data models *commuting with* permutations corresponds to database transformations for object-based data models *preserving* automorphisms.

This, however, does not yet solve all problems with applying the BP-completeness criterion to object databases. In [AK89], the Identity Query Language IQL (which is a language involving object creation) is introduced and shown to be very general and powerful. In the same article however, IQL is shown unable to express exactly the class of transformations that satisfy the modified BP-completeness criterion as discussed in the previous paragraph. In this section, we therefore introduce an adjusted version of the criterion — in terms of mappings between automorphism groups, since it was shown above that this is the most natural way to go in an attempt to translate BP-completeness criterion to the context of languages involving object creation — and show that it allows us to precisely characterize the set of transformations expressible by means of GOOD/ER programs (i.e., sequences of basic GOOD/ER operations excluding method applications).

In other words, we provide a language independent characterization for the set of pairs of ER instances $(\mathcal{I}, \mathcal{I}')$ for which there exists a GOOD/ER program mapping $\mathcal{I}$ to $\mathcal{I}'$.

Definition 5.30 (GOOD/ER-implication) *Let $\mathcal{I}$ and $\mathcal{I}'$ be ER instances. $\mathcal{I} \overset{GOOD/ER}{\Longrightarrow} \mathcal{I}'$ holds if there exists a GOOD-program (i.e., a sequence of basic GOOD/ER operations excluding method applications) that maps $\mathcal{I}$ to $\mathcal{I}'$.*

□

One might wonder why we so explicitly exclude the powerful mechanism of methods from this study. The reason for this is that for the problem we study in this section, this power is not *needed*. As mentioned before, the additional power offered by methods comes solely from their ability to model recursion. In this section, we study the problem of expressive power on "instance-level", that is, given two instances, we consider the problem of finding necessary and sufficient conditions for the existence of a GOOD/ER program that maps one to the other. Suppose there is indeed a GOOD/ER program, involving recursive methods, mapping one given instance to another given instance. Since both instances are known in every possible detail, we also know exactly *how many times* (say, n) these recursive methods will call themselves when the program is applied to the given input instance. Hence we can simply replace the initial application of any recursive method in the given program by n copies of the method's body.

However, when studying the expressive power of database manipulations defined as *mappings* from one set of instances to another set of instances, rather than between individual instances, the power of methods is indeed needed, as shown in [VdBVGAG92, VdB93], where the main result of this section is used to characterize the expressive power of database manipulations defined as mappings.

Returning to the problem to be dealt with in this section, remember that the characterization should be stated in terms of a mapping between the respective automorphism groups of $\mathcal{I}$ and $\mathcal{I}'$, with the con-

straint that an automorphism and its image under the given mapping should coincide on the intersection of $\mathcal{I}$ and $\mathcal{I}'$. In [AK89], it was shown that a condition based on an "ordinary" mapping between the automorphism groups is too weak to precisely characterize the expressive power of an object creating query language. Hence we strengthen the condition by demanding the existence of a *homomorphism* between the automorphism groups, as captured in the following definition:

Definition 5.31 (Extension morphism) *Let $\mathcal{I}$ and $\mathcal{I}'$ be two ER instances. An* extension morphism *of type $(\mathcal{I}, \mathcal{I}')$ is a group homomorphism $h : Aut(\mathcal{I}) \rightarrow Aut(\mathcal{I}')$ such that*

$$\forall v \in V_{\mathcal{I}} \cap V_{\mathcal{I}'}, \forall a \in Aut(\mathcal{I}) : a(v) = h(a)(v) \tag{5.1}$$

□

We refer to property 5.1 as the *extension property*. To understand the meaning of this name, consider the case where $\mathcal{I}$ is a subinstance of $\mathcal{I}'$. The condition then simply says that the image under h of any automorphism should coincide with that automorphism on $\mathcal{I}$.

The step from extension morphisms to an adapted BP-completeness criterion is simple. We recall that a language is BP-complete if it can express exactly all generic transformations. Hence the following definition:

Definition 5.32 (Generic Transformations) *Let $\mathcal{I}$ and $\mathcal{I}'$ be two ER instances. The pair $(\mathcal{I}, \mathcal{I}')$ is a generic transformation if there exists an extension morphism of type $(\mathcal{I}, \mathcal{I}')$.*
□

We are now ready to state the main result of this section, being that GOOD/ER expresses precisely all generic transformations.

Theorem 5.33 *Let $\mathcal{I}$ and $\mathcal{I}'$ be two ER instances. Then the following are equivalent:*

1. $\mathcal{I} \overset{GOOD/ER}{\Longrightarrow} \mathcal{I}'$.
2. *$(\mathcal{I}, \mathcal{I}')$ is a generic transformation.*

A first proposition shows that GOOD/ER *only* expresses generic transformations.

Proposition 5.34 *If for two ER instances $\mathcal{I}$ and $\mathcal{I}'$, $\mathcal{I} \overset{GOOD/ER}{\Longrightarrow} \mathcal{I}'$ holds, then $(\mathcal{I}, \mathcal{I}')$ is a generic transformation.*

Proof The fact that $\mathcal{I} \overset{\text{GOOD/ER}}{\Longrightarrow} \mathcal{I}'$ holds, implies the existence of a GOOD/ER-program that maps $\mathcal{I}$ to $\mathcal{I}'$. We prove the existence of an extension morphism h of type $(\mathcal{I}, \mathcal{I}')$ by induction on the length of this program.

Let us first assume that the GOOD/ER-program consists of zero operations, so $\mathcal{I}$ equals $\mathcal{I}'$. The identity-function on $Aut(\mathcal{I})$ is then the required extension morphism of type $(\mathcal{I}, \mathcal{I}')$.

For the general case, the induction-hypothesis is as follows: for each pair of instances for which $\mathcal{I} \overset{\text{GOOD/ER}}{\Longrightarrow} \mathcal{I}'$, such that the GOOD/ER-program, mapping $\mathcal{I}$ to $\mathcal{I}'$ consists of at most ℓ basic GOOD/ER-operations, there exists an extension-morphism of type $(\mathcal{I}, \mathcal{I}')$.

Given a GOOD-program ∇ consisting of $\ell + 1$ basic GOOD/ER operations $\nabla_1, \ldots, \nabla_{\ell+1}$, which maps an instance $\mathcal{I}$ to an instance $\mathcal{H}'$, we now have to prove the existence of an extension-morphism h' of type $(\mathcal{I}, \mathcal{H}')$.

Let $\mathcal{I}'$ be the result of applying the first ℓ operations of ∇ to $\mathcal{I}$. By the induction-hypothesis, we know that there exists an extension-morphism h of type $(\mathcal{I}, \mathcal{I}')$.

We now show how to change the extension-morphism h into an extension-morphism h' of type $(\mathcal{I}, \mathcal{H}')$, depending on which kind of basic GOOD/ER operation $\nabla_{\ell+1}$ actually is. Note first of all that the behavior of an automorphism on an instance is completely determined by its behavior on the entities in that instance. Indeed, a relationship $(P_1 : e_1, \ldots, P_k : e_k)$ must be mapped by an automorphism a to the relationship $(P_1 : a(e_1), \ldots, P_k : a(e_k))$, whereas a value must obviously be mapped to itself. Hence, when describing automorphisms in the remainder of this proof, we shall suffice with describing their behavior on entities.

- Suppose $\nabla_{\ell+1}$ is an entity addition or abstraction. If no entities are actually added, the definition of h' is obvious. If an entity e is added as a "result" of an embedding m of the pattern of the entity addition in $\mathcal{I}'$, then for all $a \in Aut(\mathcal{I})$, an entity e_a is also added as a result of the embedding $h(a) \circ m$ (cf. Lemma 5.20). We define $h''(a)(e) = e_a$. Furthermore, we define $h''(a) = h(a)$ on $V_{\mathcal{I}'}$.

  To see that h' is an extension-morphism of type $(\mathcal{I}, \mathcal{H}')$, let m be an embedding of the pattern of $\nabla_{\ell+1}$ in $\mathcal{I}'$, and let a_1 and a_2 be two automorphisms of $\mathcal{I}$. Suppose three entities e_1, e_2 and e_3 are added to $\mathcal{I}'$ as a result of the respective embeddings $m, h(a_1) \circ m$ and $h(a_2) \circ h(a_1) \circ m$. Then $h'(a_1)(e_1) = e_2$ and $h'(a_2)(e_2) = e_3$, so $h'(a_2) \circ h'(a_1)(e_1) = e_3$. But since the entity added by $h(a_2) \circ h(a_1) \circ m$ is e_3, $h'(a_2 \circ a_1)(e_1) = e_3$. The same reasoning holds for newly added relationships and values, hence h' is a group homomorphism. Since it is an extension of h, $h'(a)$ equals a on $\mathcal{I} \cap \mathcal{H}'$, for all $a \in Aut(\mathcal{I})$.

- Suppose $\nabla_{\ell+1}$ is a relationship addition. If no relationships are actually added, the definition of h' is obvious. If a relationship r is added as a "result" of an embedding m of the pattern of the relationship addition in $\mathcal{I}'$, then for all $a \in Aut(\mathcal{I})$, a relationship r_a is also added as a result of the embedding $h(a) \circ m$. We define $h'(a)(r) = r_a$. Furthermore, we define $h'(a) = h(a)$ on $V_{\mathcal{I}'}$.

  The proof that h' is an extension-morphism of type $(\mathcal{I}, \mathcal{H}')$ is totally analogous to the case where $\nabla_{\ell+1}$ is an entity addition.

- Suppose $\nabla_{\ell+1}$ is an attribute update. Then for any attribute edge (e, A, v), to be updated to (e, A, v') and for all $a \in Aut(\mathcal{I})$, the attribute edge $(h(a)(e), A, h(a)(v))$ is updated to $(h(a)(e), A, h(a)(v')) = (h(a)(e), A, v')$. Consequently, every automorphism of $\mathcal{I}'$ is also an automorphism of $\mathcal{H}'$,[5] so we can take $h' = h$.

[5] Unless v' is a value not in $\mathcal{I}$, in which case each automorphism of $\mathcal{I}'$ may be straightforwardly extended to an automorphism of $\mathcal{H}'$.

- Suppose $\nabla_{\ell+1}$ is an entity deletion. Then for any deleted entity e, and for all $a \in Aut(\mathcal{I})$, the entity $h(a)(e)$ is also deleted. The same holds for all relationships involving either e or $h(a)(e)$. As a result, the restriction of an automorphism of $\mathcal{I}'$ to $\mathcal{H}'$ is an automorphism of $\mathcal{H}'$, so we can take $h'(a)$ equal to $h(a)$ restricted to $\mathcal{H}'$, for each $a \in \mathrm{AUT}(\mathcal{I})$.

- Suppose $\nabla_{\ell+1}$ is a relationship deletion. Then for any deleted relationship r and for all $a \in Aut(\mathcal{I})$, the relationship $h(a)(r)$ is also deleted. As a result, the restriction of an automorphism of $\mathcal{I}'$ to $\mathcal{H}'$ is an automorphism of $\mathcal{H}'$, so we can take $h'(a)$ equal to $h(a)$ restricted to $\mathcal{H}'$, for each $a \in \mathrm{AUT}(\mathcal{I})$. ■

Next, we show that the GOOD/ER language can express *any* generic transformation $(\mathcal{I}, \mathcal{I}')$. This is proved in two steps. First we study the special case where $\mathcal{I}'$ is a superinstance of $\mathcal{I}$ (i.e., of *monotonic* transformations). We give a GOOD/ER program that, when applied to $\mathcal{I}$, results in a superinstance $\langle \mathcal{I}' \rangle$ of $\mathcal{I}'$, which contains information derived from the extension morphism h. Then we state how this superinstance may be restricted to $\mathcal{I}'$ by means of another GOOD/ER program (cf. Proposition 5.44).

In the second step, we consider arbitrary instances $\mathcal{I}'$. We therefore first describe an extension of $\mathcal{I}$ that also includes $\mathcal{I}'$, as well as an adaptation of the extension morphism h to this superinstance. This way we can apply the result of the first step, showing that $\mathcal{I}$ GOOD/ER-implies this superinstance. Finally we show how this superinstance can be restricted to $\mathcal{I}'$ by means of yet another GOOD/ER program (cf. Proposition 5.46).

In prepartion of the introduction of the superinstance $\langle \mathcal{I}' \rangle$ mentioned above, we first review some graph-theoretical notions (see for instance [Hof82, Section 1.4]).

First, the *orbit* of a node in a graph is usually defined as the set of all nodes in that graph that are the image of that node under some automorphism of the graph. We slightly adapt this definition using the notion of extension-morphism:

Definition 5.35 (Orbit) *Let $\mathcal{I}$ be a subinstance of $\mathcal{I}'$, and let h be an extension morphism of type $(\mathcal{I}, \mathcal{I}')$. Let n be a node of $\mathcal{I}'$. The* orbit *of n w.r.t. h is defined as the set*

$$\mathit{orb}_h(n) = \{n' \in V_{\mathcal{I}'} \mid \exists a \in \mathit{Aut}(\mathcal{I}) : h(a)(n) = n'\}$$

In each orbit, an arbitrary but fixed node is chosen, called the representative *of the orbit. $\mathit{Orbits}_h(\mathcal{I}' - \mathcal{I})$ is the set of all the orbits of nodes of $\mathcal{I}' - \mathcal{I}$ w.r.t. h.*
□

It can easily be seen that $Orbits_h(\mathcal{I}' - \mathcal{I})$ is a partition of $V_{\mathcal{I}'-\mathcal{I}}$.

Second, the *stabilizer* of a node in a graph is usually defined as the set of all automorphisms of the graph that fix this node. We also adapt this definition to the notion of extension-morphism:

Definition 5.36 (Stabilizer) *Let $\mathcal{I}$ be a subinstance of $\mathcal{I}'$, and let h be an extension morphism of type $(\mathcal{I}, \mathcal{I}')$. Let $n \in V_{\mathcal{I}'-\mathcal{I}}$. The* stabilizer *of n w.r.t. h is defined as the set*

$$\mathit{st}_h(n) = \{a \in \mathit{Aut}(\mathcal{I}) \mid h(a)(n) = n\}$$

□

It can easily be seen that $st_h(n)$ is a subgroup of $Aut(\mathcal{I})$.

The group-theoretical notion of *(left) coset* allows us to "characterize" nodes of a graph in terms of orbits and stabilizers.

Definition 5.37 (Coset) *Let G be a subgroup of $Aut(\mathcal{I})$ and let $a \in Aut(\mathcal{I})$. A* coset *of G is defined as $a \circ G = \{a \circ b \mid b \in G\}$. CosetAut($\mathcal{I}$) is the set of all cosets of all subgroups of Aut($\mathcal{I}$).*
□

The following lemma is of crucial importance to the proof of the main result of this section. It establishes a one-to-one correspondence between nodes added by a generic transformation $(\mathcal{I}, \mathcal{I}')$ on one hand, and orbits and cosets of stabilizers on the other hand:

Lemma 5.38 *Let $\mathcal{I}$ be a subinstance of $\mathcal{I}'$, and let h be an extension morphism of type $(\mathcal{I}, \mathcal{I}')$. Let O be an orbit in $\mathcal{I}' - \mathcal{I}$ with representative n_O. Then there exists a one-to-one correspondence between O and the set of cosets of $st_h(n_O)$.*

Proof First, let n be a node of O. Then by definition, there is an automorphism a of $\mathcal{I}$ such that $h(a)(n_O) = n$. We claim that $a \circ st_h(n_O)$ is a unique coset corresponding to n. To show that $a \circ st_h(n_O)$ indeed determines a unique coset, we have to prove that it is independent of the chosen automorphism a. Indeed, suppose there is some other automorphism b of $\mathcal{I}$ such that $h(b)(n_O) = n$. Then $h(a)(n_O) = h(b)(n_O)$. Applying $h(b^{-1})$ to both sides of this equation yields $h(b^{-1} \circ a)(n_O) = n_O$, so $b^{-1} \circ a$ is a member of $st_h(n_O)$. As a result, $(b^{-1} \circ a) \circ st_h(n_O) = st_h(n_O)$. Applying b to both sides of the latter equation yields $a \circ st_h(n_O) = b \circ st_h(n_O)$.

Second, to a coset $a \circ st(n_O)$, corresponds the unique node $h(a)(n_O)$. ■

With this result, we are ready to define the instance $\langle \mathcal{I}' \rangle$, which extends the "target"-instance $\mathcal{I}'$ of a generic transformation $(\mathcal{I}, \mathcal{I}')$ with information derived from the corresponding extension morphism.

Definition 5.39 *Let $\mathcal{I}$ be a subinstance of $\mathcal{I}'$, and let h be an extension morphism of type $(\mathcal{I}, \mathcal{I}')$. We define the extension $\langle \mathcal{I}' \rangle$ of $\mathcal{I}'$ w.r.t. $\mathcal{I}$ and h as follows.*

- *$\langle \mathcal{I}' \rangle$ includes all entities of $\mathcal{I}'$, as well as*
  - *the elements of Orbits$_h$($\mathcal{I}' - \mathcal{I}$) considered as entities, each of a unique type;*
  - *the elements of Aut($\mathcal{I}$) considered as entities, all of type* AUT*;*
  - *the elements of CosetAut($\mathcal{I}$) considered as entities, each typed by a unique name for the subgroup corresponding to the coset.*

  We assume that all these entities and their types do not occur in $\mathcal{I}'$.
- *$\langle \mathcal{I}' \rangle$ includes all relationships and roles of $\mathcal{I}'$, as well as*
  - *for each pair of different entities e_1, e_2 in $\mathcal{I}$ of the same type, the relationship $(\alpha_1^{\lambda(e_1)} : e_1, \alpha_2^{\lambda(e_1)} : e_2)$ of type $\mathtt{diff}^{\lambda(e_1)}$, as well as its corresponding roles;*

- *for each automorphism $a \in Aut(\mathcal{I})$ and each entity e in $\mathcal{I}$, the relationship $(\beta_1^{\lambda(e)} : a, \beta_2^{\lambda(e)} : a(e))$ of type e, as well as its corresponding roles, assuming that $\bigcup_{E \in \text{E-TYPE}} \mu[\text{E-TYPE}](E) \subset$ R-TYPE;*
- *for each coset $b \circ G \in CosetAut(\mathcal{I})$ and each automorphism $a \in b \circ G$, the relationship $(\gamma_1^{\lambda(G)} : b \circ G, \gamma_2^{\lambda(G)} : a)$ of type $\Gamma^{\lambda(G)}$, as well as its corresponding roles;*
- *for each orbit $O \in Orbits_h(\mathcal{I}' - \mathcal{I})$ and each entity $e \in O$, the relationship $(\epsilon_1^{\lambda(O)} : O, \epsilon_2^{\lambda(O)} : e)$ of type $\in^{\lambda(O)}$, as well as its corresponding roles;*
- *For each orbit-representative n_O and each automorphism $a \in Aut(\mathcal{I})$, the relationship $(\delta_1^{(\lambda(\mathit{st}_h(n_O)),\lambda(n_O))} : a \circ \mathit{st}_h(n_O), \delta_2^{(\lambda(\mathit{st}_h(n_O)),\lambda(n_O))} : h(a)(n_O))$ of type $\Delta^{(\lambda(\mathit{st}_h(n_O)),\lambda(n_O))}$, as well as its corresponding roles.*

We assume that all these relationship types and role names do not occur in $\mathcal{I}'$. The set of roles of types α_1^E for all $E \in$ E-TYPE is referred to as the set of roles of type α_1. The same holds for all other roles and relationships.

- *$\langle \mathcal{I}' \rangle$ includes no attributes or values besides those of $\mathcal{I}'$.*

□

The motivation behind this definition of the instance $\langle \mathcal{I}' \rangle$ may be found in the structure of the proof of the forthcoming Proposition 5.44. In this proof it is shown how, given a pair of instances $(\mathcal{I}, \mathcal{I}')$ for which there exists an extension-morphism, a GOOD/ER program can be constructed that maps $\mathcal{I}$ to $\mathcal{I}'$. In a first phase, this program adds to $\mathcal{I}$ its automorphisms (the "functionality" of which is represented by means of the relationships labeled with entities), subgroups of $Aut(\mathcal{I})$ and their cosets (linked to their members by means of the Γ-relationships), as well as the orbits of $\mathcal{I}' - \mathcal{I}$. We choose to add these mappings and sets *themselves* to the instance $\mathcal{I}$, since the alternative would be to add nodes *representing* them, which would just clutter up all forthcoming definitions and proofs. In a second phase, the GOOD/ER program then adds the nodes of $\mathcal{I}'$ together with the relationships of types Δ and $\in$. These relationships link it to the coset and the orbit by which it is uniquely identified as a result of Lemma 5.38.

Other particularities in the definition of $\langle \mathcal{I}' \rangle$ (such as the need for labeling relationships with entities, or the presence of the `diff`-relationships) are explained in the course of the proof of Proposition 5.44.

In the following definition, a given extension-morphism of type $(\mathcal{I}, \mathcal{I}')$ is extended to a mapping (which we show to be a group isomorphism) from $Aut(\mathcal{I})$ to $Aut(\langle \mathcal{I}' \rangle)$:

Definition 5.40 *Let $\mathcal{I}$ be a subinstance of $\mathcal{I}'$ and let h be an extension morphism of type $(\mathcal{I}, \mathcal{I}')$. Let $\langle \mathcal{I}' \rangle$ be the extension of $\mathcal{I}'$ according to Definition 5.39.*

The group homomorphism $\langle h' \rangle : Aut(\mathcal{I}) \to Aut(\langle \mathcal{I}' \rangle)$ is defined as follows. Let $a \in Aut(\mathcal{I})$.

1. *$\langle h' \rangle(a)(n) = h(a)(n)$, for each node n of $\mathcal{I}'$;*
2. *$\langle h' \rangle(a)(O) = O$, for $O \in Orbits_h(\mathcal{I}' - \mathcal{I})$;*

3. $\langle h' \rangle(a)(b) = a \circ b$, *for* $b \in$ *Aut*$(\mathcal{I})$;

4. $\langle h' \rangle(a)(b \circ G) = a \circ b \circ G$, *for* $b \circ G \in$ *CosetAut*$(\mathcal{I})$.

The behavior of $\langle h' \rangle(a)$ *on the relationships of* $\langle \mathcal{I}' \rangle$ *is defined as* $\langle h' \rangle(a)((P_1 : n_1, \ldots, P_k : n_k)) := (P_1 : \langle h' \rangle(a)(n_1), \ldots, P_k : \langle h' \rangle(a)(n_k))$.
□

Lemma 5.41 $\langle h' \rangle$ *is a group isomorphism.*

Proof We first have to show that $\langle h' \rangle$ is a well-defined mapping, i.e., that it maps automorphisms of $\mathcal{I}$ to automorphisms of $\langle \mathcal{I}' \rangle$. Let d be an automorphism of $\mathcal{I}$. It follows directly from the definition that $\langle h' \rangle(d)$ is a one to one function of $V_{\langle \mathcal{I}' \rangle}$ onto itself, preserving all node labels. We now show that $\langle h' \rangle(d)$ also preserves all relationships in $\langle \mathcal{I}' \rangle$.

- By the first item of its definition, $\langle h' \rangle(d)$ preserves all relationships of $\mathcal{I}'$;
- Since $\langle h' \rangle(d)$ is a one to one function, it preserves the `diff`-relationships;
- If, for some $a \in$ *Aut*$(\mathcal{I})$ and $e \in V_{\mathcal{I}}$, $(\beta_1^{\lambda(e)} : a, \beta_2^{\lambda(e)} : a(e))$ is a relationship in $\langle \mathcal{I}' \rangle$, then $(\beta_1^{\lambda(e)} : \langle h' \rangle(d)(a), \beta_2^{\lambda(e)} : \langle h' \rangle(d)(a)(e)) = (\beta_1^{\lambda(e)} : d \circ a, \beta_2^{\lambda(e)} : d \circ a(e))$ is also a relationship in $\langle \mathcal{I}' \rangle$;
- If $(\gamma_1^{\lambda(G)} : c \circ G, \gamma_2^{\lambda(G)} : a)$ is a relationship in $\langle \mathcal{I}' \rangle$, then $a \in c \circ G$, so $d \circ a \in d \circ c \circ G$. As a result, $(\gamma_1^{\lambda(G)} : \langle h' \rangle(d)(c \circ G), \gamma_2^{\lambda(G)} : \langle h' \rangle(d)(a)) = (\gamma_1^{\lambda(G)} : d \circ c \circ G, \gamma_2^{\lambda(G)} : d \circ a)$ is also a relationship in $\langle \mathcal{I}' \rangle$;
- If $(\epsilon_1^{\lambda(O)} : m, \epsilon_2^{\lambda(O)} : O)$ is a relationship in $\langle \mathcal{I}' \rangle$, then $m \in O$. Consequently, by the definition of orbits, $(\epsilon_1^{\lambda(O)} : \langle h' \rangle(d)(m), \epsilon_2^{\lambda(O)} : \langle h' \rangle(d)(O)) = (\epsilon_1^{\lambda(O)} : h(d)(m), \epsilon_2^{\lambda(O)} : O)$ is also a relationship in $\langle \mathcal{I}' \rangle$;
- If $(\delta_1^{(\lambda(\mathit{st}_h(n_O)), \lambda(n_O))} : h(a)(n_O), \delta_2^{(\lambda(\mathit{st}_h(n_O)), \lambda(n_O))} : a \circ \mathit{st}_h(n_O))$ is a relationship in $\langle \mathcal{I}' \rangle$, then $(\delta_1^{(\lambda(\mathit{st}_h(n_O)), \lambda(n_O))} : \langle h' \rangle(d)(h(a)(n_O)), \delta_2^{(\lambda(\mathit{st}_h(n_O)), \lambda(n_O))} : \langle h' \rangle(d)(a \circ \mathit{st}_h(n_O))) = (\delta_1^{(\lambda(\mathit{st}_h(n_O)), \lambda(n_O))} : h(d \circ a)(n_O), \delta_2^{(\lambda(\mathit{st}_h(n_O)), \lambda(n_O))} : d \circ a \circ \mathit{st}_h(n_O))$ is also a relationship in $\langle \mathcal{I}' \rangle$.

To show that $\langle h' \rangle$ is a group homomorphism, it can be trivially verified that for every node n of $\langle \mathcal{I}' \rangle$ and for every $a, b \in$ *Aut*$(\mathcal{I})$, $\langle h' \rangle(a \circ b)(n) = (\langle h' \rangle(a) \circ \langle h' \rangle(b))(n)$.

We now show that $\langle h' \rangle$ is injective. If $a \neq b \in$ *Aut*$(\mathcal{I})$, there is a node n in $\mathcal{I}$ for which $a(n) \neq b(n)$. Since h is an extension-morphism, $\langle h' \rangle(a)(n) = a(n) \neq b(n) = \langle h' \rangle(b)(n)$, so $\langle h' \rangle(a) \neq \langle h' \rangle(b)$.

Finally, we prove that $\langle h' \rangle$ is surjective. Let therefore $\langle a' \rangle \in$ *Aut*$(\langle \mathcal{I}' \rangle)$. Because $\langle a' \rangle$ has to preserve relationships of type $\in$, and each entity of $\mathcal{I}' - \mathcal{I}$ participates in precisely one relationship of type $\in$, $\langle a' \rangle$ must map entities (and hence also relationships and values) of $\mathcal{I}' - \mathcal{I}$ to entities (respectively relationships and values) of $\mathcal{I}' - \mathcal{I}$. As a result, $\langle a' \rangle$ maps nodes of $\mathcal{I}$ to nodes of $\mathcal{I}$, and hence $\langle a' \rangle|_{V_{\mathcal{I}}} \in$ Aut$(\mathcal{I})$. We claim that $\langle a' \rangle = \langle h' \rangle(\langle a' \rangle|_{V_{\mathcal{I}}})$.

- Since orbits (as nodes of $\langle \mathcal{I}' \rangle$) have a label which is unique to $\langle \mathcal{I}' \rangle$, they must be fixed by $\langle a' \rangle$ as well as by $\langle h' \rangle(\langle a' \rangle|_{V_\mathcal{I}})$;

- Let n be a node of $\mathcal{I}$. Since h is an extension-morphism, $h(\langle a' \rangle|_{V_\mathcal{I}})(n) = \langle a' \rangle|_{V_\mathcal{I}}(n) = \langle a' \rangle(n)$, so $\langle a' \rangle = \langle h' \rangle(\langle a' \rangle|_{V_\mathcal{I}})$ on $\mathcal{I}$;

- Let b be an automorphism of $\mathcal{I}$. By the definition of $\langle h' \rangle$, $\langle h' \rangle(\langle a' \rangle|_{V_\mathcal{I}})(b) = \langle a' \rangle|_{V_\mathcal{I}} \circ b$. The image under the automorphism $\langle a' \rangle(b)$ of a node n of $\mathcal{I}$ must equal the target of the n-labeled edge leaving the node $\langle a' \rangle(b)$. Since $\langle a' \rangle$ is an automorphism of $\langle \mathcal{I}' \rangle$, this target must in turn equal the node $\langle a' \rangle(b(n)) = \langle a' \rangle|_{V_\mathcal{I}}(b(n))$. We conclude that $\langle a' \rangle(b)$ and $\langle h' \rangle(\langle a' \rangle|_{V_\mathcal{I}})(b)$ represent the same automorphism, so $\langle a' \rangle$ and $\langle h' \rangle(\langle a' \rangle|_{V_\mathcal{I}})$ map b to the same node;

- Let $G \in$ *CosetAut($\mathcal{I}$)* such that $G = \{b_1, \ldots, b_n\} \subset$ *Aut*$(\mathcal{I})$. By the relationships of type Γ, an automorphism c of $\langle \mathcal{I}' \rangle$ must map G to the coset $\{c(b_1), \ldots, c(b_n)\}$. In other words, the behavior of an automorphism of $\langle \mathcal{I}' \rangle$ on *Aut*$(\mathcal{I})$, determines its behavior on *CosetAut($\mathcal{I}$)*. Since $\langle a' \rangle = \langle h' \rangle(\langle a' \rangle|_{V_\mathcal{I}})$ on *Aut*$(\mathcal{I})$, $\langle a' \rangle$ also equals $\langle h' \rangle(\langle a' \rangle|_{V_\mathcal{I}})$ on *CosetAut*$(\mathcal{I})$;

- By Lemma 5.38, the behavior of an automorphism of $\langle \mathcal{I}' \rangle$ on $\mathcal{I}'$-$\mathcal{I}$ is determined by its behavior on *Orbits*$_h(\mathcal{I}' - \mathcal{I}) \cup$ *CosetAut*$(\mathcal{I})$. Since $\langle h' \rangle(\langle a' \rangle|_{V_\mathcal{I}})$ and $\langle a' \rangle$ are equal on *Orbits*$_h(\mathcal{I}' - \mathcal{I}) \cup CosetAut(\mathcal{I})$, they are also equal on $\mathcal{I}' - \mathcal{I}$.

- By the definition of $\langle h' \rangle$, $\langle a' \rangle$ and $\langle h' \rangle(\langle a' \rangle|_{V_\mathcal{I}})$ obviously coincide on the relationships and values of $\langle \mathcal{I}' \rangle$. ■

This proof concludes the extension of $\mathcal{I}$ into a superinstance $\langle \mathcal{I}' \rangle$ of $\mathcal{I}'$, using information derived from the extension-morphism h. Note that this extension is stated in a purely *descriptive* way, independent of the GOOD/ER language. On the other hand, in the proof of Proposition 5.44, $\mathcal{I}$ is extended to $\langle \mathcal{I}' \rangle$ by means of a GOOD/ER program. In preparation of this proof, the following definition introduces a series of subinstances of $\langle \mathcal{I}' \rangle$, as well as corresponding morphisms relating their automorphism groups to *Aut*$(\mathcal{I})$. In the proof of Proposition 5.44, these subinstances will turn out to be the "intermediate" results of the GOOD/ER program extending $\mathcal{I}$ to $\langle \mathcal{I}' \rangle$.

Definition 5.42 *Let $\mathcal{I}$ be a subinstance of $\mathcal{I}'$, and let h be an extension morphism of type $(\mathcal{I}, \mathcal{I}')$. Let $\langle \mathcal{I}' \rangle$ be the extension of $\mathcal{I}'$ w.r.t. $\mathcal{I}$ and h as defined in Definition 5.39.*

Then the instances $\mathcal{K}'_1, \ldots, \mathcal{K}'_7$ are defined as follows:

1. *$\mathcal{K}'_1$ equals $\mathcal{I}$*

2. *$\mathcal{K}'_2$ equals $\mathcal{K}'_1$ extended with the* `diff`*-relationships of $\langle \mathcal{I}' \rangle$*

3. *$\mathcal{K}'_3$ equals $\mathcal{K}'_2$ extended with Aut$(\mathcal{I})$ and the e-relationships (with $e \in \bigcup_{E \in$ E-TYPE $} \mu$[E-TYPE$](E)$) of $\langle \mathcal{I}' \rangle$*

4. *$\mathcal{K}'_4$ equals $\mathcal{K}'_3$ extended with CosetAut$(\mathcal{I})$ and the Γ-relationships of $\langle \mathcal{I}' \rangle$*

5. *$\mathcal{K}'_5$ equals $\mathcal{K}'_4$ extended with Orbits$_h(\mathcal{I}' - \mathcal{I})$*

6. *$\mathcal{K}'_6$ equals $\mathcal{K}'_5$ extended with the entities of $\mathcal{I}'$ and the $\in$- and Δ-relationships of $\langle\mathcal{I}'\rangle$*

7. *$\mathcal{K}'_7$ equals $\langle\mathcal{I}'\rangle$.*

The mappings $\langle h'\rangle_1, \ldots, \langle h'\rangle_7$ are defined as:

$$\langle h'\rangle_j : \mathit{Aut}(\mathcal{I}) \to \mathit{Aut}(\mathcal{K}'_j) : a \mapsto \langle h'\rangle(a)|_{V_{\mathcal{K}'_j}} \quad (1 \leq j \leq 7)$$

□

Lemma 5.43 *The mappings $\langle h'\rangle_j (1 \leq j \leq 7)$ as defined in Definition 5.42 are group isomorphisms.*

Proof This proof is structured as follows. We first show that each $\langle h'\rangle_j$ is a well-defined, injective group homomorphism. For each pair of instances $\mathcal{K}'_i$ and $\mathcal{K}'_{i+1}$ $(1 \leq i \leq 6)$, we then give a bijection between their automorphism groups. Given the property that an injective group homomorphism between two finite groups with equal cardinality is a group isomorphism, it follows that each $\langle h'\rangle_j$ is a group isomorphism.

We first prove that $\langle h'\rangle_1$ is well-defined, in other words, that for each $a \in \mathit{Aut}(\mathcal{I})$, $\langle h'\rangle_1(a) = \langle h'\rangle(a)|_{V_{\mathcal{K}'_1}}$ is in $\mathit{Aut}(\mathcal{K}'_1)$. To show that $\langle h'\rangle_1(a)$ is well-defined, note that, because $\langle h'\rangle(a) \in \mathit{Aut}(\langle\mathcal{I}'\rangle)$, it has to preserve relationships of type $\in$. Since each entity of $\mathcal{I}' - \mathcal{I}$ participates in precisely one relationship of type $\in$, $\langle h'\rangle(a)$ must map entities (and hence also relationships and values) of $\mathcal{I}' - \mathcal{I}$ to entities (respectively relationships and values) of $\mathcal{I}' - \mathcal{I}$. As a result, $\langle h'\rangle_1(a)$ maps nodes of $\mathcal{I}$ to nodes of $\mathcal{I}$. Since $\langle h'\rangle(a)$ is an automorphism, $\langle h'\rangle_1(a)$ is also injective and surjective, and preserves node labels. To show that $\langle h'\rangle_1(a)$ also preserves edges, let (x, α, y) be an edge in $\mathcal{K}'_1$, i.e. in $\mathcal{I}$. As already shown, $\langle h'\rangle(a)(x)$ and $\langle h'\rangle(a)(y)$ are still nodes of $\mathcal{K}'_1$. But by the definition of $\langle h'\rangle$ and since h is an extension morphism, $\langle h'\rangle(a)(x) = a(x)$ and $\langle h'\rangle(a)(y) = a(y)$. Since a is an automorphism of $\mathcal{I}$, $(a(x), \alpha, a(y))$ is still an edge of $\mathcal{K}'_1$. So $\langle h'\rangle_1$ is well-defined.

We next show that $\langle h'\rangle_j$ is also well-defined for $1 < j \leq 7$. Since for all $1 < j \leq 6$, a node in $\mathcal{K}'_j - \mathcal{K}'_{j-1}$ has either a node label not in $\mathcal{K}'_{j-1}$, or an outgoing edge with a label not in $\mathcal{K}'_{j-1}$, while $\mathcal{K}'_j$ always contains *all* nodes or edges of $\langle\mathcal{I}'\rangle$ with these new labels, $\langle h'\rangle_j(a)$ always maps nodes of $\mathcal{K}'_j$ to nodes of $\mathcal{K}'_j$, and also preserves edges. $\mathcal{K}'_7$ equals $\langle\mathcal{I}'\rangle$, so $\langle h'\rangle_7(a)$ equals $\langle h'\rangle$, which has already been shown a group-isomorphism.

To prove that for all $1 \leq j \leq 7$, $\langle h'\rangle_j$ is injective, let $a \neq b \in \mathit{Aut}(\mathcal{I})$. Consequently, there is a node n of $\mathcal{I}$ (and thus of $\mathcal{K}'_j$ for each j) for which $a(n) \neq b(n)$. By Definitions 5.31 (of extension morphisms) and 5.40 (of $\langle h'\rangle$), it follows that $\langle h'\rangle(a)(n) \neq \langle h'\rangle(b)(n)$, and thus $\langle h'\rangle(a)_j(n) \neq \langle h'\rangle(b)_j(n)$.

Since $\forall a, b \in \mathit{Aut}(\mathcal{I})$, $\langle h'\rangle_j(a \circ b) = \langle h'\rangle(a \circ b) = \langle h'\rangle(a) \circ \langle h'\rangle(b) = \langle h'\rangle_j(a) \circ \langle h'\rangle_j(b)$, $\langle h'\rangle_j$ is a group homomorphism for all j.

Recall that the only thing left to be done, is to give a bijection between the automorphism groups of all pairs of instances $\mathcal{K}'_i$ and $\mathcal{K}'_{i+1}$ $(1 \leq i \leq 6)$. We only give details for the first two pairs of instances. The other bijections can be constructed analogously.

Since in $\mathcal{K}'_2$, only `diff`-relationships are added to $\mathcal{I}$, an automorphism a of $\mathcal{I}$ can be extended straightforwardly to an automorphism of $\mathcal{K}'_2$ by letting it map a relationship $(\alpha_1 : e_1, \alpha_2 : e_2)$ to the relationship $(\alpha_1 : a(e_1), \alpha_2 : a(e_2))$. Since the automorphism a is an injective mapping, this extension is an automorphism of $\mathcal{K}'_2$. Obviously, the restriction of an automorphism of $\mathcal{K}'_2$ to the nodes of $\mathcal{I}$ is a unique automorphism of $\mathcal{I}$.

An automorphism a of $\mathcal{K'}_2$ can be extended so it maps an AUT-node b in $\mathcal{K'}_3$ to the AUT-node $a \circ b$. This extension is obviously unique. Conversely, if an automorphism of $\mathcal{K'}_3$ maps an AUT-node a to a node b, it means that $b = c \circ a$, where c is the restriction of that automorphism to $\mathcal{I}$. Consequently, if two automorphisms of $\mathcal{K'}_3$ are equal on $\mathcal{K'}_2$, they must be equal, so restricting such an automorphism yields a unique automorphism of $\mathcal{K'}_2$. ■

Recall that our current aim is to show that GOOD/ER can express any "increasing" generic transformation, in other words, that the existence of an extension morphism for an instance $\mathcal{I}$ and a superinstance $\mathcal{I'}$ is a sufficient condition for the existence of a GOOD/ER program that, when applied to $\mathcal{I}$, results in $\mathcal{I'}$. The GOOD/ER program to be constructed in the proof of the following Proposition has $\langle \mathcal{I'} \rangle$ as an intermediate result, hence the proof contains a "constructive" definition for $\langle \mathcal{I'} \rangle$.

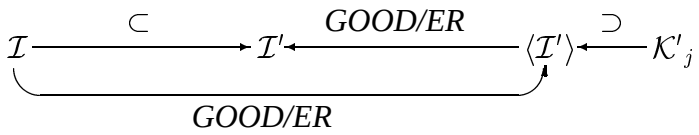


Figure 5.28: An overview of instances, used in the proof of Proposition 5.44

Proposition 5.44 *Let $\mathcal{I}$ be a subinstance of $\mathcal{I'}$ for which there exists an extension-morphism h of type $(\mathcal{I}, \mathcal{I'})$. Then $\mathcal{I} \overset{GOOD/ER}{\Longrightarrow} \mathcal{I'}$.*

Proof This proof is structured as follows. We build a GOOD/ER program that, when applied to $\mathcal{I}$, results in instance $\langle \mathcal{I'} \rangle$ as defined in Definition 5.39. It is shown that the intermediate resulting instances of this stepwise construction equal the instances $\mathcal{K'}_j$ (where j indicates the number of the step) from Definition 5.42. We then give a GOOD/ER program that, when applied to $\langle \mathcal{I'} \rangle$, results in the instance $\mathcal{I'}$ (cf. Figure 5.28).

Step 1 : The input to the program is the instance $\mathcal{I}$, which equals $\mathcal{K'}_1$.

Step 2 : We next add the `diff`-relationships. First, for each entity type E, we add a $\texttt{diff}^E$-relationship between any two entities of type E. Next, we delete any `diff`-relationships with identical participants. Figure 5.29 shows the pictorial representation of these operations. Obviously, the instance resulting from the application of these operations to $\mathcal{K'}_1$ equals the instance $\mathcal{K'}_2$.

Step 3 : We next add AUT-nodes together with relationships typed with the entities of $\mathcal{I}$. This operation can be accomplished with the single entity addition *ENTADD*$[\mathcal{K'}_2, \mathsf{AUT}, \emptyset, \emptyset, \{e \in V_{\mathcal{I}} \mid \lambda(e) \in \text{E-TYPE}\}, \{(< (\beta_2^{\lambda(e)}, e) >, \beta_1^{\lambda(e)}) \mid e \in V_{\mathcal{I}}, \lambda(e) \in \text{E-TYPE}\}]$. It uses $\mathcal{K'}_2$ as pattern, and adds entities of type AUT, with for each entity e in $\mathcal{K'}_2$ a relationship of type e linking the AUT-node to e (cf. Figure 5.30).

To see that this operation has the desired effect, reconsider Lemma 5.20. Since the identity function on $\mathcal{K'}_2$ is an embedding of the pattern of this operation, each automorphism of $\mathcal{K'}_2$ is in fact an

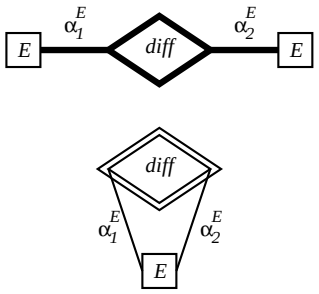


Figure 5.29: The GOOD/ER operations of Step 2 for the entity type E

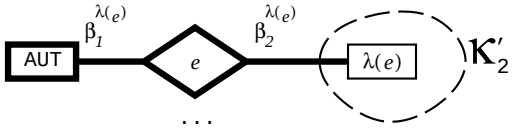


Figure 5.30: The GOOD/ER operation of Step 3

embedding. By the presence of the `diff`-edges, these automorphisms are *all* the possible embeddings of the pattern. Hence precisely one `AUT`-node will be added for each element of *Aut*($\mathcal{K}'_2$), which by Lemma 5.43 is isomorphic to *Aut*($\mathcal{I}$). We may assume without loss of generality that these newly added nodes are the elements of *Aut*($\mathcal{I}$) themselves. By the choice of the pattern and the relationship types of the entity addition, if a relationship $(\beta_1^{\lambda(e)} : a, \beta_2^{\lambda(e)} : e')$ is added as the result of an embedding $a \in$ *Aut*($\mathcal{I}$), then indeed $a(e) = e'$. Consequently, the result of this operation indeed equals $\mathcal{K}'_3$.

Step 4 : We next add the elements of *CosetAut*($\mathcal{I}$) together with the Γ-relationships. Therefore, for each subgroup $G = \{a_1, \ldots, a_n\}$ of *Aut*($\mathcal{I}$), the following four GOOD/ER operations are applied consecutively to $\mathcal{K}'_3$ (cf. Figure 5.31):

1. an entity addition with $\mathcal{K}'_3$ as pattern, of entities of type X (which we assume to be a "new" entity type), linked by means of relationships of type Γ^X to all the automorphisms of G;
2. an entity abstraction of the Γ^X-relationships *ENTABS*$[\mathcal{P}, e, \Gamma^X, \lambda(G), \Gamma^Y]$ where the pattern $\mathcal{P}$ contains a single entity e of type X;
3. a relationship addition of relationships of type Γ^G, for each Γ^Y-relationship and Γ^X-relationship "sharing" an entity of type X as participant;
4. an entity deletion of all X-entities.

By the observation used in the explanation of the correctness of Step 3, one can see that the first GOOD/ER operation of this step results in the addition of an entity of type X for each subgroup G (linked by means of Γ^X-relationships to the members of G), but also in the addition of an entity for

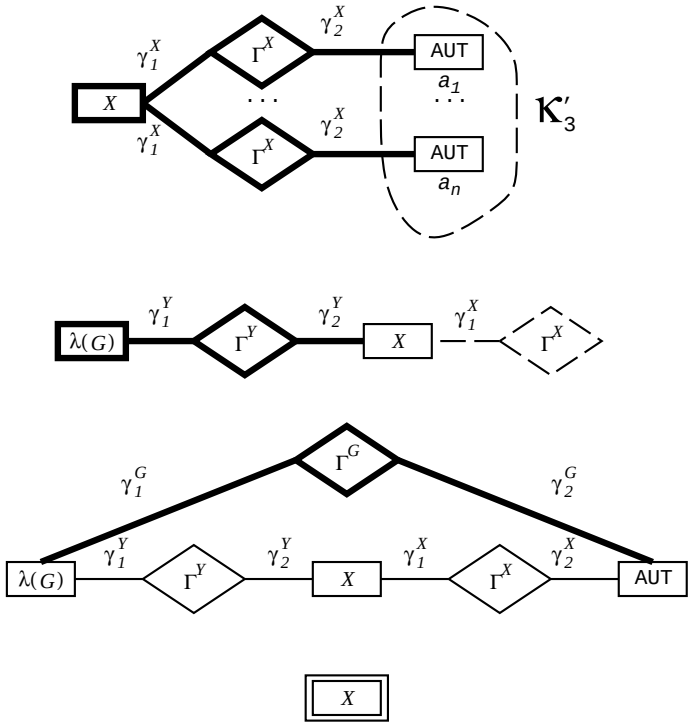


Figure 5.31: The GOOD/ER operations of Step 4 for the subgroup $G = \{a_1, \ldots, a_n\}$

each $a \in \mathit{Aut}(\mathcal{K}'_3)$ (which is (group-)isomorphic to $\mathit{Aut}(\mathcal{I})$), linked by means of Γ^X-relationships to the members of $a \circ G$. By the semantics of entity additions, however, in general *several* entities are added for one such coset: e.g., if some subgroup contains n automorphisms, then the entity addition for that particular subgroup adds n entities, since an embedding from the pattern followed by an automorphism of that subgroup, results in the addition of another entity of type X, corresponding to the same subgroup.

However, the resulting instance should contain *exactly* one entity for each coset. Recalling Definition 5.25, entity abstraction allows grouping entities according to common relationships. Hence the following three GOOD/ER operations group X-entities that represent the same set (with the Γ^X-relationships indicating the "elements"), thereby create a unique $\lambda(G)$-entity for each coset. Without loss of generality, we may assume that the entity abstractions add the elements of $\mathit{CosetAut}(\mathcal{I})$ themselves. Consequently, the result of this step equals $\mathcal{K}'_4$.

We conclude this step with a calculation, which is used later on in this proof.

$$\forall a \in \mathit{Aut}(\mathcal{I}), \forall 4 \leq j \leq 7, \forall G \text{ subgroup of } \mathit{Aut}(\mathcal{I}) : \langle h' \rangle_j(a)(G) = \langle h' \rangle(a)(G) = a \circ G \qquad (5.2)$$

Step 5 : We next add the elements of $\mathit{Orbits}_h(\mathcal{I}' - \mathcal{I})$. An isolated entity can be added easily by means of an entity addition using an empty pattern. So we apply one entity addition to $\mathcal{K}'_4$, using an empty pattern, for each element of O of $\mathit{Orbits}_h(\mathcal{I}' - \mathcal{I})$, of an entity typed by a unique name for that orbit. Without loss of generality, we may assume that these entity additions add the orbits O themselves. Consequently, the result of this step equals $\mathcal{K}'_5$.

We also conclude this step of the construction with a calculation, which is used later on in the proof.

$$\forall a \in \mathit{Aut}(\mathcal{I}), \forall 5 \leq j \leq 7, \forall O \in \mathit{Orbits}_h(\mathcal{I}' - \mathcal{I}) : \langle h' \rangle_j(a)(O) = \langle h' \rangle(a)(O) = O \qquad (5.3)$$

Step 6 : We now add the entities of $\mathcal{I}' - \mathcal{I}$ together with their attributes and the Δ- and $\in$-relationships. Therefore, one entity addition is applied to $\mathcal{K}'_5$ for each element of $\mathit{Orbits}_h(\mathcal{I}' - \mathcal{I})$ (cf. Figure 5.32). Given the representative n_O of an orbit O, the entity addition has $\mathcal{K}'_5$ as pattern, and adds entities of type $\lambda(n_O)$, together with two relationships. A relationship of type $\in^{\lambda(O)}$ links the newly added entity to the orbit O, while the other relationship, of type $\Delta^{(\lambda(\mathit{st}_h(n_O)),\lambda(n_O))}$, links the newly added entity to the stabilizer $\mathit{st}_h(n_O)$. Additionally, the entity addition adds the attributes of n_O, which are identical for all entities in O. In summary, if A_1 through A_n are all attributes in ATTR such that $n_O \in \mathrm{owner}(A_i)(1 \leq i \leq n)$, then the entity addition for Step 6 is $\mathit{ENTADD}[\mathcal{K}'_5, \lambda(n_O),$
$\{(A_1, \mu[\mathrm{ATTR}](A_1)(n_O)), \ldots, (A_n, \mu[\mathrm{ATTR}](A_n)(n_O))\}, \emptyset, \{\in^{\lambda(O)}, \Delta^{(\lambda(\mathit{st}_h(n_O)),\lambda(n_O))}\},$
$\{(< (\epsilon_2^{\lambda(O)}, O) >, \epsilon_1^{\lambda(O)}), (< (\delta_2^{(\lambda(\mathit{st}_h(n_O)),\lambda(n_O))}, \mathit{st}_h(n_O)) >, \delta_1^{(\lambda(\mathit{st}_h(n_O)),\lambda(n_O))})\}]$

To see that these operations have the desired effect, recall that $\langle h' \rangle_5$ is surjective (cf. Lemma 5.43), so for each $b \in \mathit{Aut}(\mathcal{K}'_5)$, there is an $a \in \mathit{Aut}(\mathcal{I})$ such that $\langle h' \rangle_5(a) = b$. As a result, if n_O is added with relationships $(\epsilon_1^{\lambda(O)} : n_O, \epsilon_2^{\lambda(O)} : O)$ and $(\delta_1^{(\lambda(\mathit{st}_h(n_O)),\lambda(n_O))} : n_O, \delta_2^{(\lambda(\mathit{st}_h(n_O)),\lambda(n_O))} : \mathit{st}_h(n_O))$, then an entity n_O^b is also added with relationships $(\epsilon_1^{\lambda(O)} : n_O^b, \epsilon_2^{\lambda(O)} : \langle h' \rangle(a)(O))$ and $(\delta_1^{(\lambda(\mathit{st}_h(n_O)),\lambda(n_O))} : n_O^b, \delta_2^{(\lambda(\mathit{st}_h(n_O)),\lambda(n_O))} : \langle h' \rangle(a)\ (\mathit{st}_h(n_O)))$. The notation n_O^b indicates that this is the entity, added by the same entity addition as n_O as a result of the automorphism b of $\mathcal{K}'_5$. Recalling the equations (5.2) and (5.3), the relationships involving n_O^b equal $(\epsilon_1^{\lambda(O)} : n_O^b, \epsilon_2^{\lambda(O)} : O)$ and

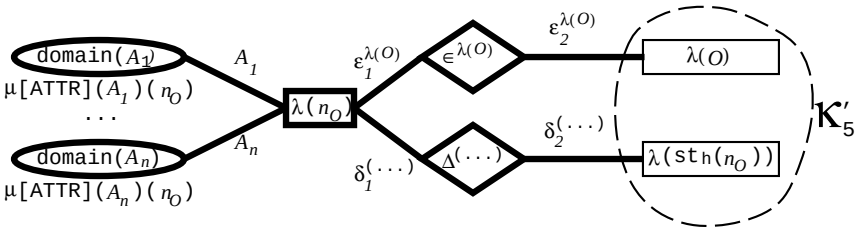


Figure 5.32: The GOOD/ER operations of Step 6 for the orbit O

$(\delta_1^{(\lambda(st_h(n_O)),\lambda(n_O))} : n_O^b, \delta_2^{(\lambda(st_h(n_O)),\lambda(n_O))} : a(st_h(n_O)))$. Consequently, we have added an entity for each pair consisting of an orbit and an arbitrary coset of the stabilizer of the representative of that orbit. We now want to apply Lemma 5.38 to $\mathcal{K}'_6$. In fact, this Lemma concerns the instance $\langle\mathcal{I}'\rangle$, but the only difference between $\langle\mathcal{I}'\rangle$ and $\mathcal{K}'_6$, is that $\mathcal{K}'_6$ lacks the relationships of $\mathcal{I}' - \mathcal{I}$. Since the absence of these relationships does not invalidate the proof of Lemma 5.38, we may apply it here. It follows that for each entity of $\mathcal{I}' - \mathcal{I}$ *exactly* one entity is added. Without loss of generality, we may assume that these entity additions add the nodes of $\mathcal{I}' - \mathcal{I}$ themselves. Consequently, the result of this step equals $\mathcal{K}'_6$.

Step 7 : Finally, we add the relationships of $\mathcal{I}' - \mathcal{I}$. For each such relationship, say $(P_1 : e_1, \ldots, P_n : e_n)$, a relationship addition is applied with $\mathcal{K}'_6$ as pattern, of a relationship $(P_1 : e_1, \ldots, P_n : e_n)$. Obviously, these operations add *at least* enough relationships. To see that they do not add too many relationships, note that for each such relationship and for each $b \in Aut(\mathcal{K}'_6)$, a relationship $(P_1 : b(e_1), \ldots, P_n : b(e_n))$ is also added. Since $\langle h'\rangle_6$ is surjective (cf. Lemma 5.43), there exists an automorphism a of $\mathcal{I}$ such that $b = \langle h'\rangle_6(a)$, so the relationship is actually $(P_1 : \langle h'\rangle_6(a)(e_1), \ldots, P_n : \langle h'\rangle_6(a)(e_n))$, or, by the definition of $\langle h'\rangle_6$ and $\langle h'\rangle$, $(P_1 : h(a)(e_1), \ldots, P_n : h(a)(e_n))$. Since $h(a)$ is an automorphism of $\mathcal{I}'$, this relationship must be present in $\mathcal{I}'$, and hence in $\langle\mathcal{I}'\rangle$. Consequently, the resulting instance *is* $\mathcal{K}'_7$, which equals $\langle\mathcal{I}'\rangle$.

Summarizing, the application to $\mathcal{I}$ of the six GOOD programs described above, results in $\langle\mathcal{I}'\rangle$. Restricting $\langle\mathcal{I}'\rangle$ to the given instance $\mathcal{I}'$ can be done very easily by deleting all entities typed AUT or by some identifier for an orbit or a subgroup of $Aut(\mathcal{I})$, as well as all diff-relationships. ■

We still have to prove that GOOD/ER is able to express *all* generic transformations $(\mathcal{I}, \mathcal{I}')$, even those where $\mathcal{I}'$ is not a super-instance of $\mathcal{I}$. Proving this becomes easy if we use Proposition 5.44. Before stating the final Proposition, leading to the proof of Theorem 5.33, we define a kind of superinstance for two instances, containing (almost) all "information" of both these instances.

Definition 5.45 *Let $\mathcal{I}$ and $\mathcal{I}'$ be ER instances. We define the instance $\mathcal{J}_{\mathcal{I},\mathcal{I}'}$ as follows:*

$$\begin{aligned} V_{\mathcal{J}_{\mathcal{I},\mathcal{I}'}} \;=\; & V_{\mathcal{I}} \cup V_{\mathcal{I}'} \cup \\ & \{l, \textit{an entity not in } V_{\mathcal{I}} \cup V_{\mathcal{I}'}\} \cup \\ & \{(P_1^K : l, P_2^K : e) \mid e \in V_{\mathcal{I}-\mathcal{I}'}, \lambda_{\mathcal{I}}(e) = K \in \textit{E-TYPE}\} \cup \end{aligned}$$

$$
\begin{array}{rcl}
 & & \{(P_1^{\neg} : e_1, \ldots, P_n^{\neg} : e_n) \mid (P_1 : e_1, \ldots, P_n : e_n) \in V_{\mathcal{I}-\mathcal{I}'}\} \\
W_{\mathcal{J}_{\mathcal{I},\mathcal{I}'}} & = & W_{\mathcal{I}} \cup \\
 & & \{(x, \alpha, y) \in W_{\mathcal{I}'} \mid \alpha \notin \textit{ATTR}\} \cup \\
 & & \{\textit{all roles corresponding to the relationships in } V_{\mathcal{J}_{\mathcal{I},\mathcal{I}'}}\} \\
\lambda_{\mathcal{J}_{\mathcal{I},\mathcal{I}'}}(n) & = & \lambda_{\mathcal{I}}(n) \qquad (n \in V_{\mathcal{I}}) \\
 & = & \lambda_{\mathcal{I}'}(n) \qquad (n \in V_{\mathcal{I}} - V_{\mathcal{I}'}) \\
 & = & \texttt{difference} \qquad (n = l) \\
 & = & R^K \qquad (n \in \{(P_1^K : l, P_2^K : e) \mid e \in V_{\mathcal{I}-\mathcal{I}'}, \lambda_{\mathcal{I}}(e) = K \in \textit{E-TYPE}\}) \\
 & = & R^{\neg} \qquad (n \in \{(P_1^{\neg} : e_1, \ldots, P_n^{\neg} : e_n) \mid (P_1 : e_1, \ldots, P_n : e_n) \in V_{\mathcal{I}-\mathcal{I}'}, \\
 & & \qquad\qquad \lambda_{\mathcal{I}}((P_1 : e_1, \ldots, P_n : e_n)) = R\}) \\
\pi_{\mathcal{J}_{\mathcal{I},\mathcal{I}'}} & = & \pi_{\mathcal{I}'}
\end{array}
$$

We assume that `difference` *is a new entity type, that for each entity type K, R^K is a new relationship type that for each relationship type R, $R^{\neg}$ is a new relationship type, and that for each role P, P^K and $P^{\neg}$ are new roles.*

□

Besides containing all entities and relationships of both $\mathcal{I}$ and $\mathcal{I}'$, $\mathcal{J}_{\mathcal{I},\mathcal{I}'}$ also indicates which entities and relationships of $\mathcal{I}$ are absent in $\mathcal{I}'$. For entities, say of type K, this is indicated by linking them to the "auxiliary" entity l by means of relationships of type R^K. For relationships (say of type R), this is indicated by adding relationships of type $R^{\neg}$ with the same participants. Not incorporating the attributes of $\mathcal{I}'$ in the instance $\mathcal{J}_{\mathcal{I},\mathcal{I}'}$ ensures that the superinstance is indeed a well-defined instance, since the union of two instances is in general not an instance: conflicts may arise with the uniqueness of attributes (cf. Definition 5.4). Attributes are therefore dealt with separately in the proof of the following proposition, which concludes the proof of the Theorem 5.33:

Proposition 5.46 *Let $\mathcal{I}$ and $\mathcal{I}'$ be instances such that $(\mathcal{I}, \mathcal{I}')$ is a generic transformation. Then $\mathcal{I} \overset{GOOD/ER}{\Longrightarrow} \mathcal{I}'$.*

Proof In this proof, we give (or prove the existence of) three GOOD/ER programs. The first one maps $\mathcal{I}$ to the instance $\mathcal{J}_{\mathcal{I},\mathcal{I}'}$, the second maps $\mathcal{J}_{\mathcal{I},\mathcal{I}'}$ to $\mathcal{I}'$ but with the attributes of $\mathcal{I}$, and the third maps the latter instance to $\mathcal{I}'$.

The existence of the first transformation is shown using Proposition 5.44. Therefore we define the following mapping h' from *Aut*($\mathcal{I}$) to *Aut*($\mathcal{J}_{\mathcal{I},\mathcal{I}'}$). Let $a \in$ *Aut*($\mathcal{I}$). Let h be the extension-morphism corresponding to the generic transformation $(\mathcal{I}, \mathcal{I}')$.

1. $h'(a)(n) = a(n)$, for $n \in V_{\mathcal{I}}$;
2. $h'(a)(n) = h(a)(n)$, for $n \in V_{\mathcal{I}'}$;
3. $h'(a)(l) = l$;

4. the behavior of h' on the relationships of $\mathcal{J}_{\mathcal{I},\mathcal{I}'}$ is defined as $h'(a)((P_1 : n_1, \ldots, P_k : n_k)) := (P_1 : h'(a)(n_1), \ldots, P_k : h'(a)(n_k))$.

We omit the tedious but straightforward verification that h' is indeed an extension morphism of type $(\mathcal{I}, \mathcal{J}_{\mathcal{I},\mathcal{I}'})$. Applying Proposition 5.44, we know that there exists a GOOD/ER program that maps $\mathcal{I}$ to $\mathcal{J}_{\mathcal{I},\mathcal{I}'}$.

The second GOOD/ER program deletes from $\mathcal{J}_{\mathcal{I},\mathcal{I}'}$ all entities and relationships not in $\mathcal{I}'$ (cf. Figure 5.33). It first deletes all relationships of some type R for which there exists a relationship of type $R^\neg$ with the same participants. Next, all relationships of type $R^\neg$ are deleted. Then all nodes are deleted which are linked to l, which is supposed to be the single entity of type `difference`. Finally the entity l itself is deleted. Let $\bar{\mathcal{I}}$ be the outcome of this second GOOD/ER program.

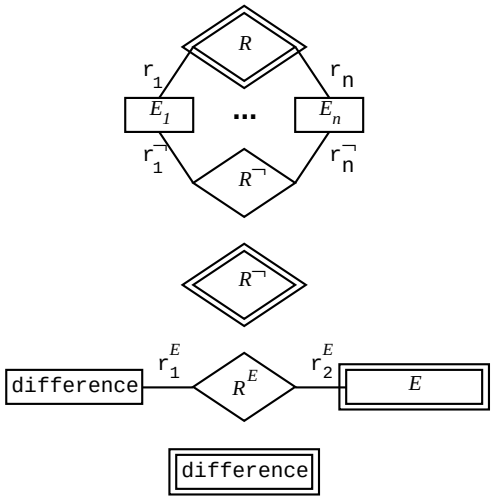


Figure 5.33: The GOOD/ER operations for the proof of Proposition 5.46 for the relationship type R and the entity type E

The only difference left between $\bar{\mathcal{I}}$ and the instance $\mathcal{I}'$ is that entities which were also present in $\mathcal{I}$, still have their attribute values from $\mathcal{I}$. Therefore, let e be an entity in $\mathcal{I} \cap \mathcal{I}'$ whose α-attribute has the value d_1 in $\mathcal{I}$ (and hence in $\bar{\mathcal{I}}$) and the value d_2 in $\mathcal{I}'$. Then we apply to $\bar{\mathcal{I}}$ the attribute update *ATTUPD*$[\bar{\mathcal{I}}, \alpha, e, d_2]$. Such an attribute update is performed for each attribute to be modified. The reasoning which shows that all these attribute updates together have the desired effect, is similar to the proof of Step 7 in the proof of Proposition 5.44.

Combining the above three GOOD/ER programs, we get the desired GOOD/ER program mapping $\mathcal{I}$ to $\mathcal{I}'$. ■

Combining Propositions 5.34 and 5.46, Theorem 5.33 easily follows.

We conclude this section with a few corollaries, in which some simple "classes" of transformations are shown to be computable in GOOD/ER.

Corrolary 5.47 *If $\mathcal{I}$ and $\mathcal{I}'$ are instances with empty intersection, then $\mathcal{I} \overset{GOOD/ER}{\Longrightarrow} \mathcal{I}'$.* ■

Indeed, the homomorphism mapping each automorphism of $\mathcal{I}$ to the identity on $\mathcal{I}'$ always satisfies the extension property.

Corrolary 5.48 *Let $\mathcal{I}$ be the empty ER instance, and $\mathcal{I}'$ an arbitrary instance. Then $\mathcal{I} \overset{GOOD/ER}{\Longrightarrow} \mathcal{I}'$.* ■

This is just a specialization of the previous Corollary. It shows that any ER instance can be generated by a GOOD/ER program, starting from scratch.

Corrolary 5.49 *Let $\mathcal{I}$ be an instance such that Aut$(\mathcal{I}) = \{id_{\mathcal{I}}\}$, and let $\mathcal{I}'$ be an arbitrary instance. Then $\mathcal{I} \overset{GOOD/ER}{\Longrightarrow} \mathcal{I}'$.* ■

The intuition behind this Corollary is the fact that in an instance which has only the identity mapping as automorphism, any node is clearly distinguishable from any other node by means of some pattern (e.g., the instance itself), and hence any conceivable transformation on it may be expressed by a GOOD/ER transformation.

Chapter 6

Conclusions

In this thesis, we studied the applicability of the theory of graph grammars and graph rewrite systems to the definition of syntax and semantics of visual database languages. In Chapters 4 and 5, we successfully demonstrated the theory's applicability in two different ways. In this concluding chapter, we contrast these two approaches, and discuss the outcome of our study.

The major differences between the definitions of respectively GOQL/EER (cf. Section 4.2) and GOOD/ER (cf. Section 5.2) may be summarized as follows (cf. Table 6.1):

	syntax	semantics	
GOQL/EER	constructive (graph rewrite rules)	attribute evaluations	operational (translation)
GOOD/ER	descriptive	graph rewriting	denotational (I/O pairs)

Table 6.1: Contrasting GOQL/EER to GOOD/ER

Syntax : The syntax of GOQL/EER is defined in a *constructive* way, namely by means of the graph rewrite rules of a graph grammar.

In contrast, the syntax of GOOD/ER is defined in a purely *declarative* manner, namely by summing up the constituents of "sentences" in the language. The latter is clearly illustrated in e.g., Definition 5.14 of entity addition syntax.

Semantics : The semantics of GOQL/EER is defined in an *operational* way, namely by providing a translation from GOQL/EER queries to SQL/EER queries, in the form of attribute evaluations within the graph rewrite rules of the graph grammar. Note that the actual evaluation of GOQL/EER queries on a database instance is not incorporated in the definition.

In contrast, the semantics of GOOD/ER is defined in a *denotational* manner, by an algorithmic description of the resulting instance of the application of a basic GOOD/ER operation to a given input instance.

Hence the most prominent difference lies in the role played by graph rewrite rules in the respective definitions: whereas both syntax and semantics of GOQL/EER queries are defined *by means of* graph

rewrite rules, basic GOOD/ER operations are *themselves* graph rewrite rules. The latter is easily motivated: looking upon database manipulation as graph rewriting follows quite naturally from looking upon database instances as graphs. In contrast to what is suggested by the table of contents of this thesis, the study of the GOOD/ER language chronologically preceded the study of GOQL/EER. Having defined the semantics of GOOD/ER as graph rewriting, the question then arose whether graph rewrite rules could also be used to *define* a graph-based language, the semantics of which has basically nothing to do with graph rewriting. The (positive) answer to this question is the graph grammar specification of GOQL/EER, as listed in Appendix C.

Another difference between GOQL/EER and GOOD/ER lies in their respective expressive power. Even though GOQL/EER is a query language, while GOOD/ER is a database manipulation language, we can still compare them on the basis of their capacity to "identify" information within a database. As an example, the GOOD/ER program shown in Figure 5.19 deletes all cash cards that do not access any account. But before this can happen, the program first has to *identify* these accounts. From the fact that a negative condition cannot be expressed in a GOQL/EER pattern, it follows readily that no GOQL/EER query can possibly perform such an identification, while in GOOD/ER, negation is in a sense "simulated" using deletion.

On a more general level, GOOD/ER is a generically complete language (as discussed extensively in Section 5.3) while GOQL/EER can only express conjunctive queries (i.e., queries whose equivalent in logic consists simply of a conjunction of atomic conditions). Intuitively, this major difference in expressive power between the two languages discussed in this thesis, stems from the fact that while GOOD/ER uses pattern matching combined with the powerful paradigm of graph rewriting, GOQL/EER uses only pattern matching.

6.1 Lessons Learned from GOQL/EER

From the act of specifying GOQL/EER by means of a graph grammar, some important lessons may be drawn. First, we have shown that the use of an attributed graph grammar allows the seemless integration of the definition of both (abstract) syntax and semantics of a graph-based language into one single specification, in the same way the use of attributed string grammars enables a similarly integrated specification of textual languages. Second, when defining graph-based, and particularly diagrammatic languages, the ability to work with graphs as "basic primitives" in the graph rewrite rules surely has its advantages over having to work with some general purpose specification language in which these graphs first have to be (sometimes cryptically) encoded themselves. It allows one to concentrate on the essentials of the graph-based language itself, rather than on the details of the applied specification language.

Besides, the specification process of GOQL/EER also taught us a few general lessons on how to write specifications in PROGRES, or, more generally, in an operational specification language based on programmed graph rewrite systems. Most importantly, both the process of writing, as well as the structure of the resulting specification, allowed us to derive the following rudimentary "life-cycle" for the construction of a graph grammar specification for a visual language:

1. the given visual representation of the language must first be mapped onto a graph model (cf. Section 4.2.1). This correspondence is called a *visual metaphor* by Haber et al. [HIL94], who

applied this idea to the visual representation of database schemes. The major motivation they offer for making a clear separation between the visual representation of a database scheme and the database scheme itself, is that as a consequence, the visual model becomes more flexible. Recently it was argued by Cattarci et al. [CCM95] that the idea of a visual metaphor should be extended to visual query languages as well, thereby providing an a posteriori motivation for the work presented in this thesis.

2. Next, the basic components of this graph model, i.e., the nodes, edges, and non-structural characteristics should be recognized, and mapped to a graph scheme, i.e., a collection of node classes, node types, and edge types, including attribute declarations.

3. Certain restrictions on the way in which these basic components may be composed into syntactically correct graphs (such as the fact that the graph should be acyclic), cannot be captured in a graph scheme alone. These restrictions should be reflected in a set of graph rewrite rules.

4. If an operational definition of the semantics of the language is to be incorporated into the graph grammar specification, then the foregoing three steps should be reiterated. Concretely, the design of the graph model should be reevaluated, resulting in the addition and/or modification of both the graph scheme and the set of graph rewrite rules.

As mentioned in Chapter 1, going through this life-cycle is most certainly facilitated by the operational character of the PROGRES specification language in addition to the availability of the PROGRES environment supporting the editing, analysis and execution of PROGRES specifications.

Besides the at times unstable character of this environment (within six months, four different releases of the system were used!), the most significant problem encountered while applying the life-cycle described above to the PROGRES specification of GOQL/EER was caused by the diverging way in which nodes and edges are treated in the PROGRES formalism. The exact problem was extensively covered in Section 4.2. In summary, by the absence of an edge class hierarchy, productions cannot take edge types as parameters. Consequently, language elements such as attributes and roles, which in the visual representation of GOQL/EER have a clear “edge-like“ character, have to be modeled by nodes. This in turn imposes the use of meta-attributes for the enforcement of crucial structural integrity constraints, which surely does not add to the clarity and understandability of the resulting specification.

A secondary problem encountered in the process of writing the GOQL/EER specification was the absence of mechanisms for structuring this specification. Indeed, as remarked in Chapter 3, PROGRES’ “sections” are merely a syntactical way of structuring a specification, and have no semantical meaning whatsoever. In principle, all declarations (of node types, paths, productions,…) in the GOQL/EER specification could be included in the specification in any arbitrary order. Structuring mechanisms for graph grammar based specification languages are therefore currently a topic of active research [EE94].

6.2 Lessons Learned from GOOD/ER

The most prominent lesson to be learned from the definition of GOOD/ER is the fact that with their typical form of an action to be performed on the outcome of a query, manipulations on a database

representable as a graph may be naturally represented by graph rewrite rules. As a major benefit, this representation offers an equally natural graphical syntax. The benefits of the Graph-Oriented Object Database model are demonstrated in an experimental environment called *xgood* [GPT93, GPTVdB93], supporting the editing and execution of GOOD programs.

Instead of developing a general purpose graph rewrite formalism (such as PROGRES) it appeared to be more suitable to define a number of categories of rules specially tailored towards typical database manipulations such as the addition of entities or the update of attributes. Forcing the user to apply only rewrite rules that fall within these categories, allows the enforcements of structural integrity constraints on the database instance. At the end of Section 5.2, it was also shown how on top of these tailored graph rewrite rules, the usual collection of typical programming constructs (such as sequences and procedures) traditionally found textual languages, may be superimposed.

Another major benefit of formalizing a graph-oriented database manipulation language by means of graph-theoretical notions was clearly illustrated in Section 5.3, in which we formally characterized the expressive power of the GOOD/ER language. Formalizing the expressive power of object-creating query languages was, at the time of writing, considered a tough and open problem, as e.g., mentioned by Abiteboul and Kanellakis in their influential paper on object identity and query languages [AK89, p.161]. In the proof of Theorem 5.33, in which we present a solution to this problem, the graph-theoretical characteristics of GOOD/ER are most prominently used. This first of all supports the claim made in Section 1.2 that the formal definition of a language enables the formal study of properties of that language. More specifically, the fact that the results presented in Section 5.3, and especially their proofs, rely so heavily on the graph-based nature of the GOOD/ER model, generally argues in favor of a graph-based look on object-oriented databases, being (at least from the structural point of view) networks of objects.

6.3 Open Issues for Future Research

Any research activity raises questions, perhaps even more than it answers. The issue of formally defining visual languages by means of graph rewriting, being part of the currently very active area of research in visual language formalization, certainly makes no exception to this rule. We therefore conclude this thesis with a series of open problems, raised, but not answered, in the course of preparing and writing this thesis:

- How does the work presented in this thesis compare precisely to other work on visual language definition, done in the visual language research community? Are definitions using graph rewrite systems easier to write, more readable, more compact,…? This should be verified experimentally, by defining one and the same visual language by means of graph rewrite systems, as well as by means of some other technique.

  Can we eventually conclude from such a comparison that graph rewrite systems may eventually replace attributed string grammars grammars as *the* standard mechanism for formalizing languages in general?

- The borderline between the concrete and abstract syntax (discussed in Section 1.3) is not that clear cut, especially in the context of visual languages. Would it be desirable and/or feasible

to incorporate layout information, spatial relationships, or other aspects from the definition of a visual language into a graph grammar-based specification for them?

- How can graph rewrite systems be used to define visual languages outside the realm of databases? Is the applicability of graph rewrite systems restricted to diagrammatic languages, or can they also cope with iconic ones?
- If PROGRES would offer mechanisms for structuring a specification, for instance a modularization concept [EE94], would this allow for improved specifications of visual languages? Would it for instance be possible to specify modules, reusable in the definition of different languages?
- …

Appendix A

Notational Conventions

$\mathcal{F}(S)$	the set of all finite sets with elements from the set S
$\{x_1, \ldots, x_n\}$	the set containing the elements x_1 through x_n
$\emptyset$	the empty set
$\mathcal{B}(S)$	the set of all finite bags or multisets with elements from the set S
S^*	the set of all finite lists of elements from the set S
S^+	the set of all finite non-empty lists of elements from the set S
$< x_1, \ldots, x_n >$	a list with elements x_1 through x_n
$S_1 \times \ldots \times S_n$	the cartesian product of the sets S_1 through S_n
$f : S_1 \times \ldots \times S_n \to T$	a total function from with domain $S_1 \times \ldots \times S_n$ and range T
$f : S_1 \times \ldots \times S_n \nrightarrow T$	a partial function from with domain $S_1 \times \ldots \times S_n$ and range T
$f \mid_V$	the restriction of the function f to the subset V of its domain

E, E_n	entity types
e, e_n	entities
R, R_n	relationship types
r, r_n	relationships
P, P_n	roles
D, D_n	data types
d, d_n	(data) values
A, A_n	attributes
C, C_n	components
T, T_n	constructors

V, V_G	set of vertices (or nodes) (of a graph G)
v, v_n	vertices or nodes
W, W_G	set of edges (of a graph G)
$\alpha, \beta, \ldots$	edge labels
m, m_n	embeddings
i, h	isomorphism, homomorphism
a, b	automorphisms

Appendix B

EBNF-grammar for SQL/EER

Auxiliary Production Rules

STRING ::= CHAR [STRING]
CHAR ::= A| …| Z| a| …| z| 0| …| 9| …
DATAOPNS ::= STRING
ATTRIBUTE ::= STRING
ROLE ::= STRING
AGGROPNS ::= STRING
ENTITYTYPE := STRING
INTEGER := STRING
DATAPRED := STRING
COMPONENT := STRING
CONSTRUCTION := STRING
RELSHIPTYPE := STRING
VARIABLE := STRING

Queries

QUERY ::= SFW-TERM
| TERM
SFW-TERM ::= [**select** TERMLIST] [**from** DECLLIST] [**where** FORMULA]
TERMLIST ::= TERM [, TERMLIST]
DECLLIST ::= DECL [, DECLLIST]

Declarations

DECL ::= VARIABLE **in** ENTITYTYPE
| VARIABLE **in** RELSHIPTYPE
| VARIABLE **in** RANGEUNION

Ranges

RANGEUNION ::= RANGE [**union** RANGEUNION]
RANGE ::= ENTITYTYPE
| RELSHIPTYPE
| TERM

Terms

TERM ::= VARIABLE
| STRING
| DATAOPNS TERM
| ’(’ TERM DATAOPNS TERM ’)’
| DATAOPNS ’(’ TERMLIST ’)’
| TERM ’.’ ATTRIBUTE
| TERM ’.’ COMPONENT
| ATTRIBUTE
| COMPONENT
| TERM ’.’ ROLE
| TERM ’.’ ENTITYTYPE
| **distinct** TERM
| TERM ’[’ INTEGER ’]’
| **ind** ’(’ TERM ’)’
| AGGROPNS ’(’ TERM ’)’
| TERM ’.’ INTEGER
| ’(’ SFW-TERM ’)’

Formulas

FORMULA ::= DATAPRED TERM
| TERM DATAPRED TERM
| DATAPRED ’(’ TERMLIST ’)’
| TERM ’=’ TERM
| TERM **in** TERM
| TERM **is** null
| TERM **is-a** ENTITYTYPE
| RELSHIPTYPE ’(’ PARTLIST ’)’
| PARTICIPANT RELSHIPTYPE PARTICIPANT
| TERM
| **not** ’(’ FORMULA ’)’
| ’(’ FORMULA **and** FORMULA ’)’
| ’(’ FORMULA **or** FORMULA ’)’
| **exists** DECLLIST ’:’ FORMULA
| **forall** DECLLIST [**with** FORMULA] ’:’ FORMULA
PARTLIST ::= PARTICIPANT [’,’ PARTLIST]

```
PARTICIPANT      ::= ROLE ':' TERM
                   | TERM
```

Appendix C

The PROGRES specification for GOQL/EER

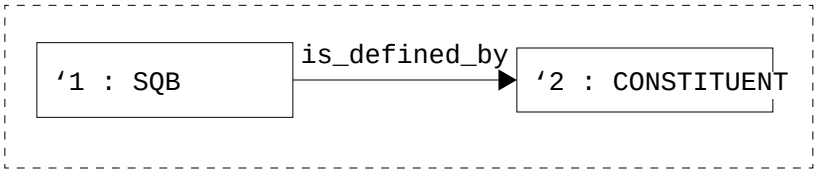
The transactions GOQL1 through GOQL14 at the end of this specification generate the abstract syntax graphs of almost all (partial) GOQL/EER queries, as well as of the graphical part of the HQL/EER queries presented in Chapter 4. The precise correspondence is as follows:

GOQL1	Figure 4.3
GOQL2	Figure 4.5
GOQL3	Figure 4.8
GOQL4	Figure 4.10
GOQL5	Figure 4.12
GOQL6	Figure 4.15
GOQL7	Figure 4.20
GOQL8	Figure 4.21
GOQL9	Figure 4.22
GOQL10	Figure 4.26
GOQL11	Figure 4.27
GOQL12	Figure 4.28
GOQL13	Figure 4.32
GOQL14	Figure 4.33

```
spec GOQL

   from LongStrings import

      types
         text;

      functions
         EmptyText      : -> text,
         Text           : ( string) -> text,
         &&             : ( text, string) -> text,
         ==             : ( text, string) -> boolean,
         Concat         : ( text, text) -> text;

   end;

   function concom : ( S1 : text ; S2 : text ) -> text =
      [ S1 == "" :: S2
      | S2 == "" :: S1
      | Concat ( S1 && ", ", S2 ) ]
   end;

   function conand : ( S1 : text ; S2 : text ) -> text =
      [ S1 == "true" :: S2
      | S2 == "true" :: S1
      | Concat ( S1 && " and ", S2 ) ]
   end;

   section FixedGraphScheme

      node class QUERY_ELEM end;

      node class PART_OF_COMPLEX is a QUERY_ELEM end;

      node class ENT_REL is a QUERY_ELEM end;

      node class ENTITY is a ENT_REL, PART_OF_COMPLEX end;

      node class RELSHIP is a ENT_REL end;

      node class ROLE
         meta
            rel : type in RELSHIP [1:1];
            ent : type in ENTITY [0:n];
      end;

      edge type role2e : ROLE -> ENTITY [1:1];

      edge type role2r : ROLE -> RELSHIP [1:1];

      node class VALUE is a QUERY_ELEM end;

      node class ATOMIC_VALUE is a VALUE, PART_OF_COMPLEX
         intrinsic
            Value : string;
      end;
```

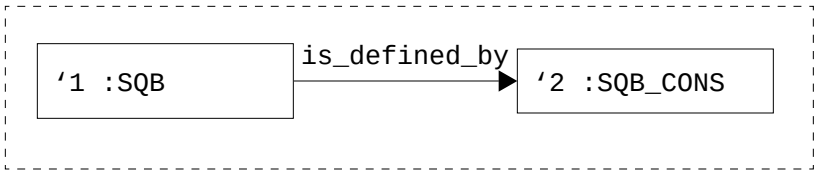
```
node class COMPLEX_VALUE is a VALUE
   derived
      Elem_Type : type in QUERY_ELEM [0:n];
end;

node class ATTRIBUTE
   meta
      entrel : type in ENT_REL [0:n];
      val : type in VALUE [1:1];
end;

edge type attribute2er : ATTRIBUTE -> ENT_REL [1:1];

edge type attribute2v : ATTRIBUTE -> VALUE [1:1];

node class COMPONENT
   meta
      cent : type in ENTITY [0:n];
      comp : type in COMPLEX_VALUE [1:1];
end;

edge type component2e : COMPONENT -> ENTITY [1:1];

edge type component2c : COMPONENT -> COMPLEX_VALUE [1:1];

node class SET_VALUE is a COMPLEX_VALUE
   intrinsic
      Singleton : boolean := false;
end;

edge type cont : SET_VALUE -> PART_OF_COMPLEX;

node class MVALUE is a COMPLEX_VALUE end;

node class BAG_VALUE is a MVALUE end;

node class LIST_VALUE is a MVALUE end;

node class MMEMBER
   intrinsic
      Index : integer := 0;
end;

node type mmember : MMEMBER end;

edge type contains_mm : MVALUE -> MMEMBER;

edge type mm_is_poc : MMEMBER -> PART_OF_COMPLEX [1:1];

node class DERIVED_SQL
   derived
      SFW_Term : text = EmptyText;
end;
```

```
node class SQB is a BAG_VALUE, DERIVED_SQL, QUERY_ELEM
   redef derived
      Elem_Type =
         ((self.=OutCons=> : CONSTITUENT [1:1]).
          -cons_is_n-> : QUERY_ELEM [1:1]).type;
      SFW_Term =
       Concat (
        Concat (
         Concat (
          Text ( "( select " ),
          concom (
           concom ( EmptyText, all self.=OutCons=>.Term ),
           concom ( EmptyText, all self.=OutSQB_Cons=>.SFW_Term ) ) ),
         Concat (
          Text ( " from " ),
          concom ( EmptyText,
                   all self.-is_defined_by->.Declaration ) ) ),
        Concat (
         Text ( " where " ),
         conand ( Text ( "true" ),
                  all self.-is_defined_by->.Formula ) ) )
       && " )" ;
end;

node type sqb : SQB end;

node class CONSTITUENT
   intrinsic
      Formula : text := Text ( "true" );
      Declaration : text := EmptyText;
      Term : text := EmptyText;
      Output : boolean := false;
end;

node type constituent : CONSTITUENT end;

edge type is_defined_by : SQB -> CONSTITUENT;

edge type cons_is_n : CONSTITUENT -> QUERY_ELEM [1:1];

node class SQB_CONS is a CONSTITUENT, DERIVED_SQL
   redef derived
      SFW_Term = self.(-cons_is_n-> : SQB [1:1]).SFW_Term;
end;

node type sqb_cons : SQB_CONS end;

path OutCons : SQB -> CONSTITUENT [0:n] =
   '1 => '2 in
```

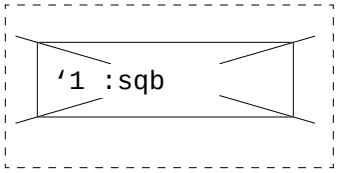



```
   condition '2.Output;
end;
```

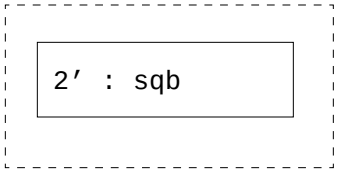
```
   path OutSQB_Cons : SQB -> SQB_CONS [0:n] =
      ‘1 => ‘2 in
```




```
      condition ‘2.Output;
   end;
end;

section Productions

   production Add_first_SQB ( out NewS : SQB ) =
```

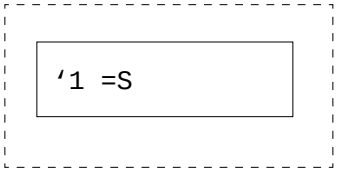



```
      ::=
```

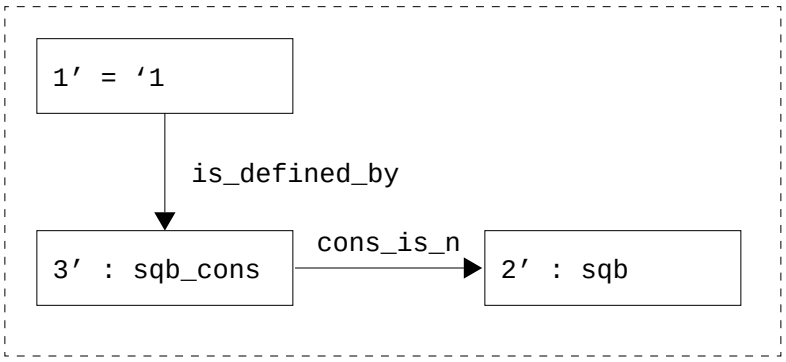



```
      return NewS := 2’;
   end;
   (* Add_first_SQB *)
```

```
production Add_SQB ( S : SQB ; out NewS : SQB ; out SC : SQB_CONS ) =
```

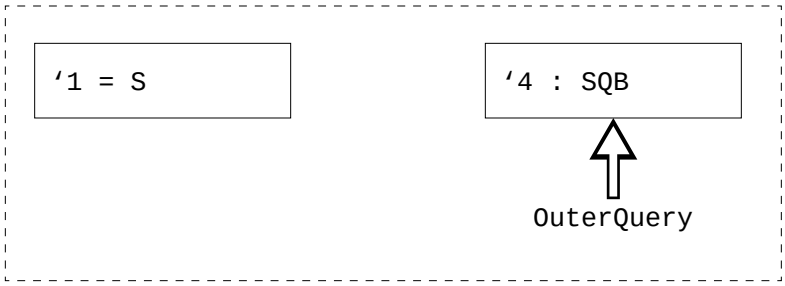



```
   ::=
```

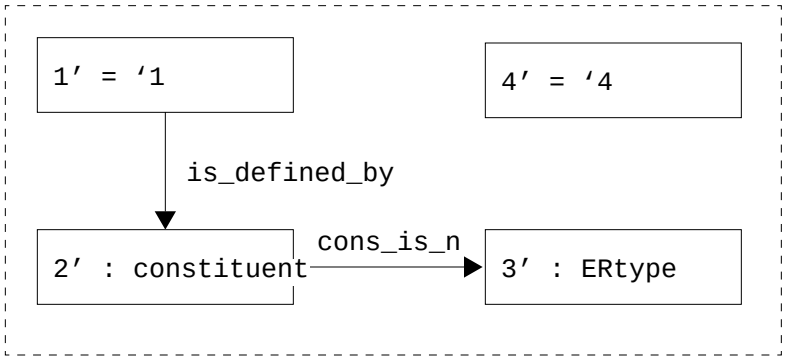



```
   return NewS := 2’;
          SC := 3’;
end;
(* Add_SQB *)

restriction OuterQuery : SQB =
   not def <-cons_is_n-
end;

path recCons : SQB -> CONSTITUENT =
-is_defined_by-> & (-cons_is_n-> & instance of SQB & -is_defined_by->) *
end;
```

```
production Add_ER
   ( S : SQB ; VarName : string ; ERtype : type in ENT_REL ;
     out E : ENT_REL ; out C : CONSTITUENT )                =
```

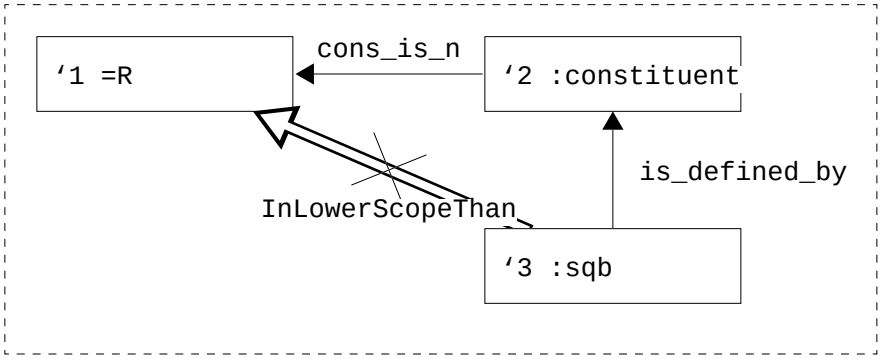



```
   ::=
```

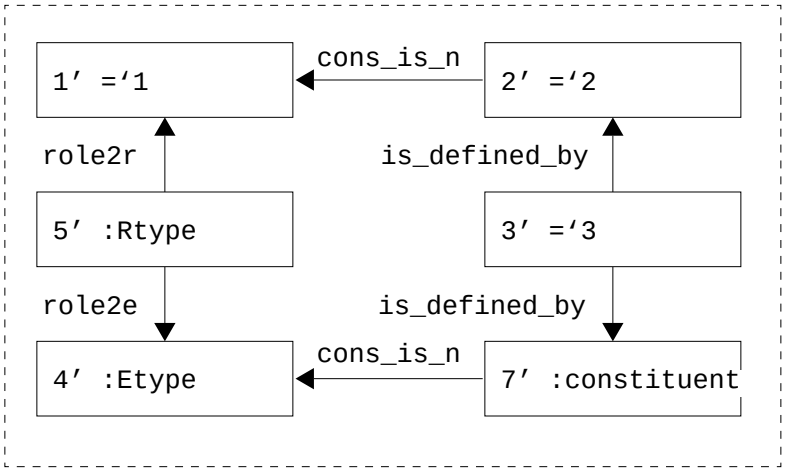



```
   folding { ‘1, ‘4 };
   condition not (Text ( VarName ) in ‘4.=recCons=>.Term);
   transfer 2’.Term := Text ( VarName );
            2’.Declaration := Text ( VarName ) && " in "
                              && string ( ERtype );
   return E := 3’;
          C := 2’;
end;
(* Add_ER *)

path InLowerScopeThan : SQB -> QUERY_ELEM =
   (<-cons_is_n- & instance of SQB_CONS & <-is_defined_by-) +
   & -is_defined_by-> & -cons_is_n->
end;
```

```
production Add_Role
   ( R : RELSHIP ; Etype : type in ENTITY ; Rtype : type in ROLE ;
     out E : ENTITY ; out C : CONSTITUENT )
=
```

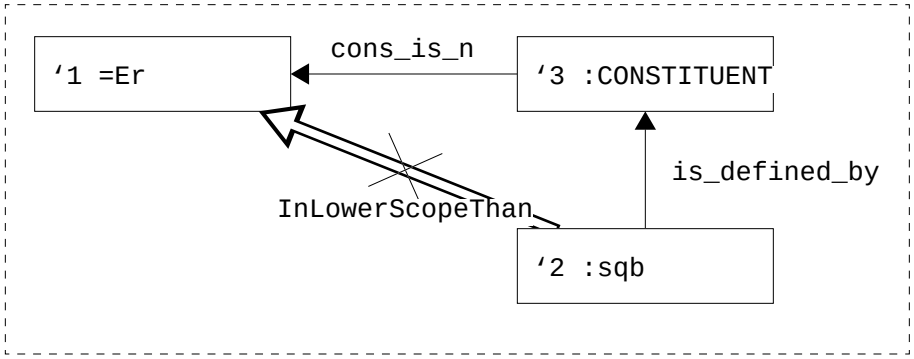



```
   ::=
```

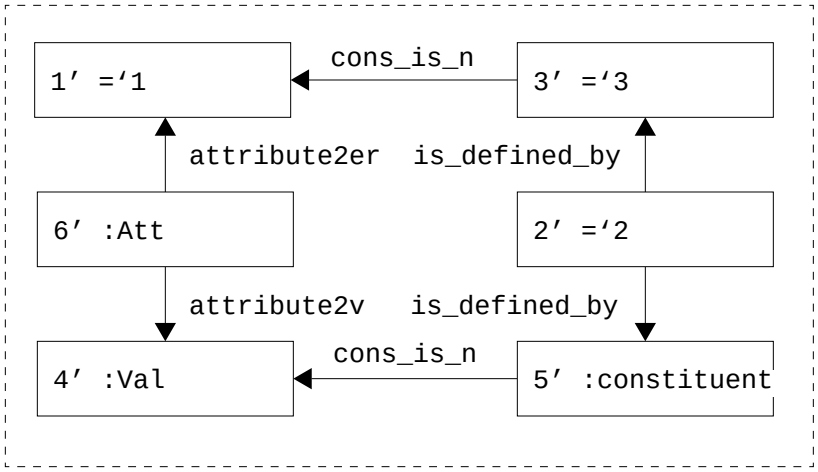



```
   condition Rtype.rel = R.type;
             Etype in Rtype.ent;
   transfer 7’.Term := ‘2.Term && "." && string ( Rtype );
   return E := 4’;
          C := 7’;
end;
(* Add_Role *)
```

```
production Add_Attribute
   ( Er : ENT_REL ; Att : type in ATTRIBUTE ; Val : type in VALUE ;
     out v : VALUE ; out C : CONSTITUENT )
=
```

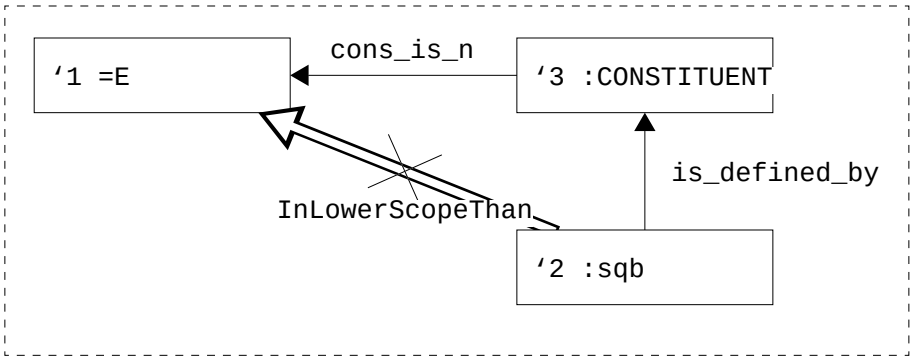



```
   ::=
```

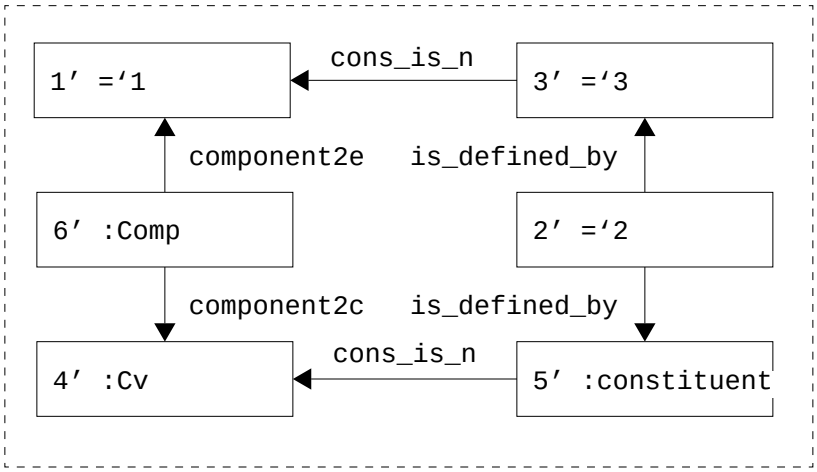



```
   condition Er.type in Att.entrel;
             Att.val = Val;
   transfer 5’.Term := ‘3.Term && "." && string ( Att );
   return v := 4’;
          C := 5’;
end;
(* Add_Attribute *)
```

```
production Add_Component
 ( E : ENTITY ; Comp : type in COMPONENT ; Cv : type in COMPLEX_VALUE ;
   out e : COMPLEX_VALUE ; out C : CONSTITUENT )                        =
```

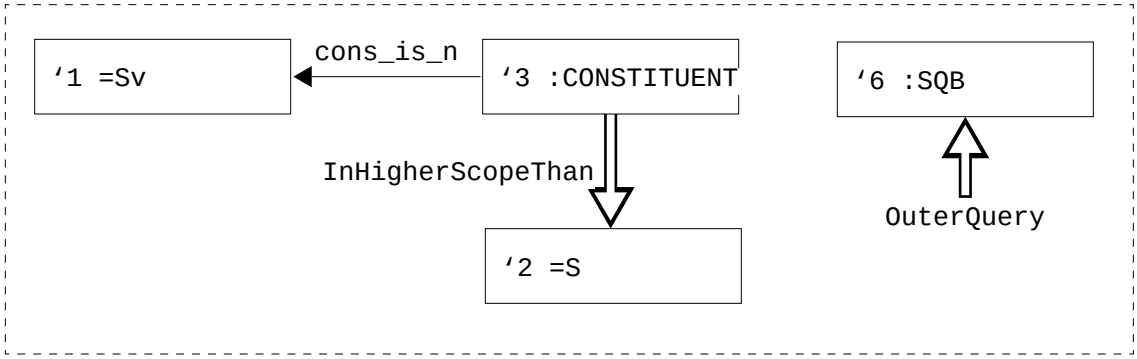



```
   ::=
```

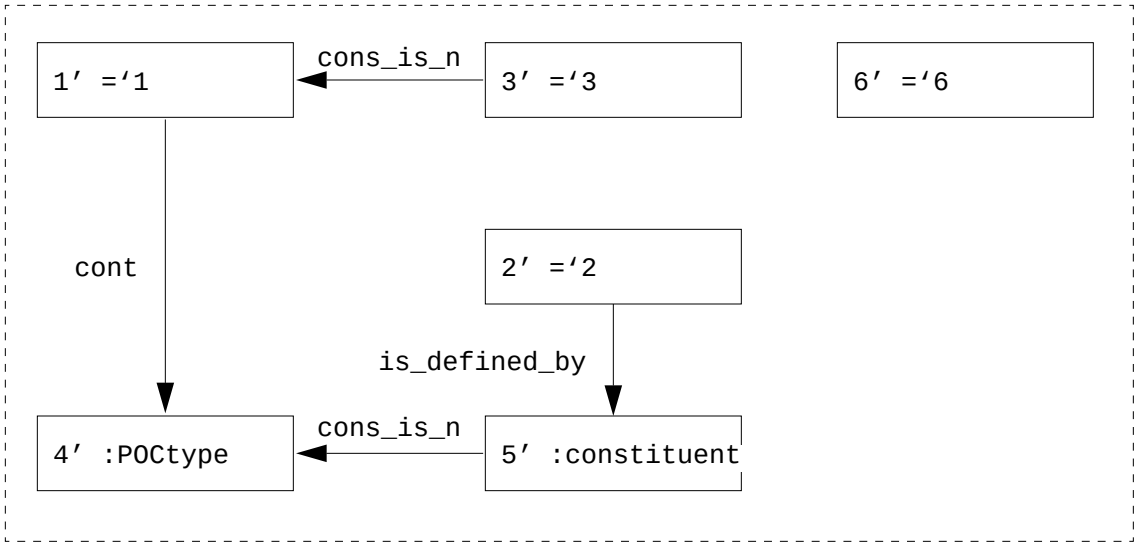



```
   condition E.type in Comp.cent;
             Comp.comp = Cv;
   transfer 5’.Term := ‘3.Term && "." && string ( Comp );
   return e := 4’;
          C := 5’;
end;
(* Add_Component *)

path InHigherScopeThan : CONSTITUENT -> SQB =
   <-is_defined_by- &
      (-is_defined_by-> & instance of SQB_CONS & -cons_is_n->
      & instance of SQB) *
end;
(* InHigherScopeThan *)
```

```
production Add_to_Set
   ( Sv : SET_VALUE ; S : SQB ; POCtype : type in PART_OF_COMPLEX ;
      VarName : string ; out POC : PART_OF_COMPLEX ;
      out C : CONSTITUENT ) =
```

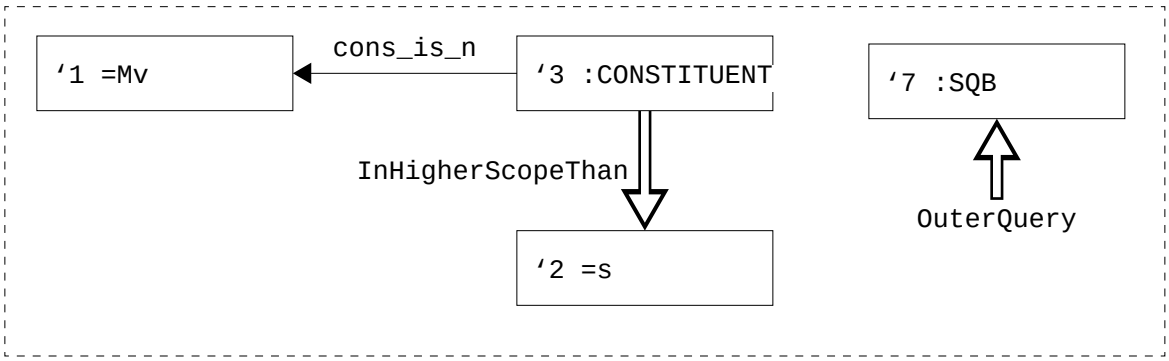



```
   ::=
```

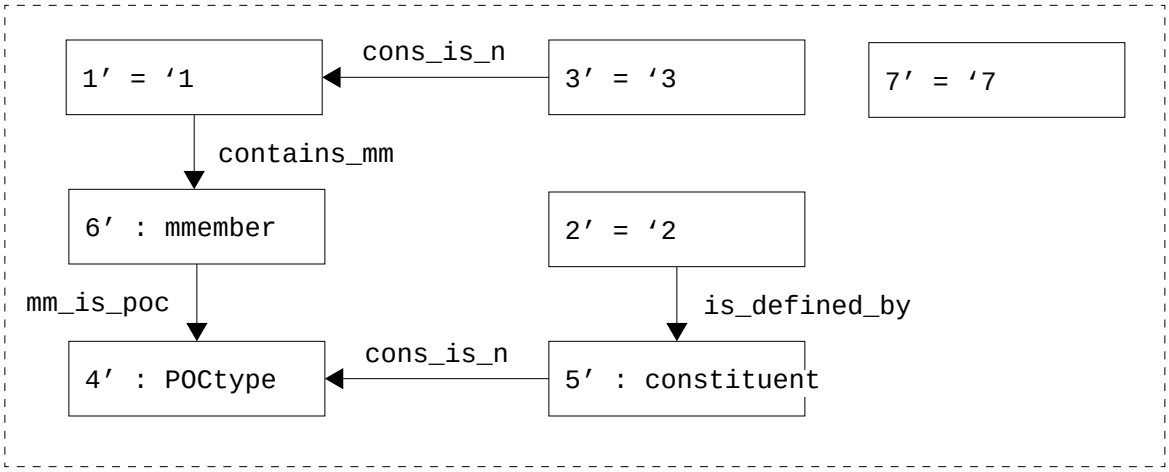



```
   folding { ‘2, ‘6 };
   condition POCtype in Sv.Elem_Type;
             ‘1.Singleton implies (card ( ‘1.-cont-> ) = 0);
             not (Text ( VarName ) in ‘6.=recCons=>.Term);
   transfer 5’.Term := Text ( VarName );
            5’.Declaration := Concat ( Text ( VarName )
                              && " in ", ‘3.Term );
   return POC := 4’;
          C := 5’;
end;
(* Add_to_Set *)
```

```
production Add_to_Mvalue
   ( Mv : MVALUE ; s : SQB ; POCtype : type in PART_OF_COMPLEX ;
     VarName : string ; out c : PART_OF_COMPLEX ; out C : CONSTITUENT )
=
```

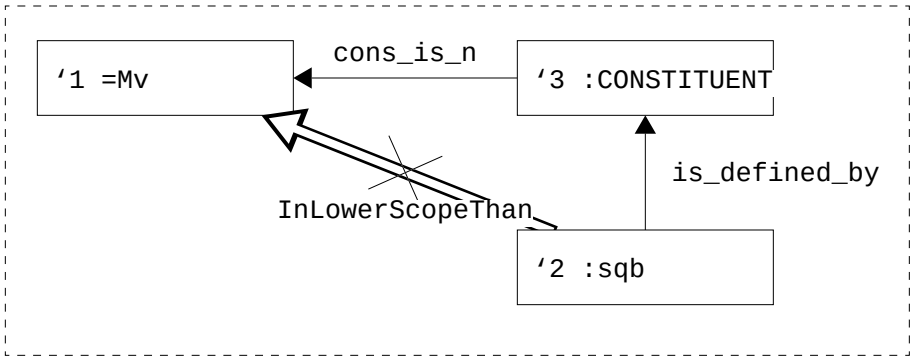



```
   ::=
```

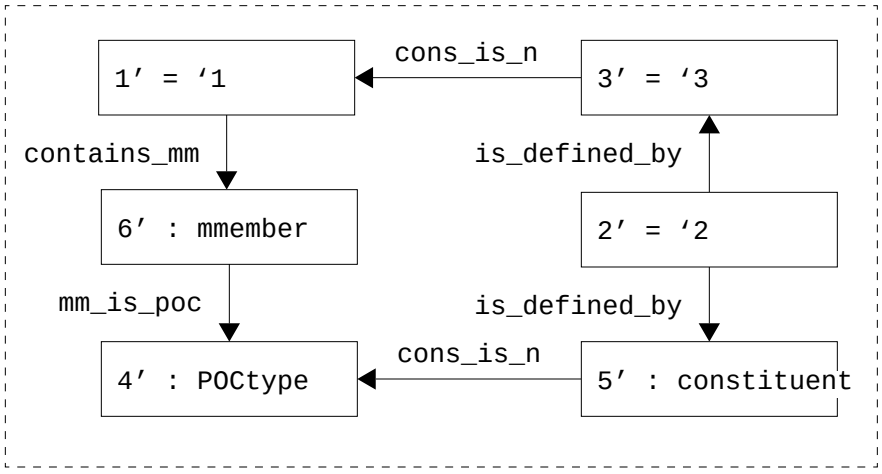



```
   folding { ‘2, ‘7 };
   condition POCtype in ‘1.Elem_Type;
             not (Text ( VarName ) in ‘7.=recCons=>.Term);
   transfer 5’.Term := Text ( VarName );
   5’.Declaration :=
      Concat
      ( Text ( VarName & " in " ), [ ‘3.type = sqb_cons ::
                                       (‘3 : SQB_CONS).SFW_Term
                                   | ‘3.Term ]                  );
   return c := 4’;
          C := 5’;
end;
(* Add_to_Mvalue *)
```

```
production Add_Indexed_to_Mvalue
 ( Mv : MVALUE ; POCtype : type in PART_OF_COMPLEX ; Ind : integer ;
   out P : PART_OF_COMPLEX ; out C : CONSTITUENT )                    =
```

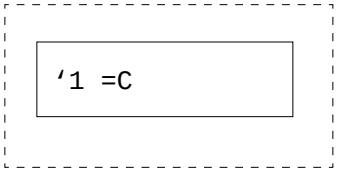



```
   ::=
```

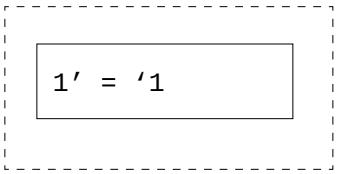



```
   condition POCtype in ‘1.Elem_Type;
   transfer 6’.Index := Ind;
            5’.Term := ‘3.Term && "[" && string ( Ind ) && "]";
   return P := 4’;
          C := 5’;
end;
(* Add_Indexed to Mvalue *)
```

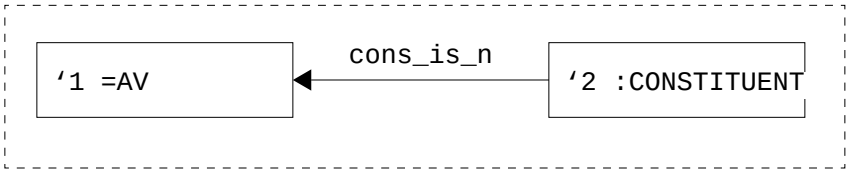
```
production Select ( C : CONSTITUENT ) =
```



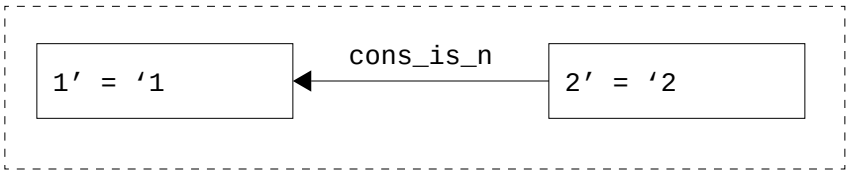

```
   ::=
```




```
condition card ( (‘1.<-is_defined_by- : SQB [1:1]).-contains_mm-> ) = 0;
 transfer 1’.Output := true;
end;
(* Select *)

production Assign_Value ( AV : ATOMIC_VALUE ; Val : string ) =
```

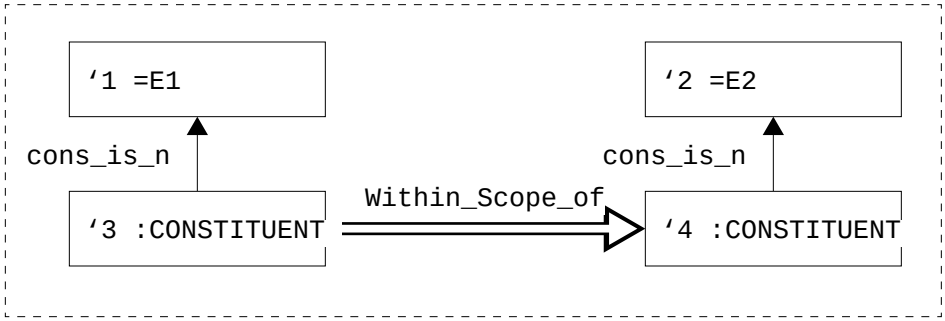



```
   ::=
```

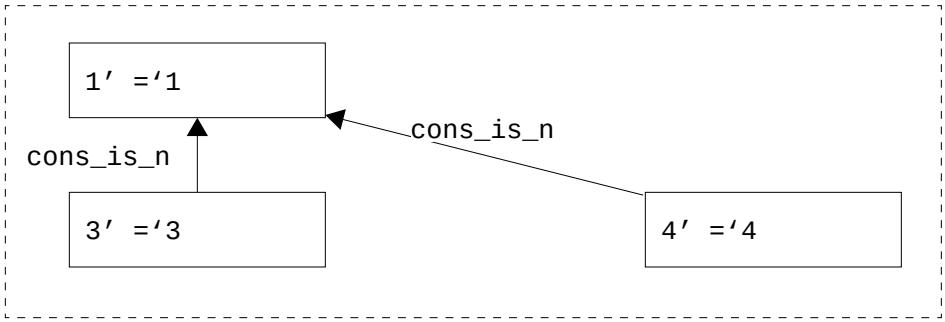



```
   transfer
     2’.Formula := conand ( ‘2.Formula,
                            Concat ( ‘2.Term && " = ", Text ( Val ) ) );
     1’.Value := Val;
end;
(* Assign_Value *)

path Within_Scope_of : CONSTITUENT -> CONSTITUENT =
   <-is_defined_by- & (<-cons_is_n- & <-is_defined_by-) *
   & -is_defined_by->
end;
(* Within_Scope_of *)
```

```
production Merge_Entities ( E1, E2 : ENTITY ) =
```

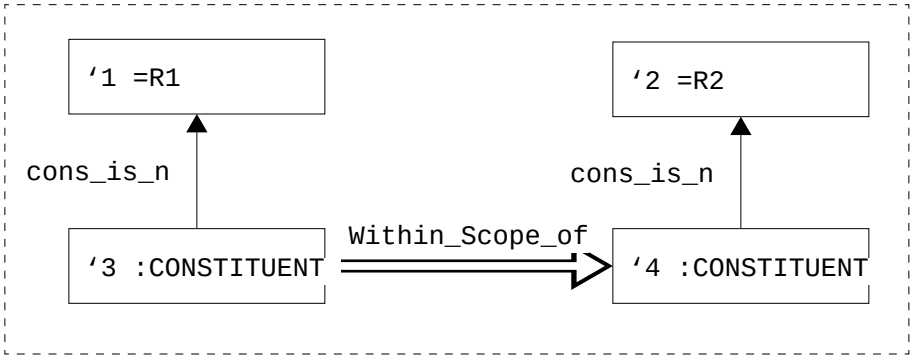



```
   ::=
```

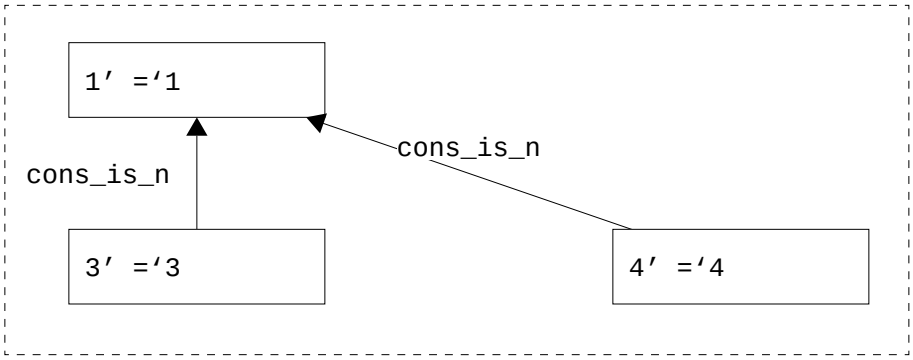



```
   condition E1.type = E2.type;
   embedding redirect <-role2e- from ‘2 to 1’;
             redirect <-mm_is_poc- from ‘2 to 1’;
             redirect <-cons_is_n- from ‘2 to 1’;
             redirect <-attribute2er- from ‘2 to 1’;
             redirect <-component2e- from ‘2 to 1’;
             redirect <-cons_is_n- from ‘2 to 1’;
   transfer
      3’.Formula := conand ( ‘3.Formula,
                             Concat ( ‘3.Term && " = ", ‘4.Term ) );
end;
(* Merge_Entities *)
```

```
production Merge_Relationships ( R1, R2 : RELSHIP ) =
```

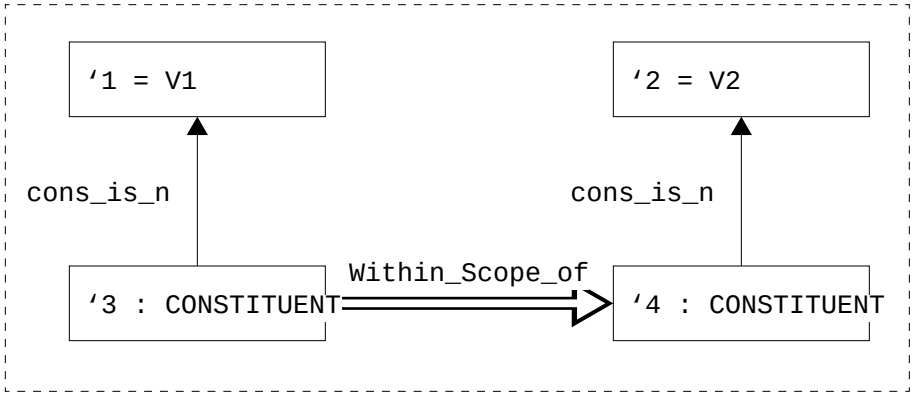



```
   ::=
```

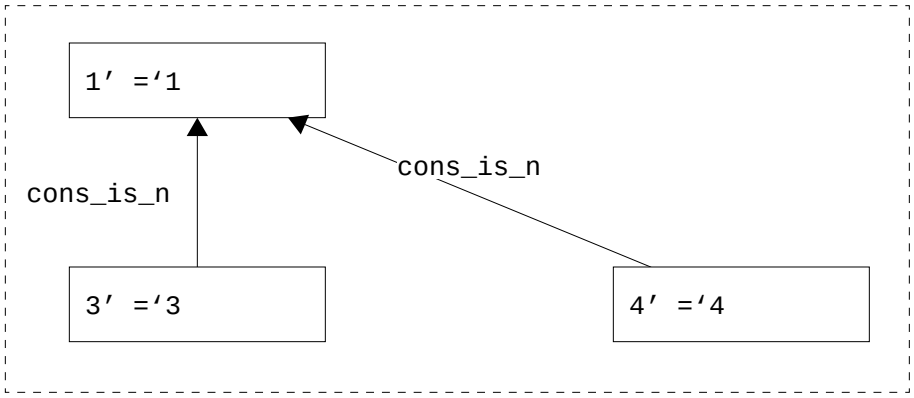



```
   condition ‘2.type = ‘1.type;
   embedding redirect <-role2r- from ‘2 to 1’;
             redirect <-attribute2er- from ‘2 to 1’;
             redirect <-cons_is_n- from ‘2 to 1’;
   transfer
      3’.Formula := conand ( ‘3.Formula,
                             Concat ( ‘3.Term && " = ", ‘4.Term ) );
end;
(* Merge_Relationships *)
```

```
production Merge_Atomic_Values ( V1, V2 : ATOMIC_VALUE ) =
```

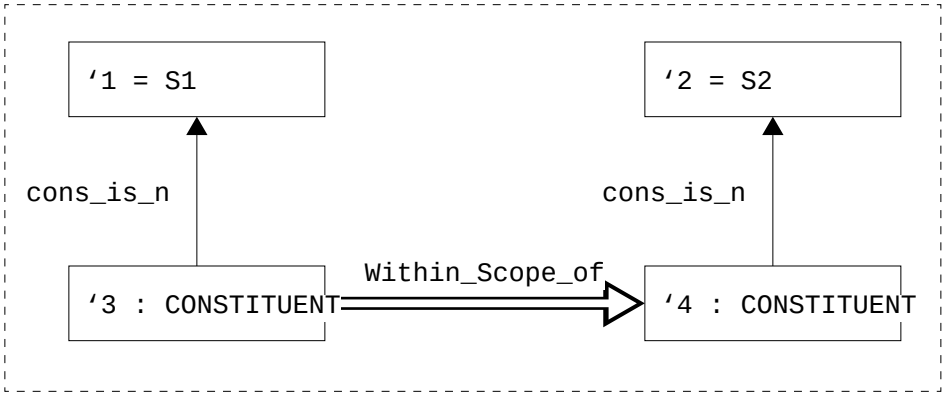



```
::=
```

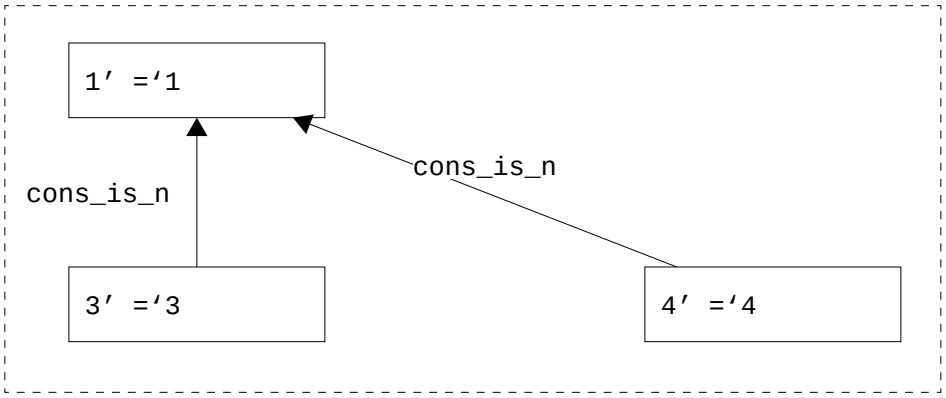



```
   condition ‘2.type = ‘1.type;
   embedding redirect <-cont- from ‘2 to 1’;
             redirect <-mm_is_poc- from ‘2 to 1’;
             redirect <-attribute2v- from ‘2 to 1’;
             redirect <-cons_is_n- from ‘2 to 1’;
   transfer
      3’.Formula := conand ( ‘3.Formula,
                             Concat ( ‘4.Term && " = ", ‘3.Term ) );
end;
(* Merge_Atomic_Values *)
```

```
production Merge_Set_Values ( S1, S2 : SET_VALUE ) =
```

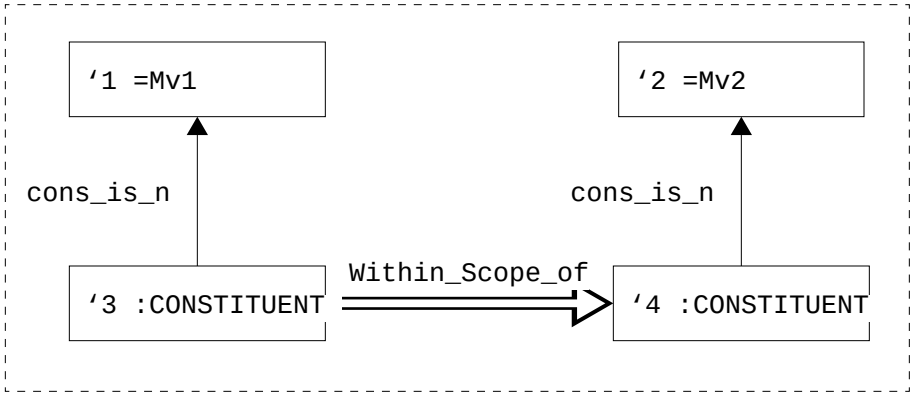



```
::=
```

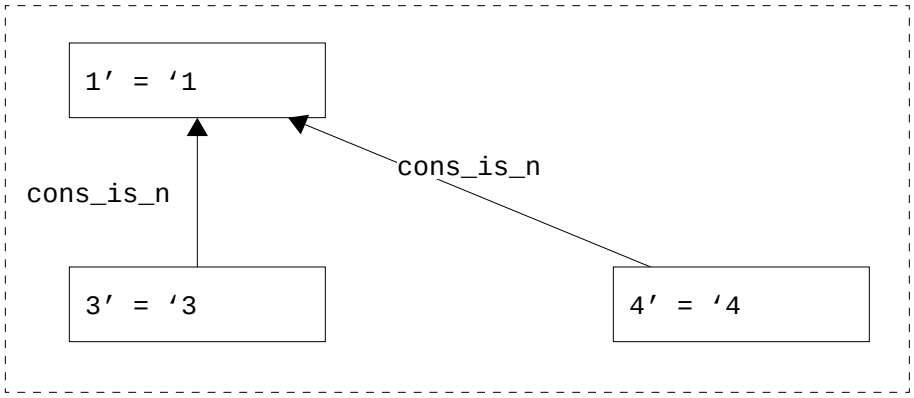



```
   condition ‘2.type = ‘1.type;
   embedding redirect -cont-> from ‘2 to 1’;
             redirect <-attribute2v- from ‘2 to 1’;
             redirect <-component2c- from ‘2 to 1’;
             redirect <-cons_is_n- from ‘2 to 1’;
   transfer
      3’.Formula := conand ( ‘3.Formula,
                             Concat ( ‘3.Term && " = ", ‘4.Term ) );
end;
(* Merge_Set_Values *)
```

```
   production Merge_MValues ( Mv1, Mv2 : MVALUE ) =
```




```
      ::=
```




```
      condition ‘2.type = ‘1.type;
                not (‘1.type = sqb);
      embedding redirect -contains_mm-> from ‘2 to 1’;
                redirect <-attribute2v- from ‘2 to 1’;
                redirect <-component2c- from ‘2 to 1’;
                redirect <-cons_is_n- from ‘2 to 1’;
      transfer
         3’.Formula := conand ( ‘3.Formula,
                                Concat ( ‘3.Term && " = ", ‘4.Term ) );
   end;
   (* Merge_MValues *)

end;
(* Productions *)

section VariableGraphScheme

   section NodeClasses

      node class ENTRY_STATION is a ENTITY end;

      node class ATM is a ENTRY_STATION end;

      node class CASHIER_STATION is a ENTRY_STATION end;

      node class CONSORTIUM is a ENTITY end;

      node class BANK is a ENTITY end;
```

```
   node class ACCOUNT is a ENTITY end;

   node class GENERIC_TRANSACTION is a ENTITY end;

   node class CASHIER_TRANSACTION is a GENERIC_TRANSACTION end;

   node class REMOTE_TRANSACTION is a GENERIC_TRANSACTION end;

   node class CASHIER is a ENTITY end;

   node class CASH_CARD is a ENTITY end;

   node class CUSTOMER is a ENTITY end;
end;

section NodeTypes

   node type entry_station : ENTRY_STATION end;

   node type atm : ATM end;

   node type cashier_station : CASHIER_STATION end;

   node type set_of_cashier_station : SET_VALUE
      redef derived
         Elem_Type = CASHIER_STATION;
   end;

   node type consortium : CONSORTIUM end;

   node type bank : BANK end;

   node type list_of_bank : LIST_VALUE
      redef derived
         Elem_Type = BANK;
   end;

   node type account : ACCOUNT end;

   node type account_s : SET_VALUE
      redef intrinsic
         Singleton := true;
      redef derived
         Elem_Type = ACCOUNT;
   end;

   node type list_of_account : LIST_VALUE
      redef derived
         Elem_Type = ACCOUNT;
   end;

   node type generic_transaction : GENERIC_TRANSACTION end;

   node type cashier_transaction : CASHIER_TRANSACTION end;

   node type remote_transaction : REMOTE_TRANSACTION end;
```

```
node type cashier : CASHIER end;

node type list_of_cashier : LIST_VALUE
   redef derived
      Elem_Type = CASHIER;
end;

node type cash_card : CASH_CARD end;

node type list_of_cash_card : LIST_VALUE
   redef derived
      Elem_Type = CASH_CARD;
end;

node type customer : CUSTOMER end;

node type entered_on : RELSHIP end;

node type eo_es : ROLE
   redef meta
      rel := entered_on ;
      ent := ENTRY_STATION ;
end;

node type eo_t : ROLE
   redef meta
      rel := entered_on ;
      ent := GENERIC_TRANSACTION ;
end;

node type owned_by : RELSHIP end;

node type teller : ROLE
   redef meta
      rel := owned_by ;
      ent := ATM ;
end;

node type owner : ROLE
   redef meta
      rel := owned_by ;
      ent := CONSORTIUM ;
end;

node type entered_by : RELSHIP end;

node type eb_ct : ROLE
   redef meta
      rel := entered_by ;
      ent := CASHIER_TRANSACTION ;
end;

node type eb_c : ROLE
   redef meta
      rel := entered_by ;
      ent := CASHIER ;
end;

node type started_by : RELSHIP end;
```

```
node type sb_rt : ROLE
   redef meta
      rel := started_by ;
      ent := REMOTE_TRANSACTION ;
end;

node type sb_cc : ROLE
   redef meta
      rel := started_by ;
      ent := CASH_CARD ;
end;

node type concerns : RELSHIP end;

node type c_t : ROLE
   redef meta
      rel := concerns ;
      ent := GENERIC_TRANSACTION ;
end;

node type c_a : ROLE
   redef meta
      rel := concerns ;
      ent := ACCOUNT ;
end;

node type accesses : RELSHIP end;

node type a_a : ROLE
   redef meta
      rel := accesses ;
      ent := ACCOUNT ;
end;

node type a_cc : ROLE
   redef meta
      rel := accesses ;
      ent := CASH_CARD ;
end;

node type has : RELSHIP end;

node type ha_cc : ROLE
   redef meta
      rel := has ;
      ent := CASH_CARD ;
end;

node type ha_c : ROLE
   redef meta
      rel := has ;
      ent := CUSTOMER ;
end;

node type holds : RELSHIP end;
```

```
node type ho_c : ROLE
   redef meta
      rel := holds ;
      ent := CUSTOMER ;
end;

node type ho_a : ROLE
   redef meta
      rel := holds ;
      ent := ACCOUNT ;
end;

node type money : ATOMIC_VALUE end;

node type time : ATOMIC_VALUE end;

node type address : ATOMIC_VALUE end;

node type list_of_address : LIST_VALUE
   redef derived
      Elem_Type = address;
end;

node type _string : ATOMIC_VALUE end;

node type int : ATOMIC_VALUE end;

node type bool : ATOMIC_VALUE end;

node type entry_station_location : ATTRIBUTE
   redef meta
      entrel := ENTRY_STATION ;
      val := address ;
end;

node type generic_transaction_entry_time : ATTRIBUTE
   redef meta
      entrel := GENERIC_TRANSACTION ;
      val := time ;
end;

node type generic_transaction_amount : ATTRIBUTE
   redef meta
      entrel := GENERIC_TRANSACTION ;
      val := money ;
end;

node type atm_cash_on_hand : ATTRIBUTE
   redef meta
      entrel := ATM ;
      val := money ;
end;

node type atm_dispensed : ATTRIBUTE
   redef meta
      entrel := ATM ;
      val := money ;
end;
```

```
node type cashier_name : ATTRIBUTE
   redef meta
      entrel := CASHIER ;
      val := _string ;
end;

node type bank_name : ATTRIBUTE
   redef meta
      entrel := BANK ;
      val := _string ;
end;

node type bank_location : ATTRIBUTE
   redef meta
      entrel := BANK ;
      val := address ;
end;

node type consortium_name : ATTRIBUTE
   redef meta
      entrel := CONSORTIUM ;
      val := _string ;
end;

node type customer_name : ATTRIBUTE
   redef meta
      entrel := CUSTOMER ;
      val := _string ;
end;

node type customer_residence : ATTRIBUTE
   redef meta
      entrel := CUSTOMER ;
      val := list_of_address ;
end;

node type cash_card_password : ATTRIBUTE
   redef meta
      entrel := CASH_CARD ;
      val := _string ;
end;

node type cash_card_serial_number : ATTRIBUTE
   redef meta
      entrel := CASH_CARD ;
      val := int ;
end;

node type cash_card_limit : ATTRIBUTE
   redef meta
      entrel := CASH_CARD ;
      val := money ;
end;

node type account_blocked : ATTRIBUTE
   redef meta
      entrel := ACCOUNT ;
      val := bool ;
end;
```

```
      node type account_balance : ATTRIBUTE
         redef meta
            entrel := ACCOUNT ;
            val := money ;
      end;

      node type consortium_consists_of : COMPONENT
         redef meta
            cent := CONSORTIUM ;
            comp := list_of_bank ;
      end;

      node type bank_owns : COMPONENT
         redef meta
            cent := BANK ;
            comp := set_of_cashier_station ;
      end;

      node type bank_proper_acct : COMPONENT
         redef meta
            cent := BANK ;
            comp := account_s ;
      end;

      node type bank_manages : COMPONENT
         redef meta
            cent := BANK ;
            comp := list_of_account ;
      end;

      node type bank_employs : COMPONENT
         redef meta
            cent := BANK ;
            comp := list_of_cashier ;
      end;

      node type bank_issues : COMPONENT
         redef meta
            cent := BANK ;
            comp := list_of_cash_card ;
      end;
   end;
end;
```

```
section Transactions

  transaction GOQL1 =
    use s : SQB;
        av1, av2 : VALUE;
        r, e1, e2 : ENT_REL;
        c1, c2, c3, c4, c5 : CONSTITUENT
    do
        Add_first_SQB ( out s )
      & Add_ER ( s, "h", has, out r, out c1 )
      &
      Add_Role ( (r : RELSHIP), cash_card, ha_cc, out e1, out c2 )
      & Add_Role ( (r : RELSHIP), customer, ha_c, out e2, out c3 )
      & Add_Attribute
        ( (e1 : ENTITY), cash_card_password, _string, out av1,
          out c4 )
      & Add_Attribute
        ( (e2 : ENTITY), customer_name, _string, out av2, out c5 )
      & Select ( c4 )
      & Select ( c5 )
    end
  end;
  (* GOQL1 *)

  transaction GOQL2 =
    use s : SQB;
        cv1, cv2 : COMPLEX_VALUE;
        e1, e2 : ENT_REL;
        e3, e4 : PART_OF_COMPLEX;
        c1, c2, c3, c4, c5, c6, c7, c8 : CONSTITUENT;
        av1, av2 : VALUE
    do
        Add_first_SQB ( out s )
      & Add_ER ( s, "ba1", bank, out e1, out c1 )
      & Add_ER ( s, "ba2", bank, out e2, out c2 )
      & Add_Component
        ( (e1 : ENTITY), bank_employs, list_of_cashier, out cv1,
          out c3 )
      & Add_Component
        ( (e2 : ENTITY), bank_employs, list_of_cashier, out cv2,
          out c4 )
      & Add_to_Mvalue
        ( (cv1 : MVALUE), s, cashier, "ca1", out e3, out c5 )
      & Add_to_Mvalue
        ( (cv2 : MVALUE), s, cashier, "ca2", out e4, out c8 )
      & Merge_Entities ( (e4 : ENTITY), (e3 : ENTITY) )
      & Add_Attribute ( e1, bank_name, _string, out av1, out c6 )
      & Add_Attribute ( e2, bank_name, _string, out av2, out c7 )
      & Select ( c6 )
      & Select ( c7 )
    end
  end;
  (* GOQL2 *)
```

```
transaction GOQL3 =
  use s : SQB;
      r1, r2, r3 : ENT_REL;
      e1, e2, e3, e4, e5, e6 : ENTITY;
      c1, c2, c3, c4, c5, c6, c7, c8, c9, c10 : CONSTITUENT;
      av1 : VALUE
  do
      Add_first_SQB ( out s )
    & Add_ER ( s, "acc", accesses, out r1, out c1 )
    & Add_ER ( s, "has", has, out r2, out c2 )
    & Add_ER ( s, "hol", holds, out r3, out c3 )
    & Add_Role ( (r1 : RELSHIP), account, a_a, out e1, out c4 )
    &
    Add_Role ( (r1 : RELSHIP), cash_card, a_cc, out e2, out c5 )
    &
    Add_Role ( (r2 : RELSHIP), customer, ha_c, out e3, out c6 )
    & Add_Attribute
      ( e3, customer_name, _string, out av1, out c10 )
    & Add_Role
      ( (r2 : RELSHIP), cash_card, ha_cc, out e4, out c7 )
    &
    Add_Role ( (r3 : RELSHIP), customer, ho_c, out e5, out c8 )
    & Add_Role ( (r3 : RELSHIP), account, ho_a, out e6, out c9 )
    & Merge_Entities ( e1, e6 )
    & Merge_Entities ( e2, e4 )
    & Merge_Entities ( e3, e5 )
    & Select ( c10 )
  end
end;
(* GOQL3 *)

transaction GOQL4 =
  use s : SQB;
      e1 : ENT_REL;
      e2 : PART_OF_COMPLEX;
      c1, c2, c3, c4, c5 : CONSTITUENT;
      av1, av2 : VALUE;
      cv1 : COMPLEX_VALUE
  do
      Add_first_SQB ( out s )
    & Add_ER ( s, "co", consortium, out e1, out c1 )
    & Add_Component
      ( (e1 : ENTITY), consortium_consists_of, list_of_bank,
        out cv1, out c2 )
    & Add_Indexed_to_Mvalue
      ( (cv1 : MVALUE), bank, 2, out e2, out c3 )
    & Add_Attribute
      ( e1, consortium_name, _string, out av1, out c4 )
    & Assign_Value ( (av1 : ATOMIC_VALUE), "General Banking" )
    & Add_Attribute
      ( (e2 : ENTITY), bank_name, _string, out av2, out c5 )
    & Select ( c5 )
  end
end;
(* GOQL4 *)
```

```
transaction GOQL5 =
  use s, s1, s2 : SQB;
      sc1, sc2 : SQB_CONS;
      av1, av2, av3 : VALUE;
      av4 : PART_OF_COMPLEX;
      e1, e2 : ENT_REL;
      c1, c2, c3, c4, c5, c6 : CONSTITUENT
  do
      Add_first_SQB ( out s )
    & Add_SQB ( s, out s1, out sc1 )
    & Add_SQB ( s, out s2, out sc2 )
    & Add_ER ( s1, "b1", bank, out e1, out c1 )
    & Add_ER ( s2, "b2", bank, out e2, out c2 )
    & Add_Attribute
      ( (e1 : ENTITY), bank_name, _string, out av1, out c3 )
    & Add_Attribute
      ( (e1 : ENTITY), bank_location, address, out av3, out c5 )
    & Add_Attribute
      ( (e2 : ENTITY), bank_location, address, out av2, out c4 )
    & Select ( c4 )
    & Add_to_Mvalue ( s2, s, address, "a", out av4, out c6 )
    & Merge_Atomic_Values
      ( (av3 : ATOMIC_VALUE), (av4 : ATOMIC_VALUE) )
    & Select ( c3 )
    & Select ( c6 )
    & Select ( sc1 )
  end
end;
(* GOQL5 *)

transaction GOQL6 =
  use s, s1, s2 : SQB;
      e1 : ENT_REL;
      cv1, cv2 : COMPLEX_VALUE;
      e2, e3 : PART_OF_COMPLEX;
      c1, c2, c3, c4, c5 : CONSTITUENT;
      sc1, sc2 : SQB_CONS
  do
      Add_first_SQB ( out s )
    & Add_SQB ( s, out s1, out sc1 )
    & Add_SQB ( s1, out s2, out sc2 )
    & Add_ER ( s, "co", consortium, out e1, out c1 )
    & Add_Component
      ( (e1 : ENTITY), consortium_consists_of, list_of_bank,
        out cv1, out c2 )
    & Add_to_Mvalue
      ( (cv1 : MVALUE), s1, bank, "b", out e2, out c3 )
    &
    Add_Component
    ( (e2 : ENTITY), bank_owns, set_of_cashier_station, out cv2,
      out c4 )
    & Add_to_Set
```

```
      ( (cv2 : SET_VALUE), s2, cashier_station, "cs", out e3,
        out c5 )
    & Select ( sc1 )
    & Select ( sc2 )
    & Select ( c5 )
    & Select ( c3 )
    & Select ( c1 )
  end
end;
(* GOQL6 *)

transaction GOQL7 =
  use s, s1, s2, s3 : SQB;
      sc1, sc2, sc3 : SQB_CONS
  do
      Add_first_SQB ( out s )
    & Add_SQB ( s, out s1, out sc1 )
    & Add_SQB ( s, out s2, out sc2 )
    & Add_SQB ( s1, out s3, out sc3 )
  end
end;
(* GOQL7 *)

transaction GOQL8 =
  use s, s1 : SQB;
      sc1 : SQB_CONS;
      r1, r2 : ENT_REL;
      c1, c2 : CONSTITUENT
  do
      Add_first_SQB ( out s )
    & Add_SQB ( s, out s1, out sc1 )
    & Add_ER ( s1, "r1", R, out r1, out c1 )
    & Add_ER ( s, "r2", R, out r2, out c2 )
    & Merge_Relationships ( (r1 : RELSHIP), (r2 : RELSHIP) )
   end
end;
(* GOQL8 *)

transaction GOQL9 =
  use s : SQB;
      e1 : ENT_REL;
      sv : COMPLEX_VALUE;
      e2 : PART_OF_COMPLEX;
      av1, av2 : VALUE;
      c1, c2, c3, c4, c5 : CONSTITUENT
  do
      Add_first_SQB ( out s )
    & Add_ER ( s, "e1", bank, out e1, out c1 )
    &
    Add_Component
    ( (e1 : ENTITY), bank_owns, set_of_cashier_station, out sv,
      out c2 )
    &
```

```
    Add_to_Set
    ( (sv : SET_VALUE), s, cashier_station, "cs", out e2, out c3
    )
    & Add_Attribute ( e1, bank_name, _string, out av1, out c4 )
    & Add_Attribute
      ( (e2 : ENTITY), entry_station_location, address, out av2,
        out c5 )
  end
end;
(* GOQL9 *)

transaction GOQL10 =
  use s, s1 : SQB;
      sc : SQB_CONS;
      e1 : ENT_REL;
      sv : COMPLEX_VALUE;
      e2 : PART_OF_COMPLEX;
      av1, av2 : VALUE;
      c1, c2, c3, c4, c5 : CONSTITUENT
  do
      Add_first_SQB ( out s )
    & Add_SQB ( s, out s1, out sc )
    & Add_ER ( s, "e1", bank, out e1, out c1 )
    &
    Add_Component
    ( (e1 : ENTITY), bank_owns, set_of_cashier_station, out sv,
      out c2 )
    & Add_to_Set
      ( (sv : SET_VALUE), s1, cashier_station, "cs", out e2,
        out c3 )
    & Add_Attribute ( e1, bank_name, _string, out av1, out c4 )
    & Add_Attribute
      ( (e2 : ENTITY), entry_station_location, address, out av2,
        out c5 )
  end
end;
(* GOQL10 *)

transaction GOQL11 =
  use s : SQB;
      e : ENT_REL;
      av1, av2, av3 : VALUE;
      c1, c2, c3, c4 : CONSTITUENT
  do
      Add_first_SQB ( out s )
    & Add_ER ( s, "e", cash_card, out e, out c1 )
    & Add_Attribute
      ( e, cash_card_serial_number, int, out av1, out c2 )
    &
    Add_Attribute ( e, cash_card_limit, money, out av2, out c3 )
    & Add_Attribute
      ( e, cash_card_password, _string, out av3, out c4 )
    & Assign_Value ( (av3 : ATOMIC_VALUE), "password" )
  end
end;
(* GOQL11 *)
```

```
transaction GOQL12 =
  use s : SQB;
      e1 : ENT_REL;
      e2 : PART_OF_COMPLEX;
      av : VALUE;
      cv : COMPLEX_VALUE;
      c1, c2, c3, c4 : CONSTITUENT
  do
      Add_first_SQB ( out s )
    & Add_ER ( s, "c", consortium, out e1, out c1 )
    & Add_Attribute
      ( e1, consortium_name, _string, out av, out c2 )
    & Assign_Value ( (av : ATOMIC_VALUE), "Banks United" )
    & Add_Component
      ( (e1 : ENTITY), consortium_consists_of, list_of_bank,
        out cv, out c3 )
    & Add_to_Mvalue
      ( (cv : MVALUE), s, bank, "b", out e2, out c4 )
  end
end;
(* GOQL12 *)

transaction GOQL13 =
  use s : SQB;
      cv1, cv2, av3, av4 : VALUE;
      e1, e2 : ENT_REL;
      av1, av2 : PART_OF_COMPLEX;
      c1, c2, c3, c4, c5, c6, c7, c8 : CONSTITUENT
  do
      Add_first_SQB ( out s )
    & Add_ER ( s, "c1", customer, out e1, out c1 )
    & Add_ER ( s, "c2", customer, out e2, out c2 )
    & Add_Attribute
      ( (e1 : ENTITY), customer_residence, list_of_address,
        out cv1, out c3 )
    & Add_Attribute
      ( (e2 : ENTITY), customer_residence, list_of_address,
        out cv2, out c4 )
    & Add_to_Mvalue
      ( (cv1 : MVALUE), s, address, "a1", out av1, out c5 )
    & Add_to_Mvalue
      ( (cv2 : MVALUE), s, address, "a2", out av2, out c6 )
    & Merge_Atomic_Values
      ( (av1 : ATOMIC_VALUE), (av2 : ATOMIC_VALUE) )
    & Add_Attribute
      ( e1, customer_name, _string, out av3, out c7 )
    & Add_Attribute
      ( e2, customer_name, _string, out av4, out c8 )
    & Assign_Value ( (av4 : ATOMIC_VALUE), "John" )
    & Select ( c7 )
  end
end;
(* GOQL13 *)
```

```
    transaction GOQL14 =
      use s1, s2 : SQB;
          sc1 : SQB_CONS;
          r1, e3 : ENT_REL;
          e1, e2 : ENTITY;
          c1, c2, c3, c4, c5, c6, c7 : CONSTITUENT;
          av1, av2, av3 : VALUE
      do
          Add_first_SQB ( out s1 )
        & Add_SQB ( s1, out s2, out sc1 )
        & Add_ER ( s1, "ho", holds, out r1, out c1 )
        & Add_Role ( (r1 : RELSHIP), account, ho_a, out e1, out c2 )
        &
        Add_Role ( (r1 : RELSHIP), customer, ho_c, out e2, out c3 )
        & Add_Attribute
          ( e1, account_balance, money, out av1, out c4 )
        & Add_Attribute
          ( e2, customer_name, _string, out av2, out c5 )
        & Select ( c5 )
        & Add_ER ( s2, "ac", account, out e3, out c6 )
        & Add_Attribute
          ( (e3 : ENTITY), account_balance, money, out av3, out c7 )
        & Select ( c7 )
      end
    end;
    (* GOQL14 *)

    transaction MAIN =
      GOQL1
    end;
    (* MAIN *)

  end;
  (* Transactions *)

end.
```

Samenvatting

Graaf Herschrijfsystemen en Visuele Database Talen

De gebruiksvriendelijkheid van een informatie- ofte database-systeem kan in belangrijke mate verbeterd worden door het visualiseren van diverse aspecten van de bijhorende gebruikersinterface. Eén aspect van een dergelijke interface dat zich hiertoe uitstekend leent, is de taal om de database te ondervragen (de zogenaamde *querytaal*) of te manipuleren. Nieuwe inzichten in mens-machine interactie, evenals recente hardware-evoluties, hebben dan ook geleid tot de ontwikkeling van een brede waaier van talen en hulpmiddelen voor visuele interactie met informatiesystemen.

Helaas zijn slechts weinige van de in de literatuur besproken formalismen ook formeel onderbouwd. Bij talen die wèl formeel gedefinieerd zijn, wordt vaak gebruik gemaakt van uitgebreide stringgrammatica's. In de regels van een dergelijke grammatica worden, naast de gebruikelijke terminale en niet-terminale symbolen, ook speciale operatoren gebruikt om ruimtelijke (of meer-dimensionale) verbanden tussen de symbolen aan te geven. Het merkwaardige hieraan is dat stringgrammatica's in origine ontwikkeld werden om *tekstuele* (en dus één-dimensionale) talen formeel te kunnen definiëren. Eind jaren '60 werden echter *graafgrammatica's* ingevoerd, onder andere om op grafen gebaseerde talen te kunnen formaliseren. Visuele, en vooral dan diagrammatische talen, laten zich immers uitstekend uitdrukken in termen van grafen en graafherschrijving.

Een regel in een graafgrammatica bestaat uit een tweetal grafen. De toepassing van zo'n regel op een graaf komt neer op het vervangen van een isomorf voorkomen van de ene graaf (de zogenaamde *linkerkant*) door een isomorfe kopie van de andere graaf (de zogenaamde *rechterkant*). De grammatica definieert dan de taal van alle grafen die kunnen worden verkregen door toepassing van een willekeurige sequentie regels op een "initiële" graaf. Nauw verwant met graafgrammatica's zijn de zogenaamde *graafherschrijfsystemen*. In tegenstelling tot grammatica's bestaan herschrijfsystemen uit *gestructureerde* verzamelingen regels, en kunnen zodoende gebruikt worden om een graaf te "herschrijven" tot een andere graaf.

In dit proefschrift wordt onderzocht hoe graafherschrijfsystemen kunnen worden gebruikt om zowel syntax als semantiek van visuele talen voor database-systemen formeel te definiëren. We bestuderen daarbij twee mogelijkheden.

In een eerste benadering vertrekken we van een uit de literatuur gekende techniek om uit een grafische representatie voor database-schema's een eenvoudige visuele querytaal af te leiden. Het formuleren van een query in een dergelijke taal bestaat uit het samenstellen van grafische componenten uit een gegeven (visueel gepresenteerd) database-schema tot een *patroon*.

Zo kan bijvoorbeeld het diagram van figuur C.1 op twee manieren bekeken worden. Het kan ener-

Figure C.1: Een diagram met een dubbele betekenis

zijds worden gelezen als een database-schema. In dat geval geeft het aan dat in een database die volgens dit schema is opgebouwd, informatie kan opgeslagen worden over klanten en de bankpas(sen) die ze bezitten, evenals de naam van die klanten en het paswoord van de bankpassen. Anderzijds kan men het diagram ook lezen als een vraag, waarbij de vraagsteller zijn/haar interesse uitdrukt in de naam van in de database aanwezige klanten, evenals het paswoord van bankpassen die ze bezitten. In sectie 4.1 van dit proefschrift wordt de grafische querytaal GOQL/EER ingevoerd, waarmee queries kunnen geformuleerd worden, gebruik makend van de grafische componenten van een Extended Entity Relationship diagram. In sectie 4.2 wordt vervolgens aangetoond hoe zowel de (abstracte) syntax als de semantiek van deze taal formeel gedefinieerd kunnen worden door middel van een graafgrammatica. De abstracte syntax van een GOQL/EER query wordt uitgedrukt door middel van een graaf, terwijl de semantiek wordt gedefinieerd door middel van een vertaling naar de tekstuele querytaal SQL/EER. Deze vertaling is geïntegreerd in de graafgrammatica, en maakt gebruik van knoop-attributen in de graaf.

De graafgrammatica is geschreven in PROGRES (ontwikkeld aan de RWTH Aachen), het op dit ogenblik meest expressieve specificatie formalisme gebaseerd op graafherschrijfregels. De benodigde concepten van deze taal worden herhaald in hoofdstuk 3 van dit proefschrift, en aldaar geïllustreerd door middel van een specificatie voor de taal van EER diagrammen.

Vanuit de observatie dat puur grafische talen zowel het specificeren als het lezen van queries vaak eerder compliceren dan vereenvoudigen, worden in sectie 4.3 de tekstuele taal SQL/EER en de visuele taal GOQL/EER samengevoegd tot een *hybride* taal HQL/EER. In deze taal kunnen queries geformuleerd worden door middel van een willekeurige combinatie van grafische en tekstuele elementen.

In een tweede benadering richten we onze aandacht op database-manipulatietalen. Zoals eerder vermeld, kunnen EER diagrammen geformaliseerd worden door middel van grafen. Wanneer we daarnaast ook database-*instances* bekijken als grafen, blijkt het een zeer natuurlijke benadering om graafherschrijving te gebruiken als database-manipulatie paradigma.

Bij het toepassen van een graafherschrijfregel op een graaf die een database-instance voorstelt, moet echter wel gebruik gemaakt worden van een aangepaste semantiek. Volgens de eerder beschreven semantiek wordt een herschrijfregel immers toegepast op *één* voorkomen van zijn linkerkant. Een database-manipulatie bestaat echter in het algemeen uit een query, tezamen met de beschrijving van een manipulatie die moet uitgevoerd worden op het resultaat van deze query. Uitgedrukt in termen van graafherschrijving betekent dit dat we een herschrijfregel “uitputtend” moeten toepassen op *alle* voorkomens van zijn linkerkant in de graaf.

Bij wijze van illustratie toont figuur C.2 een graafherschrijfregel die volgende database-update uitdrukt:

Geef elke bankpas als paswoord de naam van de eigenaar van deze pas.

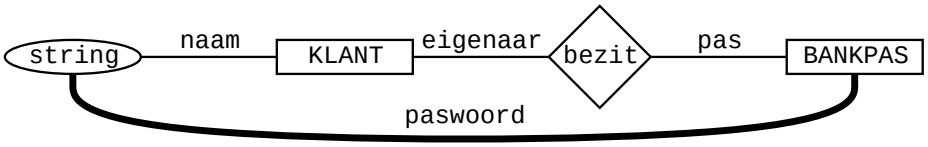


Figure C.2: Een graafherschrijfregel die een database-update uitdrukt

De linkerkant van deze herschrijfregel wordt gevormd door het deel van de figuur in dunne lijntjes, terwijl de gehele figuur de rechterkant vormt.

Door aan dit model voor graafherschrijfregels ook nog een aantal typische programmeerconstructies toe te voegen (zoals sequenties, procedures,…) verkrijgen we de in hoofdstuk 5 van dit proefschrift beschreven Graph-Oriented Object Database language GOOD/ER, gebaseerd op het Entity Relationship model.

In sectie 5.3 bestuderen we tot slot de expressieve kracht van de taal GOOD/ER. Een dergelijke studie komt neer op het exact wiskundig karakteriseren van de verzameling transformaties die door middel van de gegeven taal uitdrukbaar zijn. Dit gebeurt doorgaans door middel van een zogenaamd *volledigheidscriterium*, een conditie waaraan een tweetal database-instances moet voldoen opdat de ene in de ander zou kunnen omgezet worden door middel van een programma in de beschouwde taal. Het formele bewijs van de karakterisatie voor GOOD/ER maakt sterk gebruik van het feit dat de taal geformaliseerd is door middel van grafentheorie en graafherschrijving. Dit levert een bijkomende ondersteuning voor wat met dit proefschrift is aangetoond, met name dat graafherschrijfsystemen een beloftevol middel zijn voor het formeel definiëren van visuele (database-)talen.

Curriculum Vitae

De schrijver van dit proefschrift werd op 6 april 1968 te Wilrijk (thans Antwerpen, België) geboren. In 1986 behaalde hij het Getuigschrift van Hoger Secundair Onderwijs (Wetenschappelijke A) aan het Sint-Jan Berchmanscollege te Antwerpen. In 1988 behaalde hij het Kandidaatsdiploma in de Wetenschappen, Groep Wiskunde aan het Rijksuniversitair Centrum Antwerpen (RUCA), en in 1990 het Licentiaatsdiploma in de Wetenschappen, Groep Wiskunde, richting Informatica aan de Universitaire Instelling Antwerpen (UIA).

In september 1990 begon hij aan de UIA aan zijn promotie-onderzoek, onder begeleiding van prof. dr. J. Paredaens. Dit onderzoek zette hij in 1991 voort als bursaal in dienst van het Instituut voor Wetenschappelijk Onderzoek in Nijverheid en Landbouw (IWONL). Vanaf 15 februari 1992 was hij werkzaam als Assistent-in-opleiding bij de Vakgroep Informatica van de Rijksuniversiteit te Leiden, onder begeleiding van prof. dr. G. Engels. Daar voltooide hij het in dit proefschrift beschreven onderzoek in het kader van de ESPRIT Basic Research Working Group COMPUGRAPH II (Computation by Graph Transformation).

Bibliography

[ACPB95] Marc Andries, Luca Cabibbo, Jan Paredaens, and Jan Van den Bussche. Applying an update method to a set of receivers. In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 208–218. ACM Press, 1995.

[ACS90] Michele Angelaccio, Tiziana Catarci, and Giuseppe Santucci. QBD: A Graphical Query Language with Recursion. *IEEE Transactions on Software Engineering*, 16(10):1150–1163, 1990.

[ADD+91] Annamaria Auddino, Yves Dennebouy, Yann Dupont, Edi Fontana, Stefano Spaccapietra, and Zahir Tari. SUPER: A Comprehensive Approach to Database Visual Interfaces. In *IFIP 2.6 2nd Working Conf. on Visual Database Systems*, September 1991.

[ADS80] G. Ausiello, A. D'Atri, and D. Sacca. Graph-Algorithms for the Synthesis and Manipulation of Database Schemes. In *Proceedings of the 6th International Workshop on Graph-Theoretic Concepts in Computer Science (WG'80)*, volume 100 of *Lecture Notes in Computer Science*, pages 212–, 1980.

[AE94] Marc Andries and Gregor Engels. Syntax and Semantics of Hybrid Database Languages. In Hartmut Ehrig and H.J. Schneider, editors, *Graph Transformations in Computer Science – International Conference and Research Center for Computer Science, Schloss Dagstuhl, January 4-8, 1993*, volume 776 of *Lecture Notes in Computer Science*, pages 19–36, Berlin, 1994. Springer.

[AE97] Marc Andries and Gregor Engels. A Hybrid Query Language for the Extended Entity Relationship Model. *Journal of Visual Languages and Computing*, 8(1), March 1997. Special Issue on Visual Query Systems (to appear).

[AGP+92] Marc Andries, Marc Gemis, Jan Paredaens, Inge Thyssens, and Jan Van den Bussche. Concepts for graph-oriented object manipulation. In A. Pirotte, C. Delobel, and G. Gottlob, editors, *3rd International Conference on Extending Database Technology, Proceedings*, number 580 in Lecture Notes in Computer Science, pages 21–38, Berlin, 1992. Springer.

[AK89] Serge Abiteboul and Paris C. Kanellakis. Object Identity as a Query Language Primitive. In Clifford et al. [CLM89], pages 159–173.

[And94] Marc Andries. An Exhaustive Semantics for Structured Graph Grammar Rules. In *Fifth International Workshop on Graph Grammars and their Application to Computer Science (Williamsburg VA), Book of abstracts*, pages 179–182, November 1994.

[AP91] Marc Andries and Jan Paredaens. Macro's for the GOOD-transformation language. Technical Report 91-20, University of Antwerp (U.I.A.), 1991.

[AP92] Marc Andries and Jan Paredaens. A Language for Generic Graph-Transformations. In Schmidt and Berghammer [SB92], pages 63–74.

[AP96] Marc Andries and Jan Paredaens. On instance-completeness for database query languages involving object creation. To appear in the Journal of Computer and System Siences, 1996.

[Ban78] François Bancilhon. On the completeness of query languages for relational data bases. In *Proceedings of the 7th Symposium on Mathematical Foundations of Computer Science*, volume 64 of *Lecture Notes in Computer Science*, pages 112–123, Berlin, 1978. Springer.

[BCCL91] C. Batini, T. Catarci, M.F. Costabile, and S. Levialdi. Visual Query Systems. Technical Report 04.91, Universita degli Studi di Roma "La Sapienza", Dipartimento di Informatica e Sistemistica, March 1991.

[BH86] D. Bryce and R. Hull. SNAP: A Graphics-Based Schema Manager. In *Proceedings of the International Conference on Data Engineering*, pages 151–164, 1986.

[BJ93] Peter Buneman and Sushil Jajodia, editors. *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, volume 22:2 of *SIGMOD Record*. ACM Press, June 1993.

[BM90] Christian J. Breiteneder and Thomas A. Mück. A Graph Grammar Driven ER CASE Environment. In Spaccapietra [Spa90], pages 375–392.

[CAE+76] D.D. Chamberlain, M.M. Astrahan, K.P. Eswaran, P.P. Griffiths, R.A. Lorie, J.W. Mehl, P. Reisner, and B.W. Wade. SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control. *IBM Journal of Research and Development*, 20(6):560–575, 1976.

[CCM95] Tiziana Catarci, Maria F. Costabile, and Maristella Matera. Visual Metaphors for Interacting with Databases. *SIGCHI bulletin*, 27(2):15–17, April 1995.

[CERE87] Bogdan Czejdo, Ramez Elmasri, Marek Rusinkiewicz, and David W. Embley. Graphical query languages for semantics database models. In *1987 National Computer Conference*, volume 56 of *AFIPS Conference Proceedings*, pages 615–623. AFIPS Press, June 1987.

[CGT90] Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic Programming and Databases*. Surveys in Computer Science. Springer-Verlag, Berlin, 1990.

[CH80] Ashok K. Chandra and David Harel. Computable Queries for Relational Databases. *Journal of Computer and System Sciences*, 21:156–178, 1980.

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

[CLM89] J. Clifford, B. Lindsay, and D. Maier, editors. *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, volume 18:2 of *SIGMOD Record*. ACM Press, 1989.

[CM90] Mariano Consens and Alberto Mendelzon. GraphLog: a visual formalism for real life recursion. In PODS90 [POD90], pages 404–416.

[Cod72] E.F. Codd. Relational Completeness of Data Base Sublanguages. In Randall Rustin, editor, *Data Base Systems*, number 6 in Courant Computer Science Symposium, pages 65–98. Prentice Hall, Englewood Cliffs, 1972.

[Cou91] Bruno Courcelle. Recursive queries and context-free grammars. *Theoretical Computer Science*, 78:217–244, 1991.

[Cru92] Isabel F. Cruz. DOODLE: A Visual Language for Object-Oriented Databases. In Michael Stonebraker, editor, *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, volume 21:2 of *SIGMOD Record*, pages 71–80. ACM Press, 1992.

[CTODL95] Gennaro Costagliola, Genoveffa Tortora, Sergio Orefice, and Andrea De Lucia. Automatic Generation of Visual Programming Environments. *IEEE Computer*, pages 56–66, March 1995.

[CTYY89] Shi-Kuo Chang, Michael J. Tauber, Bing Yu, and Jing-Sheng Yu. A Visual Language Compiler. *IEEE Transactions on Software Engineering*, 15(5):506–525, May 1989.

[EE94] Gregor Engels and Hartmut Ehrig. Towards a Module Concept for Graph Transformation Systems: The Software Engineering Perspective. In G. V. Feruglio, editor, *Proceedings Colloquium on Graph Transformation and its Application in Computer Science*, March 1994. Also as Technical Report 93-34, Leiden University, Dept. of Comp. Science, The Netherlands.

[EF94] Jürgen Ebert and Angelika Fränzke. A Declarative Approach to Graph Based Modeling. In Ernst W. Mayr, Gunther Schmidt, and Gottfried Tinhofer, editors, *Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science*, volume 904 of *Lecture Notes in Computer Science*, pages 38–50, Berlin, 1994. Springer.

[EGH+92] Gregor Engels, Martin Gogolla, Uwe Hohenstein, Klaus Hülsmann, Perdita Löhr-Richter, Gunter Saake, and Hans-Dieter Ehrich. Conceptual modelling of database applications using an extended ER model. *Data & Knowledge Engineering*, 9(2):157–204, December 1992.

[EHH+89] Gregor Engels, Uwe Hohenstein, Klaus Hülsmann, Perdita Löhr-Richter, and Hans-Dieter Ehrich. CADDY: Computer-Aided Design of Non-Standard Databases. In N. Madhavji, H. Weber, and W. Schäfer, editors, *International Conference on System Devlopment Environments & Factories*, London, 1989. Pitman Publ.

[EK76] Hartmut Ehrig and Hans-Jörg Kreowski. Parallelism of Manipulations in Multidimensional Information Structures. In *Proceedings of the 5th Symposium on Mathematical Foundations of Computer Science*, volume 45 of *Lecture Notes in Computer Science*, pages 284–239, Berlin, 1976. Springer.

[EK80] Hartmut Ehrig and Hans-Jörg Kreowski. Applications of Graph Grammar Theory to Consistency, Synchronization, and Scheduling in Data Base Systems. *Information Systems*, 5:225–238, 1980.

[EKR90] Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Graph-Grammars and Their Application to Computer Science, International Workshop*, volume 532 of *Lecture Notes in Computer Science*, Berlin, 1990. Springer.

[EL85] Ramez Elmasri and James A. Larson. A Graphical Query Facility for ER Databases. In Peter P. Chen, editor, *Entity-Relationship Approach: The Use of ER Concept in Knowledge Representation*, pages 236–245. IEEE CS Press/North-Holland, 1985.

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

[Eng90] Gregor Engels. Elementary Actions on an Extended Entity-Relationship Database. In Ehrig et al. [EKR90], pages 344–362.

[GF79] C.C. Gotlieb and A.L. Furtado. Data Schemata Based on Directed Graphs. *Journal of Computer and System Sciences*, 8(1), 1979.

[GG87] Ephraim P. Glinert and Jakob Gonczarowski. A (Formal) Model for (Iconic) Programming Environments. In *INTERACT'87, Proceedings of the 2nd IFIP Conference on Human-Computer Interaction* [IFI87], pages 283–290.

[GG90] *Proceedings of the International Workshops on Graph-Grammars and Their Application to Computer Science*, volume 73,153,291,532 of *Lecture Notes in Computer Science*. Springer, 1978–1990.

[Gli90a] Ephraim P. Glinert, editor. *Visual Programming Environments: Applications and Issues*. IEEE Computer Society Press, Los Alamitos, CA, 1990.

[Gli90b] Ephraim P. Glinert, editor. *Visual Programming Environments: Paradigms and Systems*. IEEE Computer Society Press, Los Alamitos, CA, 1990.

[Gog94] Martin Gogolla. *An extended entity-relationship model : fundamentals and pragmatics*, volume 767 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1994.

[GPT93] Marc Gemis, Jan Paredaens, and Inge Thyssens. A Visual Database Managment Interface Based on GOOD. In R. Cooper, editor, *Interfaces to Database Systems*, Workshops in Computing, pages 155–175. Springer, 1993.

[GPTVdB93] Marc Gemis, Jan Paredaens, Inge Thyssens, and Jan Van den Buscche. GOOD: A Graph-Oriented Object Database System. In Buneman and Jajodia [BJ93], pages 505–510. Video presentation.

[GPVdBVG94] Marc Gyssens, Jan Paredaens, Jan Van den Bussche, and Dirk Van Gucht. A graph-oriented object database model. *IEEE Transactions on Knowledge and Data Engineering*, 6(4):572, August 1994.

[GPVG89] Marc Gyssens, Jan Paredaens, and Dirk Van Gucht. A Uniform Approach toward Handling Atomic and Structured Information in the Nested Relational Database Model. *Journal of the ACM*, 36(2):790–825, October 1989.

[GPVG90a] Marc Gyssens, Jan Paredaens, and Dirk Van Gucht. A Graph-Oriented Object Database Model. In PODS90 [POD90], pages 417–424.

[GPVG90b] Marc Gyssens, Jan Paredaens, and Dirk Van Gucht. A Graph-Oriented Object Model for End-User Interfaces. In H. Garcia-Molina and H.V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, volume 19:2 of *SIGMOD Record*, pages 24–33. ACM Press, 1990.

[GR87] Monika Gerstendörfer and Gabriele Rohr. Which task in which representation on what kind of interface. In *INTERACT'87, Proceedings of the 2nd IFIP Conference on Human-Computer Interaction* [IFI87], pages 513–518.

[GR89] Eric J. Golin and Steven P. Reiss. The Specification of Visual Language Syntax. In *IEEE Proceedings of the Workshop on Visual Languages*, pages 105–110, Los Alamitos, CA, October 1989. IEEE, IEEE Computer Society Press.

[Gra90] Mike Graf. Visual Programming and Visual Languages: Lessons Learned in the Trenches. In Glinert [Gli90b], pages 452–455.

[Haa95] Erik de Haas. Categorial Graphs. Submitted for publication, 1995.

[HE90] Uwe Hohenstein and Gregor Engels. Formal Semantics of an Extended Entity-Relationship based Query Language. In Spaccapietra [Spa90], pages 177–194.

[HE91] Uwe Hohenstein and Gregor Engels. SQL/EER – Syntax and Semantics of an Entity-Relationship-Based Query Language. Technical Report 91-02, Technische Universität Braunschweig, March 1991.

[HE92] Uwe Hohenstein and Gregor Engels. SQL/EER – Syntax and Semantics of an Entity-Relationship-Based Query Language. *Information Systems*, 17(3):209–242, 1992.

[HG88] Uwe Hohenstein and Martin Gogolla. A Calculus for an Extended Entity-Relationship Model Incorporating Arbitrary Data Operations and Aggregate Functions. In C. Batini, editor, *Proceedings of the 7th International Conference on Entity-Relationship Approach*, pages 129–148, 1988.

[HIL94] Eben M. Haber, Yannis E. Ioannidis, and Miron Livny. Foundations of Visual Metaphors for Schema Display. *Journal of Intelligent Information Systems*, 3:1–38, 1994.

[Hof82] Christoph M. Hoffmann. *Group-Theoretic Algorithms and Graph Isomorphism*, volume 136 of *Lecture Notes in Computer Science*. Springer, 1982.

[Hoh93] Uwe Hohenstein. *Formale Semantik eines erweiterten Entity-Relationship Models*. Teubner-Texte zur Informatik. B.G. Teubner Verlagsgesellschaft, 1993.

[Hou92] Teruhisa Houchin. DUO: Graph-Based Database Graphical Query Expression. In Qiming Chen, Yahiko Kambayashi, and Ron Sacks-Davis, editors, *Proceedings of The Second Far-East Workshop on Future Database Systems*, volume 3 of *Advanced Database Research and Development Series*, pages 286–295, Singapore, April 1992. World Scientific.

[IEE88] IEEE. *IEEE Proceedings of the Workshop on Visual Languages*, Los Alamitos, CA, October 1988. IEEE Computer Society Press.

[IFI87] IFIP. *INTERACT'87, Proceedings of the 2nd IFIP Conference on Human-Computer Interaction*, Amsterdam, 1987. Elsevier Science Publishers B.B. (North Holland).

[Kan88] Hannu Kangassalo. Concept D: a graphical language for conceptual modelling and data base use. In *IEEE Proceedings of the Workshop on Visual Languages* [IEE88], pages 2–11.

[KL89] M. Kifer and G. Lausen. F-Logic: a Higher-Order Logic for Reasoning about Objects, Inheritance and Scheme. In Clifford et al. [CLM89], pages 134–146.

[KM89] Michel Kuntz and Rainer Melchert. Pasta-3's Graphical Query Language: Direct Manipulation, Cooperative Queries, Full Expressive Power. In Peter Apers and Gio Wiederhold, editors, *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pages 97–105. Morgan Kaufmann, 1989.

[Knu68] Donald E. Knuth. Semantics of Context-Free Languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[Knu71] Donald E. Knuth. Examples of Formal Semantics. In *Symposium on Semantics of Algorithmic Languages*, volume 188 of *Lecture Notes in Mathematics*, pages 212–235, 1971.

[Kun92] Michel Kuntz. The gist of GIUKU: Graphical interactive intelligent utilities for knowledgeable users of data base systems. *SIGMOD Record*, 21(1), 1992.

[LG94] John P. Lee and Georges G. Grinstein, editors. *Database Issues for Data Visualization, IEEE Visualization '93 Workshop*, volume 871 of *Lecture Notes in Computer Science*, Berlin, October 1994. Springer.

[LP91] Mark Levene and Alexandra Poulovassilis. An object-oriented data model formalised through hypergraphs. *Data & Knowledge Engineering*, 6(3):205–224, 1991.

[Mar89] Leo Mark. A Graphical Query Language for the Binary Relationship Model. *Information Systems*, 14(3):231–246, 1989.

[Mar94] Kim Marriott. Constraint Multiset Grammars. In *Proceedings 1994 IEEE Symposium on Visual Languages*, pages 118–125, Los Alamitos, CA, October 1994. IEEE, IEEE Computer Society Press.

[Miu94] T. Miura. Nesting quantification in a visual data manipulation language. *Data & Knowledge Engineering*, 12(2):167–196, March 1994.

[MP80] Dan McCue and George Poonen. Evaluation of an E-R query langauge (Abstract). In Peter P. Chen, editor, *Entity-Relationship Approach to Systems Analysis and Design*, page 463. North-Holland, 1980.

[MR83] Victor M. Markowitz and Yoav Raz. ERROL: An Entity-Relationship, Role Oriented, Query Language. In Carl G. Davis, Sushil Jajodia, Peter Ann-Beng Ng, and Raymond T. Yeh, editors, *Entity-Relationship Approach to Software Engineering*, pages 329–346. North-Holland, 1983.

[New91] Paulisch F. Newbery. *The Design of an Extendible Graph Editor*, volume 704 of *Lecture Notes in Computer Science*. Springer, 1991. Dissertation, University of Karlsruhe.

[NS90] Manfred Nagl and Andy Schürr. A Specification Environment for Graph Grammars. In Ehrig et al. [EKR90], pages 599–609.

[OPT+92] S. Orefice, G. Polese, M Tucci, G. Gortora, G. Costagliola, and C. K. Chang. A 2d Interactive Parser for Iconic Languages. In *Proceedings of the 1992 IEEE Workshop on Visual Languages*, pages 207–213, Los Alamitos, CA, September 1992. IEEE, IEEE Computer Society Press.

[Pag81] F. G. Pagin. *Formal Specification of Programming Languages: A Panoramic Primer*. Prentice Hall, Englewood Cliffs, New Jersey, 1981.

[Par78] Jan Paredaens. On the expressive power of the relational algebra. *Information Processing Letters*, 7(2):107–111, February 1978.

[POD90] *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*. ACM Press, 1990.

[PPT91] Peter Peelman, Jan Paredaens, and Letizia Tanca. G-Log: A Declarative Graphical Query Language. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Proceedings 2nd International Conference on Deductive and Object-Oriented Databases*, volume 566 of *Lecture Notes in Computer Science*, pages 108–128, Berlin, December 1991. Springer.

[PPT95] Jan Paredaens, Peter Peelman, and Letizia Tanca. G-Log: A Graph-Based Query Language. *IEEE Transactions on Knowledge and Data Engineering*, 7(3):436–453, June 1995.

[PR69] J. Pfaltz and A. Rosenfeld. Web Grammars. In *Proc. International Joint Conference on Artificial Intelligence*, pages 609–619, 1969.

[Pra71] Terrence W. Pratt. Pair Grammars, Graph Languages and String-to-Graph Translation. *Journal of Computer and System Sciences*, 5:560–595, 1971.

[PVdBA+92] Jan Paredaens, Jan Van den Bussche, Marc Andries, Marc Gemis, Marc Gyssens, Inge Thyssens, Dirk Van Gucht, Vijay Sarathy, and Lawrence Saxton. An Overview of GOOD. *SIGMOD Record*, 21(1):49–53, 1992.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, 1991.

[RS95] Jan Rekers and Andy Schürr. A Graph Grammar approach to Graphical Parsing. In *Proc. VL'95 11th Int. IEEE Symp. on Visual Languages*. IEEE Computer Society Press, September 1995.

[SB92] G. Schmidt and R. Berghammer, editors. *Proceedings of the 17th International Workshop on Graph-Theoretic Concepts in Computer Science*, volume 570 of *Lecture Notes in Computer Science*, Berlin, 1992. Springer.

[SBMW93] Gary Sockut, Luanne Burns, Ashok Malhotra, and Kyu-Young Whang. GRAQULA: A graphical query language for entity-relationship or relational databases. *Data & Knowledge Engineering*, 11:171–202, 1993.

[SBOO95] E. Sukan, N.H. Balkir, G. Ozsoyoglu, and Z.M. Ozsoyoglu. VISUAL: A graphical Icon-Based Query Language. Unpublished Manuscript, 1995.

[Sch70] H.-J. Schneider. Chomsky-Systeme für partielle Ordnungen. Technical Report Arbeitsbericht IMMD-3-3, Universität Erlangen, 1970.

[Sch89] Andy Schürr. Introduction to PROGRESS, an Attribute Grammar Based Specification Language. In M. Nagl, editor, *Proceedings of the 15th International Workshop on Graph-Theoretic Concepts in Computer Science*, volume 411 of *Lecture Notes in Computer Science*, pages 151–165, Berlin, 1989. Springer.

[Sch90a] Andy Schürr. PROGRESS: a VHL-Language Based on Graph Grammars. In Ehrig et al. [EKR90], pages 641–659.

[Sch90b] Andy Schürr. PROGRESS-Editor: A text-oriented hybrid editor for PROgrammed Graph REwriting SyStems. In Ehrig et al. [EKR90], page 67.

[Sch91a] Jens Schacht. Visuelle Spezifikation von komplexen Aktionen auf erweiterten Entity-Relationship-Datenbanken. Master's thesis, Technische Universität Braunschweig, Germany, March 1991. In German.

[Sch91b] Andy Schürr. *Operationales Spezifizieren mit programmierten Graphersetzungssystemen*. PhD thesis, RWTH Aachen, 1991. Deutsche Universitäts Verlag, Wiesbaden. In German.

[Shn83] Ben Shneiderman. Direct Manipulation: A Step Beyond Programming Languages. *IEEE Computer*, 16(8):57–69, 1983.

[Spa90] S. Spaccapietra, editor. *Proceedings of the 9th International Conference on Entity-Relationship Approach*. North Holland, 1990.

[VAO93] K. Vadaparty, Y.A. Aslandogan, and G Ozsoyoglu. Towards a Unified Visual Database Access. In Buneman and Jajodia [BJ93], pages 357–366.

[VdB93] Jan Van den Bussche. *Formal aspects of object identity in database manipulation*. Doctoral thesis, University of Antwerp (UIA), 1993.

[VdBP91] Jan Van den Bussche and Jan Paredaens. On the Expressive Power of Structured Values in Pure OODB's. In *Proceedings of the Tenth ACM Symposium on Principles of Database Systems*, pages 291–299. ACM Press, 1991.

[VdBVGAG92] Jan Van den Bussche, Dirk Van Gucht, Marc Andries, and Marc Gyssens. On the Completeness of Object-Creating Query Languages for Nearly-Deterministic Queries. In *Proceedings 33rd Symposium on Foundations of Computer Science*, pages 372–379. IEEE Computer Society Press, October 1992.

[Wit92] Kent Wittenburg. Earley-Style Parsing for Relational Grammars. In *Proc. 1992 IEEE Workshop Visual Languages*, pages 192–199, Los Alamitos, CA, 1992. IEEE CS Press.

[WMS+92] Kyu-Young Whang, Ashok Malhotra, Gary Sockut, Luanne Burns, and K.-S. Choi. Two-Dimensional Specification of Universal Quantification in a Graphical Database Query Language. *IEEE Transactions on Software Engineering*, 18(3):216–224, March 1992.

[Zlo77] M. M. Zloof. Query-by-Example : a data base language. *IBM Systems Journal*, 16(4):324–343, 1977.

[ZS92] Albert Zündorf and Andy Schürr. Nondeterministic Control Structures for Graph Rewriting Systems. In Schmidt and Berghammer [SB92], pages 48–62.