Ludwig Maximilians Universität München

LMU

Institut für Informatik
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen
Oettingenstraße 67 D-80538 München

visXcerpt

Conception of a visual interface to a query and transformation language for the Web

Beginn der Arbeit : 4.August 2002
Abgabe der Arbeit : 4.Februar 2003

Betreuer : Dipl. Inf. Sebastian Schaffert,
Prof. Dr. François Bry

Sacha Berger

Abstract

This thesis covers the conception of a visual query and transformation interface for XML, based on pattern matching. The interface is a front end for Xcerpt [bry02gentle], a descriptive query and transformation language for XML. The system aims at easing the task of querying semi-structured data for the casual end-user who often feels uncomfortable with textual programming languages. The goal of this thesis is to fill the gap between the interactive nature of current research on XML query systems, like QURSED [petropoulos02qursed] and BBQ [munroe00bbq], and research about visual query languages for XML and object oriented database systems. While visual languages usually provide a good approach to statical representation of program or query structure, the editing and browsing of programs is rarely covered by visual languages. Interactive query systems on the other hand do not provide a static view of programs that can be understood or stored outside the context of the query environment. In contrast, the visual Xcerpt interface presented in this thesis covers both aspects: visual representation of Xcerpt programs and an editor model tailored for this view. Due to the widespread acceptance of QBE-like [zloof77query] database interfaces by casual users, an intuitive pattern based approach for querying XML is desirable. VisXcerpt, the system presented in this thesis, fulfills this requirement.

Zusammenfassung

Diese Diplomarbeit behandelt die Konzeption einer auf *Pattern Matching* basierten Anfrage- und Transformationsschnittstelle für XML. Die Schnittstelle wird der deskriptiven XML Anfrage- und Transformationssprache Xcerpt [bry02gentle] vorgeschaltet. Das Ziel des vorgestellten Systems ist es, die Anfrage an semi-strukturierte Datenbanken für den unerfahrenen Endanwender zu vereinfachen, welcher üblicherweise Probleme mit textuellen Programmiersprachen hat. Das Ziel ist es eine Brücke zwischen aktuellen XML Anfragesystemen, wie QURSED [petropoulos02qursed] und BBQ [munroe00bbq], und der Forschung im Bereich visueller Anfragesprachen für XML und objektorientierte Datenbanken zu schlagen. Viele visuelle Sprachen bieten eine gute statische Darstellung von Programm- und Anfragestruktur, während das Erstellen und Browsen von Programmen selten behandelt wird. Interaktive Anfragesysteme hingegen bieten meist keine statische Repräsentation von Anfragen die außerhalb der Anfrageumgebung sinnvoll ist. Die visuelle Xcerpt Schnittstelle in dieser Diplomarbeit hingegen, deckt beide Aspekte ab: eine visuelle Repräsentation von Xcerpt Programmen und ein Bearbeitungsmodell für diese Sicht. Aufgrund der Beliebtheit QBE-artiger [zloof77query] Datenbankschnittstellen unter Endanwendern, ist ein *Pattern Matching* basierter Ansatz wünschenswert. Die hier vorgestellte Anfrageschnittstelle visXcerpt erfüllt diese Bedingung.

Erklärung

Hiermit versichere ich, dass ich diese Diplomarbeit selbstständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

München, den 4.2.2003 Sacha Berger

Contents

1 Motivation **4**

2 Related Work **7**
2.1 Visual Programming, Modelling and Query Languages 7
2.1.1 Visual Programming Languages 8
2.1.1.1 Imperative Languages 9
2.1.1.2 Functional Languages 11
2.1.1.3 Logic Languages 14
2.1.2 Visual Modelling Languages 17
2.1.2.1 ER Diagrams 18
2.1.2.2 UML 20
2.1.2.2.1 Static structure diagrams – class and object diagrams 20
2.1.2.2.2 Use case diagrams 21
2.1.2.2.3 Interaction diagrams 21
2.1.2.2.4 Collaboration diagrams 23
2.1.2.2.5 State chart diagrams 23
2.1.2.2.6 Implementation diagrams 24
2.1.2.3 UML's four layer meta model 25
2.1.3 Visual Query Languages 26
2.1.3.1 Kaleidoquery 26
2.1.3.2 Query by Example 30
2.2 Non visual XML Query Languages 33
2.2.1 Navigational queries with XPath, XSLT and XQuery 34
2.2.1.1 W3C's Standardisation Process 35
2.2.1.2 XPath 35
2.2.1.3 The transformation language XSLT 37
2.2.1.4 The XML query language XQuery 43
2.2.1.4.1 XPath and XQuery 43
2.2.1.4.2 FLWR Expressions 43
2.2.1.4.3 Optional Type System 44
2.2.1.4.4 Functions 45
2.2.1.4.5 Remarks on XQuery 46

2.2.2 Xcerpt: Pattern- and Logic-based Queries and Transformations 46
2.2.2.1 Xcerpt's syntax 47
2.2.2.1.1 Patterns 48
2.2.2.1.2 Variables 48
2.2.2.1.3 Rules 50
2.2.2.1.4 Programs 50
2.2.2.1.5 Goals 51
2.2.2.1.6 The Descendant construct 51
2.2.2.1.7 Disjunctions and Conjunctions 52
2.2.2.1.8 Regular expressions 52
2.3 Further Visual Languages and Query Systems for XML 52
2.3.1 XML-GL and WG-LOG 54
2.3.1.1 About Sara Comai - "Graphical Query Languages for Semi-Structured Information" 54
2.3.1.2 About Stefano Ceri et.al - "XML-GL: A Graphical Language for Querying and Restructuring XML Documents" 55
2.3.2 GraphLog 58
2.3.3 VXT 59
2.3.3.1 VXT: Visual XML Transformer 60
2.3.3.2 VXT: A Visual Approach to XML Transformations 60
2.3.4 BBQ 61
2.3.5 QURSED 61
2.3.6 Xing 62

3 visXcerpt 65
3.1 Aspects of visXcerpt 65
3.1.1 Static Aspects of visXcerpt 66
3.1.1.1 visXcerpt's Generic XML Visualisation 66
3.1.1.2 visXcerpt and graph structures 68
3.1.1.3 visXcerpt Elements 71
3.1.1.3.1 Program 71
3.1.1.3.2 Rule 71
3.1.1.3.3 Head (Construction Pattern) and Body (Query Patterns) 72
3.1.1.3.4 Query 72
3.1.1.3.5 Variable 72
3.1.1.3.6 All and Some 73
3.1.1.3.7 Descendant 73
3.1.1.3.8 Regular expression 74
3.1.1.3.9 Logical connectives 74
3.1.1.3.10 User constraints 75

3.1.2 Dynamic Aspects of visXcerpt ... 75
3.1.2.1 The visXcerpt Editor Model ... 77
3.2 Examples and Use Cases ... 78
3.2.1 Simple visXcerpt Programs ... 78
3.2.1.1 Database Identity ... 78
3.2.1.2 Querying for Parts of the Database ... 80
3.2.1.3 Using Multiple Queries in one Rule ... 83
3.2.1.4 Xcerpt Goals and the visXcerpt Query Frame ... 83
3.2.1.5 The Rule's Head ... 83
3.2.1.6 Programs with many Rules ... 86
3.2.1.7 Chaining Multiple Rules ... 88
3.2.1.8 Practical Example - a core crawler for the web ... 92
3.3 A Prototype for visXcerpt ... 94
3.3.1 Target Platform ... 97
3.3.2 Modelling the Prototype ... 99
3.3.2.1 A Motivating Example ... 99
3.3.2.2 The IXr XML meta-model ... 100
3.3.2.2.1 Properties of host Documents ... 101
3.3.2.2.2 IXr Annotation Attributes ... 101
3.3.2.2.3 IXr and the visXcerpt prototype ... 104
3.3.2.3 Assembling visXcerpt's Client Components ... 105
3.3.3 The visXcerpt Server ... 107

4 Conclusion 109
4.1 Language Conception ... 109
4.2 Editor Conception ... 110
4.3 Prototype Conception ... 111
4.4 Need for empirical evaluation and user feedback ... 112

Chapter 1

Motivation

Due to the wide acceptance of the World Wide Web, a multitude of information has been made available electronically in the past, and the growth rate of online resources is still increasing. The World Wide Web is used for information, communication, entertainment, commerce, research, et cetera. Web content can be obtained from "free for all" to "the sky is the limit" for a few selected, the quality ranges from valuable to sheer nonsense. While the WWW initially was mainly tailored for human consumption, the demand for machine understandable data is actually increasing [lee02semantic]. This is mainly due to the huge amount of online information, making feasible information retrieval for humans more and more difficult. The technology of the WWW – originally mainly document oriented – is passing through a metamorphosis into a data oriented system. This vision will be implemented through the use of the generic markup language XML[1]. XML is an instance of the semi structured data model.

The data oriented view [abiteboul00data] of the WWW brought up the need for general purpose database tools, like query and transformation systems. These demands are mainly satisfied by the standards XPath[2], XQuery [wadler02xquery] and XSLT[3]. Further query systems for XML are an active field of research [bry02gentle], [schl01b], [bry02xpath], [may01logic], [baru98features].

While reflecting over traditional database query systems, it gets apparent that textual query languages like SQL are mainly used by database experts, administrators or programmers, while visual systems, like those based on QBE [zloof77query], have a stronger acceptance among casual users. Visual query languages seem to be more intuitive and easier to apply, indeed.

As most computer users have access to the WWW, we believe in a broad user base for a visual query system for XML data on the web. Furthermore,

[1]This WWW resource is located at `http://www.w3.org/XML/`
[2]This WWW resource is located at `http://www.w3.org/TR/xpath`
[3]This WWW resource is located at `http://www.w3.org/TR/xslt`

the user should be able to describe his query, without having to bother about efficiency – the queries should be as descriptive as possible, because this leaves much space for automatic query optimisation.

For a wide acceptance of the semantic web vision, a widely spread comprehension of the semi-structured data model is necessary. For this purpose the general sensibility for recognition of hierarchical structures must be aided and supported. A visual query system for XML therefore needs a striking visualisation of hierarchical structures. To minimize the concepts, a pattern based approach is helpful. As soon as the overall nature of semistructured data and the concrete structure of an application domain have been understood, the additional effort to understand patterns is very low. A pattern can be understood as an example using the same formalism as used for a database itself. Non pattern-based query paradigms, e.g. navigational queries, require the introduction of a new formalism to express queries. A novice user hence needs to learn two formalisms for non pattern-based query systems (the database and the query formalism) while only having to learn one formalism when using a pattern-based query approach.

The existing textual query and transformation language Xcerpt [bry02gentle], [bry03pattern], [bry02towards], [bry02the] is used as back end of the proposed query system. Xcerpt is a textual query language based on graph unification. The language follows a rule-based approach comparable to Prolog. A rule consists of a construct term and *and/or* connected query terms. Thus the queried or transformed XML data itself can also be interpreted as term. Term based rules have been used for a long time in logic programming, and reasoning (aka. program evaluation) is usually based on unification. Unfortunately a classical term interpretation for XML data is too rigid compared to the expressive power of XML data. While term unification relies on the arity and the argument order of the matched terms, XML patterns with varying arity and varying child element order can be interpreted as equal, based on partial equality. Those type of patterns are known as incomplete patterns. For unification of incomplete patterns on graph structures, a technique called simulation unification is used [bry02towards].

In *"A Gentle Introduction into Xcerpt, a Rule-based Query and Transformation Language for XML"* [bry02gentle] the syntax of Xcerpt is explained with many examples. An outlook about different possible procedural semantics is also given. The declarative semantics of Xcerpt as far as defined is explained in *"Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification"* [bry02towards]. *"Pattern Queries for XML and Semistructured Data (revised version)"* [bry03pattern] focuses only on syntax and semantics of query patterns in Xcerpt and visXcerpt. In *"The XML Query Language Xcerpt: Design Principles, Examples, and Semantics"* [bry02the] an overview of the declarative semantic and language features of Xcerpt is given. This article

also presents planned features for future development. The syntax of Xcerpt will also be explained in detail in this thesis.

Chapter 2

Related Work

2.1 Visual Programming, Modelling and Query Languages

As Margaret M. Burnett says *"Visual programming is programming in which more than one dimension is used to convey semantics"* [burnett99visual]. The same is applicable to visual modelling and visual query languages. As mentioned by Paul A. Fishwick [fishwick00aesthetic] a concise metaphoric mapping with sufficient symbolism is needed for aesthetic programs or models to be executable on a computer.

Actually most software is written in textual languages. The reason is that text is easier and cheaper to cope with than images or other artistic n-dimensional data. Development in the past decades successfully introduced the graphical user interface, replacing the textual operating system shells. Performance gain of the hardware enabled this evolution. Visual query languages were developed back in the 70ies [zloof77query] and are widely accepted, e.g. in modern database systems. More recent development introduced UML [omg01uml], [rumbaugh98uml], a visual modelling language for software and business models.

While looking at concrete visual languages, it seems that the constructs are chosen according to aesthetic preferences of the developers, hardly pursuing possible general visual vocabularies. Nevertheless there are some abstract concepts for querying, programming and modelling that one has always to cope with: nesting or inclusion and association. In some languages those concepts are solved in a rigid way, i.e. the extensions and positions of the constructs in the dimensions of the languages are fixed, according to their relationships (e.g. nested boxes or circles indicating inclusion). In others, those relationships are modelled so that dimensional information is independent of relationship (e.g. arrows indicating association). Usually

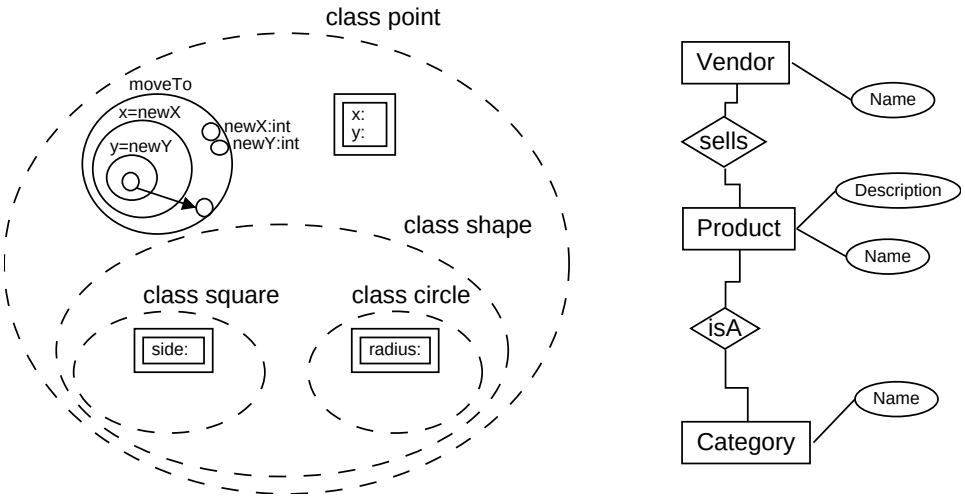

Figure 2.1: Two different approaches for visualisation of structures: On the left a VIPR expression that represents hierarchical structure through spatial inclusion. On the right an ER-diagram where structure is represented through edges between visual components.

most languages mix both paradigms.

In Paul A. Fishwick's research [fishwick00aestetic] the concept of *aesthetic programming* is introduced. Based on visual approaches in programming, this concept describes a methodology for definition or modelling of new visual models. The novelty in this work is that a general methodology is proposed. The research indicates a shift in development away from ad-hoc methods to a general structured methodology of visual system development. The potential impact of those methodologies on visual languages can be compared to the impact of object oriented programming on software development. More complex tasks can now be covered, because methodologies provide interfaces and guidelines for people working together on large projects.

The following sections will present different visual programming languages, query systems and modelling languages. They are meant as inspiration for possible visual constructs and not so much comments about the qualities, advantages and draw backs of the presented languages. Usually there will not be too much in depth coverage of the semantics of those systems, the focus will be on the syntax.

2.1.1 Visual Programming Languages

Visual programming languages must be distinguished from Visual programming environments. Visual programming languages fulfil Burnett's [burnett99visual] definition and use more than one dimension to convey seman-

tics, while visual programming environments use visual interfaces to generate parts of textual programs. Therefore systems like *Visual Basic (R)* are not visual programming languages as often proclaimed, but visual programming environments. They provide a classic textual programming language and user interfaces to browse the structures and constructs of projects written in this language. Usually a graphical editor for user interfaces is also provided.

The borderline between textual and visual languages is blurred and could be placed near languages like *Haskell* and *Python*, where indenting is used rather than braces to indicate block structure. If the use of n-dimensional constructs goes beyond indenting text, a normal text editor is no longer usable to develop programs. Hence visual programming languages usually need their own visual programming environments, but they are rarely mentioned in publications about visual programming languages.

Early work in the field of visual programming had *"[...] advantages that seemed exciting, when demonstrated with 'toy' programs, but ran into difficult problems when attempts were made to extend them to more realistically-sized programs. These problems led to an early disenchantment with visual programming, causing many to believe that visual programming was inherently unsuited to 'real' work – that it was just an academic exercise."* [burnett99visual]. As a consequence visual programming shifted away from the domain of general purpose programming languages to domain specific and specialised languages. For those applications users are experts in the domain, but not in programming. Domain specific visual languages with a high degree of system abstraction received a better acceptance, because users did not need additional knowledge beyond their specialised domain knowledge. Digital signal processing is one of the domains where visual programming is often used.

The ideas of visual programming are usually similar to those of classical/textual programming languages. Properties like e.g. Turing completeness, side effect handling, control flow or level of descriptiveness, are relevant for textual and visual programming as well. Therefore the following overview is divided in the three categories *imperative*, *functional* and *logic*languages, which are also used to categorise textual programming languages.

2.1.1.1 Imperative Languages

Imperative programs can roughly be described as sequences or blocks of instructions/assignments. These sequences can be grouped into procedures. The entities that values are assigned to, are called variables and can be compared to named memory areas. Instructions can be computations or procedure calls. Most imperative languages provide support for primitive data types, like strings, numbers and boolean values, and also for structured values and pointers to values.

Conditions can be expressed through `IF-THEN-ELSE` or `CASE` constructs, forking the sequence of assignments. Sequences can be repeated, `WHILE` a condition holds, or `UNTIL` a condition is reached. It is usually possible to iterate over a sequence of values – `FOR` every pass the value of a variable is set to an actual iteration value (e.g. a number) and can be accessed in the block that is iterated.

Procedures may be defined to require parameters and they may return a result value. When a procedure is called, the control flow in the calling sequence is suspended and the block of the procedure is executed. When a procedure returns (this happens when its sequence of assignments is done) the calling block continues its execution.

VIPR [citrin94design] is a visual imperative and object oriented language. The fact that the language is object oriented is irrelevant here, because this property is orthogonal to the imperative nature of this language. In "The design of a completely visual OOP language" [citrin94design] the visual imperative object-oriented programming language VIPR is presented. VIPR's semantic is inspired by C++, but it can be understood in terms of a small amount of graphical rewriting rules. Features like continuations and dynamic dispatch, which exist implicitly in C++, can be made explicit for the sake of a better program comprehension. The static view of programs as well as the visualisation of dynamic execution can be presented with the same visual metaphors.

In VIPR, programs are represented as nested circles. The notation can be seen as viewing a flowchart from the point of a bead moving along the control flow lines. The nested circles can also be interpreted as segments of a pipeline with a point of view inside this pipeline. VIPR defines objects and classes by aggregating definitions of data and procedures visually. Interfaces are defined by hiding all the private methods and data. A dynamically dispatched method contains branches to all the appropriate implementations of the method, and the correct branch is selected dynamically based on the type of the object being called.

Procedures and functions are represented as sets of nested rings outside the ring of the main procedure. A procedure call is an arrow pointing to the procedure, with nodes attached representing the arguments. Nodes attached to the outermost ring of the procedure represent parameters. Each procedure must have at least one parameter, representing the continuation or the return address. The assignment of arguments to parameters is done through positional matching.

Classes and objects are shown as aggregations of methods. In essence, a VIPR class is a grouping of individual methods, each of which looks like a strand of fibre being viewed head-on. Subclasses are enclosed inside the aggregate representing the superclass. This is contrary to the usual way of modelling class hierarchies, where subclasses indicate their superclass. This implies that the superclass is modified when a subclass is defined, but the

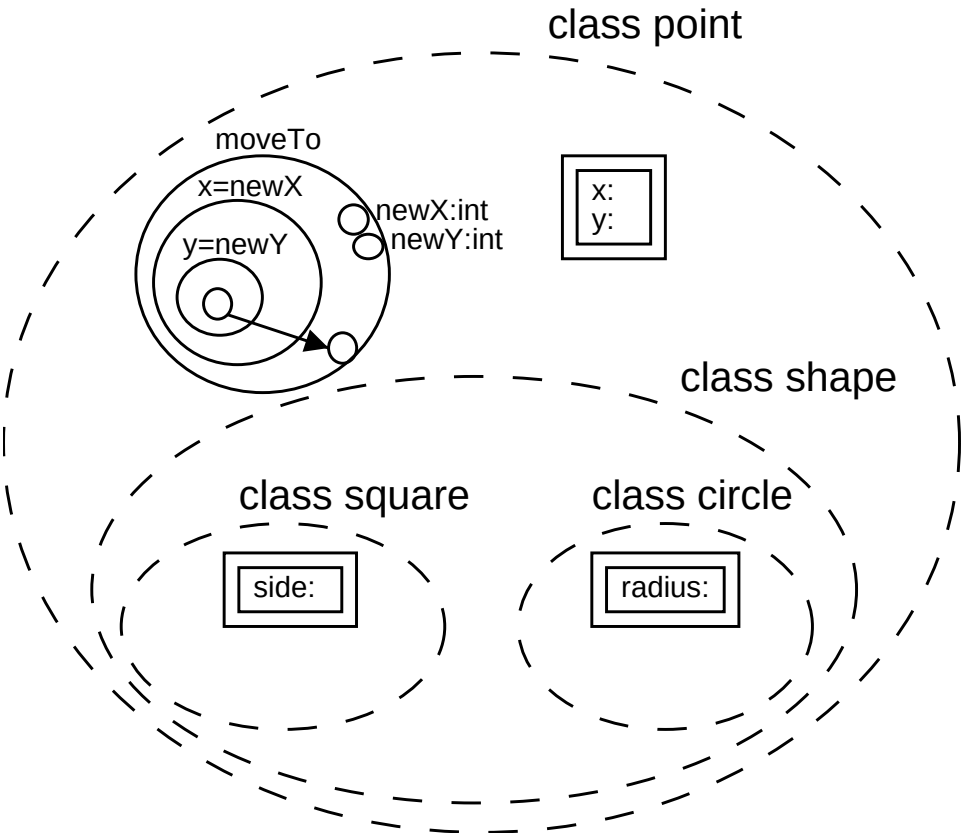

Figure 2.3: Three VIPR classes are presented here. The class named *point* is the base class of *shape*. *square* and *circle* are subclasses of *shape*. All of them share the *moveTo* method and the coordinates. *square* and *circle* have further class members.

system environment can automatically deduce this for the user. Thus only single inheritance is supported (see figure 2.3).

The use of arrows in a visual environment usually leads to scalability problems concerning readability. Thus VIPR programs may be enriched by text when it simplifies the representation. Nevertheless, the semantic of the programs is fully determined through non textual components. Therefore the textual elements may be seen as documentation or syntactically restricted comments (the textual representation of visual elements is the syntactical C++ equivalent)

2.1.1.2 Functional Languages

The main difference between pure functional and imperative programming is that functional variables do not represent storage or memory areas, but represent values or functions. Once a value is assigned to a variable this value may not be altered any more. Hence the result of evaluating a function or

querying a variable has no side effects, which means it will always return the same result. Therefore the control flow – the order of evaluation – is no longer explicitly determined in a program. Optimisation based on restructuring the evaluation order and reducing duplicate calculations can automatically be deduced. Lazy evaluation and caching of results as well as automatic parallelisation with syncronisation are advantages of modern functional languages.

Most functional languages have functions and data types as their basic constructs. A function is defined in terms of other functions and constructors. A constant is a function of arity 0. Inside a function, local functions (and therefore also variables) can be defined using so called `LET` or `WHERE` clauses. There are various ways to express conditions, e.g. pattern matching. Other features found in many functional programming languages are *currying* (aka. *'Schoenfinkeln'*) and higher-order functions.

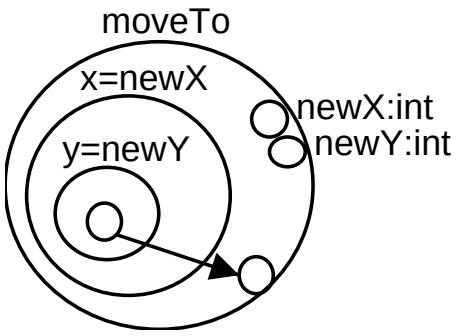

Figure 2.2: This illustration shows a VIPR method. It is called "moveTo" and accepts two arguments: newX and newY. First newX is assigned to the object member x (the object is not shown here), then newY to y. The method returns a void or null value. The arguments and the return argument are attached to the outer ring.

While people concerned with imperative programming often argue that the lack of side effects is of disadvantage for optimisation, it is easy to emulate by sequencing functions taking state objects and returning state objects. This programming pattern is known as monadic programming [jones93imperative].

The success of visual programming in the field of digital signal processing (mentioned in the introduction of this section) is strongly related to functional programming. A simple, non technical entertainment oriented and high level system is the open source software called BEAST[1]. BEAST can be used to build sound synthesiser networks based on simple synthesiser components. Each of those components has input and output slots that are connected by edges. Each instance of such a component can be seen as a function without side effects. This methodology emulates the classical construction of analogue synthesisers.

A general purpose visual programming language is Visual Haskell [john94visual]. The roots of the project also originate from the signal processing community.

Visual Haskell is a visual functional programming language based on the textual functional language Haskell. It is very tightly bound to Haskell.

[1]This WWW resource is located at `http://beast.gtk.org/`

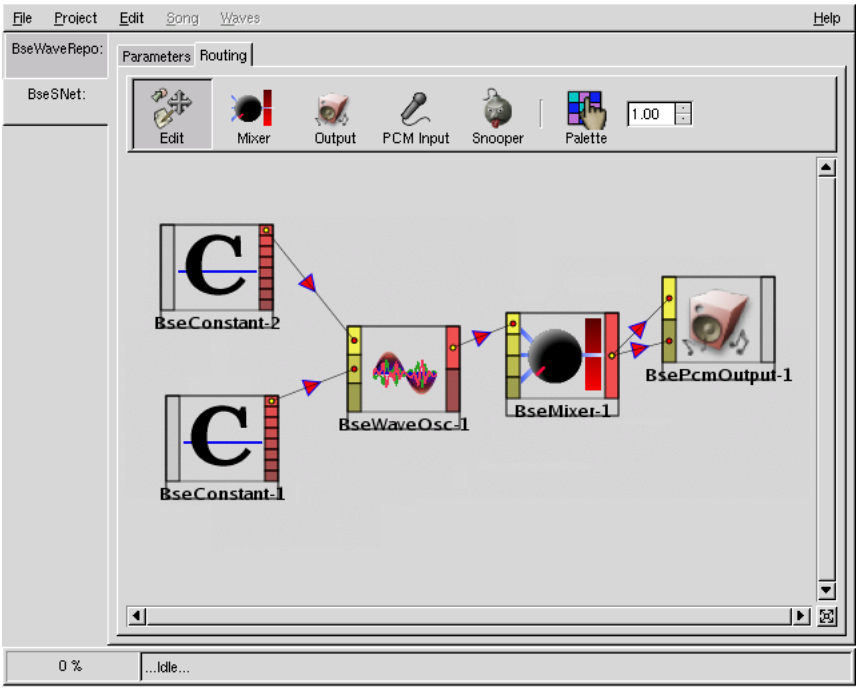

Figure 2.4: A screen shot of a synthesiser network in BEAST. The arguments passed through filters are signals. Signals are time based values. The two constant values on the left are used to influence the sample player *BseWaveOsc-1*. A virtual amplifier further modifies the signal before the output audio bus is fed with it.

This way Haskell programs can be seen or edited in the textual and in the visual representation. The goal is to provide a visual representation to all constructs found in Haskell.

The main concept behind visual Haskell is the data-flow diagram. This seems to be generally the widespread visual view for functional programming languages. A function is defined as a box labelled with its name and optionally an icon or a visual representation. Those icons are used to provide new graphical elements for application-specific data types and functions. This way a visual metaphor is provided for a function. The visual metaphor. At one of the outer sides of the boxes bound (usually the right side) there are slots for the input arguments, at the opposite side (usually the left side) a slot for the output is available. At the inner side of the input argument slots' side, visual patterns can be attached with further slots to pass the pattern variables to concrete function bodies. This is done through arrows. Different patterns and different corresponding bodies can be stacked (separated by a line) inside the function definition box.

An icon representing a function may have a special grey box representing a visual variable for the first argument of the function. The special focus on the first argument of the function is due to a wide spread functional methodology called *currying* - with currying a n-ary function may be applied to m ($< n$) arguments resulting in an $(n-m)$-ary function (Thus compilers or virtual machines can break down every function application into a series of unary function applications). Functions in functional libraries are often designed so that applying something to the first argument results in a very practicable specialised function. This can be done in a cheerful way in visual Haskell, because of the visual variables inside iconic function representation.

2.1.1.3 Logic Languages

In classical logic programming neither functions nor procedures exist, only rules and facts are known. A fact is a predicate followed by an n-tuple of terms. A term can contain variables. Prolog is a widespread logic programming language. In Prolog rules are restricted to a subset of Horn clauses, disjunctions of facts with exactly one positive fact. A more popular view of the Horn clauses used in Prolog is an implication from a body (which consists of a conjunction of predicates) to a head (consisting of exactly one predicate). A special Horn clause is called fact; this Horn clause has an empty body interpreted as *true*. Program execution of logic programs is a kind of reasoning on the facts and rules the program consists of. The result of a program execution can be seen as proof of the rules and facts. It can be true or false, and free variables can be interpreted as being bound to the facts that fulfil the program.

Alike to the implementation of the imperative paradigm that has been presented in the section about functional programming, the implementation

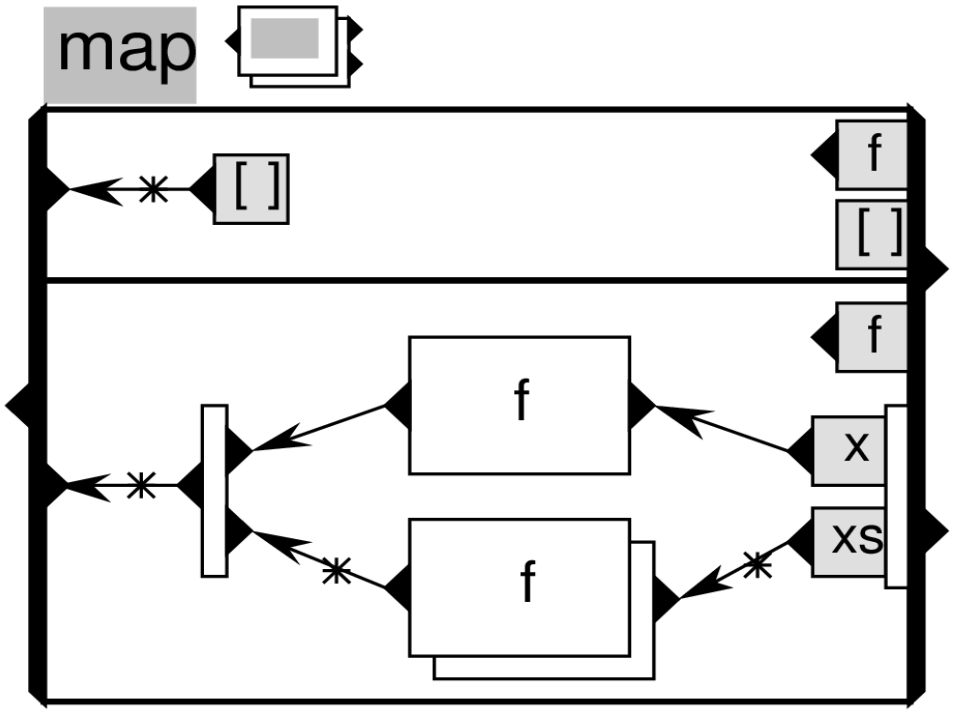

Figure 2.5: The classical *map* function in visual Haskell. Right beside the name there is an icon used to depict the function. It is used (for purposes of recursion) in the function itself with f as argument (the last occurrence of f). Function arguments are represented by the external triangles on the right, the external triangle on the left is the result of the function. Two patterns are used: the first for mapping f to the empty list – the empty list is returned, another for the recursion case. The list constructor is the vertical white bar used twice in the second pattern – once for deconstruction into head x and tail xs, another time for construction of the mapped list right before the functions output.

of functions in logic programming will be presented now. To implement functions in logic programs, rules are needed that relate the result to the arguments. To represent $f(x) = y$ in a logic program, one may use a relation $f(x, y)$. From the point of view of a relation, it is irrelevant which parameters are the input of a function and which are the output, thus functions implemented in logic programming languages can often be evaluated in both directions.

The execution of logic programs is usually based on resolution, which itself uses unification to either check equality of terms or provide variable bindings. A runtime system for logic programs is an inference system.

In logic programming there is no explicit control flow specification, only rules and facts. For visual programming, this means that only formulas and terms need to be visualised. Visual logic programming languages appear rarely to provide visual metaphors of those very abstract concepts, but provide domain specific concept visualisation. A visual logic language [puigsegur97from] with expressions based on acyclic graphs and sets is presented.

The visual language presented in *"From Queries to Answers in Visual Logic Programming"* [puigsegur97from] is comparable to Prolog in its expressiveness.

Terms are represented through directed acyclic graphs consisting of labels and variables. Variables are anonymous and are represented by circles, the parts of terms with explicit names are represented by labels written in a round box. If a term f is composed of other terms or variables, an arrow is drawn from the corresponding member to *f*. Sharing a variable by using multiple arrows pointing away from it, is used to represent multiple occurences of the same variable in a term representation. Because there is no explicit order for arrows pointing to a term node, terms could be interpreted as having unordered application of arguments, but this is not explicitly noted in [puigsegur97from]. The examples seem to indicate an implicit order for arrows from left to right.

Predicates are interpreted to represent set inclusions. Therefore one argument of a predicate is omitted in arrow based application and is represented by the visual extension of the predicate visualisation. A predicate is visualised as rectangle with its name written in the top left corner. All further predicate arguments not treated as the set representing argument are applied with arrows as mentioned earlier. No explicit order is given for the occurence of the set representing variable in a textual transcription of predicates, but examples indicate that the first position is used for the implicit set representation variable. To express a subset relationship, nesting the box that represents the subset into the box representing the superset can be used. To express intersection of sets, the boxes that represent the sets are layed out in an intersecting way.

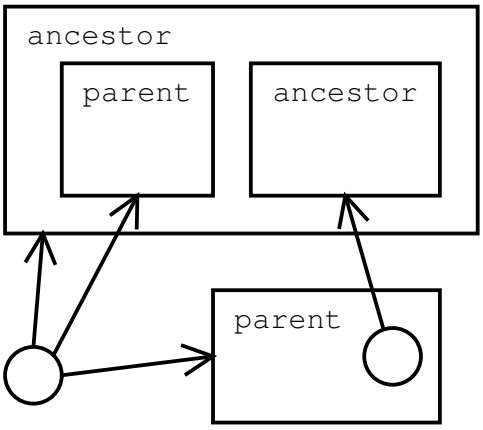

Figure 2.7: In this example predicates and variables are used. It models the ancestor predicate: the left variable has its parents as ancestors (through inclusion in the big ancestor box) as well as the ancestors of its parents.

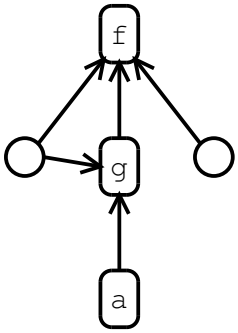

Figure 2.6: This graph represents a term. Because the variables are unnamed, there are ambigous textual representations, but *f(X,g(X,a),Y)* is one of them.

The smallest complete unit of description in this language is a diagram. A diagram is composed of various predicates and terms connected to one graph. A diagram represents a visual equivalent to a textual formula. Queries to a program (a set of formulas) can be expressed through use of explicitly marked variables. The variables are labelled by a questionmark to indicate that their bindings are to be returned as result.

The use of abstraction over one predicate argument for set representation reminds of currying in functional programming. To some extent this language supports a blend of functional and relational programming style.

visXcerpt, the query and transformation language proposed in this thesis (in the next chapter), is also a visual logic programming language. Expressions in visXcerpt are based on a visualisation of XML data, the program evaluation is based on an unification algorithm called simulation unification.

2.1.2 Visual Modelling Languages

Visual modelling is accepted more than visual programming. A reason for this could be that models are based on structure and structures are mentally

coped with in a static[2], easily visualised way. Furthermore the main goal of modelling is to provide an interface for humans to understand data, thus everything intuitive is welcome to support modelling. In contrast to modelling, the main goal of programming is 'to get things done'. Even though the most important rating for programming languages is based on its readability for humans, programs need to be executable by machines. The deepest level of detail, necessary to instruct a machine without ambiguity, can be omitted in models, because human intuition fills the gap when comprehension of high level structure is focused.

Some well known modelling languages are presented now. The main purpose of modelling languages is documentation, with programs automatically derivable from models created in some modelling languages. Some systems even provide support for reverse engineering of software source code[3]. Thus it is possible to transform back programs into models representing the structure of those programs.

2.1.2.1 ER Diagrams

A popular diagrammatic modelling language is the Entity Relationship diagram. It is based on the Entity Relationship model. The ER model combines ideas of the network model, the relational model and the entity-set model. The main concepts of the ER model are called *entity set* and *relationship set*. Two or more entity sets are related through a relationship set. The relationships can be quantified as *1:1*, *1:n* or *m:n* relationships.

In ER diagrams, an entity set is visualised as a rectangular box and the relationship set is shown as diamond shaped box. The name of entities or name of relationships is placed inside its box. Associating entities to relations is performed by connecting them through lines. Beneath those lines the optional quantifiers *1*, *n* or *m* can be placed. A later extension introduced attributes. Attributes are ovals containing names and are associated to exactly one entity set. They represent the atomic members of an entity set.

The Entity Relationship model has not been implemented in any database system, but it is commonly used during the database design phase, because it provides less restrictions and higher abstraction than most widely implemented database models. For the relational database model (and also for other models), methodologies translating the entity relationship model into the implementation model exist. Although those translations can be derived automatically, it is common to refine the translations manually.

[2]static must be understood as visually representable in 2D or 3D without temporal aspects like e.g. animation. A problem with animation is that the user needs to interact to fully understand a dynamic model.

[3]This WWW resource is located at `http://www.rational.com/`

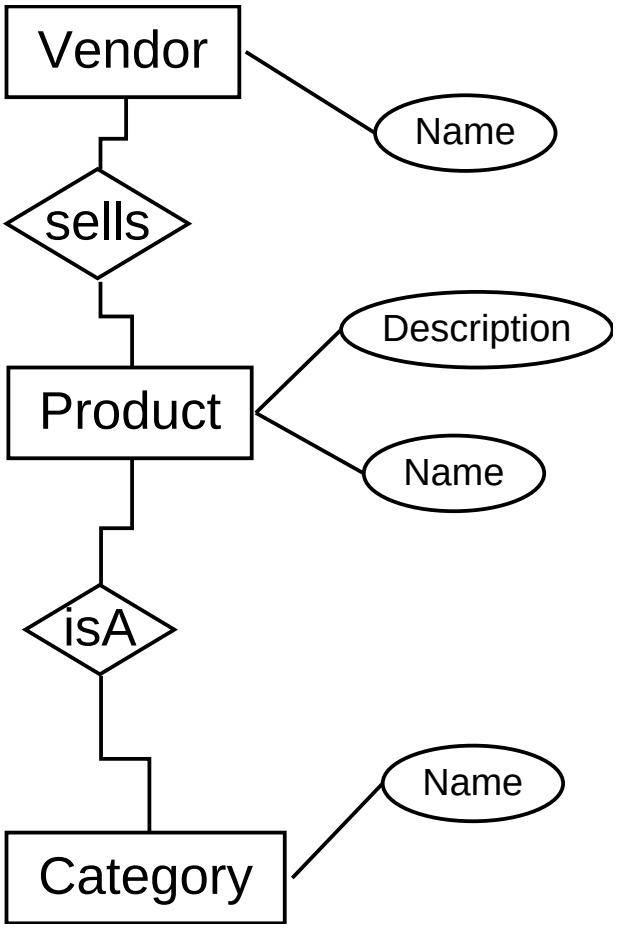

Figure 2.8: A database with vendors, products and categories is modelled here. The vendors sell products and products belong to categories. It is possible to find out which vendors sell which products and which products are sold by which vendors. The same applies for the *isA* relationship. Vendors and categories have a name. Products have a description and a name. The attributes are considered to be of primitive type, like e.g. string.

2.1.2.2 UML

UML is the abbreviation for the *Unified Modelling Language*, a standart managed by the OMG[4]. UML is used to model object oriented software (or in general object oriented architectures). Programming is not in the scope of UML. An UML model is used as plan for implementation, it can be compared to blue prints for house construction.

UML is a diagrammatic language with various diagram types. The topics covered by each of those diagram types are mostly non-overlapping, so as to make clear which diagram type to use for which aspect of the model.

The different diagram types will be presented now. A following description is based on [omg01uml] and [rumbaugh98uml].

2.1.2.2.1 Static structure diagrams – class and object diagrams

Those diagrams are used to model object oriented aspects of the project. These diagrams are widely accepted for applications implementing UML partially. While a class diagram represents relationships and properties of classes, an object diagram is used as example for concrete instantiations.

First of all objects and classes are modelled as boxes. Some visualised aspects are contained inside the boxes, others use lines and arrows between boxes. The aspects modelled are:

- **Inheritance/generalisation:** This is shown as a line with a big arrow head pointing towards the superclass. Usually those lines use vertical and horizontal components only. If a class generalises many classes the same arrow head is usually shared.

- **Attributes/operations:** Primitive attribute members and operations are contained as vertical list inside the class box.. Primitives are usually strings, boolean values and numbers. For other members, association or composition is used. Operations may contain parameter lists enclosed in parentheses. It is a purely textual feature.

- **Types:** The type of parameters, attributes and operations is appended, separated by a colon. It is a purely textual feature.

- **Visibility:** The visibility of a member is modelled by prepending a textual symbol like + for public and - for protected. Standard colour icons for the visibility attributes exist. It is questionable if those icons make the visibility aspect a non textual aspect, just because they happen not to be part of the usual font sets, after all it does not provide a new dimension.

[4]This WWW resource is located at `http://www.omg.org/`

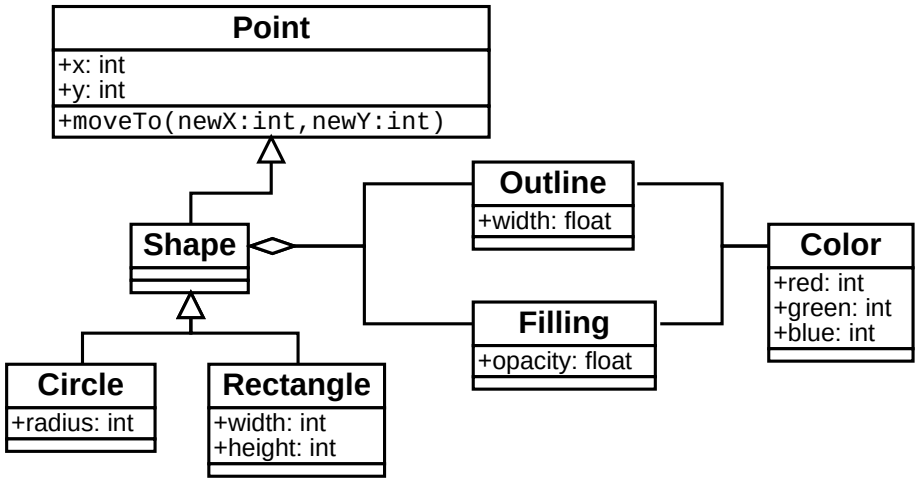

Figure 2.9: A static UML class diagram. Every box represents a class. *Circle* and *Rectangle* are derived from *Shape* which is derived from *Point*. A *Shape* has an *Outline* and a *Filling* aggregated to it (they may not be shared between instances of shapes). *Outline* and filling have both a *Colour* associated to them.

- **Association/composition/aggregation:** Those features are used for non primitive members of classes. They are mainly represented as lines between two classes. They may be directed and a name must be associated to such a line. The name is usually written below the line. Composition and aggregation are special cases of associations and are shown through diamond arrow heads either filled (composition) or hollow (aggregation).
- **Multiplicity:** On both ends of an association the relations multiplicity may be written in textual form..

2.1.2.2.2 Use case diagrams

Use case diagrams are used to model relationships between use cases and actors. They are only for documentation purpose, no automatic program generation is known that benefits of use case diagrams.

A use case diagram is a graph with use cases and actors as nodes. The arcs are so called *use case relations* and *actor relations*. Actors are drawn as stick-figures with labels below, use cases are abbreviated by ovals with labels inside.

2.1.2.2.3 Interaction diagrams

Interaction diagrams are realised through *sequence diagrams*. A *sequence diagram* is used to model the life line of an object, the activation and the

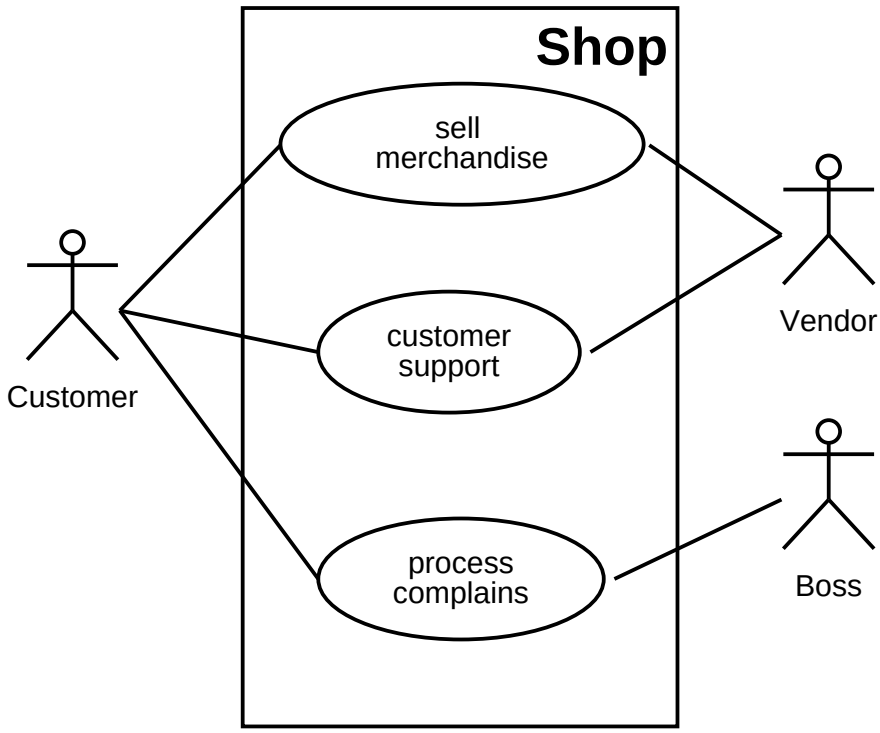

Figure 2.10: This use case diagram is about the relationships between *vendor*, *boss* and *customer*. The use cases are the three ovals in the middle.

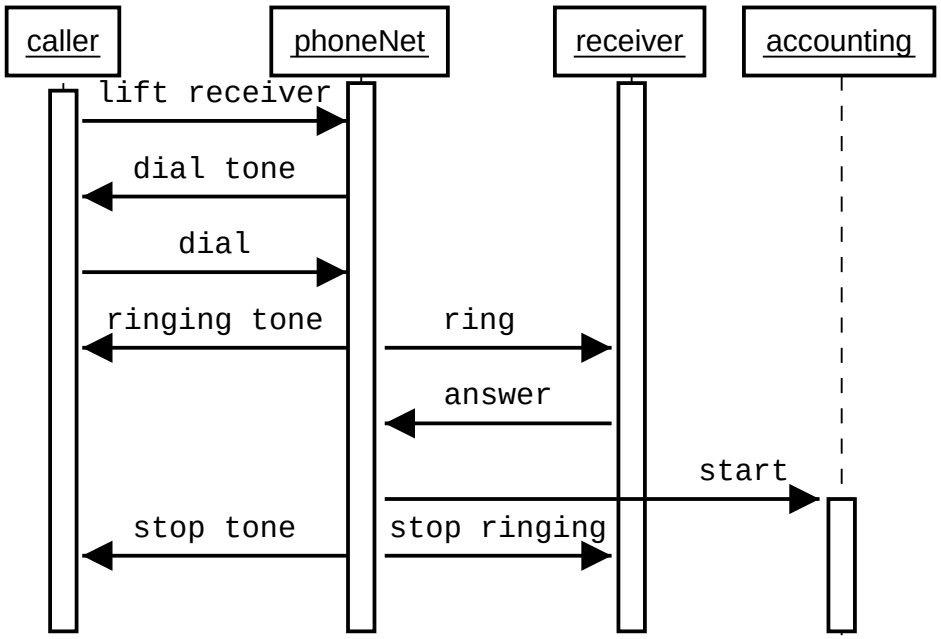

Figure 2.11: With this sequence diagram, the process of making a phone call is described. The objects involved are caller, phoneNetwork, the person who is called and an accounting system. Various messages are passed between the objects.

message passing. *Sequence diagrams* are two-dimensional diagrams: one dimension (usually the vertical) is used to display evolution through time, the other to distinguish between different objects. Therefore an object is a vertical line with a thick bar on the line indicating the life line of the object. Message passing is modelled by arrows pointing from one object to the next at a certain state transition on the life lines of both objects. When message passing is instantaneous, the arrows are horizontal.

2.1.2.2.4 Collaboration diagrams

A special case of object diagrams are *collaboration diagrams*. They provide links between objects. They represent message passing between objects and actors. Therefore also stick men are allowed in *collaboration diagrams*. The expressiveness is comparable to the *sequence diagram* without temporal aspects.

2.1.2.2.5 State chart diagrams

A *state chart diagram* can be used to describe the behaviour of an object. It is a graph of states as nodes and state transitions as directed edges. A state is displayed as rectangle, an edge as arrow. A node is labelled with the

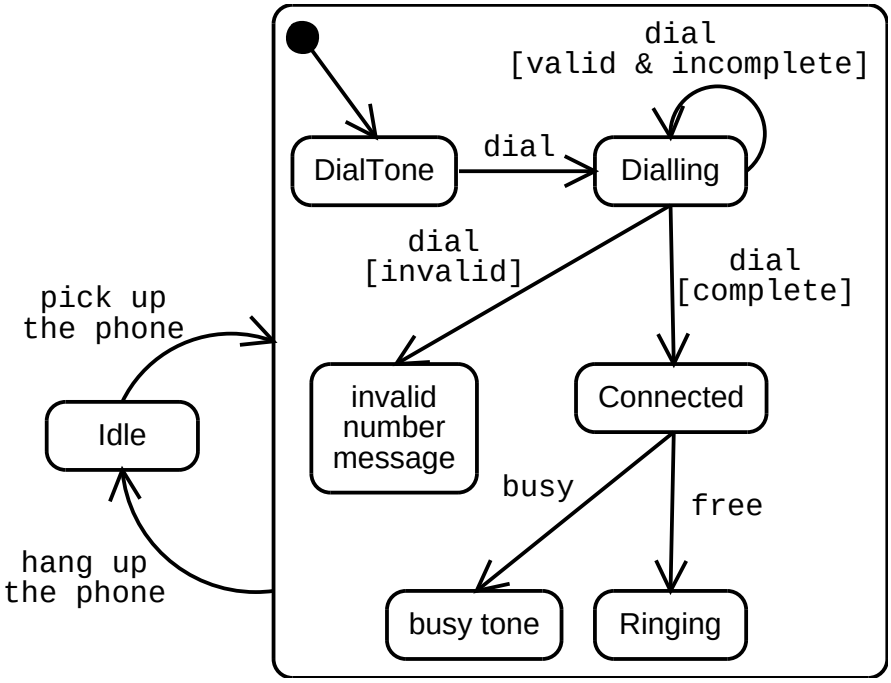

Figure 2.12: Here, a state chart diagram is used to model the procedure of making a phone call. Opposed to the previously presented sequence diagram, the involved objects are not obvious. Hence the states and the events responsible for state changes are modelled here.

state name inside the bounds. It may also be divided in two areas where the first contains the name and the other an activity in process while in this state. Edges are labelled with a name and optional preconditions (in textual form). A special state, the start state, is a little empty circle. An end state is a double bordered circle. State diagrams can be nested hierarchically, indicating sub state machines. Entering a sub state machine begins at the starting state of the sub machine. Reaching an end state means leaving a sub state.

A special case of a *state chart diagram* is the *activity diagram*. An *activity diagram* has mostly actions (a UML-term) as states. It represents some kind of flow diagram and has therefore some visual constructs for handling conditions, like a diamond shaped box for conditionals and plates for merging control flow.

2.1.2.2.6 Implementation diagrams

Implementation diagrams show aspects of the concrete implementation. There are two types of implementation diagrams:

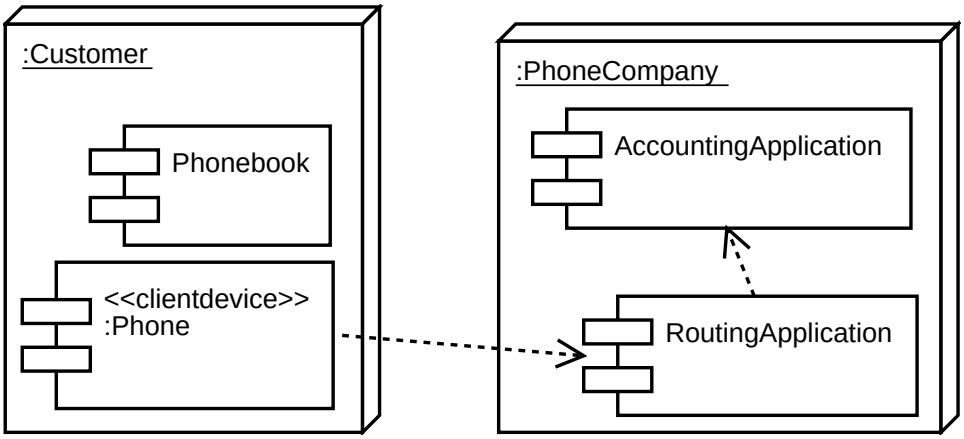

Figure 2.13: A deployment diagram is used here to model the relationships of physical instances and processes involved in a telephone network system.

- **Component diagrams:** They show the dependencies of implementation components. A *component diagram* is a graph of components connected through edges representing simple dependency, containment and implementation relationships
- **Deployment diagrams:** They represent the configuration of runtime objects like processes, and objects that are part of them. A *deployment diagram* is a graph of nodes representing processes with communication associations as edges. The processes are displayed as bevelled boxes containing *component diagrams*.

2.1.2.3 UML's four layer meta model

All those different visual diagrammatic languages in UML are based on the same four layer meta model. It consists of:

- **meta meta model:** The infrastructure used to model meta models. It is mostly the definition of a graph
- **meta model:** Meta models are the languages described above (like class/object diagrams, collaboration diagrams,...)
- **model:** A model is the output of the concrete use of the former meta models, or in terms of programming languages: modules and classes.
- **user objects:** User objects are instances of former classes.

UML itself mostly covers *meta models*. Research work about visual *meta-meta modelling* can be read in [fishwick00aesthetic].

2.1.3 Visual Query Languages

As read in [catarci93fundamental] *"Visual Query Systems (VQLs) may be defined as query systems essentially based on the use of visual representations to depict the domain of interest and express the related requests."* Among the first visual query languages is Query By Example (QBE), which is still quite popular in education and whose descendants can be found in popular commercial database systems like Microsoft Access. Common to almost all visual query languages is the fact that they are tailored for the casual user. This is not surprising, because casual users have a big demand for querying databases easily. Thus a lot of visual query languages for almost all data models have been investigated, with the goal of achieving general acceptance of their underlying data model.

A further property among visual query languages is the use of patterns. One advantage of pattern based queries is, that the same formalism can be used to express the data and the core of the query itself. For visual query languages fulfilling the former definition of [catarci93fundamental], a visual representation of the data reveals the core of the visual queries without extra effort. Casual users benefit from this consistency, because after understanding the structure of the data model, they do not need to learn a new formalism for the query language.

In [cruz90a], the connection between graphs and most applications is mentioned and a query language based on graphs is proposed. This generalisation leads to a query language that could be used in many different domains. With the casual users in mind, the use of a less abstract visual formalism that is nearer to the desired target application domain is advisable. In [fishwick00aestetic] a methodology for support of project specific aesthetic is described, but the design of a new VQL specially tailored for every new project is possibly not feasible. The right balance between abstraction (and hence application independence) and concreteness (and therefore intuitiveness) is an unsolved problem that is possibly only solvable through intuition and experience.

2.1.3.1 Kaleidoquery

The short description given here is based on [murray98kaleidoquery].

Kaleidoquery is a visual query language for object databases. It is a visual syntax that is back-ended by the textual query language OQL. The main philosophy behind Kaleidoquery is a filter flow metaphor. This can be compared to a functional paradigm. Kaleidoquery is modelled towards the needs of casual users that want to create ad-hoc queries. The iconic nature is useful for users not aware of the database schema – they can deduce it from the visualisation.

A query can be explained as a chain of filters, with all database extends

passing through this chain and being successively reduced until the intended result is left. The first element of this filter chain is an icon representing a certain class in the database. The database author is urged to provide icons to database classes. The chosen icon is the lowest member of the filter structure. For a projection of some attributes, a vertical arrow is used pointing to an oval with the attributes that should be projected. A filter network without projection as last filter step has no output. This is useful for joins, that are explained later.

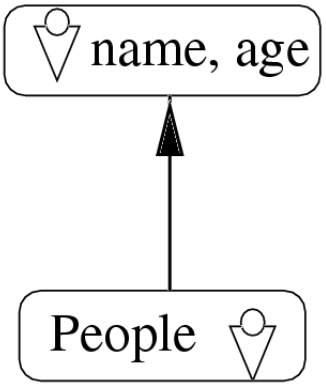

Figure 2.14: This Kaleidoquery example queries all objects in the class *People* and returns their name and age as result projection.

Further simple constraints (like less, equal,...) can be placed in textual notation on this uprising filter path. To express conjunction of constraints, all constraints are attached vertically by arrows. For disjunctions, the filter flow is split into many parallel vertical filter flows, each strand representing one disjunct. Horizontal arrangement of disjuncts and vertical arrangement of conjuncts is quite common in visual languages. Negation is expressed by inverting the corresponding simple constraints colours.

Joins are expressed by parallel filter expressions sharing a simple binary constraint, like equality. One side of the equality is then located in one filter while the other side is in the other subfilter. A self-join is also possible, when two arrows are drawn from one class oval.

Object oriented databases may store complex structures where objects may contain other objects as members. Refining a query based on certain members of selected objects along the filter flow is a common operation. Shifting the selection to the members of an object is called navigation. Navigation is represented by a horizontal arrow with the name of the member written above it.

In object databases a common operation is checking for membership of objects. For this purpose a curly horizontal arrow with the word *member* is used. The same curly arrow is also used for the *for all* and the *exists* quantifiers found in OQL.

Further expressions like functions, grouping and ordering constructs are expressed by wrapping the ovals with further ovals. The corresponding operation is included in the wrapping oval. This corresponds to the nested term structure of the textual OQL counterparts.

Kaleidoquery has a very imperative nature in the sense that the queries are refined step by step. The user has to be aware of query optimisation while using constructs like conjunction. For example using more restrictive

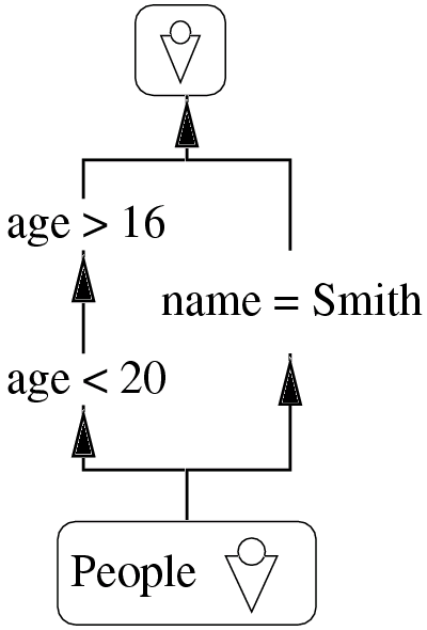

Figure 2.15: All people that are either named Smith or are between 16 and 20 years old are queried and returned. This explains the use of constraints, disjunction and conjunction.

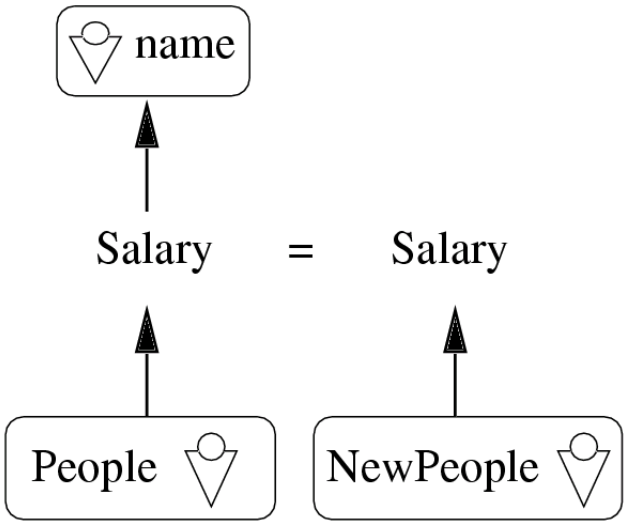

Figure 2.16: An equi-join is expressed over two databases. All names of instances in the left database which have the same salary as instances in the right database are returned.

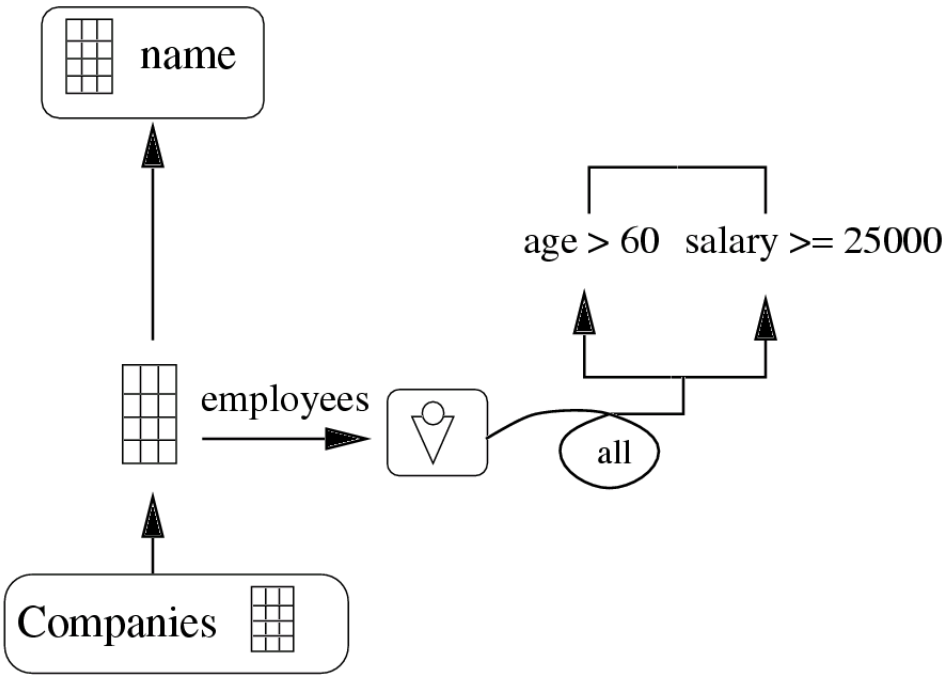

Figure 2.17: In this example navigation and a quantifier is used. The boxes with checked pattern represents companies. This query returns company names where all employees are either older than 60 years or have a salary of more than 25000.

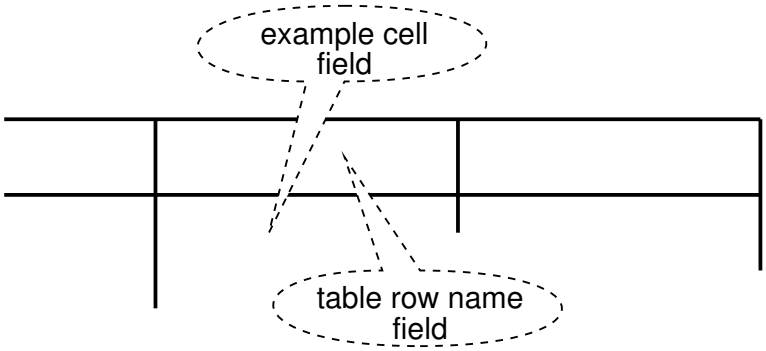

Figure 2.19: An empty table skeleton in QBE consists of (any number of) rows with a table row name and an area for values.

constraints at first is better than using them later in the filter flow. Generally, the use of edges (arrows in Kaleidoquery) tend to render cluttered views for big programs, because of line crossing. However, even big Kaleidoquery programs do not seem to get cluttered. A reason could be that edges are not used to express structure.

2.1.3.2 Query by Example

As example for a visual descriptive query language QBE will now be explained. The description is based on [zloof77query].

QBE (short for Query-by-Example) is a database management language. The base idea is that the user fills in examples of the data he/she intends to modify or query. The underlying data model for QBE is the relational data model. Because relational databases can be seen as a set of tables, the example queries edited by the user are tables.

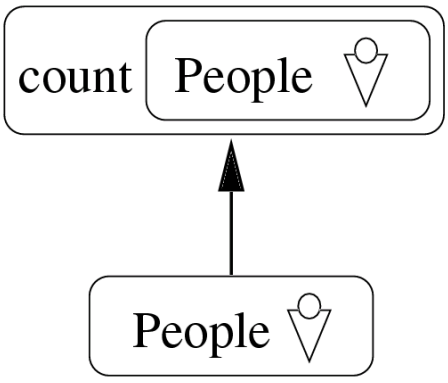

Figure 2.18: An aggregation function is used to count the people in the database. Functions, grouping and ordering constructs are nested according to their corresponding term structure in OQL.

QBE is a visual query language language, because it relies on tables which provide arrangement of information in two dimensions. Except underlined text (to indicate variables), further constructs of QBE are purely textual.

In QBE a query starts with an empty table skeleton based on the schema of the data. The user may now fill in fixed values for constraints of the result data. For different alternative constraints, more (implicitly *OR*-ed)

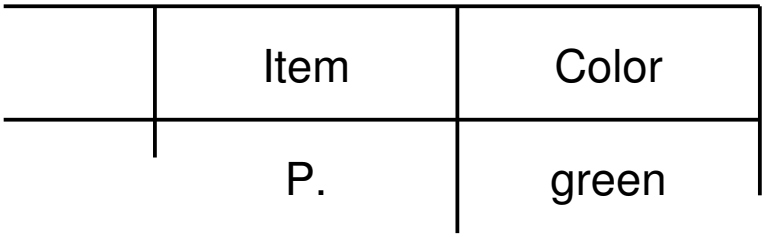

Figure 2.20: This example queries all green items in the database. The queried database consists of at least one table with at least two columns – *Item* and *Colour*.The result table presented to the user only has one column – *Item*.

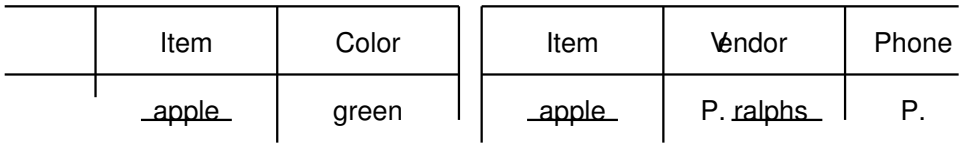

Figure 2.21: An example that uses the *apple* variable to join over two tables. The result is a list of all vendors (and their phone numbers) selling *green* products. The *Ralph's* variable is unused, but this doesn't matter. Unused variables may be introduced as examples for in-code documentation purpose.

cells are filled into the table. The result of executing such a query is a table containing only tuples found in the database fulfilling the constraints. The result schema is the same. To return data in different schemata, the user may use the keyword `P.` (for print) in columns relevant to the result.

Variables have a name and are represented by the underlined name. A `P.` can be prefixed to the variable name if the table is also to be printed. In QBE it is considered good programming style to choose names that can be seen as example instances of the desired variable bindings. The same variable can also be used in different database tables within a query - this pattern is called link. Thus joins over different relations can be formulated. Temporary result tables can be introduced. They may contain only variables of queried tables occurring in the same query.

For further qualification of variables or constant values, inequality operators and negation can be used, as well as arithmetic expressions. There is also a so called `Conditions` box: expressions over variables, arithmetic expressions, equations and *AND* and *OR* concatenated expressions can be placed here. It is intended to be used when conditions are difficult to express in table cells.

For grouping and aggregation further keywords are provided:

- `ALL.` prevents omission of duplicate elements in the result,
- `CNT.` counts the matching tuples,
- `SUM.` adda all matched numeric values together,

Item	Price
P. carrot	0.89
P. apple	0.79

Figure 2.22: An example using implicit OR to query for products that cost 0,79 or 0,89.

Item	Price
P. apple	<4.50
apple	> 0.79

Figure 2.23: This shows the implicit AND. An item is only printed to the result table, if it costs more than 0,79 and also less than 4,50.

Item	Price	ID
	price1	0815
	price2	7 987
P.	P. price3	P.

CONDITIONS
price3 > price2 + price1

Figure 2.24: A condition box is used here to express that items are used that cost more that the items with *ID* 0815 and 7987 together. *Item*, *Price* and *ID* are returned as result.

- `MIN.` and `MAX.` get the minimum/maximum of the matched values and
- `AVG.` calculates the average of the matched numeric values.

To be able to fully manage a database, not only querying but also inserting, deleting and updating is necessary. The keywords for this are `D.` for delete, `U.` for update and `I.` for insert. `I.` can also be used to create new tables in conjunction with keywords describing the valid database types. These operations are used similar to the print operation.

QBE has been used as conceptual framework for many successful commercial database systems like MS Access. For a measure of feature completeness and for the sake of comparability, the expressiveness of query languages for relational databases are compared to the relational algebra or to the tuple calculus (which provide the same expressiveness) : QBE can directly be translated into the tuple calculus and is hence as expressive as for example SQL, but it is much easier to use especially for casual users – the knowledge about filling in forms is wider spread, than the knowledge about textual database languages.

2.2 Non visual XML Query Languages

XML has been proposed by the W3C as resort out of the so called *browser war*, initiated by the rivalling companies Netscape and Microsoft. Both tried to extend the document standard HTML with incompatible extensions in form of new markups, in the hope to establish them as new de-facto standards, urging the market to accept their product as the better solution. While the pureness of HTML as a generalised markup language is vulnerable to feature extensions, the generic markup language XML is destined as meta language for HTML-like languages, and therefore encourages application developers to define new markups. Thus the former violations of HTML got transformed to productive extensions encouraged by the new standard. At the same time the extension task was shifted from the browser developers to the web site authors, urging the browser developers[5] to provide the generic extension mechanisms.

Infrastructure for the proposal of new languages without compromising standards was only one aspect of the escape out of the *browser war*. Infrastructure to provide implementations for those potentially infinite new

[5] An interesting aspect of this competence drift is that the browser developers are all members of the corresponding standardisation bodies. Therefore they seem interested in preventing new power struggles as seen in the browser war.

languages is the other task. Therefore one of the first subsequent requirements after the XML core definitions, was the ability to translate XML based languages into other languages like e.g. HTML. This requirement was first fulfilled by XSLT, a navigational transformation language inspired by functional programming and query languages.

XML's generic nature soon attracted the attention of the database community – XML is in fact an instantiation of the semi structured data model, originating from the database research community [abiteboul00data].

The high expressiveness of XSLT also led to applications that originated from database domain, but this was done (and is still done) mainly to ride the hype of XML as new technology. XSLT is tailored towards document processing and often cumbersome if used as query language. Thus a query language had to be proposed for XML. Nevertheless XPath, the central aspect of XSLT, got quiet popular and was therefore used as inspiration while developing XQuery, a new XML query language. In fact XQuery incorporates a new XPath version and at the same time is the root of its definition. This is sensible for the sake of concept compatibility throughout the XML community.

The following section will describe XPath, XSLT and XQuery in more detail. After that Xcerpt, a completely different approach to the task of transformation and querying of XML, will be presented. Opposed to the navigational nature of XPath based languages, Xcerpt is a pattern based descriptive language.

For proposal of a good visual XML query language it is useful to look at existing textual query languages, to be able to adopt popular aspects or paradigms for querying XML. The comparison and the former section about visual query languages will hopefully be enlightening for preferring the pattern based approach as base of a visual XML query language.

2.2.1 Navigational queries with XPath, XSLT and XQuery

The following query languages, as well as XML, are standards proposed by the W3C[6]. The W3C's is a standardisation consortium. W3C standards are called recommendations. W3C recommendations usually precede concrete products. This process differs from the work of other standardisation organisations (like e.g. the Internet engineering task force – IETF[7]) that usually standardise existing technologies. The W3C consist of members from industry and academia.

[6]This WWW resource is located at `http://www.w3.org/`
[7]This WWW resource is located at `http://www.ietf.org/`

2.2.1.1 W3C's Standardisation Process

The W3C's standardisation process is transparent and can publicly be followed by everyone on the WWW. Only members are allowed to contribute to the standardisation process. A recommendation is the final and stable version of the standard emitted by the W3C[8]. Such a recommendation is likely to be integrated in various related commercial products, because often the leading companies working on concerned topics are members of the W3C and are also the main contributors. The W3C's recommendation process passes through the following stages (as defined in the *W3C Technical Reports and Publications*[9] page)

- **Note :** A Note does not represent commitment by W3C to pursue work related to the Note
- **Working Draft :** These are draft documents and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to use W3C working drafts as reference material or to cite them as other than "work in progress".
- **Candidate Recommendations :** A Candidate Recommendation is work that has received significant review from its immediate technical community. It is an explicit call to those outside of the related Working Groups or the W3C itself for implementation and technical feedback.
- **Proposed Recommendation :** A Proposed Recommendation is work that

  1. represents consensus within the group that produced it and
  2. has been proposed by the Director to the Advisory Committee for review.

- **Recommendation :** A Recommendation is work that represents consensus within W3C and has the Director's stamp of approval. W3C considers that the ideas or technology specified by a Recommendation are appropriate for widespread deployment and promote W3C's mission.

2.2.1.2 XPath

The navigational aspect of XSLT and XQuery is expressed through XPath expressions. XPath has reached the state of a W3C recommendation. XPath should not be seen as a language on its own, but rather as query expression

[8]This WWW resource is located at `http://www.w3.org/`
[9]This WWW resource is located at `http://www.w3.org/TR/`

for hierarchical data, like regular expressions for textual data. Although simple query tools for evaluation of XPath expressions on XML data[10] exist, it is usually considered to be a component of a host language. XPath is part of XSLT and XQuery.

An XPath expression is applied to so called XML contexts and returns a (maybe empty) list of XML contexts. An XML context is a node or location inside an XML document. An XPath is a sequence of selection primitives and conditions, each applied to the selected contexts of the preceding selection primitive. The primitives are instances of the so called *Axis* and are the simplest possible XPath expressions. Those axis provide the ability to select

- named children or any child of a context node
- the parent node of a context node
- the previous/following nodes of a context node in document order [11].
- the previous/following sibling nodes of a context node
- named (or any) children or descendant at any depth below a context node
- a context node itself

If a path primitive (or a whole XPath expression) is applied to a list of contexts, the results of applying the XPath to each context are concatenated. Conditions are either boolean functions operating on contexts or further XPaths. If a condition is an XPath it is interpreted as successful if the XPath's result is not the empty list.

Although XPath is defined to process tree structures, nothing enforces an implementation to work on physical tree structures. There is ongoing and very promising research [olteanu02xpath] about processing XPath expressions on streams of XML events – known as SAX[12] streams. This has the advantage of not having to manage an in-memory representation of the XML document. This is highly relevant for processing very large databases. It is even possible to support approximated queries on infinite streams.

[10] Xmllint's shell mode can be used for this purpose. Xmllint is part of the libxml library, hosted at `http://xmlsoft.org/`

[11] Document order describes the order in which the nodes – or least their closing tags – occur in a textual XML representation. For a tree view of XML data this corresponds to the order of a depth-first left-to-right traversal of the tree

[12] This WWW resource is located at `http://www.saxproject.org/`

```
1 /html/body//a[    contains(./text(),"Xcerpt")
2               and starts-with(./@href,"http:")
3             ]
4
```

Figure 2.25: This XPath expression matches hyperlinks (namely *a* elements) at any depth (by using *"//"*) in the *body* of an *html* document, if they link to absolute URLs (*href* starts with *http:*) and the link text contains the word *Xcerpt*. It is an absolute XPath expression, because it starts with a slash. The XPath expression is written in abbreviated syntax.

2.2.1.3 The transformation language XSLT

XSLT is the first XPath based language proposed by the W3C[13] and has reached the state of a W3C recommendation. XSLT is intended to be used for transformation of XML documents into other XML, HTML or even plain text documents. A template based approach is used. An XSLT stylesheet is applied to a document [14] and returns a new document.

Some major components of the visXcerpt prototype have been implemented using XSLT. For the reader unfamiliar with XSLT and interested in studying the implementation of the prototype, XSLT is explained in more detail here. Nevertheless, XSLT is mostly irrelevant for the design of the visXcerpt language itself.

The whole process of transformation is driven by recursive application of templates. The selection of the context and the restriction to the template to apply to the context selected is expressed in XPath. Every template has an XPath expression that has to match with a context in order to be applied. If many templates match the same context, various precedence rules determine wich template is actually chosen. The template consists mainly of a portion of XML content[15] which may contain hooks for further template application. Those so called `apply-template` elements establish the recursion step. They have an XPath expression for selection of the next recursion step's context. The syntax of the XSLT language itself is XML based.

With the recursive template mechanism, generating new XML content based on existing content is defined. Further aspects of XSLT are:

- **Including data from the input document:** Textual data can be

[13] This WWW resource is located at `http://www.w3.org/`

[14] Although applied to only one document, it is possible to specify the use of further documents either as input through an appropriate XPath-function, or as included templates.

[15] A portion can be plain text, or any number of elements. It is not necessary that there is only one root element, but all elements have to be well formed. It is therefore not possible to open tags in one template application and close them in another one.

```
...
<xsl:template match="a">
  <xsl:copy select=".">
    <apply-templates select="*|text()"/>
    (see <xsl:value-of select="./@href"/>)
  </xsl:copy>
</xsl:template>
...

```

Figure 2.26: This example promotes the *template*, *apply-template*, *value-of* and the *copy* elements of XSLT. A template is presented that extends HTML hyperlinks with a textual representation of the URL they point to. The *a*-element is copied with all its attributes, the content is recursively processed and a *"(see http://...)"* is appended.

included with the `value-of` element. An XPath expression is supplied to select the context that is to be included. If the expression evaluates to more than one context, only the first one is considered. If the context is not a text-node or an attribute, nothing is copied. To copy an element with all its attributes, the `copy` element is used.

- **Variables and parameters:** To be able to transfer data from one template invocation to a subsequent template application, templates can be parametrised. Therefore in a template, named parameters with default values can be defined. A template application can then set those parameters to other values, but it is not forced to. Assignment to parameters is restricted to the template application, and further applications of the same template are not affected by the assignment. It is therefore considered to be a side effect free assignment, comparable to *let* expressions in pure functional programming languages. Variables are similar, but may only be defined and used inside of templates, or globally inside of the stylesheet. Template variables are not accessible outside their containing template; global variables are accessible in the whole stylesheet.

- **Further conditions:** The template application according to an XPath selection and the template choice according to a matching XPath is a powerful conditional. It can be compared to the guard based conditional of modern functional languages like Haskell[16] or Clean[17]. Nevertheless, it is sometimes useful to be able to restrict template application, and therefore have the application hooks surrounded

[16] This WWW resource is located at `http://www.haskell.org/tutorial/`

[17] Haskell's guards are described in the "gentle introduction" in the section about pattern matching semantics at http://www.haskell.org/tutorial/patterns.html#sect4.1

```
1  <xsl:stylesheet version="1.0"
2             xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
3    <xsl:variable name="TITLE">
4      <xsl:value-of select="/html/head/title/*"/>
5    </xsl:variable>
6
7    <xsl:template match="/">
8      <xsl:variable name="COUNT">
9        <xsl:value-of select="count(//h1)"/>
10     </xsl:variable>
11     <h1>Index</h1>
12     <xsl:apply-templates select="//h1">
13       <xsl:with-param name="COUNT">
14         <xsl:value-of select="$COUNT"/>
15       </xsl:with-param>
16     </xsl:apply-templates>
17   </xsl:template>
18
19   <xsl:template match="h1">
20     <xsl:param name="COUNT">???</xsl:param>
21     <xsl:apply-templates select="*"/>
22     ( <xsl:value-of select="position()"/>
23       of
24       <xsl:value-of select="$COUNT"/>
25     )
26     in <xsl:value-of select="$TITLE"/>
27   </xsl:template>
28 </xsl:stylesheet>
29
```

Figure 2.27: Parameters and variables are used to transport values accross contexts. *TITLE* is a global variable. It is invoked in the third template by its name *$TITLE*. The second template passes the parameter *COUNT* to the third template. *COUNT* has the default value *"???"* if the template is called without parameter.

```
1  ...
2  <xsl:template match="h1">
3    <xsl:copy>
4      <xsl:if test="contains(text(),"Xcerpt")">
5        <strong>[RELEVANT] </strong>
6      </xsl:if>
7      <xsl:apply-templates/>
8    </xsl:copy>
9  </xsl:template>
10 ...
11
```

Figure 2.28: This template can be used to prefix an annotation "[**RELEVANT**]" in *h1* headings containing the ord *Xcerpt*.

with conditional constructs. For this purpose the `if` conditional can be used. The test is either a boolean test or an XPath expression that has to be productive to be interpreted as true expression. There is no *else* to choose between two alternatives. For choices there is a `choice` conditional with the same semantic as the widely known `switch/case` conditional in C or Java.

- **Namespaces and XML abstraction:** XSLT is an XML based language, therefore it uses XML elements as syntactical elements. The construction patterns are also in XML syntax. To be able to distinguish between construction patterns and XSLT elements the XML namespace mechanism is used. To be able to use XSLT-elements as construction patterns, there exists a so called `namespace-alias` element that defines a namespace to translate into the XSLT namespace in the output element. This mechanism was widely used for the prototypic implementation of visXcerpt, which is partly implemented using XSLT. When it is inconvenient to use XML patterns, because e.g. an element name is parametrised, or because only attributes are to be created in a template, there exists an abstraction over XML elements. The XSLT elements `attribute`, `element`, `text`, `comment` and `processing-instruction` are defined for this purpose. The content of all of those elements is interpreted as the corresponding child content or value, `attribute` and `element` have an attribute `name` for the corresponding names of the constructed components.

- **Defining functions:** Functions as known in programming languages are very similar to templates. Applying templates is in fact like having a function applied to some XML content from the input document and having the return value transferred to the output document. To be able to call functions from inside other functions, it is possible to give

```
...
<xsl:element name="a">
  <xsl:attribute name="href">
    http://www.xcerpt.org/
  </xsl:attribute>
  <xsl:text>
    The Xcerpt Homepage
  </xsl:text>
</xsl:element>
...
```

Figure 2.29: This is the part of a template. It produces a HTML hyperlink to the Xcerpt homepage by using the XML abstraction of XSLT.

```
...
<xsl:template match="body">
  <xsl:apply-templates/>
  <xsl:call-template name="footer"/>
</xsl:template>

<xsl:template name="footer">
  <hr/>
  (c)2003 - Sacha BERGER.
</xsl:template>
...
```

Figure 2.30: A function named *footer* is provided to insert a copyright note in various situations. The first template calls the function defined in the second template.

templates a `name` attribute and to use the `call-template` element with the name of the template to call. This feature is mostly used to centralise functionality used in different places of the transformation stylesheet.

- **The `for-each` element:** The `for-each` element has a `select` attribute containing an XPath, and a construction pattern as content that is applied to the data matched by the XPath expression. It can be compared to an anonymous function (a lambda function) in functional programming languages. The `for-each` expression is an anonymous template applied to the Xath expression of its `select` attribute.

- **Reordering data:** Using the `sort` element inside an `apply-template` or a `for-each` element it is possible to sort the items matched by the XPath. It is possible to sort lexically, numerically and language dependent. Using the `select` attribute it is possible to specify an

```
...
<xsl:template match="body">
  <body>
    <h1>Index</h1>
    <ol>
      <xsl:for-each select=".//h1">
        <li>
          <xsl:apply-templates select="*"/>
        </li>
      </xsl:for-each>
    </ol>
    <hr/> <!-- CONTENT -->
    <xsl:apply-templates="*"/>
  </body>
</xsl:template>
...

```

Figure 2.31: This template may be used to enrich the body of a HTML document with an index list containing all *h1*-headings found in the original document.

XPath expression relative to each item, based on which the sorting has to be done.

- **Modularising templates :**The `import` element is used to import templates defined in external stylesheet documents. In case of conflict with locally defined templates, the local ones have higher precedence. The `apply-imports` element is used inside templates as application hook restricted to imported templates. It has no XPath expression and therefore is only applicable to the actual context.

XSLT is a widely accepted W3C recommendation with several implementations available. It has reached the W3C status of a recommendation which means that the standard is considered to be stable. The expressiveness of XSLT is comparable to the expressiveness of a general programming language, because it is Turing complete[18]. Nevertheless, it is not convenient to use XSLT for general programming tasks, because of the rigid information flow from input document to output document. The Turing completeness of XSLT ensures that all transformation requirements can be achieved with XSLT, but the drawback is that there is no termination guarantee for the transformation process.

[18]The Turing machine markup language TMML (found at http://www.unidex.com/turing) can be processed with XSLT, which demonstrates the turing completness of XSLT. A further proof for Turing completeness can be found in [kepser02proof]

```
1 for $b in document("http://www.bn.com/bib.xml")//book
2   let $e := $b/author[contains(string(.), "Suciu")]
3   where exists($e)
4   return
5     <book>
6       { $b/title }
7       { $e }
8     </book>
9
```

Figure 2.32: This example (found in [w3c02use]) uses a complete FLWR expression. It is used to return *book*-elements with title and author, where the name of the author contains *Suciu*.

2.2.1.4 The XML query language XQuery

Like XSLT and XPath, XQuery [w3c02xquery-wd] is a language definition maintained by the W3C[19]. It has not yet achieved the state of a recommendation, it is filed as a working draft. This indicates that XQuery is work in progress.

From a sloppy point of view, XQuery looks like a mix of XPath and SQL [fatdog01quick]. From an academic point of view *"XQuery is a typed, functional language for querying XML"* [wadler02xquery]. Opposed to XSLT, XQuery does not use an XML syntax exclusively, although construction patterns can be written as XML portions. As proven in [kepser02proof] XQuery is Turing complete.

2.2.1.4.1 XPath and XQuery

In the most simple form, XPath expressions are already XQuery programs. The XPath standard that is included in XQuery is XPath2, which is also a working draft. While comparing the XPath2 working draft [w3c02xpath-wd] with the XQuery working draft [w3c02xquery-wd] it is apparent that both standards are strongly related. As described in [wadler02xquery], it is possible to express all XQuery programs in a reduced language called XQuery core. XPath expressions are not part of this reduced language and can be rewritten using the core language.

2.2.1.4.2 FLWR Expressions

Beside XPath, the most important syntactical construct is the so called FLWR[20] expression. FLWR stands for **for**, **let**, **where** and **return**. The meaning of those four parts will be roughly explained now:

[19] This WWW resource is located at `http://www.w3.org/`

[20] FLWR is pronounced like *flower*

- **for :** This part is written like `for $`*var* `in` *queryExpr FLWRExpr* . This will iterate over all elements of *queryExpr* and bind them to the variable `$`*var*[21]. For every binding of `$`*var FLWRExpr* is called. `$`*var* can be used in *FLWRExpr*. Both *queryExpr* and *FLWRExpr* can be FLWR expressions. Variables may also be used in XPath expressions. The use of variables is one of the main differences between XPath1 and XPath2. It is also possible to write `for $`*var1* `in` *queryExpr1* `, $`*var2* `in` *queryExpr2 FLWRExpr*, expressing to iterate over the elements of *queryExpr2* inside of the iteration over the elements of *queryExpr2*.

- **let :** `let $`*var* `:=` *FLWRExpression* is used to assign content to `$`*var*. The content can be empty or contain any number of elements, it may even be a boolean value. As usual in functional languages, there can not be more than one assignment to the same variable in the same scope.

- **where :** A `where` construct is a boolean condition that must be fulfilled in the scope of the `where` condition. It is comparable to the `where` of SQL.

- **return :** `return` indicates which content has to be returned to the output. It may contain XML construction patterns, variables and further FLWR expressions. If not explicitly provided, it implicitly surrounds the whole XQuery expression.

For valid FLWR expressions the four parts are not always necessary. The examples present some of those cases. They were extracted from the comprehensive and very educative collection [w3c02use] of XQuery examples and use cases maintained by the W3C[22].

2.2.1.4.3 Optional Type System

XQuery incorporates a type system based on XML Schema[23]. The type system does not have to be implemented in a valid XQuery implementation. Nevertheless, it is likely that an implementation using the type system can incorporate much more optimisations than an untyped implementation.

XQuery provides a textual syntax that in a straight forward manner represents the XML Schema language. It is also possible to include schemas defined in external resources. The following features can be used in a typed XQuery engine:

[21] Variables always start with a $-Sign in XQuery
[22] This WWW resource is located at `http://www.w3.org/`
[23] This WWW resource is located at `http://www.w3.org/XML/Schema`

```
1  define function toc($e as element )
2    as element*
3  {
4    let $n := local-name( $e )
5    return
6      if ($n = "section")
7      then <section>
8               { toc($e/*) }
9             </section>
10     else if ($n = "title")
11     then $e
12     else ()
13 }
14
15 <toc>
16   {
17     toc( document("book.xml")/book )
18   }
19 </toc>
20
```

Figure 2.33: This generates a table of contents, listing all the *section* elements and their *title* elements. The function is typed to require an element and return an arbitrary number of elements. Line 17 calls the function and embeds the result in a *toc* element.

- **validate:** with `validate { expr }` an expression *expr* can be validated against a schema;

- **assert as:**`assert` checks the type on an expression statically (e.g. to compile time), hence the checked expression cannot be variable;

- **treat as:** this checks the type dynamically (e.g. to runtime). The expression may be variable. It is comparable to casting in C++ and Java.

2.2.1.4.4 Functions

Another major feature of XQuery is functions. A lot of functions are already predefined in the standard, like `unordered`, `distinct`, `empty` and `not`. New functions can be defined with `define function` *functionName* `(`*argumentType1* `$arg1, ..., `*argumentTypeN* `$argN) returns` *resultType* `{...}`. It is possible to call functions recursively.

2.2.1.4.5 Remarks on XQuery

XQuery uses a textual non-XML syntax. Nevertheless a XML representation is available. This syntax can be used for implementations without XQuery parsers, or for processing with further XML aware tools. The rationale for the textual syntax is the ease of readability for humans as compared with the XML syntax.

The previously mentioned XQuery core language is a subset of full blown XQuery. All XQuery programs can be rewritten as expressions in XQuery core. XQuery core differs from XQuery by following restrictions:

- only one variable may occur per `for`-clause,
- no `where`-clauses are allowed,
- only simple path expressions of the form `iteratorVariable/Axis::NodeTest` are allowed,
- only simple element and attribute constructors are allowed. This means that no compositions of CDATAs, elements and attributes are allowed,
- `sortBy` is allowed,
- `if/then/else` is allowed,
- function calls to the predefined functions are allowed (but no new function definitions).

The rationale behind a core language, is the availability of a simple implementation base.

2.2.2 Xcerpt: Pattern- and Logic-based Queries and Transformations

Xerpt is a query and transformation language actively developed at the Institute for Computer Science of the University of Munich[24]. The main concept behind Xcerpt is pattern matching based on a non-standard unification. The roots for this paradigm can be found in logic programming languages like Prolog. Logic programming languages are descriptive programming languages, in the sense that no explicit control flow is given. Instead the problem or the desired result is described, the necessary computational operations are automatically derived. Usually specifying the problem consists of specifying patterns that describe the input and result structure.

[24]This WWW resource is located at `http://www.informatik.uni-muenchen.de/`

Input and result are bound together through the use of variables, sharing their information through multiple occurrence. They represent a bondadge to equal values at all their occurrences.

The evaluation of logic programs is mainly based on equality of terms. This ranges from simple analysis of equality of two variable free terms to the verification and/or assignment to variables in terms. This procedure is called unification and is usually applied to two terms, either failing if there is no equality between the terms or succeeding with the most general occupations of the involved variables.

In classical logic programming, patterns are expressed through terms. In Xcerpt, XML elements are used as patterns. Although documents in the physical XML data model constitute trees[25], various reference mechanisms as ID/ID-REF[26] pairs, XLinks[27] or RDF[28] reveal a graph structure in XML data. Hence equality related processes need to be applied on graphs rather than trees in Xcerpt. For this purpose a special unification called *simulation unification* [bry02towards] is used.

The detailed semantics of Xcerpt are explained in [bry02the] and also in [bry02gentle]. In this thesis a visual XML query and transformation language based on Xcerpt is presented. For this purpose almost only syntactic properties of Xcerpt are relevant. Therefore Xcerpt's semantics will not be explained in a formal way, but only as far as necessary to understand the syntax in conjunction with some examples. VisXcerpt's semantics does not differ of that of Xerpt and will therefore not be formally introduced.

2.2.2.1 Xcerpt's syntax

Xcerpt, like XQuery, has two different syntaxes: one XML based and another textual syntax. Both can be used to represent query programs, input and output documents. In fact the pattern description parts of the non XML syntax are derived from the semi-structured formalism and have more expressiveness than the XML core standard, as it supports the specification of ordered and unordered content[29] (see below). Because the non-XML syntax is easier to read, it will be used in the examples in this section. For the implementation of visXcerpt it turned out more appropriate to work based on the XML syntax. This is a hint that the use of different syntaxes for one language as in XQuery and Xcerpt is a useful innovation.

Xcerpt's data notation are terms representing XML data. They are called database terms. An XML element is represented by its name followed

[25] terms and trees are very similar – terms can be seen as serialisation of trees

[26] The ID/ID-REF mechanism is part of the core XML definition.

[27] This WWW resource is located at `http://www.w3.org/TR/xlink/`

[28] This WWW resource is located at `http://www.w3.org/TR/REC-rdf-syntax/`

[29] XML only supports ordered content. This is useful for document oriented data. Database applications usually do not imply any order on its content

by braces with the content, separated by commas. Text nodes are quoted in double quotes. There are four different kinds of braces : square brackets `[]`, double square brackets `[[ ]]`, curly braces {} and double curly braces {{ }}. To represent XML data the double square brackets are used. XML was at first tailored for the document management community. In documents, the order of elements is usually relevant. The XML formalism is useful as database formalism as well, but databases usually do not impose an order on its element. XML does not provide a mechanism to represent unordered information. In Xcerpt the different bracket types are used therefore: curly braces represent unordered data and square brackets represent ordered information.

e difference between single and double brackets is mainly relevant for query patterns and will therefore be described in the section about Patterns.

2.2.2.1.1 Patterns

The base of Xcerpt's syntax are XML patterns. They are used for querying as well as for constructing results. Patterns are instances of XML elements. A variable free pattern used for construction would simply result in exactly this pattern to be constructed. Used as query pattern, it would match all it's occurrences in the document or database queried.

For querying, it is highly relevant to be able to query elements containing some content specified, among non specified content. A query pattern is therefore needed to be a partial pattern of the database items it matches. To express the difference between partial and total match patterns, the single and double braces are used: single braces are used for partial match patterns, double braces for total match. For construction only double braces are allowed, because partial match is not meaningful for construction terms.

2.2.2.1.2 Variables

Variable expressions can either be variables or so called "`as`" constructs. Variables are special nodes in a pattern without content (or leaves, if the pattern is seen as a tree). The "`as`" construct is used to restrict a variable by a pattern. Using the "`as`" construct it is possible to specify the pattern that is searched for while being able to retrieve instances of it for construction. The "`as`" construct is therefore only meaningful in query patterns and not allowed in construction patterns. While in the non-XML notation "`as`" constructs are names attached by a curly arrow to a pattern, in the XML notation it is an element containing its pattern.

It is planned to allow variables in element names, but the relevant syntax has not yet been established.

The next question is how to distinguish construction and query patterns and how to associate them. This is covered by the definition of rules and

```
greengrocer {
  products {
    product {
      type { "vegetable" },
      name { "cabbage" },
      price {
        unit { "piece" },
        value { "0.59" }
      },
      vendor { "DeRuiter" }
    },
    product {
      type { "fruit" },
      name { "cherry" },
      price {
        unit { "kilo" },
        value { "2.19" }
      }
    },
    vendor { "Lafayette" }
  },
  vendors {
    vendor {
      country { "holland" },
      name { "DeRuiter" }
    },
    vendor {
      country { "france" },
      name { "Lafayette" }
    }
  }
}

```

Figure 2.34: This database term models a database comparable to the greengrocer's database in the visXcerpt example section presented later. There is no order in the elements, all elements are total (databases never use partial terms)

```
greengrocer {{
  products {{
    product {{ }}
  }}
}}

```

Figure 2.35: This pattern matches with all *product* elements of the former database. The pattern uses partial, unordered matching.

```
1 greengrocer {{
2   products {{
3     var PRODUCT -> product {{
4       name { var NAME }
5     }}
6   }}
7 }}
8
```

Figure 2.36: Variables are used in this example. *NAME* is a variable retrieving the content of the *name* element, *PRODUCT* is in an "as"-construct retrieving all *product* elements.

programs.

2.2.2.1.3 Rules

A rule consists of one construction pattern and one or many *and*-connected or *or*-connected queries. It is written like a pattern with a root element called `rule`. The construction pattern is contained in an element called `cons`, the queries in elements called `query`. To avoid name conflicts with patterns containing elements called `rule`, the XML notation of Xcerpt makes use of namespaces. Namespaces is an ongoing research aspect for the non-XML Xcerpt notation. A query consists of a query pattern and optionally a resource locator. The optional resource locator is the first sub term and is called `in`. Variables are allowed in `in`-terms, allowing variable resource locators. The resource locator indicates the external data source that is to be queried. If no resource locator is given, the actual program is queried. What this means, is explained later in the section about programs.

In the tradition of logic programming languages, the construction part is called head and all query parts of a rule together are called body. It is considered as good programming style to first write the construction part and then the queries. The queries of a rule can be considered to be implicitly *and*-connected. The exact meaning is explained in Xcerpt's semantics specification.

2.2.2.1.4 Programs

Programs are a set of rules. Using the XML notation, the rules are rooted by a `program` element. For the non-XML notation the document containing the rules implicitly is the program. A program is executed by querying it with a pattern. The most simple useful query that a program can be asked is an empty variable. If a program consists of only one rule, all alternative constructions that this rule can produce are collected by this variable.

```
rule {
  cons {
    product-list [
      all var PRODUCT
    ],
  },
  query {
    in [ "file:database/greengrocer.xml" ],
    greengrocer {{
      products {{
        var PRODUCT
      }}
    }}
  }
}

```

Figure 2.37: This example presents a rule querying all *product* elements in a database that is read from a file. All *product* elements retrieved are aggregated in a *product-list* element.

If rules without query patterns involving resource locators are contained in a program, the actual program is queried. This technique is called chaining: a rule querying the current program, queries all rules that are productive, hence query patterns match with productive construction patterns. Chaining is fundamental for code modularisation, recursion and therefore Turing completeness.

2.2.2.1.5 Goals

If a single rule is applied to a database, the result is usually clear, because the rule is the only entity determining the result. For a program, every rule may potentially contribute to the result of a program evaluation. To be able to constrain the result to a desired subset of the elements matched by all program rules, a goal must be used. A goal is simply a query pattern. It can be seen as chained to some rules of the evaluated program. By now the goal is an interactive feature of Xcerpt, specified at evaluation time to activate the processor.

2.2.2.1.6 The Descendant construct

Sometimes it is necessary to specify a pattern that may occur at any depth level in the queried data. For this purpose the `descendant` element is used. It contains a term that is not necessarily the direct child term of the parent of the `descendant` element. A `descendant` element must always contain exactly one child term. The `descendant` element is only valid in query

```
rule {
  cons {
    product-list [
      all var PRODUCT
    ],
  },
  query {
    in [ "file:database/greengrocer.xml" ],
    desc var PRODUCT -> product
  }
}

```

Figure 2.38: Using the descendant construct, the previous example can be expressed shorter. *PRODUCT* is in an "as"-construct.

terms.

2.2.2.1.7 Disjunctions and Conjunctions

If a rule's body contain more than one query the queries need to be connected in a conjunction or a disjunction. The `or` and the `and` construct can be used. Those constructs are n-ary relations, connecting all their members. Construction patterns may not be located in `or`/`and` constructs.

2.2.2.1.8 Regular expressions

It is possible to use regular expressions to restrict textual matching. Regular expressions are written between slashes. By the time of writing they were only allowed at the level of textual content and attribute values, but it is planned that they are also usable for element names.

2.3 Further Visual Languages and Query Systems for XML

In this section related work about visually querying and transforming XML is presented. The different approaches reach from visual query and transformation languages to systems used to browse or query data in an interactive fashion.

```
rule {
  cons {
    product-list [
      all var PRODUCT
    ],
  },
  query {
    in [ "file:database/greengrocer.xml" ],
    desc var PRODUCTS -> product {{
      or {
        vendor { var DUTCHVENDOR },
        vendor { /Van.*/ }
      },
    greengrocer {{ vendors {{ vendor {{
      country { /[hH]olland/ },
      name { var DUTCHVENDOR }
    }} }} }}
  }
}

```

Figure 2.39: This example uses disjunction and implicit conjunction. Implicit conjunction is expressed over two patterns, one matching *product* elements, the other matching vendor names. The variable *DUTCHVENDOR* is used for an equijoin in the conjuncts. A disjunction is used to either match products with names of dutch vendors or with names starting with "Van" (expressed by a regular expression). A regular expression is also used for matching "Holland" case-insensitive.

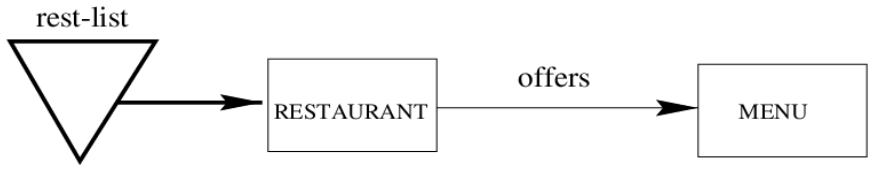

Figure 2.40: In this WG-Log query a *rest-list* element is constructed with all *Restaurant* elements offering menus as query pattern. The triangle aggregates all matched elements it points to. The *offers* relation is specified in a schema. Schema information could also be expressed in WG-Log.

2.3.1 XML-GL and WG-LOG

Two articles present the visual query languages XML-GL and WG-LOG: "Graphical Query Languages for Semi-Structured Information" [comai00graphical] by Sara Comai compares both and "XML-GL: A Graphical Language for Querying and Restructuring XML Documents" [ceri99xmlgl] presents only XML-GL.

2.3.1.1 About Sara Comai - "Graphical Query Languages for Semi-Structured Information"

Sara Comai's PhD Thesis [comai00graphical] presents two graphical query languages called XML-GL and WG-LOG. XML-GL presented here is not equal to the language XML-GL presented in [ceri99xmlgl], but it is likely to be an evolutionary step due to the continuing research on this language. Furthermore, this article is condensed and thus covers less details than the older paper [ceri99xmlgl]. Both languages are based on graphs.

WG-Log is based on G-Log, a database language intended to query complex objects through the use of graphs and pattern matching. The patterns are explicitly based on schemas. A query consists of one or many graphs, each representing a rule. Opposed to other approaches like Xing found in [erwig00visual] and visXcerpt, the query and the construction part are integrated in one structure (a graph), while Xing and visXcerpt separate construction and query parts. The querying graph structure is spanned through thin or red lines, while the construction structure is spanned by green or thick lines. This indicates very clearly the relation of how the information flows between query and construction part – they share the same nodes, making variables obsolete. A possible drawback is that complicated graphs could tend to be cluttered with many edges, but this has not been further investigated.

Due to schema use, the query patterns can be smaller than their hypothetical untyped equivalent. On the other hand, WG-Log is only applicable to schema based data, while semi structured data encourages the use of

schema free data. This domain is covered by XML-GL.

XML-GL uses pairs of graphs arranged horizontally and separated by a line to represent rules. Graphs consist of boxes with names inside, representing element names. Complex programs may consist of various rules. The graphs are used as query and construction patterns. Because XML data is tree structured, simple queries can be expressed with trees as query patterns. Graphs are used to model joins. Information is carried from the query side to the construction side by tag name match. This means that an element with a certain name in the query pattern is carried to the corresponding position in the construction pattern, where the same name occurs. Various constructs exist to use unnamed data and to overcome ambiguity. If e.g. a name occurs more than ones in a query graph, a line connecting the relevant query and construction node is used.

XML-GL is explained in more detail in [ceri99xmlgl].

2.3.1.2 About Stefano Ceri et.al - "XML-GL: A Graphical Language for Querying and Restructuring XML Documents"

In [ceri99xmlgl], XML-GL is presented as visual query and transformation language for XML. The use of XML-GL does not rely on the availabilty of schema information. If some kind of schema is available, editing and creating queries becomes easier, but the evaluation model of the language does not rely on a schema.

It is on the other hand possible to express schema information with XML-GL, with more expressive power than the DTD formalism provides, but no primitive type system for the textual information, as in XML-Schema, is available.

An XML-GL program is a set of pairs (at least one pair) of XML-GL graphs arranged horizontally, the left representing the query pattern and the right for the construction pattern. An XML-GL schema is an XML-GL graph enriched with extra constructs for conditions that are not mentioned in the context of query programs.

A graph consists of labelled boxes representing elements. Their hierarchical structure is modelled by directed edges. Patterns described this way can be used for querying as well as for schema implementation – a data instance is valid, if it matches with a pattern. To denote the relevance of order for child elements, the first edge is crossed by a short stroke. If the structure is an acyclic graph instead of a tree and it is used as query pattern, it can be interpreted as using a join: the "child node" with more than one ancestor indicate that the ancestors must share identical child nodes. Hence the join is expressed over the shared child node. This is also the case, if two trees

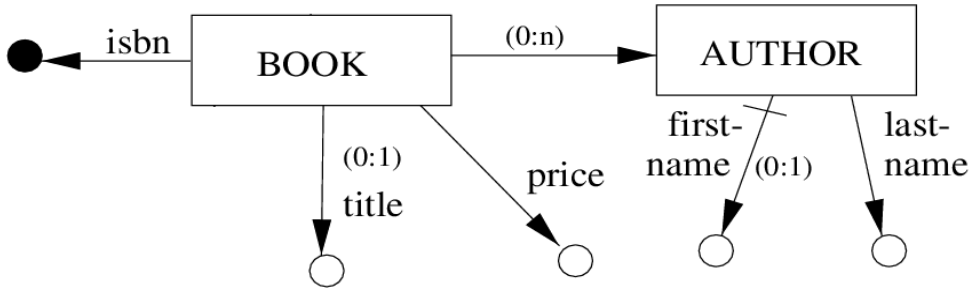

Figure 2.41: This XML-GL expression represents the DTD in figure 2.42.

```
1 <!ELEMENT BOOK       (title?,price,AUTHOR*)>
2 <!ATTLIST BOOK
3               isbn  CDATA  #REQUIRED>
4 <!ELEMENT title      PCDATA>
5 <!ELEMENT price      PCDATA>
6 <!ELEMENT AUTHOR     (first-name,last-name)>
7 <!ELEMENT first-name PCDATA>
8 <!ELEMENT last-name  PCDATA>
9
```

Figure 2.42: This DTD expresses the same as the XML-GL expression in figure 2.41, except that the content of *BOOK* is unordered in the XML-GL expression. This is not expressible in DTD syntax, because arbitrary *AUTHOR* elements can occur, opposed to a fixed number of *title* and *price* elements.

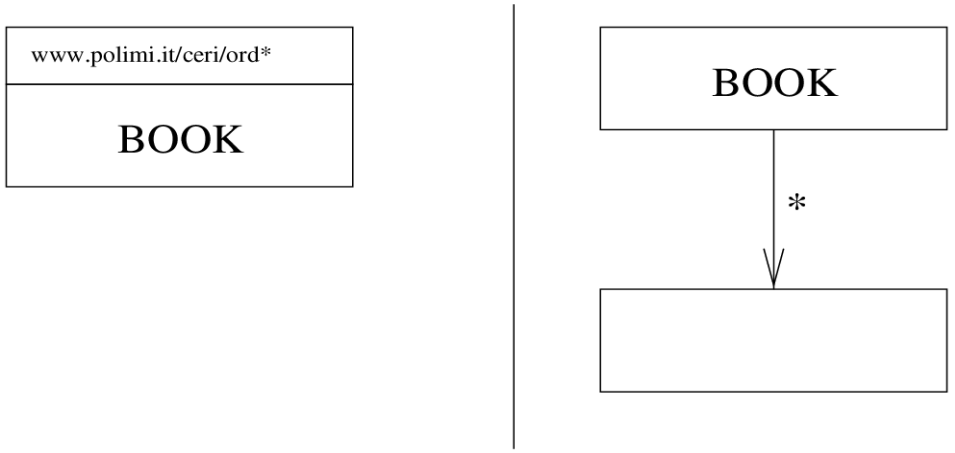

Figure 2.43: This simple XML-GL query grabs all *BOOK* elements found in resources in www.polimi.it/ceri/ starting with *ord*. The construction pattern on the right expresses to return the matched *BOOK* elements with all sub-elements at all depth. Without the asterisk, only the direct child elements are queried.

share nodes. In a schema graph the edges are labelled with multiplicity information like edges of ER-Diagrams. Another special visualisation is used for nodes containing only textual content – they are drawn as hollow circles. Filled circles depict attributes. The edges to those circles are labelled with the element name or the attribute name.

A feature used solely in schemas is an *xor* arch, crossing two edges. To model mixed content of plain text and elements, the elements and a text node circle are *xor*-ed. The edge to the textnode circle is labelled with "content" to indicate that it is not a text-only element. Therefore in this circumstance it is not possible to construct a text-only element called "content", a namespace mechanism that is actually missing in XML-GL, would be needed.

The base idea for the use of XML-GL graphs for querying has already been explained in the section about [comai00graphical]. Further constructs are explained now. There are three kinds of construction primitives for aggregation purpose:

- **Plain boxes:** They are used to construct elements. The elements have the same name as the box. For each element the query pattern has matched, an element is constructed by the box the query pattern is attached to.

- **Triangles:** They simply list all the matched elements from the query part they are attached to explicitly or by name.

- **List icons:** They react as a kind of combination of both, plain boxes

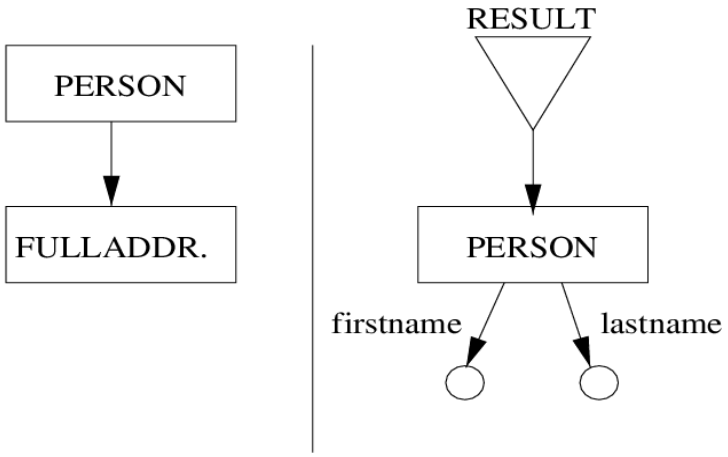

Figure 2.44: This XML-GL program aggregates all matched *PERSON* elements (those having a *FULLADDR* element) below a constructed element called *RESULT*. Only *firstname* and *lastname* children are projected to the result.

and triangles, by aggregating the matched elements according to an additional grouping condition explicitly related to by an edge.

Further constructs are available for nesting and unnesting of content, which are powerful tools to prevent recursive queries that are not expressible in XML-GL. Nevertheless it does not seem to be possible to express queries about the structure of the queried data with the constructs presented, which is usually a simple task with recursion.

2.3.2 GraphLog

Graphlog as presented in [consens89expressing] has a lot in common with WG-LOG mentioned earlier. WG-LOG is based on GraphLog. GraphLog is visual language inspired by DataLog which are variable free Prolog programs. GraphLog is intended to query and transform graph structures, having hyperdocuments in mind – the WWW was not established at the time of writing that article, but the concept of hyperdocuments was already known. In GraphLog it is possible to express patterns based on named directed edges between nodes. As in WG-LOG there is no structural separation of construction patterns and query patterns – queried and derived arrows are distinguished through their line width: thin edges depict query edges while thick edges depict so called *virtual edges*. The nodes between the edges are unnamed circles. Attribute values can be shown as rectangles with their content inside (the arrow pointing to an attribute is labelled with the name of the attribute). A doubly circled node depicts an all-quantified node.

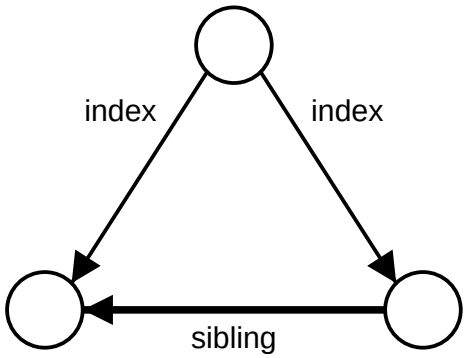

Figure 2.45: The GraphLog expression presented here introduces a new *sibling* hyperlink between two documents that share the same *index* document (also expressed through a link).

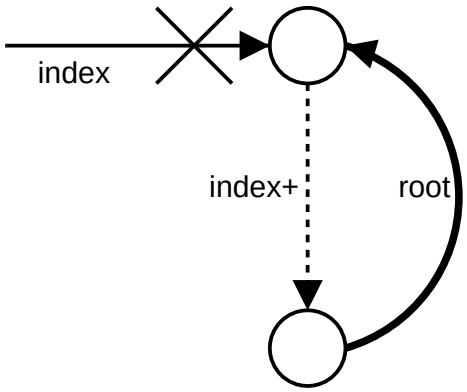

Figure 2.46: In this GraphLog expression a regular expression represents a path of one or more *index* links. A document gets a *root* link to another document if that document has no *index* link.

A third edge type is introduced in GraphLog – a dashed edge. It is used to express a regular path expression. Such an edge may therefore substitute arbitrary other edges. An edge may also be crossed-out to indicate a negation. No such edge is allowed in the physical or virtual graph.

2.3.3 VXT

VXT is a visual language integrated into an interactive runtime and editing environment. Its main purpose is transformation rather than querying. It has been presented in the two publications [pietriga01xvt2] and [pietriga01xvt].

2.3.3.1 VXT: Visual XML Transformer

The article [pietriga01xvt2] is a short summary of VXT. VXT is a pattern based language. Because the emphasis is on transformation rather than on querying, some commonly used query language features, like aggregation, are not covered.

The roots of XVT can be found in XSLT. Several rules provide a template mechanism. A rule consists of a query and a construction part. Information matched on the query side is collected through links with slots on the construction side. The links can be typed as *copy-*, *textual extraction-* or *template application*-relation.

The XML visualisation is a so called visual treemap representation. Elements are displayed as horizontally aligned nested boxes. Above the box a label indicates the element name. Text nodes are represented as diamond shaped boxes, attributes as triangles.

2.3.3.2 VXT: A Visual Approach to XML Transformations

The article [pietriga01xvt] presents VXT in more detail, compared with [pietriga01xvt2]. The additional aspects covered here are related to document type definition, further language constructs and the translation of VXT programs into XSLT.

A very intersting aspect mentioned is, that very little extension is nessecary to use this visualisation mechanism as document type definition formalism. The proposed extensions are:

- **conditions or choices:** choices are modelled by stacking the different choices vertically.
- **zero or one occurence:** Those elements are surrounded by a dashed box.
- **zero or many occurences:** Like *zero or one occurence* those elements are surrounded by a dashed rectangle, but a second empty dashed rectangle is attached to its right side by a short dashed line.
- **one or many occurences:** in this case the same notation as *zero or many occurences* is used, except that the left dashed box is drawn with a solid line.

The schema definition presented this way can be used as translucent overlay or template mechanism inside the editing window to support easier transformation writing.

2.3.4 BBQ

As presented in [munroe00bbq], BBQ is an interactive visual browsing and querying environment. Its querying functionality is back-ended by the textual query language XMAS [papakonstantinou99framework]. Querying in BBQ is schema-driven using XML DTDs, but at the same time the user is encouraged to browse the schema and data. A BBQ session consists of query cycles. A cycle is made up of query specification and query execution. The user may refine the queries until the desired result is achieved. To make this approach feasible, a lazy approach to the implementation of the query engine and browser is suggested.

Visualisation of queries and results is based on standard tree visualisations, as found in all modern visual user interface tool kits. Various icons are used to depict the different node types of schemas and results. The Java toolkit Swing is used for an implementation explained in the article. The editing of queries is based on cut/copy and paste as well as drag and drop.

A query in BBQ consists of a query and a construction part. The query part is a tree view of the DTD from the database to be queried. Every node can be refined through use of textual constraints. Joins are built through use of drag and drop. This way only equijoins can be expressed. Joins are indicated by textual annotation on the node's visual representation. Constructors are located in a second tree view. Nodes in constructors are either created as new nodes or are dragged from the query tree view. Dragging a query node to a construction tree indicates that the nodes matching the query node at execution time are included in the corresponding location of the construction tree. The naming of a dropped query node in a construction tree is based on a path notation of the query node and indicates its path to the root of the schema.

The result of executing the query is presented in a tree for browsing. The textual representation of the corresponding XMAS query is also presented. A back and forward button, as known from all modern web browsers, can be used to access all query cycles of a session.

2.3.5 QURSED

The QURSED project [petropoulos02qursed] provides a tool for authoring web-based query forms and reports. To create the query forms, the developer may use any web page editor. These forms are back-ended by a query language called TQL (tree query language). An editor for tying query forms to TQL programs (the QURSED editor) is presented. At this level the developer has to cope with the tag structure of the query forms, to provide bindings to constructs of the TQL program. By populating the form, the user ties concrete values to variables in the TQL program and the query can be executed.

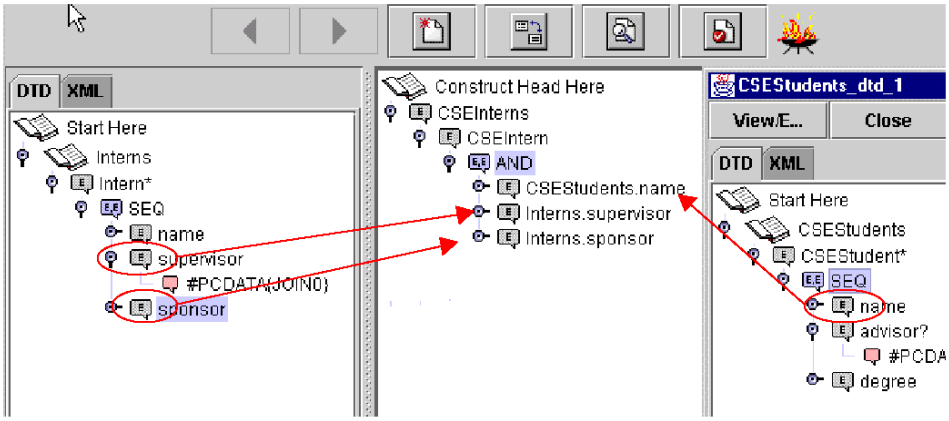

Figure 2.47: This BBQ screenshot presents the process of editing a BBQ query. Two DTD visualisations are used (on the left and on the right) to drag sub-query-patterns into the construction pattern (in the middle).

A TQL program is a tree consisting of constant nodes (they are labels that match elements in the queried XML tree), node name variables, literal variables (at leaf level), *AND* and *OR* constructs providing the ability to construct conditions based on comparison of variables to literals or to other variables. In the end a TQL program is translated into a XQuery program producing the result.

The TQL program editor is a classical tree view. The tree view used in this project is Swings JTree found in Java with nodes representing the language components. The editor model is the default behaviour provided by the windowing toolkit used to implement the TQL program editor. Like Xcerpt TQL is a pattern based language. Compared to Xcerpt, TQL is much simpler to implement, but the language is also much more restricted. Programs written in TQL tend to be longer and less readable then Xcerpt programs.

2.3.6 Xing

Xing is a form-based interface to express XML queries and transformations by so-called "document patterns", presented in the articles [erwig00visual] and [erwig-xml2000]. The form based character is motivated by the fact that forms often represent nested boxes of sections, questions and answer fields. The main idea is to let the user draw samples, of the data he expects from a query to be returned. Optionally a construction pattern of the structure for the resulting document may be provided. These two document patterns together are called a document rule. To transport information from the querying pattern to the construction pattern, variables (so called *aliases*) are used.

The definition of this system was based the on following design goals:

- do not define a(nother) textual language,
- avoid XML syntax,
- use a simple and intuitive visualisation of XML,
- employ pattern matching,
- facilitate restructuring,
- keep the system simple and intuitive.

Based on these constraints, it was nescessary to define a visualisation of XML. This is called the *document metaphor* which is strongly influenced by documents like product descriptions and forms. XML documents are visualized as follows:

- XML elements are represented by boxes (with rounded edges),
- tag names are written in bold font above the corresponding element on the left,
- elements containing only plain text are shown as the (bold) tagname followed by the text on the same line and both separated by a colon,
- attributes are displayed like the former abbreviated elements, except that the attribute name is not bold faced,
- elements containing text mixed with other elements (like HTML/XHTML documents) are not supported,
- XML name spaces are neither supported.

Document patterns are built as instances of the document metaphor. Further elements are:

- **Regular expressions:** regular expressions (and therefore also simple wildcards) may be used instead of tag names.
- **Aliases:** matched patterns may be enclosed in curly brackets prefixed by a name. It is possible to refer to this name in the result pattern.
- **Or-patterns:** *or* patterns are shown like or-statements in regular expressions - the components are arranged horizontally and joined by a vertical bar.
- **Deep queries:** Deep queries are able to match at arbitrary depth in the document tree below their occurence. They are depicted as an ellipsis prefixed to the pattern that is used for the deep query.

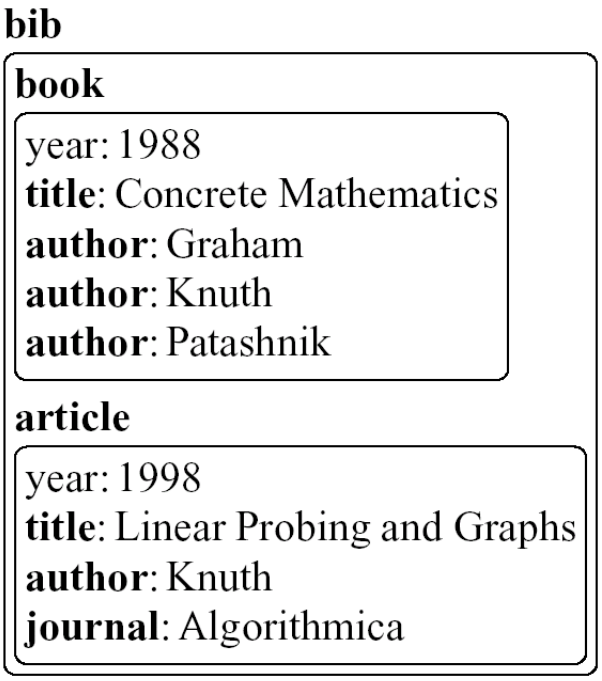

Figure 2.48: This illustration shows an example of Xing's document metaphor. The bold labels correspond to tag labels, the thin label is an attribute.

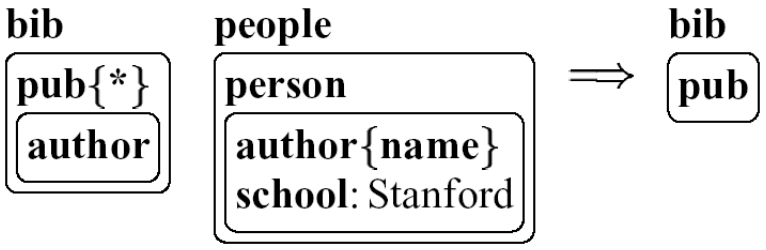

Figure 2.49: This Xing query rule generates a *bib* element containing all publications (articles, books,...) with authors from *Stanford*. The variable *author* is used to join based on names in query patterns. The variable *pub* carries the matched elements into the construction pattern on the right side.

Chapter 3

visXcerpt

3.1 Aspects of visXcerpt

After having provided a summary about querying XML data and various visual languages and systems, a new visual query system is proposed here. The semantics of the proposed system will be based on Xcerpt's semantics. Studying the related work shows that a lot of visual query systems are based on pattern matching. Pattern matching and unification in turn is a paradigm best explored in the logic programming environment. Nevertheless the most important query systems for XML (XSLT and XQuery) do not follow the logic programming paradigm. Further in-depth analysis of logical XML query systems would be interesting.

In the past visual query systems like QBE [zloof77query] have always been presented by their syntax as well as their semantic. However, QBE was proposed at a time before logic-based query systems were well analysed. Today the situation is different: A lot of work has been done in the analysis of underlying theories of logical query and programming languages. Xcerpt uses advanced techniques like constraint solving based on simulation unification, which provide a high potential of optimiseability like magic set strategy, tabelling and memorisation. For a visual XML query system based on pattern matching it therefore seems sensible to rely on the semantics of a well designed textual query language with the desired features, in this case Xcerpt. Thus the semantic of visXcerpt is equal to Xcerpt's semantic.

The remaining task to solve is to visually support Xcerpt. This will be accomplished by providing a new visual syntax. The guidelines therefore are (they will be explained in detail afterwards):

- support the comprehension of pattern structures, queried data structures and generated results. This implies
  - visually reflecting the generic nature of XML;

  - visually distinguishing Xcerpt related visual components from XML related ones;

- supporting the process of editing query programs. Thus
  - well known editing primitives should be used;
  - editing steps should (as far as possible) lead from one valid query program to another, not confronting the user with syntactically incorrect programs;
  - the user should be able to learn the language by exploration of the visual editors user interface, rather than being forced to skim a language reference manual;

- providing support to focus or to hide information without modifying the program.

The visXcerpt proposal is divided in two parts: The static aspects and the dynamic ones. The static aspects will cover the visual representation of Xcerpt programs as well as the visualisation of XML in general. The dynamic aspect of visXcerpt is the editor model through which the editing process is defined.

3.1.1 Static Aspects of visXcerpt

This section will describe all editing independent aspects of visXcerpt. They represent a visualisation of Xcerpt. Static does not mean that there is no user interaction covered by these aspects. Information hiding for example is a static aspect where user interaction may be used to hide or display parts of a document. It is categorised as static, because no information is altered that way. The static aspects are divided in two sections, one covering generic XML visualisation, another for visualising Xcerpt specific constructs. The focus on a generic XML visualisation is done, because the pattern mechanism of Xcerpt is in fact nothing else than providing sample XML documents or sample document fragments. Hence it is quite natural for patterns to be represented by something being usable for the representation of XML data. The same generic visualisation may be used to represent the query result, or it may be used to show XML documents.

3.1.1.1 visXcerpt's Generic XML Visualisation

Database terms are represented in visXcerpt through nested rectangular boxes with attached "tabs". This reminds of tabs used in graphical user interface toolkits, which supports the awareness of an underlying hierarchical structure. The child-elements visualisations are nested inside of the

box. Because visXcerpt is primarily intended to be used with XML, there is special handling of textual content (so called CDATA), attributes and namespaces[1]: Attributes are placed in a two-column table with the name in the left column and the value in the right one. The attribute table is the first content of the box. If there are no attributes, the table is omitted. Sub-terms (i.e. child nodes), are recursively visualised the same way and arranged vertically inside of the parent element's box. Textual content is simply written as plain text. In visXcerpt the namespace of an element is written on the top right corner in a smaller font and attribute namespaces are prefixed (also in smaller font) before the attribute name inside it's table cell. The width of an element box is at least the width used to represent the content, but it tries to use all width provided by the parent element. An element box is as long as needed to contain its content.

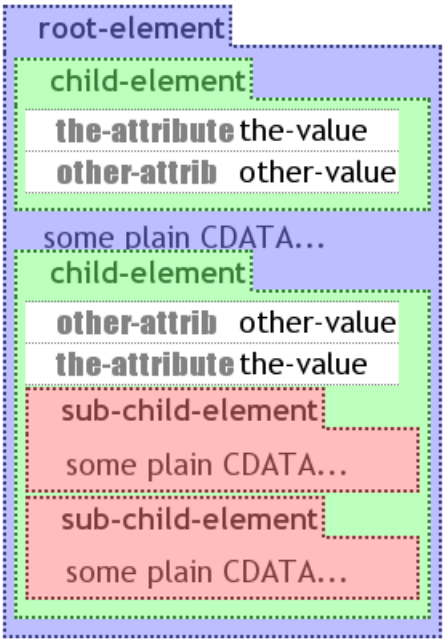

Figure 3.1: This is an XML document visualised using the generic visXcerpt visualisation. The colours are used for better distinction of depth.

Opposed to many other visual languages, this way the position and the size of all elements (except the root element's size) is determined recursively through their parent's extends and locations. An advantage of this approach is that it is not necessary to keep visual meta information along with the XML document to reproduce the same visual arrangement. On the other hand the freedom of being able to re-arrange or re-size elements in an interesting didactic way for documentation purpose is lost.

For better recognition of the element depth, every level has a different background colour. The amount of colours may be restricted and they may be reused for deeper levels. For a pleasant look and feel not too many colours should be used. It turned out that three to six colours provide a good tradeoff between tree level recognition and economic colour use. It turned out to be pleasant to use light pastel colours for the background and the same colour in a much darker tone for the foreground.

A further way to support the metaphor of nested content is to use a lightly bevelled border line instead of a plain one and to introduce some kind of visual depth (The use of the prototype showed, that this feature is

[1]This WWW resource is located at `http://www.w3.org/TR/REC-xml-names/`

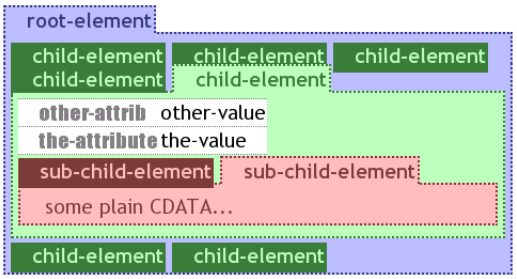

Figure 3.2: This generic visualisation uses information hiding. Some *child-element*s and a *sub-child-element* are folded.

only useful for screen presentation and not pleasant on printed versions of the visualisation)

The XML visualisation as proposed by now is partly inspired by Martin Erwig's *document metaphor* [erwig-xml2000]. The *document metaphor* in turn is inspired from forms. Forms are known by most people e.g. as ordering forms or as forms for income taxes. Forms usually have in common that associated information is grouped in a boxed way, sometimes even using borders to show grouping.

Visualisation of large documents tend to get very long while not being too wide. For browsing of large documents, scrolling through the visualisation can be cumbersome. An information hiding mechanism is introduced to hide irrelevant parts of a document. Nodes can be folded by clicking on their tab. The content is hidden while light and dark shading of colours used in the tab are reversed. The labels are then displayed like tabs on registers along with other folded sibling elements, an unfolded element's label is the last one in the row of labels. If the amount of folded labels exceeds the width of the visXcerpt display area, a line-break occurs and a new line begins. For big databases all nodes can be folded in the initial state. The user can then dig into relevant areas by iteratively unfolding relevant nodes.

It is noticeable that a certain kind of order in the two dimensional visualisation was used for the former proposal: from left to right, then from top to bottom. This is based on the occidental textual document order for two dimensional information. In other cultures this order may be adapted to another order, e.g. in the visualisation of an arabic XML document, element names may be aligned to the right and flow to the left.

3.1.1.2 visXcerpt and graph structures

XML allows the construction of graphs using reference mechanisms. Graph representation is not fully implemented in visXcerpt, its development is still in progress. The solution retained for visXcerpt is the use of hyperlinks to cope with graph structure. The graph is represented by a spanning tree and

Figure 3.3: A visXcerpt pattern representing a bibliography database. Hyperlinks are used to span references between books and authors.

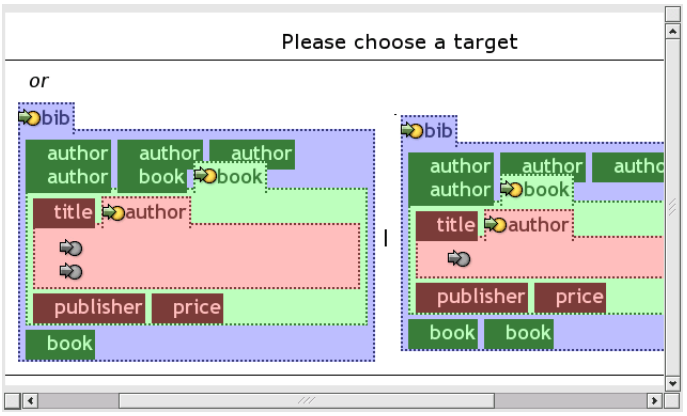

Figure 3.4: A dialog for selection of referring elements could look like this.

references used to complete the graph are spanned in the "hyperspace". This departs from the approaches taken in most other visual languages, which are using two dimensional representations of nodes and edges for graph visualisation. While a two dimensional visualisation of non-tree graphs is suitable for small graphs, a hyperlink representation scales very well to graphs of arbitrary sizes. A reference is visualised as link (see 1b in figure 3.3 on the preceding page) to the referenced element (1a in figure 3.3 on the page before). Clicking such a link (see 3a) scrolls the user's visual context to the referenced element (3b). Furthermore, when hovering with the mouse pointer over a referenced element (see 2a), all referring elements are highlighted (2b). While browsing or editing documents using visXcerpt, it is also possible to navigate back to the referencing element via a back referring link.

A back referring link is comparable to web-browser's "back"-button, but it is not useful, while trying to find any element referencing to an element. For this purpose, a dialog that shows all instances referencing to an element can be used. The dialog must be accessible dependant of the referenced element. The context menu, described later in the section about the dynamic aspects, would be a good place for accessing the dialog. As seen in figure 3.4, it displays a disjunction of elements. Each element is the whole document with maximum possible information hiding, so that the referring element is still visible. Beside the element names, a hyperlinks that scrolls the main visual representation (not the dialog's visualisation) to the corresponding position is available. Displaying the whole document for each occurence of a reference is useful to better grasp the context of the referencing element.

While this specification is quite usable for generic XML visualisation, it may also be interesting to provide an extension mechanism for the visualisation of special elements or attributes, a kind of stylesheet for special applications. Presuming such a mechanism, the visualisation of the special Xcerpt components is proposed now.

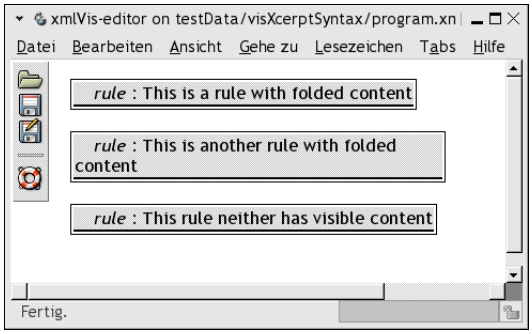

Figure 3.5: A visXcerpt program always exists in the context of a visual environment. Thus no special visual construct is available to depict a program. This window shows a visXcerpt editor with three rules (all folded for information hiding purpose).

3.1.1.3 visXcerpt Elements

The recursive nature of the former proposal for generic XML specification suggests to inherit the layout behaviour for the visual Xcerpt components. Nevertheless, direct children of Xcerpt components may be layed out in a different way. The liberal use of colours for generic visualisation suggests the use of a fixed colour scheme for Xcerpt components that is disjunct to the former colour set: white and black with more or less transparency will be used for all Xcerpt elements. Another distinction is done for textual Xcerpt keywords: they are written with italic font style.

3.1.1.3.1 Program

A Program consists of a finite set of rules (containing at least one rule). Because a program is meant to exist inside a development environment, programs do not introduce any visual representation. The rules (that are direct children of a program) are arranged as a list from top to bottom.

3.1.1.3.2 Rule

A rule is a container for construction and query patterns. The construction pattern can also be called head and the set of query patterns is also known as body. The semantics of rules is inspired by Prolog rules which in turn are Horn clauses with exactly one positive fact. The visual representation of rules is therefore also inspired by the traditional way of displaying horn clauses: head and body are arranged horizontally separated by an arrow pointing from the body to the head. The head is on the left, while the body is on the right. Because the semantic of the constructor is related to the content generated by the rule, the programmer may grasp with a quick glance at the left of the rule for what kind of content a rule is responsible.

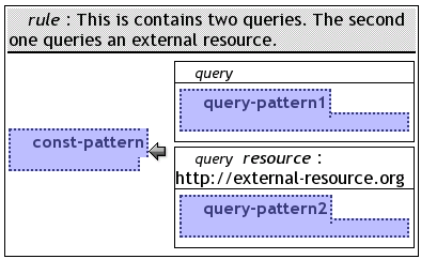

Figure 3.6: This example consists of one rule. The rule has a head with a pattern consisting of only one element called *const-pattern*. The rule's body contains two rules with both only one element as query pattern – *query-pattern1* and *query-pattern2*. (Semantically the query is boring, because it always returns only *const-pattern* – the queries are therefore obsolete.)

Every rule is surrounded by a thin black border. At the top there is a title bar containing the keyword *rule* and an optional comment about the rule. Clicking the *rule* label folds the whole rule for information hiding purpose. A folded rule is represented by its title bar (still containing the *rule* keyword and the comment)

3.1.1.3.3 Head (Construction Pattern) and Body (Query Patterns)

The head like the program is not rendered in a special way, because it is determined by its position at the left of the rule arrow. The content of the head is displayed in the usual way inherited from the generic visualisation (which means that the content is arranged vertically)

The body is treated the same way as the head.

3.1.1.3.4 Query

Queries are the common members of a body. A query may have an optional resource on which it is applied. A query is displayed approximately the same way as a rule: a black bordered box with a title bar and the keyword *query*. The optional query resource takes the place of the comment in the title bar. Multiple queries are arranged vertically if they are *and*-connected and horizontally if they are *or*-connected. Because the semantics of multiple queries are a conjunction of them, it is considered to be unintuitive to hide some of them. Therefore they are not folded when their label is clicked.

3.1.1.3.5 Variable

There are two different use cases for using variables in Xcerpt: variables placed somewhere inside a pattern and the so called *as*-construct, which restrict a variable to a sub-pattern. Variables are represented as black boxes

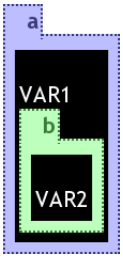

Figure 3.7: A query pattern is presented here, where *VAR1* corresponds to Xcerpt's *as*-construct and *VAR2* is a normal variable. *VAR1* is restricted to *b*-elements found in elements named *a*. *VAR2* is constrained to elements found inside of *b*-elements.

with their name written in white at the top left corner inside its bounds. If the variable represents an *as*-construct, the restricting pattern is placed inside the variable's bounds. This may be confusing regarding the structure of the pattern: the aliased pattern is a direct child of the element containing the variable, and not a variable's child. To indicate this fact, a visXcerpt implementation may use a translucent black screening instead of a solid black colour for the variable background. This would indicate that the pattern inside the bounds of the variable has some relationship to its parent element – the parent's colour is still surrounding the element. The translucent background is not implemented in the current prototype. Variables cannot be folded, because then empty variables and *as*-patterns would be indistinguishable. Variables partly implement the graph visualisation paradigm. If the mouse pointer is moved over a variable name, all occurences of this variable are hilighted.

3.1.1.3.6 All and Some

The Xcerpt *All* construct is only used in the head. It is represented by a black bordered box with a translucent screening background. The contained pattern is prefixed by the word *ALL* written in italic font, because it is an Xcerpt keyword. The *SOME* construct is currently not implemented. It can be used to take some results out of pattern matches. For this purpose a numeric value indicating how many elements are to be extracted from the set of matched elements is necessary. This number can be written between the pattern and the *SOME*- keyword (which is also in italic font). It is possible, that the number is styled like attribute values, to emphasize that it is an editable value.

3.1.1.3.7 Descendant

The descendant construct is used to match content included as child at arbitrary depth from the descendant element's position. To depict this, a

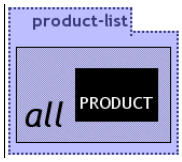

Figure 3.8: A construction pattern illustrating the use of the *all* construct. This pattern constructs an element called *product-list* containing all elements being matched my *PRODUCT* in a query pattern.

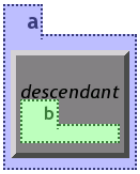

Figure 3.9: A query pattern describing the use of *descendant*. It matches any element named *a* that has an element *b* as direct or indirect child.

grey box with very strong bevel is used. A descendant need not be folded because the gain is marginal as it contains only a single element.

3.1.1.3.8 Regular expression

Regular expressions are labelled by the Keyword *regexp*. The regular expression itself is underlined with a dotted line. Regular expressions need not be folded because they tend to be one line long and only little would be gained. Regular expressions are only allowed in query patterns.

3.1.1.3.9 Logical connectives

The logical connectives *AND*, *OR* and *NOT* are displayed in different ways. Since *NOT* is currently not worked out in Xcerpt it will not be covered here, but it could be represented similarly to the *all*-construct.

In accordance with various 2-dimensional logic notations, *or* is expressed through horizontal arrangement of its members while *and* connected elements are arranged vertically. This convention turned out to be the intuitive way for most people to the question how to associate vertical/horizontal arrangement to *and* and *or* in any abstract context.

Figure 3.10: This query pattern matches any element named *"a"* that has a CDATA section as direct child which contains the string Xcerpt.

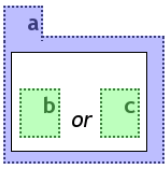

Figure 3.11: This query pattern (logical connectives are not allowed in construction patterns) matches elements names *a* which contain elements *b* or *c* as direct child.

Any number of elements can be *or*-connected. They are arranged horizontally in a white box separated by the *or* keyword.

Because queries in a rule are implicitly *and*-concatenated, not so much effort is done to visualise the *and*-construct. If used inside a query it may be indicated in a similar way as the *or*-construct – by a white box .

3.1.1.3.10 User constraints

User constraints are work in progress in Xcerpt and not documented yet. A short explanatoion is given here.

User constraints can be compared to the condition boxes in QBE [zloof77query]. They provide data structure or schema independent constraints on variables. In Xcerpt, user constraints can be placed in the body. They may be children of *and*, *or* and *query* constructs. They are not part of the pattern structure they are located in. The position of user constraints according to the term structure is restricted by the occurrence of the variables they contain: variables used in user constraints may not violate range restriction (standardisation apart). It is likely that constraints are expressed as equations.

In visXcerpt, constraints are placed in so called condition boxes. A condition box currently does not use a special syntax: it is visualised as a generic XML element with name *constraint* in the Xcerpt namespace. Inside those constraints, no further visual constructs are introduced, except those defined. This means that the equations are written as in the textual representation, except e.g. document patterns or variable occurrences: these constructs are displayed as usual. This may lead to a kind of horizontally arranged list of visXcerpt constructs separated by keywords or symbols for user constraints. User constraints are neither implemented in Xcerpt, nor in visXcerpt.

3.1.2 Dynamic Aspects of visXcerpt

So far, only so called static aspects of visXcerpt have been proposed. At this stage, visXcerpt can be used as visualisation for Xcerpt programs. To be able to use visXcerpt for programming tasks without having to rely on

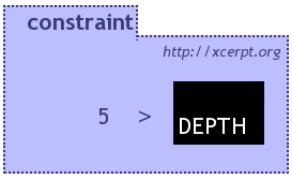

Figure 3.12: This is an user constraint restricting the variable *DEPTH* to textual values which can be interpreted numerically smaller that 5. It may occur anywhere in a rule's body, as long as it doesn't violate the variable's range restriction.

textual Xcerpt, it is necessary to edit visXcerpt documents. For textual languages the editing process is clear, because the editing process for text is generalised – any text document, no matter in which language (natural or programming) it is written, can be edited with the same text editor, e.g. *Emacs*, *Word*, *Notepad* or *Vi*. This section will first provide an overview about visXcerpt's editor model and then explain some facets in detail.

An visXcerpt editor should be aware of all the structures and components described in the former section. For textual editing, typing in new data (e.g. through use of a keyboard), cutting, copying, deleting and pasting are the most common editing primitives. These editing operations are bound to some kind of context locator – usually a cursor or a selection. A cursor is usually located between two symbols and indicates the location where something may be edited or pasted. A selection is determined by two cursor positions: one position is obtained while the process of selecting starts, the other when it ends. The biggest difference between text (and other sequential information) and visXcerpt data is the explicit tree like structure of documents. Various (usually very short) sequences of elements are arranged in a hierarchy. Unfortunately the usual editor model with a cursor as separator, cut, copy and paste operation based on intervals is tailored for one dimensional data like text documents. Flattening the XML hierarchy by implying a document order would be necessary to apply the classic editor model to visXcerpt documents. As the solution is not intuitive, the concept of a cursor as separator is unsuited for visXcerpt. Instead, another approach is proposed to provide contexts for editing primitives: everything relevant for the editing process is supplied with a context menu containing the valid editing primitives for the context. Because actions are now bound to components rather than separators, it is likely that the editing primitives are different from those used for text editing. This difference is explained later.

Another condition for an editor model of visXcerpt is that one editing step should (as far as possible) lead from one valid visXcerpt program to another valid visXcerpt program. This way an explorative approach to the visXcerpt language for new users is encouraged – "play around with the user interface, until what you see seems to represent what you have in mind".

To accomplish this, the context menus should be dynamic, providing only state transitions that are not violating validity.

While restricting the editing process to transitions leading from valid states to new valid states is helpful for casual- and unexperienced users, experienced users may feel too restricted by this feature. These users may have concrete visXcerpt programs in mind and know efficient ways to assemble them based on snippets of older programs, leading to some invalid intermediate steps. A pragmatic answer to this problem is to provide an unrestricted version of editing contexts with the abilitiy to copy, paste and delete in all circumstances.

3.1.2.1 The visXcerpt Editor Model

As mentioned earlier, cutting, copying, pasting and deleting are the main editing primitives of visXcerpt. These actions are bound to contexts and contexts are bound to elements, CDATA sections or attributes. They are invokable through a context menu. These context menus can be seen as a static element, because every element has it's own context menu.

In order to paste without inconvenience, it is necessary to be able to paste into empty elements, at the beginning and at the end of a list of sibling nodes. This is accomplished through the special paste variants

- **paste before:**paste the buffer content before the context node below the same parent
- **paste after:**paste the buffer content after the context node below the same parent
- **paste into (at beginning):**insert the buffer content as first child **into** the context node
- **paste into (at end):**insert the buffer content as last child **into** the context node

Copying, cutting and deleting is applied to the context node (including its child nodes). Editing text like CDATA, attribute names/values is initiated by clicking them. Because clicking the element names leads to folding or unfolding them, editing the element name cannot be accomplished by clicking – this feature is therefore located in the context menu.

An interesting aspect about editing is to support "learning by doing": the user is only supplied with editing operations which lead from one valid visXcerpt program to another valid visXcerpt program. One feature supporting this paradigm is to deactivate paste operations based on the paste buffer's content, if pasting would invalidate the selected context. Cutting and deleting is deactivated as well, if removing the context element invalidates the document. The current visXcerpt prototype does not support

dynamic context menus. Integration of an XML type system comparable to DTD is necessary to check for validity of visXcerpt documents and therefore also for validity of editing operations.

While cut/copy and paste is convenient for modifying existing content, creation of new documents is not supported very well. For this purpose a template document is provided. It contains all visXcerpt elements and they can be copied from there and pasted into a new document if needed. Therefore the editor can be split into two frames, one with the new program to edit and one with the template document. The previously mentioned support of dynamic context menus for "learning by doing" may influence the design of template documents. If editing operations are restricted to non-invalidating operations, more element compositions should be provided as templates: a rule element for example must have a query and a head element to be valid.

3.2 Examples and Use Cases

So far Xcerpt's and visXcerpt's syntax have been presented. Since the semantics has not been introduced in detail, a comprehensive example section is provided to explain the usage of Xcerpt and visXcerpt through *"learning by doing"*.

3.2.1 Simple visXcerpt Programs

The programs in this section will present the elementary visXcerpt features in one rule. They rely on a simple document called the greengrocer database. It is kept short and easy to understand. The database is presented now using visXcerpt's generic element visualisation.

The visXcerpt term in figure 3.13 on the next page represents the example database that the following examples will query. For layout purposes of print media, the visXcerpt pattern is split in three parts. It models a greengrocer database with the product inventory and the vendors providing the grocery. Every vendor has a name and a country attribute. Products are either vegetable or fruit and have a name and a vendor who delivered them. Furthermore they have a price that is either given in pieces or based on the weight.

3.2.1.1 Database Identity

For evaluation of the examples in the visXcerpt prototype a program and a goal (called *query*) are needed. In the following examples the goal is mostly irrelevant and will therefore always be the default goal – an unconstrained variable called *RESULT*.

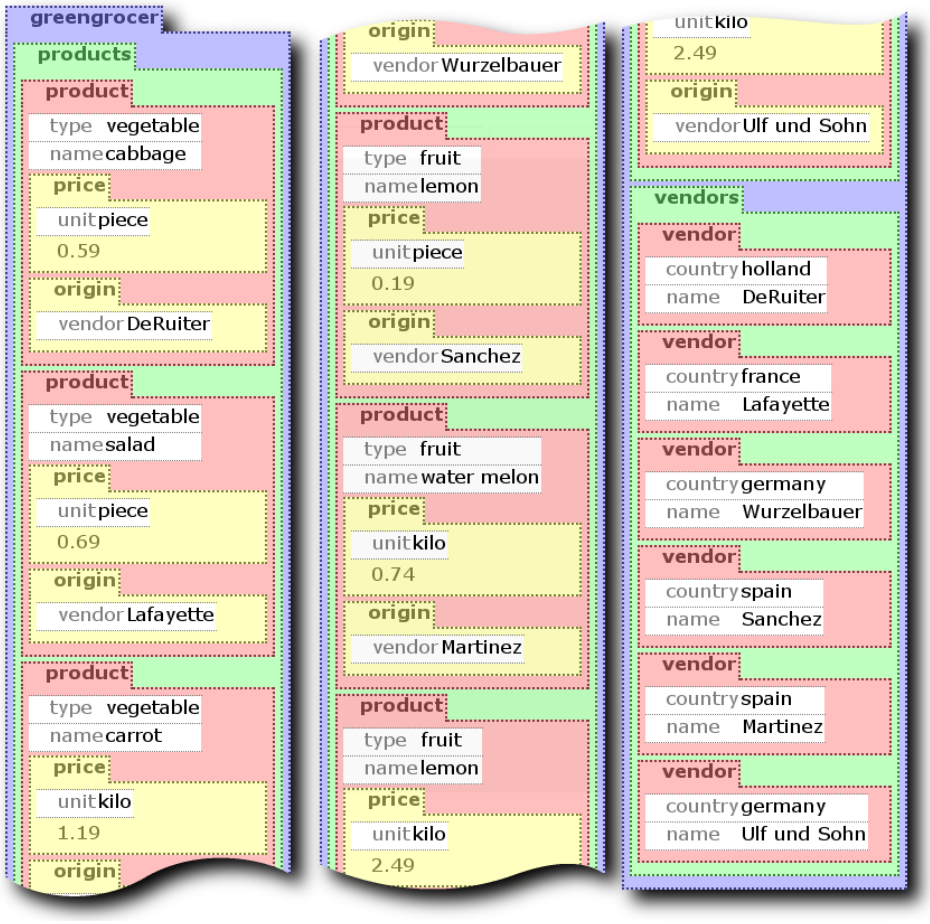

Figure 3.13: Greengrocer database. (The visualisation is cut in three parts for better print layout)

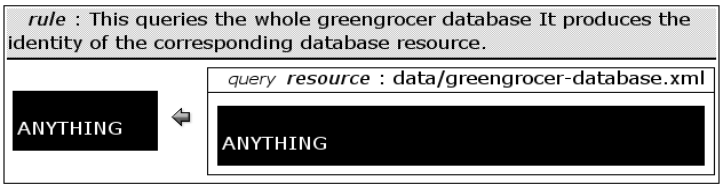

Figure 3.14: Greengrocer database identity. The *ANYTHING* variable is unrestricted in the query pattern and "undecorated" in the construction pattern.

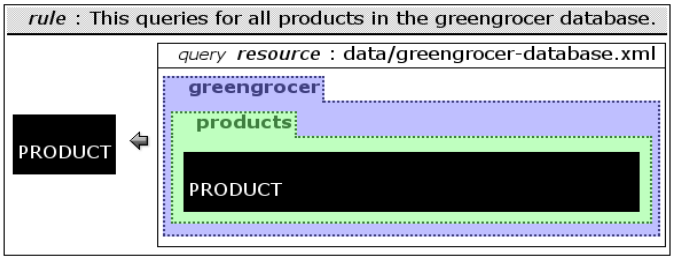

Figure 3.15: Querying all products. The variable is placed inside of *products* and matches only elements in this context.

The program in figure 3.14 consists of only one rule. The rule has one query in the body that queries an external resource – the greengrocer database. The query pattern consists of one unconstrained variable called *ANYTHING*. It thus matches the whole database. The rule's head and hence the construction pattern consists of the same variable. The result is therefore the database itself.

3.2.1.2 Querying for Parts of the Database

The following section will focus on query patterns. The result will always be bound to one variable. The head (or the construction pattern) will only consist of this variable. The result may consist of multiple elements and will therefore not always be well formed XML. In such a case the result consists of a disjunction of alternatives.

The overall structure of the program in figure 3.15 is comparable to the former identity example. In contrast, the variable is now inside a pattern. Therefore only elements being inside a structure matching this pattern will be bound to the variable. All elements fulfilling the locational restriction are passed to the head and are therefore returned as result of the program.

In the example on figure 3.17 on the facing page not all products will be returned. Instread, the result consists only of such products that are vegetables, i.e. have a "type"-attribute with the value "vegetable". In Xcerpt a variable is considered a "name" for a pattern and thus not part of the structure. Th visXcerpt on the other hand, the restricting pattern is

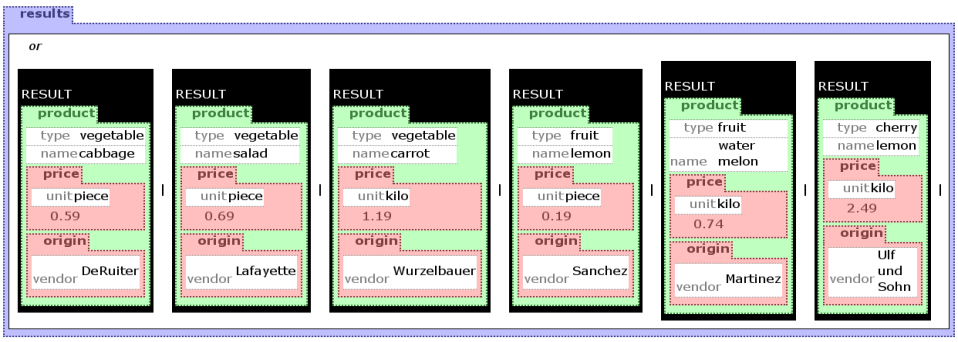

Figure 3.16: Result of querying all products, based on executing the former example. The alternative answers of the query are represented as disjunction.

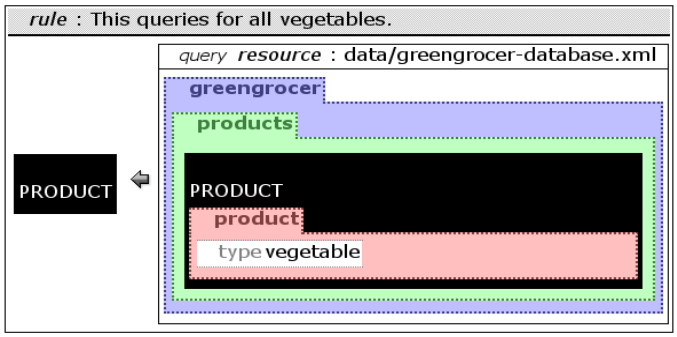

Figure 3.17: Restricting product query to vegetables. The restricting pattern is placed inside of the variable's bounds.

placed inside the variable.

The `descendant` construct is a kind of wild-card that matches any element in arbitrary depth. A `descendant` construct has to contain exactly one pattern. The corresponding pattern is then matched at any depth below the `descendant` in the database. The example in figure 3.19 on the next page behaves like the example in figure 3.17 and queries all vegetables. It is somehow simpler, as the restricting pattern only consists of the *product*-element. The variable containing the *product*-element is surrounded by a `descendant` construct. The root *greengrocer*-element and the *products*-element can be omitted.

On figure 3.20 on the next page the example uses a disjunction with the `OR` construct to select all products supplied by the vendors Sanchez and DeRuiter.

Although there exists an explicit `AND` construct in Xcerpt, conjunctions are usually implicit. The conjunction is the base for joins: variables shared between elements of a conjunction are used to express the join. An example using a join over the *vendor* attribute of *product* and the *name* attribute

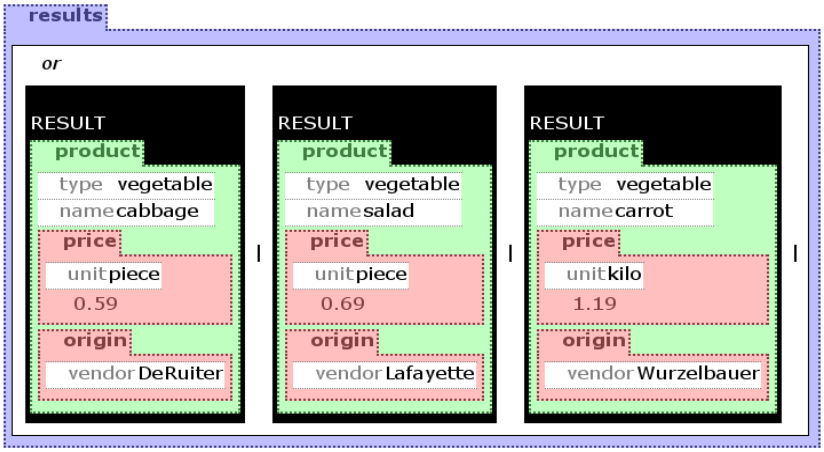

Figure 3.18: Result of querying for vegetables as presented in the former example.

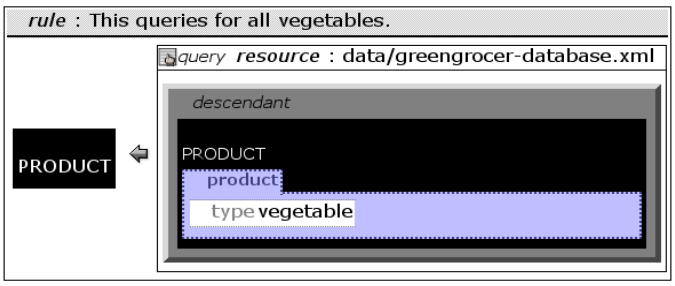

Figure 3.19: Using the `descendant` construct. The query can be compared to the former example. The pattern outside of the variable can be obmitted because the occurence of *product* elements is clear in the context of the greengrocer database.

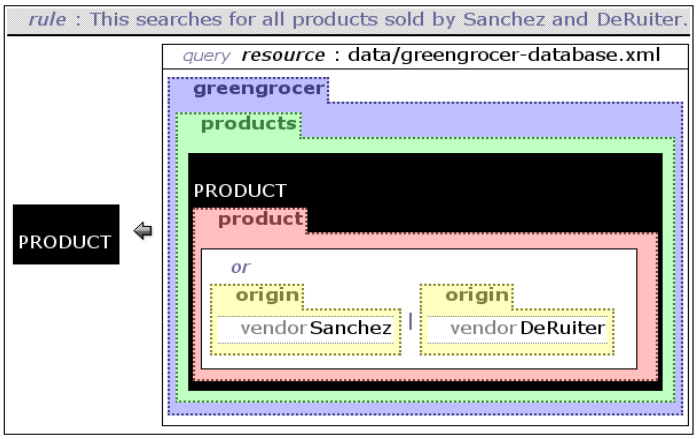

Figure 3.20: Disjunction of multiple queries. Only *product* elements having an *origin* with *vendor* attribute "Sanchez" or origin with *vendor* attribute "DeRuiter" are chosen.

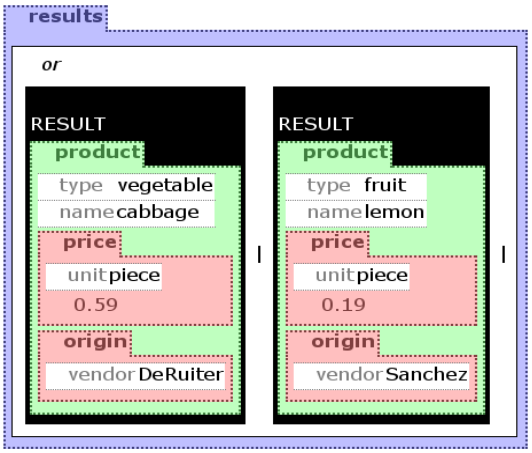

Figure 3.21: Result of the former disjunction example

of *vendor* to search for all products produced in Spain is presented on figure 3.22 on the following page.

3.2.1.3 Using Multiple Queries in one Rule

To be able to query multiple resources at the same time, it is possible to put several queries in one rule body. Queries can be connected as disjunction or conjunction. If no conjunction is specified, a conjunction is built as the queries are arranged vertically. In a conjunction shared variables can be used to model joins.

3.2.1.4 Xcerpt Goals and the visXcerpt Query Frame

A goal is a query pattern that is applied to the whole program. So far, the goal was always assumed to be an unrestricted variable, therefore the result was only determined by the program. By using a goal, it is possible to refine the result of a program. A goal is similar to a query in the rule body and hence can be used to restrict the result. A program can then be compared to a library, with the goal as a selection of concrete functionality. The query of figure 3.25 on page 85 can be applied to the previously presented program that retrieved all product elements to restrict it to the output of the disjunction program.

3.2.1.5 The Rule's Head

While the focus so far was on the rule bodies the rule's head will be explained now. In former examples the head only consisted of a variable. To

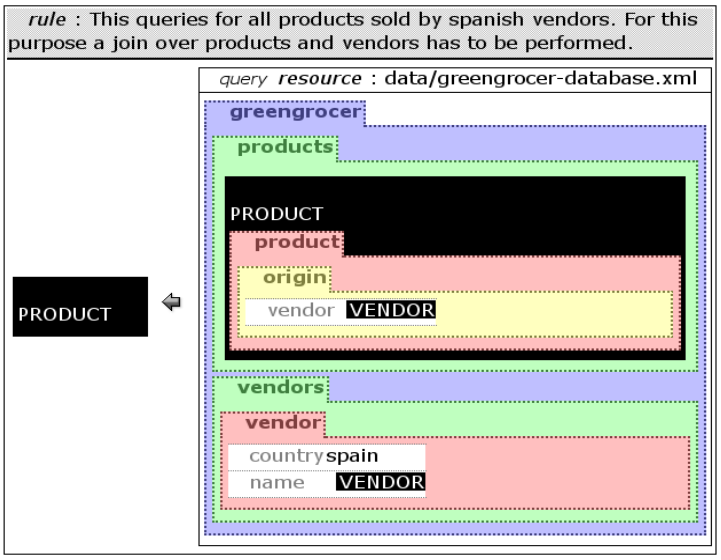

Figure 3.22: Conjunction and joins. The *VENDOR* variable is used to establish the join explained in the rule comment.

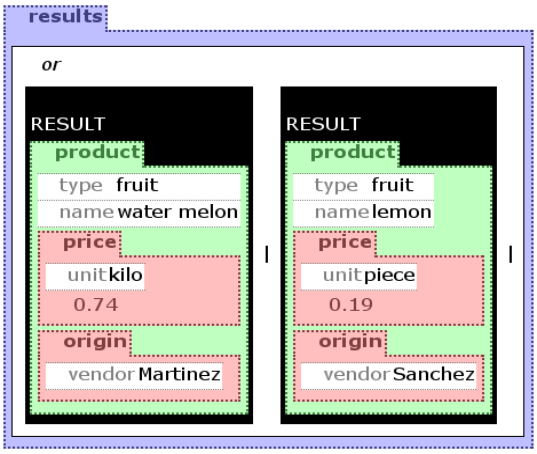

Figure 3.23: Result of the former conjunction and join example.

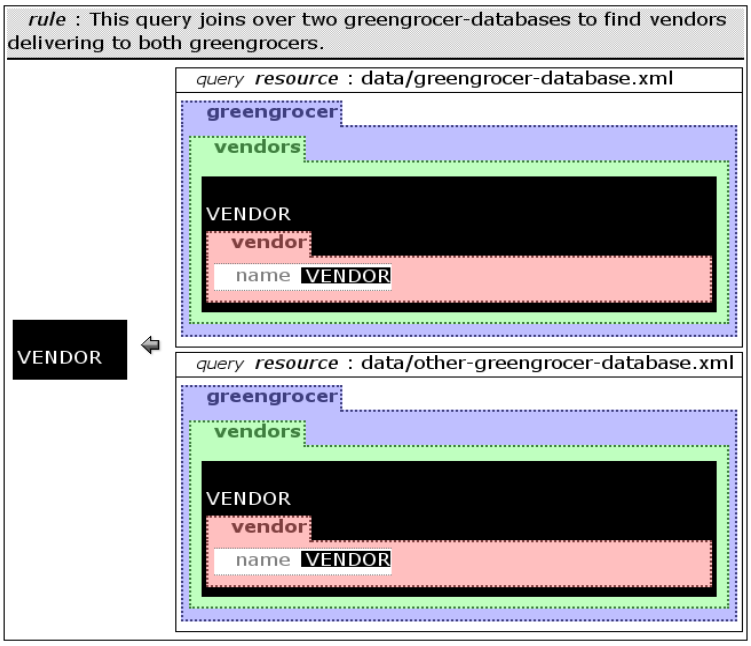

Figure 3.24: The join over multiple resources is explained in the rule comment.

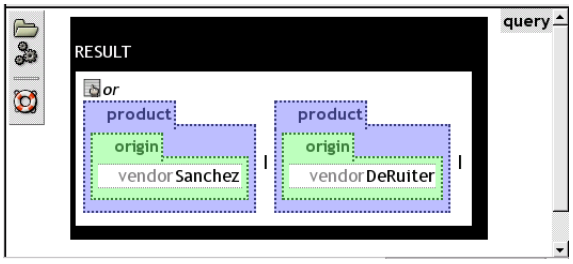

Figure 3.25: Using a query goal to query all products delivered by Sanchez and DeRuiter (used in conjunction with the former program to query all product elements)

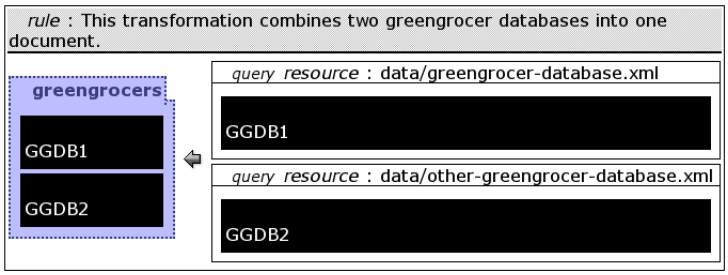

Figure 3.26: Combining two greengrocer databases

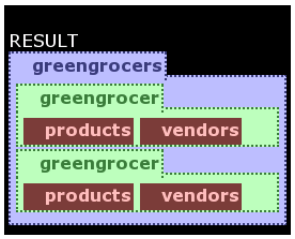

Figure 3.27: Result of combining two greengrocer DBs. (Information hiding is used and *products* and *vendors* are folded)

be able to use visXcerpt as transformation language, it is important to construct new content, but queries can only reproduce existing content of the queried resource. To produce new content, construction patterns are used. Construction patterns are located in the head of a rule.

The example of figure 3.26 combines two resources into one document. It can be compared to the identity example. The difference is that the identity of two resources is associated to two variables that are used as child of the root of the result document.

The result document shown in figure 3.27 is displayed using information hiding, because the content of the two *greengrocer* elements is similar to the greengrocer database presented earlier.

The example from figure 3.28 on the facing page is a modification of the product query from the previous section. All products are collected in an element *product-list*. To be able to collect all alternatives of the variable *PRODUCT* into the *product-list*, the `ALL`-construct is used. Without this, every product would have been packed into its own *product-list*. This leads to a result consisting of only one alternative, therefore the result is valid XML content. The `ALL`-construct is only allowed in construction patterns.

3.2.1.6 Programs with many Rules

In the general case, it will be desireable to write programs with more than one rule. As in logic programming rules are disjunctive. The variables are

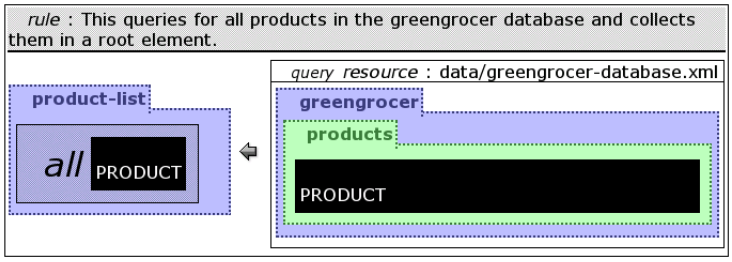

Figure 3.28: Rule to generate a product-list as explained in the rule comment.

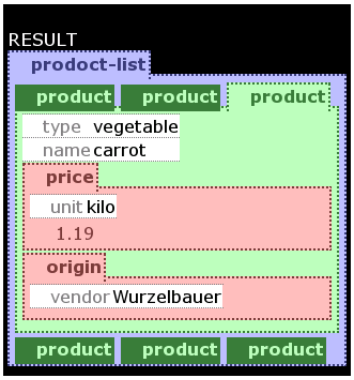

Figure 3.29: Result of generating a product-list with the former query. (All *product* elements except one is folded for layout purpose)

Figure 3.30: Rules are implicitly connected by a disjunction. This program is semantically identical to figure 3.20 on page 82.

not shared among rules, the scope is local to the rules.

The example of figure 3.30 on page 88 is comparable to the disjunction example presented earlier. The difference is, that it is implemented using two different rules.

The example in figure 3.30 presented the disjunctive nature of many rules in a program, but this is not really useful compared to the shorter variant presented earlier (see figure 3.20 on page 82). Using multiple rules in a program is especially useful to build a library of rules. Concrete queries can then be expressed easily using the query goal. An example for this is the library presented in figure 3.31 on the facing page. It can be used to query products or vendors, simply by using an appropriate query goal, like the one presented in figure 3.25 on page 85.

3.2.1.7 Chaining Multiple Rules

A striking feature of visXcerpt is the so called chaining. Chaining is characterised by the process of using the result of a rule evaluation as input for further rule application. So far, rules had a resource associated as input. The query patterns operated on this input. If a query has no associated resource, chaining is applied. The program itself is used as input in this

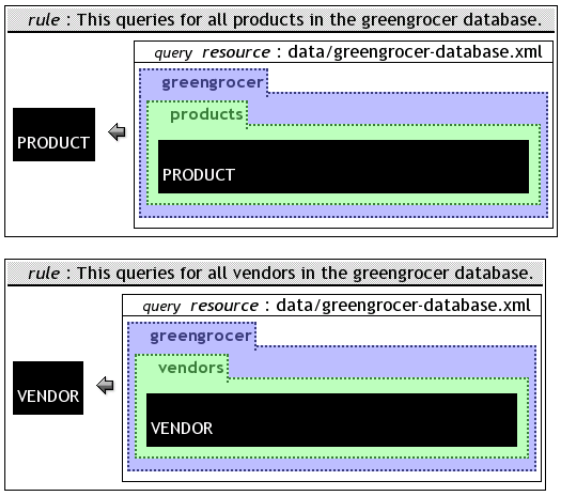

Figure 3.31: A simple greengrocer access library. The rule comments describe the functionality.

case. This leads to querying the output of all construction patterns in that program.

A simple example using the former product list is the following: an HTML product list is generated based on the product-list example. Because no module or include mechanism is defined so far in Xcerpt, the rule is added to the former example. One rule is used to build the HTML table by chaining it to the *product-list* and by transforming it. Another rule takes the table and embeds it in an HTML page. The only purpose of this rule is better readability in the sense of separation of concerns. To get the HTML representation, the query goal is restricted to the HTML representation by putting an *html* element into the unrestricted query goal variable. If the variable was not restricted in this way, three alternatives were produced by the program: the *product-list*, the *table* and the HTML page.

The use of chaining brings up the question about evaluation order: in which order are the chained rules evaluated? Two possible strategies are backward chaining and forward chaining. Both strategies are useful in certain contexts and there is continuing research about this subject. Both strategies are considered for Xcerpt.

Forward chaining is useful to pre-generate results. The operational semantics of forward chaining can roughly be explained as follows: every body/query that is productive, i.e. unifies a condition, is evaluated. A condition is either a resource specified in the query or the result of an evaluated rule. The queried conditions are consumed and the evaluated rule instantiates a new condition, based on its construction pattern. This process is iterated until a saturation is obtained, i.e. further rule evaluations are not possible. Multiple evaluation of the same rule is possible, if corre-

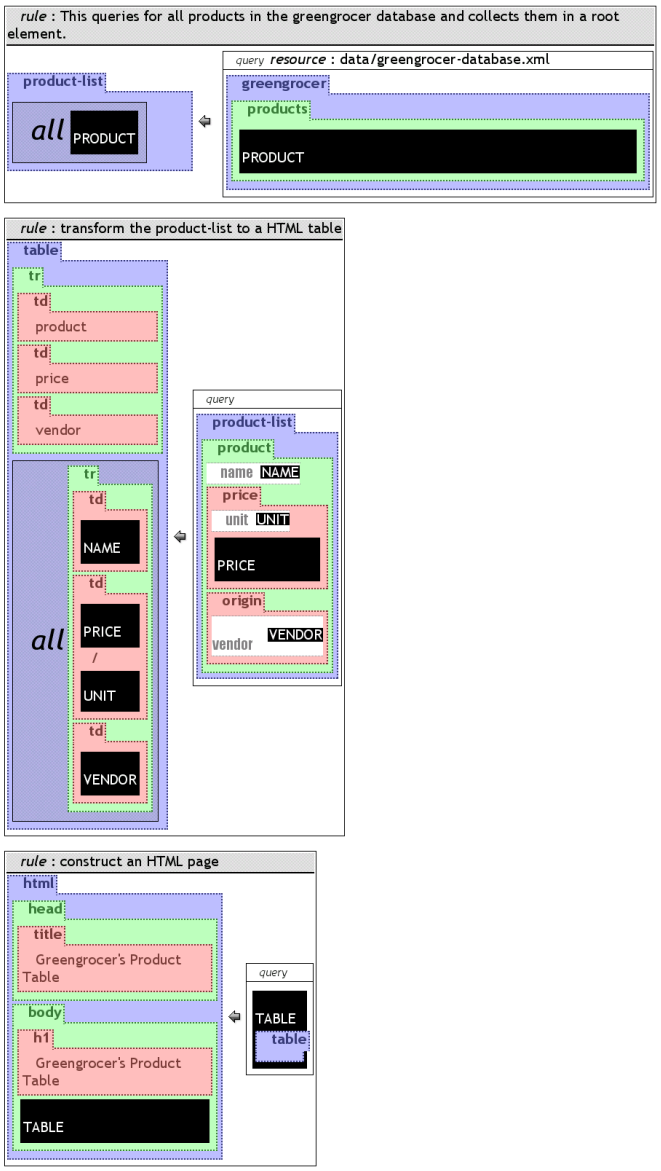

Figure 3.32: A HTML product table is generated by chaining three different rules. The presentation is separated from the query, the table could be queries separately to be included in another HTML page. The rule comments describe the features.

sponding preconditions exist. Forward chaining is often used in deductive database systems. A practical use case for forward chaining in Xcerpt is to generate the web pages of a whole web site based on an Xcerpt program. All web pages are generated at once and can be submitted to a web server.

The operational semantic of backward chaining is the opposite of the operational semantic of forward chaining. The evaluation is driven by a goal. The goal induces only evaluation of rules with heads that can be unified by the goal. The queries of those rules operate the same way recursively. Backward chaining is useful to query large or infinite structures, because irrelevant rule evaluations are usually not instantiated.

Structural recursion is an interesting aspect that requires chaining. In structural recursion, expressions are usually stated on contexts and recursion is used to operate the same way on the child elements of the contexts. This is the common behaviour of XSLT. It should be noted that the current Xcerpt prototype fails or does not terminate on the programs presented in the following. However, the necessary modifications are currently being investigated. This concerns all programs where information is passed to a variable in the head with the intention to be matched in the body. The recursions are now explained from the point of view of backward chaining.

To express structural recursion in logic programming, it is necessary to operate on a context. Because rules are chosen based on the unification with a query pattern, the context needs to be known in the query pattern. To evaluate a rule that is meant to operate on that context, the rule's head needs to provide a subpattern that can be unified with the context. This sub-pattern can be seen as input parameter. The result of the application needs to be transferred to the initiating query pattern. Thus the query pattern has a variable that matches a part of the head of the rule that evaluated the context. This part will be called output parameter.

In figure 3.33 on the following page a simple *wrapper-rule* is presented, which wraps its input parameter into a *wrapper* element. It is apparent that this rule has no query body. In Prolog these rules are called facts. The empty body has to be interpreted as *true* because it can be seen as an empty conjunction. The arrow may be replaced by another symbol for rules without body (maybe a dot). The current prototype does not support facts.

The use of *in* and *out* parameters, as mentioned above, can be compared to a function evaluation. An interesting aspect is that the reversal of a function application can be done with the same rule by exchanging the role of input and output parameters.

The mechanism of parameter transfer between rules is now used to build the structural recursive transformation in figure 3.34 on page 93.

The *rec-wrap-rule* is capable of wrapping elements and all its child elements recursively. For this purpose a further extension is necessary to the Xcerpt prototype: element names have to be capable of holding variables. This extension has already been investigated, but has not yet been imple-

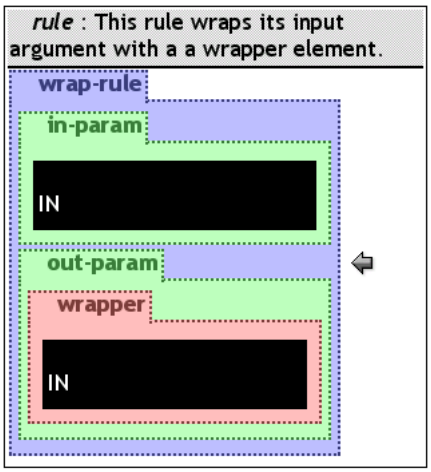

Figure 3.33: A rule without body can be called a fact. This fact simulates a function with input and output parameters. The input is put in a *wrapper* element.

mented; however it is not likely to pose any problems.

The program in figure 3.34 on page 93 consists of one rule. It is similar to the former *wrap-rule* program. The element name *ELEM-NAME* has to be interpreted as variable in this example. The input inside *in-param* is decomposed into the top level element and the child elements bound to the *REC-IN* variable. While the decomposed element name is used to construct the wrapped element inside the *out-param* element, the *REC-IN* variable is used in a recursive query of the same rule. The output of the recursive query is used to complete the *out-param* section of the construction pattern.

3.2.1.8 Practical Example - a core crawler for the web

This section will present a little library useful for construction of web crawlers. For this purpose variable resources are used. Since the current visXcerpt prototype does not support variable resources, the Xcerpt notation is used for this purpose – a dollar-sign is used as prefix for the variable name.[2] Obviously, the recursion through the link structure of the WWW will not terminate in most cases, as the WWW contains circular link structures. Nevertheless the core of this web crawler is a simple recursion through the WWW. The recursive core transforms the WWW into an infinite document. This XML representation is not intended to be materialised, but should be chained to user supplied visXcerpt queries. Apparently, this in-

[2]Prefixing a dollar-variable also works for attribute values or CDATA, because the transformed document is valid Xcerpt. Because currently variables in attribute values are broken, this feature is not disabled.

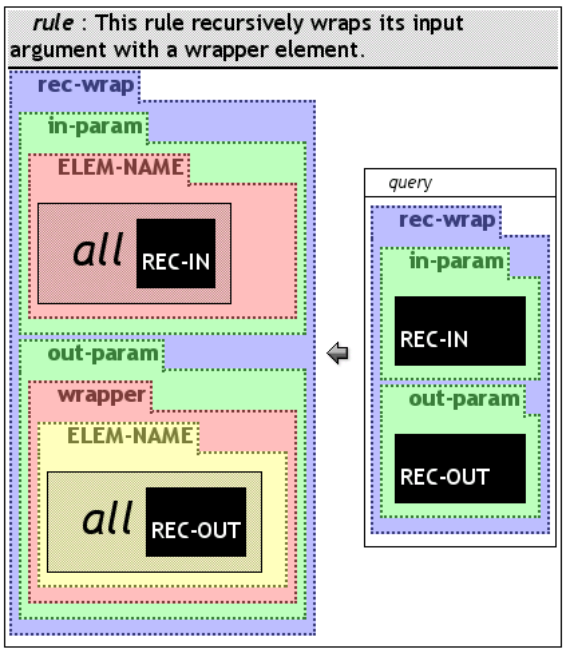

Figure 3.34: A rule for structural recursion. All elements are wrapped recursively in a *wrapper* element. Input and output parameters are used.

finite recursion assumes the Xcerpt processor to defer the construction and hence also the resource loading until it is relevant for a concrete query.

The naive crawler example in figure 3.35 on the next page explains how a recursion over the WWW structure can be expressed. To access most of the web it is sufficient to access a page of the so called web core. The web core consists of all pages that are accessible through at least one path of arbitrary length from another page of the web core, while having a path to all other pages of the web core. Those paths are established through hyperlinks. Yahoo! is a popular member of the web core with a very high "out degree"[3]. For this purpose the first rule initiates the recursion with the Yahoo! home page and wraps it in an element*web-resource*[4] element. The second rule does the same, except that the resource is variable. A second query in the body matches any *web-resource* element the program constructed (or will construct) and retrieves URLs in hyperlinks. Through a join, the URL is tied to the resource querying the page.

A problem with this approach is, that a querying and restricting rule can not make any assumption on the progress, e.g. the depth of the search or if the URL has already been visited. For this purpose, the *web-resource*

[3]The out degree corresponds to number of hyperlinks of a web site to other pages

[4]The wrapping is used to have an uniform access point to all web pages, because the HTML-Root element may be spelled differently or may even be omitted.

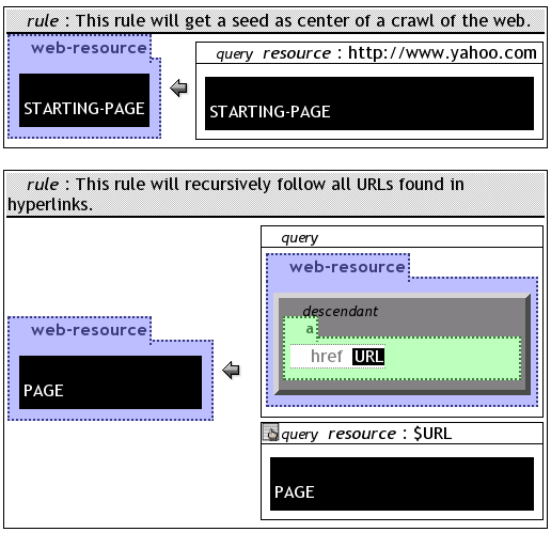

Figure 3.35: Naive web crawler (please read the rule comments)

element would need further extensions, like the URL it originates from, and information about its environment in the WWW topology. The extended crawler constructs the infinite document mentioned earlier as spanning tree of the WWW. Circular structures are not cropped, because an application may rely on them. Hence the result document may have infinite depth. A *web-resource* element is now extended by an *URL* element, a *content* element containing the concrete page and a *following-siblings* element with all web-resources accessible from this page through a hyperlink. Not covered is the aspect of differentiating between relative and absolute URLs. To implement this feature the regular expression construct is useful.

The example in figure 3.36 on the facing page is proposed to overcome the shortcomings of the former example. The program consists of three rules. The first one is used to strip the function emulation pattern off the desired *web-resource* document. The second and third rules are used to recurse through the WWW's link spanned graph structure. The third rule is used to handle pages without hyperlinks. The second rule cannot handle them, because no bounds can be found for the variable *REC-URL*, therefore the rule fails. The second query of the first rule is used to recursively call the *recursion* function. Still unhandled by this example are relative URLs.

3.3 A Prototype for visXcerpt

A main task of this thesis was the implementation of a prototype for visXcerpt. This prototype has been used to evaluate and refine the concepts of visXcerpt as presented earlier. Among the most important evaluated

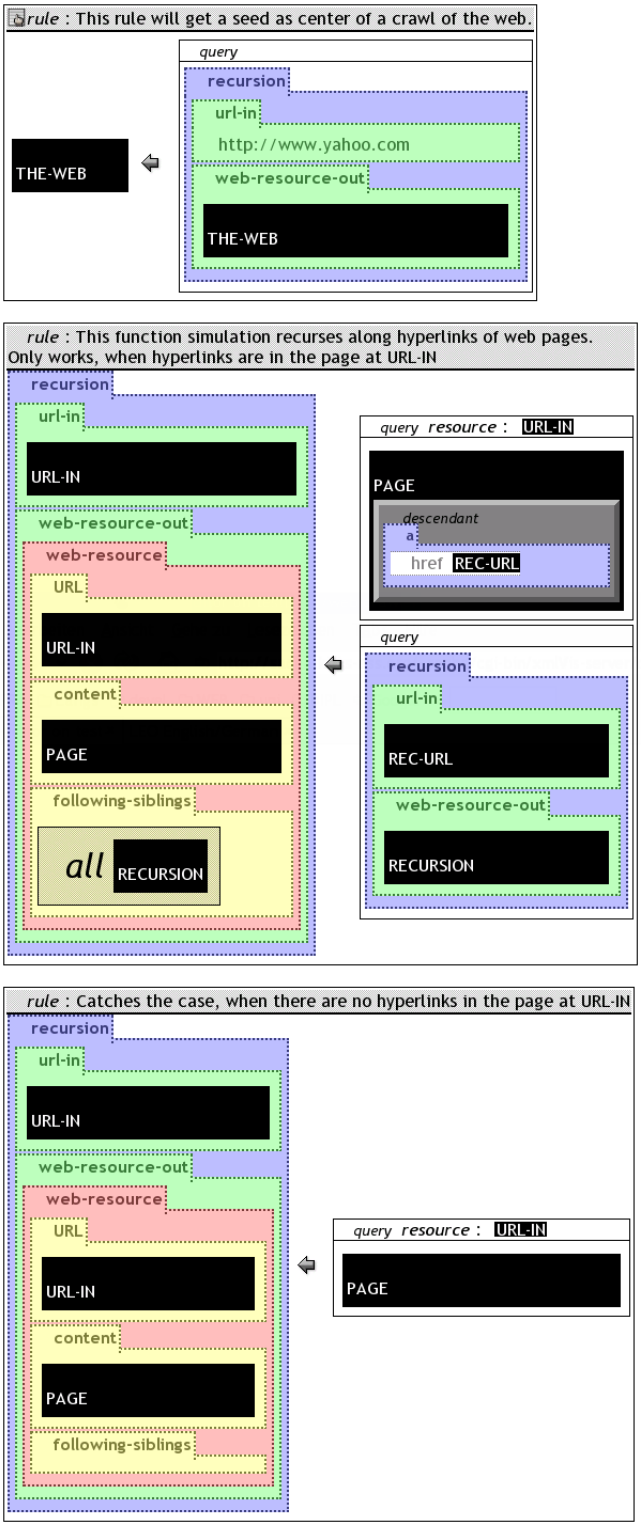

Figure 3.36: The rule comments of the revised web crawler explain the functionality.

features are:

- readability of the visualisation: The inspiration for the use of colours came after a monochrome visualisation turned out to look too cluttered
- usefulness of information hiding: while visualising large documents, it turned out to be useful to include information hiding. For the visualisation of databases it is more comfortable to start with a fully folded view, while this is unpleasant for visualising Xcerpt programs.
- editing concepts: The paste operation of the editor model was refined to contain the
  - `paste into at beginning`
  - `paste into at end`
  - `paste before`
  - `paste after`

  operation to fulfil all editing tasks comfortably. However, the editing process is still slightly clumsy and various high level or macro operations have to be investigated.

Last but not least the prototype was important for the authoring of visXcerpt examples in this thesis. Because of time and capacity restriction and because not all features are relevant for the proof of concept, not everything proposed in the former section about the aspects of Xcerpt have been implemented. The implemented parts are:

- the generic XML visualisation restricted to trees
- most static aspects of visXcerpt
- the untyped (non-validating) editing aspects
- persistence (the ability to open and save documents)
- presenting the result of visXcerpt program evaluation

The aspects that have not been implemented are:

- **schema support and type checking of editing operations** : This feature is not relevant for the evaluation of visualisation and editing aspects. The application design is separated in a server and a web browser application. This feature would require a lot of information flow from the client to the server for all editing operations modifying the document. To have pleasant user interface responsiveness some optimisation would be necessary.

- **context sensitive editing** : The restriction of editing operations to transitions that result in valid documents, and furthermore support for context sensitive creation of valid elements would be required. For this feature the former mentioned schema support is an important precondition.

- **general graph operations** : Due to lack of time they have not been implemented yet. Some parts of the graph support can be found in the behaviour of variables.

3.3.1 Target Platform

For the purpose of "separation of concern" a major implication was to use the Xcerpt prototype for program evaluation. This way the prototype is only focused towards translation of visXcerpt programs into Xcerpt programs. Another important feature was the abstraction away from layout aspects: visXcerpt programs do not contain layout and positioning information as many other visual programming languages do. A visXcerpt program can be translated into an Xcerpt program and vice versa. This enables an unambiguous translation from Xcerpt to visXcerpt. Hence Xcerpt can be used as intermediate format for persistence and transfer of visXcerpt programs. VisXcerpt is a pure in-memory representation of Xcerpt.

The abstraction away from layout and positioning information reveals the need for an automatic layout based on structural information. For this purpose a standard web browser can be used. HTML is the structured information that is rendered by the web browser. Using web standards for client user interfaces brings a high degree of system independence and benefits of highly optimised rendering engines. Forward compatibility with upcoming web browsers is very likely.

As some features are not available in web browsers, a web server based service is necessary. Persistence of visXcerpt programs is one of the reasons for this web server based service: some mechanism for loading and saving programs is necessary. Furthermore, standard web browsers do not integrate the Xcerpt prototype. The use of a Java applet is inappropriate, because various Xcerpt prototypes exist in different implementation languages other than Java. Furthermore, the persistence problem can not be solved easily in Java, due to file access restrictions.

Todays web browsers incorporate various web standards that can be used to implement web application client views. The following were used to implement the client side view of visXcerpt:

- **HTML :** This is the core of most modern web browsers. HTML incorporates properties of specific-, generalised- and generic- markup languages. It is therefore possible to specify the style and the structure

of documents. There is actually a tendency to deprecate the specific aspects – and therefore the elements responsible for document style. The styling aspects are actually shifting towards CSS. The generic aspects of HTML are realized through the use of so called `class`-attributes. `class`-attributes can be used together with most HTML elements, but the suggested semantics will always be a combination of the user supplied class-semantics and the elements core semantics. A `H1`-Element (a major heading) with a class attribute value X is likely to be interpreted at the same time as heading and as an X object. To prevent this semantic overloading, there exists the semantically almost untainted elements `div` and `span`. The only interpretation of `span` is that it is an inline element of the surrounding text. It will therefore be part of the text flow. `div` is interpreted as some kind of section and will therefore start a new block of text.

- **CSS :** CSS is used to specify the style of HTML markup. The capabilities range from text- and background- colour, font -style, -size and -family to arrangement, indenting, automatic text creation, list numbering style and simple interactive features like link action style. CSS is the first choice to make classes of elements visually distinguishable. Nested structures can easily be made apparent with CSS through the use of borders and colours. A static core mechanism for information hiding is available through various element visibility properties.

- **ECMA-script :** Based on Netscapes scripting language Java-Script[5] the ECMA standardisation organisation established ECMA-script. It is a subset of Java-Script implemented by all modern web browsers. It lacks Netscape's and Internet Explorer's specific features. ECMA-script is an imperative, dynamically typed and object oriented interpreted programming language. ECMA-script's object orientation is not based on class inheritance as used by most object oriented programming languages today, but on prototype delegation. Nevertheless, class inheritance can be simulated using prototype delegation if desired. ECMA-script is used to include calculated content into HTML pages, to accomplish various HTTP based network communication with a web server and to dynamically react on user interaction.

- **DOM :** to be able to alter the structure of an HTML document dynamically through ECMA-Script, various incompatible solutions were developed by competing browser developers. An effort called DHTML was launched to unify the way how content could be altered dynamically. The base for this is an application programming interface (aka.

[5] Java-Script is not the same as Java. The name has been chosen for marketing reasons, to benefit from the popularity of Java in Java-Script's early days. It was initially intended to be called Live-Script.

API) to access and modify the document. This application is called document object model, in short DOM. DOM provides a tree interpretation of the document where elements are nodes with lists of children and a parent node. The API is an official W3C recommendation. The standard contains the API as interface specification for Java, ECMAscript and for CORBA. It has now been adopted by most modern programming languages as a API to process HTML and XML content. The DOM API is implemented in most modern web browsers and provides access to the loaded document.

As mentioned earlier, visXcerpt is related to Xcerpt through a reversible transformation. Xcerpt itself provides an XML notation besides the more compact term notation introduced earler. To get a visXcerpt instantiation of an Xcerpt document, a transformation from an XML notation to something using the former web standards is necessary. Because there exists an XML dialect of HTML[6], an XML transformation language should be suitable for this task. Xcerpt (and hence visXcerpt) is suitable for this purpose, but some technical inconveniences with the prototype at the time of writing this diploma thesis discouraged the use of Xcerpt for a runnable prototype. The reasons was mentioned in the examples section earlier and is related to recursive function emulation. XSLT was used for the prototypic implementation of the *Xcerpt-to-visXcerpt* transformation.

3.3.2 Modelling the Prototype

This section will explain how far CSS is usable for the prototype implementation, why the XML notation of Xcerpt cannot be directly styled with CSS and what kind of HTML structures are necessary to represent Xcerpt. The need of a reversible transformation will be motivated and a generic mechanism for reversible transformations will be presented.

3.3.2.1 A Motivating Example

This section will roughly explain how a pattern element (called generic element) is translated into its visual counterpart. The real translation is slightly more complicated, because it contains further elements like buttons with event handlers and layout elements.

A first idea for the prototype was to use CSS on XML to specify a visual rendering for Xcerpt documents. For this purpose an XML aware browser could have been used. It turned out to be impossible, because default XML browsers cannot display an XML element name. The element name is a major information in visXcerpt terms that cannot be visually

[6]The XML dialect of HTML is called XHTML and is a W3C recommendation

omitted. Therefore a transformation is necessary, to extract information like element names and attribute names. The target language chosen to reflect the Xcerpt documents is HTML.

Spoken in abstract terms, the main concept of visXcerpt is to use nested objects to represent structure. The nested structure can easily be mapped to nested HTML `div`-elements with a class attribute, while the visual representation is controlled through the use of CSS. Borders around each element and background colouring can easily be made with CSS. With CSS margins and paddings of the `div`-elements a pleasant feeling of nesting and indenting is given.

While the nested structure is easy to represent, the fact that the label is outside of the nesting is problematic: For the element name, a `span`-element can be used. It can be placed along with other `span`-elements, because it is not interpreted as a text block. This feature is needed for the information hiding aspect. Unfortunately the `span` cannot be placed inside the `div`, because it would not be located outside, as specified. For this reason a generic visXcerpt element needs to be translated in two elements, one for the label and another one for the content.

By now it has been explained, why CSS and Xcerpt are not enough to implement the proposed visXcerpt language and a rough sketch about how a generic translation of pattern elements may take place have been given. If a translation for Xcerpt is necessary, a translation back from visXcerpt to Xcerpt will be necessary, at least to be able to delegate program evaluation to an existing Xcerpt implementation. A straightforward solution would be to write two translations. This approach has a major drawback: functional dependency of both transformations is very high and therefore modification on one side leads to inconsistency on the other side. The solution presented now, prevents this dependency of transformation and backward transformation through introduction of a meta-model and a generic backward transformation process.

3.3.2.2 The lXr XML meta-model

The lXr XML meta-model is an annotation mechanism for XML documents. It is used to embed information into a host document about how to generate a structurally similar, but simpler document based on the content of the host document. The lXr XML meta-model is explained independently from the visXcerpt language. It is also independent of HTML. An lXr processor is able to transform the host document without any further program or document needed to specify the transformation. In this way, the lXr transformation process differs from Xcerpt, XSLT and other transformation languages that usually involve two documents in the transformation process: one as data source, the other one as transformation program. Nevertheless, the processor can easily be written as Xcerpt or XSLT transformation pro-

gram. The lXr annotations are specified using XML attributes, optionally in a separate namespace. This way lXr annotations usually do not disturb the application structure of the host document, it should be usable for most common XML based applications. This was an important property while developing the visXcerpt prototype, because it was not clear if the intended features could be modeled in HTML. In case of HTML related problems, SVG[7] would have been used. The lXr meta-model is also applicable to SVG.

3.3.2.2.1 Properties of host Documents

The lXr XML meta-model makes some assumptions about the host document:

- Every node of the derivable document is represented by one or many sibling nodes in the host document. If many host nodes represent a derivable node, the host nodes are in a sequence not interrupted by host nodes of other derivable nodes.
- If derivable nodes are siblings (or attributes are from the same element), their supporting host lists are also siblings
- CDATA, element and attribute names, attribute values and namespaces of the derivable documents are always CDATA nodes of the host documents. They are isolated CDATA nodes without further irrelevant CDATA.
- The host nodes of a child node of a derivable node are descendants of one of the host nodes representing the derivable parent node.

The properties of the host document make clear that the host document always has more nodes, than the derivable document. The lXr annotation is done with the three attribute classes described now.

3.3.2.2.2 lXr Annotation Attributes

The first attribute class indicates the location of CDATA that represents textual information of the derivable document. They have a numerical value that indicate the position of an element containing the CDATA as sole child node. They indicate the position in the list of child nodes of the element containing the annotation. The concrete attribute names are:

- `ANPos` : Indicates an element containing an **attribute name**.
- `AVPos` : Indicates an element containing an **attribute value**.

[7] SVG is a markup language used for specification of vector graphics. It also contains grouping constructs usable to model nested structures. It is a W3C recommendation.

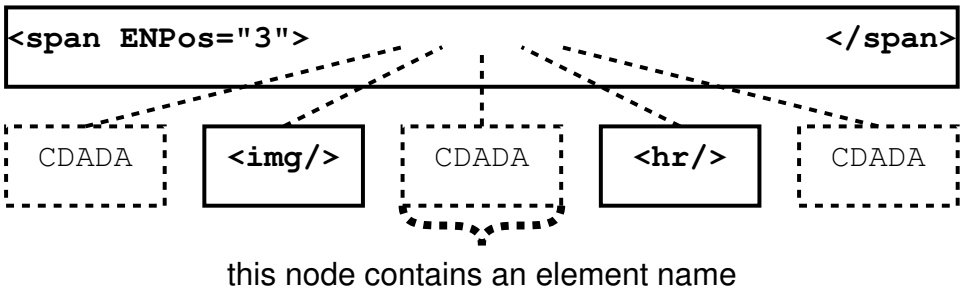

Figure 3.37: This example illustrates how a position attribute can be used to determine the CDATA containing a hosted element name.

- `ENPos` : Indicates an element containing an **element name**.
- `CDPos` : Indicates an element containing **CDATA**.
- `NSPos` : Indicates an element containing a **namespace URI**.

The host document structure allows a list of siblings to represent one derivable node. For this purpose, an annotation attribute to determine the list is necessary. Because two host element lists are always disjunct, an indicator for the bounds of the lists is used. Because it is possible that a list contains, starts or ends with some CDATA that cannot be annotated with attributes, the bounds are indicated at a suitable node in the list. The meta-model is named after those bounding indication attributes – the lXr attributes. The l stands for the length of the list on the *left* side of the element containing the lXr attribute, the r for the length on the the *right* hand side. There exist three different lXr attributes and the X is a place holder for three different characters:

- `lEr` : Indicates a list of nodes hosting an **element**.
- `lAr` : Indicates a list of nodes hosting an **attribute**.
- `lCr` : Indicates a list of nodes hosting some **CDATA**.

For example an attribute `lEr="3,5"` indicates that the host node list consists of 9 nodes: the actual one containing the lEr attribute, the three previous sibling nodes and five following siblings. This way it is easy to implement the previously described generic visXcerpt element consisting of one `div`-element and one `span`-element and optionally up to three CDATA sections of whitespace for HTML source code indenting.

With the lXr attributes and the textual positioning mechanism it is still possible to specify host documents where "digging past" the direct child nodes is possible. The second condition on host documents prevents this: if all sibling nodes are hosted by sibling lists, there exists a common

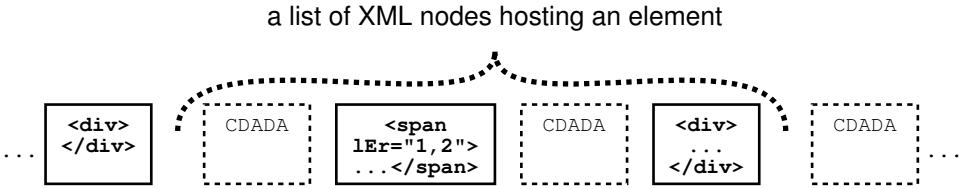

Figure 3.38: This illustration explains how lXr attributes define lists. The *span* element is annotated with an *lEr* attribute, the curly brace marks the hosted list.

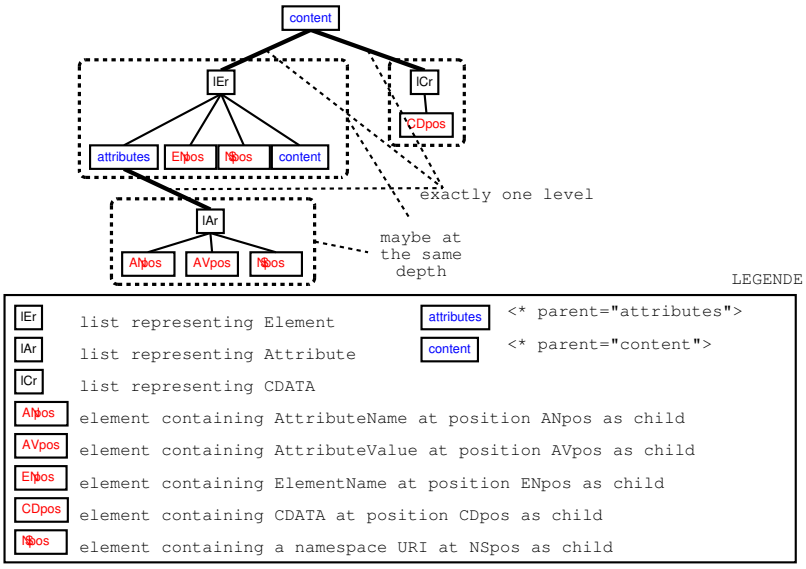

Figure 3.39: The figure illustrates the structure of the portion of a host document describing an element. An arbitrary amount of irrelevant nodes can be added, and there is no constraint on the node names for lXr annotated nodes. The thin lines depict a descendant relation, oval boxes indicate an optional sibling relation (optionally overriding the descendant relationship).

parent of those nodes. Therefore there is exactly one descendant below an element-host-node-set that is the parent of the children's host lists and one descendant indicating attributes as child nodes. In the context on an host node document fragment, the path down to an element marked as parent must not contain any other parent marked node.

- `parent="attributes"` : This marks an element that is the parent of node lists hosting **attributes**.
- `parent="content"` : This marks an element that is the parent of node lists hosting **elements and CDATA**.

With those ten attribute types, it is possible to embed information about a derivable XML document into a host document. It is possible to model

elements, CDATA, attributes and namespaces for elements and attributes. XML comments, processing instructions and entities are not expressible using the lXr meta-model. The following section will describe the role of the lXr meta-model for the visXcerpt prototype, which reaches beyond reverse transformation and is of major relevance for the dynamic aspects.

3.3.2.2.3 lXr and the visXcerpt prototype

One advantage of the lXr meta-model for the visXcerpt prototype is the generic reverse transformation. If Xcerpt programs are translated into visXcerpt programs that are enriched with lXr annotations, modifications of the transformation program are independent of the backward transformation. The consequence is, that the transformation step needs to be enriched with lXr attributes. There would be no advantage beyond functional dependent transformation and reverse transformation, if the lXr annotation would have to be manually supported. It turned out, that transformation programs that fulfill certain properties can be automatically enriched with lXr annotations. For this a transformation program has been written, enriching the construction patterns of a transformation program with lXr annotation attributes. A non-enriched transformation is called a *theme*. The term "theme" is widely used in desktop environments and graphical applications for a package or file that controls the overall look-and-feel of the applications.

As mentioned, a visXcerpt *theme* is a transformation program that is transformed into another transformation program with lXr annotated construction patterns. This lXr enrichment program is independent of the theme that has to be translated, it only depends on the transformation language the theme is based on. The transformation language chosen for theme authoring was a slightly modified and reduced subset of XSLT. The theme specification language has not been formally specified, because this is out of the focus of this thesis. It is implicitly specified through the themes and the transformation program used to enrich the themes with lXr annotations. Xcerpt and visXcerpt would have been the first choice solution as theme specification language, especially as an educative example for boot strapping of visXcerpt, but unfortunately, the current Xcerpt prototype lacks some features needed for transformations based on structural recursion. The features are actually being developed. In the chapter about examples, there is an example describing the use of structural recursion in Xcerpt. An Xcerpt based theme would use similar rules.

Another striking feature of the lXr meta-model is the usefulness for the editing process of visXcerpt. Because the whole information about the Xcerpt document is inherent in the visXcerpt document, which itself is an HTML document, modification of the HTML document is reflected in the Xcerpt document. It is in fact possible to edit a visXcerpt document using an HTML editor, if new components are generated by copying, pasting and

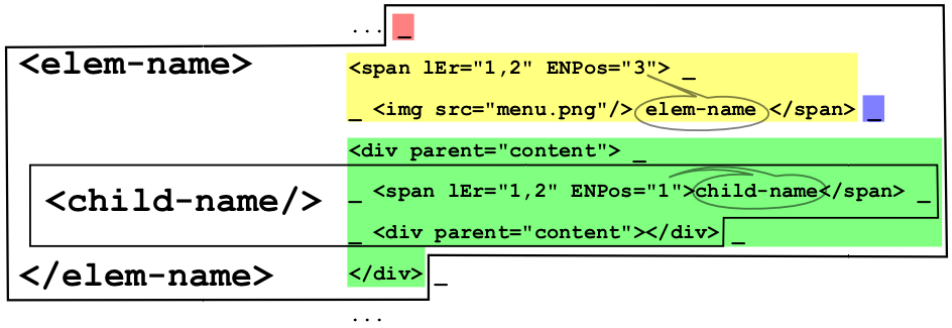

Figure 3.40: The thick XML code on the left side is hosted by the HTML code on the right side. The different colours indicate different DOM nodes (only at one level), either elements or text nodes. The HTML code is comparable to the code produced by the prototype (style and ECMA-script attributes have been omitted).

modifying existing components. By copying and pasting the lXr annotations are preserved and the modifications are delegated to the hosted Xcerpt document. A modern web browser provides the ability to modify a document in memory using ECMA-script and the DOM API. HTML documents using ECMA-script to modify themselves through the DOM are known as DHTML (short for dynamic HTML) documents. visXcerpt documents are DHTML documents. The HTML `onClick`, `onMouseOver` and `onMouseOut` event-handlers are used to invoke ECMA-script functions modifying the visXcerpt document. The ECMA-script functions needs lXr annotation awareness. If e.g. an element is to be copied, the responsible function needs to find out which DOM nodes are concerned. For this purpose an ECMA-script library has been written, providing lXr meta-model navigation, selection and query functions. Its functionality is similar to the DOM API's functionality, but additional awareness about hosting and hosted elements is supplied.

3.3.2.3 Assembling visXcerpt's Client Components

The most important components of a visXcerpt document are summarised as follows:

1. **The lXr annotated transformation of the Xcerpt documents** are enriched by event handlers that are mostly based on
2. **the generic lXr annotation aware ECMA-script library.** The library performs editing operations, reverse transformation and information hiding operations. The concrete look-and-feel of the visXcerpt document is controlled by
3. **various CSS documents.** These style sheets control the fonts, the colours, the margins, paddings and indenting and the borders of the

boxes representing elements.

Furthermore some objects relative to the whole visXcerpt document are necessary:

- menus for editing operations. They are displayed when an editing operation icon on an element is clicked. The context for the operations of the menu is the element whose icon was clicked. The visual location of the menu is near the editing operation icon (based on the screen location of the mouse event).

- icons assembled in a tool bar for loading, saving and helping.They are relative to the visual presentation context of the visXcerpt document (e.g. a frame or a window), therefore they are always visible in a tool bar and do not require the context menu.

All those components are assembled in a DHTML document. It was decided to keep the ECMA-script functions and the CSS expressions as external documents, because they are not in XML syntax and because they are immutable.

The current visXcerpt prototype provides a framework of useful client configurations, an Xcerpt result presentation and a minimal file selection document for the server-side stored repository of Xcerpt (or XML) documents. The Xcerpt result configuration is again strongly based on the pattern presentation of visXcerpt. A result consists of variable constraints. There may be an arbitrary amount of results for each variable. They are interpreted as disjunction. For this, a variable has an *or*-ed set of variable free patterns as result. visXcerpt's `OR` construct is used for the disjunction, each variable is represented by an element called result with an attribute called `variable-name`. The content is a disjunction or optionally directly a result, if only one result occurs.

A client configuration is a frame set of visXcerpt documents. Copy-and-paste operations can be performed between the documents of such a frame set. Currently three configurations exist:

- **Plain editor** : This configuration consists of only one visXcerpt document. It is not possible to use copy buffers from other documents. The plain editor is mainly used for screen shots, because it is easy to bookmark[8], resize and because it has the least screen usage overhead.

- **Two vertical frames** : This turned out to be the best compromise between screen size usage and flexibility. Two visXcerpt documents can be edited simultaneously, they share the same copy-and-paste buffer.

[8] Most web browsers behave strange while using bookmarks of frame sets.

- **Interactive Xcerpt query frames** : Three frames are available. One on the left is used for editing a visXcerpt program, two are stacked on the right side, and the bigger one contains a visXcerpt template document. It is configured to be non-editable (only the copy operation is active). This frame contains all visXcerpt components. It is intended to be used for copying new instances of visXcerpt components. Pasted into the editor frame on the left, they are editable again and can be specialised for the users query. Below is a smaller frame that is intended to contain a query. The tool bar provides a special icon for query execution. If clicked, the query is applied to the editor frame on the left. The result is presented in a new window with a plain editor configuration, thus it can be edited or saved easily.

Loading, saving, query execution and assembling of visXcerpt documents can not be performed by the web browser. Those operations are handled by the visXcerpt server explained in the next section.

3.3.3 The visXcerpt Server

The visXcerpt server is implemented using the *common gateway interface* (aka. CGI). CGI is a communication standard between web servers and server applications. Based on a given query, the applications are intended to produce result documents returned by the web server. The CGI uses environment variables and standard input to inform the application about the request, the standard output of the application returns the result document (including parts of the HTTP header, like the content type). The CGI is implemented on almost all common web servers.

Any Xcerpt program can be transformed to a visXcerpt program and vice versa without information loss. For this reason the shorter Xcerpt notation has been chosen for persistence. e.g. file storage. The first task of the visXcerpt server is to transform a request for an Xcerpt document into a visXcerpt document and to send it to a web client. Furthermore the server dispatches according to the users preferences the right visXcerpt transformation theme and the appropriate CSS document.

Two special requests to the visXcerpt server are

- **saving documents** : using a HTTP post request, the web browser sends the Xcerpt document and the intended filename to the server. The transformation from visXcerpt to Xcerpt is fulfilled by the web browser using the ECMA-script lXr library. The advantage of client side transformation is reduced data transfer – Xcerpt documents are about 8 to 10 times smaller than visXcerpt documents.[9] The server

[9]It is useful to reduce the data the client has to send to the server. Actual network

stores the document and returns notification about the success of storing the document. The notification is displayed in a small window opened by the web client. For security reasons, the document loading and storing is restricted to a special folder on the servers file system.

- **evaluating a query on a program** : using the HTTP post request, the web browser sends the Xcerpt program and an Xcerpt pattern used as query for the program. The visXcerpt server dispatches the program and the query to an integrated Xcerpt prototype. The result of the Xcerpt prototype is transformed using the user's preferred visXcerpt theme and is returned to the client. The client displays the result in a new window.

For the server side XSLT transformations (used for lXr annotation of the themes and for theme application) xsltproc, a libXSLT[10] based XSLT processor has been used. To glue all components (XSLT processor, Xcerpt prototype, user preferences handling and file storage) together, the scripting language Python has been used. Beside the author's personal preference for using python, the availability of a python library wrapping the Xcerpt prototype with a comfortable API was a reason for choosing python as implementation language. The visXcerpt server consists of less than 500 lines of python code in 13 modules.

technologies indicate a drift towards asymmetric network bandwidth, where providers have large down streams to their customers, while customers have small up streams back to their provider. Reducing the data to transfer through the upstream is therefore important.

[10]This WWW resource is located at `http://xmlsoft.org/`

Chapter 4

Conclusion

In this thesis a descriptive visual query and transformation language has been developed. Beyond the static presentational aspects of the language, the process of editing and authoring visXcerpt documents has been analysed. A prototype has been developed based solely on open web standarts. Concluding remarks regarding the language conception, editing operations and the implementation of the prototype will be given now.

4.1 Language Conception

The main difference of visXcerpt to most visual languages is the fact that the visual graph structure is no longer made up of edges and nodes. As long as the structure of the data to be visualised is a tree, nesting of elements is used to represent the structure. If the data structure is not expressible as a graph, a graph is constructed from a spanning tree with the missing edges moved into the hyperspace. This approach has proved to be powerful for large structures. If structures are represented by graphs based on nodes and edges, they tend to become so cluttered that they are unreadable. Various graph layout algorithms exist trying to maximise the readability by reducing the amount of line crossings. Unfortunately the computational complexity is usually high and the document size (its extension in its visual dimensions) often grows rapidly depending on the number of edges.

A drawback of a hyperlink-based approach is the difficulty of grasping printed representations. For established use of modelling languages this would be a major drawback, because their documentation nature often makes printing necessary. For a query and transformation language, the interactive requirements are not a drawback. Further aspects, like the template-based instantiation of new content, enforce the interactive use for the sake of "learning by doing".

With XML documents as data to represent, various mechanisms for

graph construction are given:

1. ID/ID-ref mechanism
2. internal links based on XLink
3. the use of XPath expressions
4. RDF
5. external links in a XLink base.

While the first three methods imply a spanning tree on the graph data, methods 4 and 5 do not imply a spanning tree structure. Because usually more than one spanning tree can be derived from a graph, further investigation would be necessary to find out which one to use as spanning tree for a visXcerpt document. The ranking of potential spanning trees is likely to be application dependant. Another approach could be to provide an editing mechanism to interactively extract a spanning tree.

4.2 Editor Conception

The editor concept proposed for visXcerpt documents in this thesis is a simple adaption of the copy-and-paste paradigm. To a certain extend the editing process of visXcerpt documents can be expressed in terms of prototype based object oriented languages. An element copied can be seen as the prototype of a new element pasted. The template document is like a library of prototypes. In contrast to prototype based object oriented systems the relationship to the prototype is lost. Modifications of an element are not reflected as differences to the prototype of an element. Research in the field of editing prototype based object oriented knowledge bases could be enlightening in the research for further editing mechanisms.

Extending the set of editing operations has to be handled with caution. While more features may provide easy handling of special use cases, the overall handling gets more complicated. The easy handling of user interface menues is indirectly proportional to the amount of operations located in a menu – short menues can be grasped quicker. In the HIG (GNOME Human Interface Guidelines) [benson02hig] it is suggested to restrict menues to 15 entries and to group the content to groups of 2-5 entries. The values are empirically founded. Because editing operations are accessed through a context menu and because their use is frequent, good menu design is a major concern. A good editing model should hence consider balancing the length of the menu with the complexity of the editing process.

Another amelioration of the editing step is the improvement of the template document. A dynamic template, based on a generalisation of the

edited document, is likely to be a useful tool. This is due to a tendency to reuse the same elements with slight modifications more than once in the same document. A dynamic template mechanism could provide copies of elements found in the editor document, without child elements or with fixed depth. Those elements with cropped sub-trees are comparable to an empty form that can be filled with new content. Generating templates based on schema information is a more powerful template mechanism, because thus many valid elements with arbitrary complexity can be provided.

4.3 Prototype Conception

The use of browser technology for the prototype was advantageous because no effort was necessary to lay out the visual components – the render engine of the web browser fulfills this task. The transformation aspects, actually implemented using XSLT, are interesting to implement using Xcerpt (or excangeably visXcerpt) as soon as the required features are available in the Xcerpt prototype. Nevertheless, a reimplementation in Xcerpt should reach beyond the current state of visXccerpt and introduce a type system for the lXr annotation mechanism. Many features can only be implemented satisfactorily based on a type system. Some of those features are:

- variables in attributes
- horizontally arranged elements in a disjunction

The reason is that the concerned components are treated differently depending on their parent context. Because the lXr annotation mechanism can generally be used for any XML application, a generic type system, comparable to XML Schema, should be included. The editing operations must be re-implemented to be type-aware. Cutting and deleting content must be prevented, to avoid schema violation. Content pasted may need transformation to be valid at the given context. An lXr schema must therefore be enriched by prototypes or constructors for visual element representations.

A further important feature that is not implemented in the prototype, is the support of graph structure based on references. The implementation aspects can be compared to those used for the variable handling. A generic graph handling mechanism also relies on a type system, because knowledge about reference element types or reference attribute types needs to be specified. Variables should be reimplemented using that a graph mechanism for the sake of code factorisation.

4.4 Need for empirical evaluation and user feedback

An evaluation of visXcerpt by non-computer scientists or programmers is advisable. This is the only way to rate one of visXcerpt's goals – the orientation towards casual users. A short informal discussion with some interested non-scientists about this thesis led to the conclusion that either people are not aware of structure in databases, or that they are fixed on an imperative programming style and have problems with the semantics of rules and variables in visXcerpt. The first category are consumers that browse the web, use E-Mail or write text documents. Those people are used to fill in prepared forms or search fields in WWW search engines. They are used to get the result presented in a predefined and fixed way. Searching consists of providing key words, structure is a given feature of the query result or a given query form. Consumer evaluation of visXcerpt should be focused on using and altering predefined query patterns, hiding aspects like joins, disjunctions/conjunctions, resource locations and chaining. For the result presentation an aspect to evaluate is, how far the generic visualisation of semi structured data is comprehensible to consumers.

Second category of potential casual users are those people who have some experience in using computers, but no experience in using logic programming languages. Those users usually have problems to understand the nature of variables in logic programming, because their view to variables is tainted by the imperative paradigm – variables are names for storage areas, values can be read and assigned. To solve the problem, metaphors have to be found that describe the behaviour of visXcerpt variables in a less abstract way. The unix concept of pipes is a salient example for a good metaphor. The word pipe intuitively indicates flow, blocking behaviour if one end waits, buffering and disastrous consequence, if one endpoint is removed (or killed in unix terminology). For Xcerpt and visXcerpt the following abstract concepts are candidates to be aliased with new metaphors:

- variable
- as-construct
- chaining (forward or backward chaining)

Even different metaphors for different use cases or user communities are likely to be useful. This approach and a methodology for its visualisation is explained in [fishwick00aesthetic]. The choice of spicy metaphors are considered by some as syntactic sugar and as an exchangeable feature, but they are of major importance for comprehension of the underlying semantics and

certainly support wide acceptance. Comprehension leads to trust and trust leads to acceptance. Even if an indispensable property of query, programming and transformation languages is the executability on a machine, we may not forget that programs have to be understood by humans.

Bibliography

[abiteboul00data] S. Abiteboul, P. Buneman and D. Suciu.
Data on the Web: From Relations to Semistructured Data and XML.
Morgan-Kaufmann , in year 2000

[abiteboul97querying] Serge Abiteboul.
Querying Semi-Structured Data.
ICDT , in year 1997 , page 1-18

[baru98features] C. Baru, B. Ludäscher, Y. Papakonstantinou, P. Velikhov and V. Vianu.
Features and Requirements for an XML View Definition Language: Lessons from XML Information Mediation.
http://www.db.ucsd.edu/publications/xmas.html
in year 1998

[benson02hig] Calum Benson, Adam Elman, Seth Nickell and Colin Z. Robertson.
GNOME Human Interface Guidelines (1.0).
http://developer.gnome.org/projects/gup/hig/

[bry02gentle] François Bry and Sebastian Schaffert.
A Gentle Introduction into Xcerpt, a Rule-based Query and Transformation Language for XML.
http://www.pms.informatik.uni-muenchen.de/publikationen/PMS-FB/PMS-FB-2002-11.pdf
in year 2002

[bry02the] François Bry and Sebastian Schaffert.
The XML Query Language Xcerpt: Design Principles, Examples, and Semantics.
http://www.pms.informatik.uni-muenchen.de/publikationen/PMS-FB/PMS-FB-2002-7.pdf
in year 2002

[bry02towards] François Bry and Sebastian Schaffert.
Towards a Declarative Query and Transformation Language

for XML and Semistructured Data: Simulation Unification.
http://www.pms.informatik.uni-muenchen.de/publikationen/PMS-FB/PMS-FB-2002-2_short.pdf
in year 2002

[bry02xpath] Dan Olteanu, Holger Meuss, Tim Furche and François Bry.
XPath: Looking Forward.
http://www.pms.informatik.uni-muenchen.de/publikationen/PMS-FB/PMS-FB-2002-4.pdf
in year 2002

[bry03pattern] Sacha Berger, François Bry and Sebastian Schaffert.
Pattern Queries for XML and Semistructured Data (revised version).
http://www.pms.informatik.uni-muenchen.de/publikationen/PMS-FB/PMS-FB-2003-1.pdf
in year 2003

[burnett99visual] Margaret M Burnett.
Visual Programming.
Encyclopedia of Electrical and Electronics Engeneering (John G. Webster, ed.) , in year 1999

[catarci93fundamental] Tiziana Catarci, Giuseppe Santucci and Michele Angelaccio.
Fundamental Graphical Primitives for Visual Query Languages.
Information Systems , in year 1993 , number 2 , volume 18 , page 75-98

[ceri99xmlgl] Stefano Ceri, Stefano Ceri, Ernesto Damiani, Piero Fraternali, Stefano Paraboschi and Letizia Tanca.
XML-GL: A Graphical Language for Querying and Restructuring XML Documents.
Sistemi Evoluti per Basi di Dati , in year 1999 , page 151-165

[citrin94design] W. Citrin, M. Dhoerty and B. Zorn.
The design of a completely visual OOP language.
M. Burnett, A. Goldberg, and T. Lewis, editors, Visual Object-Oriented Programming: Concepts and Environments. , in year 1995

[comai00graphical] Sara Comai.
Graphical Query Languages for Semi-Structured Information.
EDBT PhD Workshop , in year 2000

[consens89expressing] Mariano Consens and Alberto Mendelzon.
Expressing structural hypertext queries in GraphLog.
Second ACM Hypertext Conference , in year 1989 , page 269-292

[consens90graphlog] Mariano Consens and Alberto Mendelzon.
GraphLog: a visual formalism for real life recursion.
ACM Symp. on Principles of Database Systems , in year 1990 , page 404-416

[consens92visualizing] Mariano P. Consens, Isabel F. Cruz and Alberto O. Mendelzon.
Visualizing Queries and Querying Visualizations.
SIGMOD Record , in year 1992 , number 1 , volume 21 , page 39-46

[consens93hy] Mariano Consens and Alberto Mendelzon.
Hy+: A Hypergraph-based Query and Visualization System.
ACM SIGMOD '93 International Conference on Management of Data , in year 1993 , page 511-516

[cruz90a] M. Cruz, Alberto Mendelzon and P.Wood.
A graphical query language supporting recursion.
9th ACM SIGMOD Conference on Management of Data, San Francisco, CA , in year 1990 , page 323-330

[erwig-xml2000] Martin Erwig.
XML Queries and Transformations for End Users.
XML 2000 , in year 2000 , page 259-269

[erwig00visual] Martin Erwig.
A Visual Language for XML.
IEEE Symp. on Visual Languages , in year 2000 , page 47-54

[fatdog01quick] Howard Katz.
A quick intro to XQuery.
`http://www.fatdog.com/XQuery_Intro.html`

[fishwick00aesthetic] Paul A. Fishwick.
Aesthetic Programming.
`http://www.cise.ufl.edu/~fishwick/tr/00/leo.pdf`
in year 2000

[john94visual] H. John Reekie.
Visual Haskell: A First Attempt.
`http://ptolemy.eecs.berkeley.edu/~johnr/papers/visual.html`
in year 1994

[kepser02proof] Stephan Kepser.
A Proof of the Turing-completeness of XSLT and XQuery.
`http://tcl.sfs.uni-tuebingen.de/~kepser/papers/xsltxq.pdf`
in year 2002

[lee01semantic] Tim Berners-Lee and Ora Lassila.
The Semantic Web.
http://www.sciam.com/article.cfm?articleID=00048144-10D2-1C70-84A9809EC588EF21

[may01logic] Wolfgang May.
A Logic-Based Approach to XML Data Integration.
http://www.informatik.uni-freiburg.de/~may/lopix/

in year 2001

[munroe00bbq] Kevin D. Munroe and Yannis Papakonstantinou.
BBQ: A Visual Interface for Integrated Browsing and Querying of XML.
VDB , in year 2000

[murray98kaleidoquery] N. Murray, N. Paton and C. Goble.
Kaleidoquery: A Visual Query Language for Object Databases.
Proceedings of Advanced Visual Interfaces , in year 1998

[omg01uml] OMG.
OMG Unified Modelling Language Specification.
http://www.omg.org/technology/documents/formal/uml.htm

in year 2001

[papakonstantinou99framework] Y. Papakonstantinou, B. Ludascher and P. Velikhov.
A Framework for Navigation-Driven Lazy Mediators.
http://www.db.ucsd.edu/publications/webdb-domvxd.pdf

in year 1999

[petropoulos02qursed] Yannis Papakonstantinou, Michaelis Petropoulos and Vasilis Vassalos.
QURSED:Querying and Reporting Semistructured Data.
ACM SIGMOD , in year 2002

[pietriga01xvt] E. Pietriga, V. Quint and J.-Y. Vion-Dury.
VXT: A Visual Approach to XML Transformations.
ACM Symposium on Document Engineering , in year 2001

[pietriga01xvt2] E. Pietriga and J.-Y. Vion-Dury.
VXT: Visual XML Transformer.
IEEE Symposium on Visual/Multimedia Approaches to Programming and Software Engineering (Human Centric Computing Languages and Environments) , in year 2001

[puigsegur97from] Jordi Puigsegur, W. Marco Schorlemmer and Jaume Agusti.
From Queries to Answers in Visual Logic Programming.
Visual Languages , in year 1997 , page 102-109

[rumbaugh98uml] J. Rumbaugh, I. Jacobson and G. Booch.
The Unified Modeling Language Reference Manual.
Addison Wesley , in year 1998

[schl01b] T. Schlieder.
ApproXQL: Design and Implementation of an Approximate Pattern Matching Language for XML.
http://www.inf.fu-berlin.de/inst/ag-db/publications/2001/report-B-01-02.ps
in year 2001

[w3c02use] W3C.
XML Query Use Cases.
http://www.w3.org/TR/xmlquery-use-cases/

[w3c02xpath-wd] W3C.
XML Path Language (XPath) 2.0.
http://www.w3.org/TR/xpath20/

[w3c02xquery-wd] W3C.
XQuery 1.0: An XML Query Language.
http://www.w3.org/TR/xquery/

[wadler02xquery] Philip Wadler.
XQuery: a typed functional language for querying XML.
http://www.research.avayalabs.com/user/wadler/topics/xml.html

[xmlschema0] W3C.
XML Schema Part 0: Primer.
http://www.w3.org/TR/xmlschema-0/

[zhang01visual] Kang Zhang, Da-Qian Zhang and Yi Deng.
A Visual Approach to XML Document Design and Transformation.
Proceedings of 2001 IEEE Human-Centric Computing Languages and Environment , in year 2001 , page 312-319

[zloof77query] M.M.Zloof.
Query-by-Example:a data base language.
IBM System Journal , in year 1977 , number 4 , volume 16 , page 324-343