INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen

Oettingenstraße 67 D–80538 München

Grouping Structures for Semistructured Data: Enhancing Data Modelling and Data Retrieval

Sebastian Schaffert

Diplomarbeit

Beginn der Arbeit:	20. Oktober 2000
Abgabe der Arbeit:	18. April 2001
Betreuer:	Prof. Dr. François Bry
	Dipl. Inf. Dan Olteanu

Erklärung

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

München, den 10. Mai 2001 Sebastian Schaffert

Zusammenfassung

Markup-Sprachen für semistrukturierte Daten wie XML gewinnen an Bedeutung als Mittel für Datenaustausch und -speicherung. In dieser Diplomarbeit wird eine Erweiterung für das semistrukturierte Datenmodell vorgeschlagen, die es erlaubt, Daten mit speziellen Beziehungen zu gruppieren und dadurch mehr Semantik auszudrücken. Ein Datenmodel wird vorgeschlagen und der Einfluß auf Pattern Matching und Lokalisierung untersucht. Außerdem werden Algorithmen vorgestellt, die mit auf dieser Weise erweiterten Daten umgehen können.

Abstract

Markup languages for semistructured data like XML are of growing importance as means for data exchange and storage. In this thesis, an enhancement for the semistructured data model is proposed that allows to group data with special relationships, thus allowing to express more semantics. A data model is suggested and the implications on pattern matching and localization are investigated. Furthermore, algorithms are presented that make use of such enhanced data.

Danksagung

TEXT

Contents

Part I

Grouping Constructs for Semistructured Data

Chapter 1

Introduction

1.1 Introduction

Languages for semistructured data (SSD) like XML have by now gained widespread acceptance as a data exchange format (see [Abi97] for an introductory work). Also growing is the importance of the SSD data model for database management. Query languages like XQuery [xqu01] and its predecessors QUILT [RCF00] and XQL [Rob99] are visible signs of this development.

In this paper, an enhancement called *grouping constructs* to the SSD data model is suggested. This enhancement allows to establish explicit semantic relationships between data items in semistructured databases. Usually these relationships are given either implicitly through the meaning of element names or are implemented in the application software processing the data.

In the first part, an overview about *grouping constructs* is given on two introductory examples. It is also discussed how these grouping constructs could be introduced in existing SSD implementations like XML. The first part finishes with a formal discussion of the proposed grouping constructs and the foundations for a declarative localization language (similar to e.g. XPath [xpa99]). This localization language not only copes with grouping constructs but also uses them for a more efficient localization.

Extending this localization language is the main topic of the second part, where data retrieval in SSD with grouping constructs is examined more thoroughly. Furthermore, a method for computing aggregated answers is introduced. The last chapter of this part covers the introduction of variables in SSD with and without grouping.

Some algorithms for computing the ideas that have been presented before are given in the third part.

The thesis finishes with an overview over related work and some suggestions for future research.

In this introductory chapter, an overview over semistructured data is given, followed by two examples to show the relevance of grouping constructs in two different applications.

1.2 Semistructured Data

In this section, a short introduction about semistructured data is given. The ideas presented here follow the proposals made in the introductory work [Abi97]. The reader may want to consult [Abi97] for a more thorough discussion.

Semistructured Data intends to fill a space between data that is completely structured (relational databases) and data that is unstructured (binary images, flowing text).

Naturally, this definition is very imprecise, as it may or may not apply to certain types of data. But this also shows the necessity of such data models.

An example that is proposed in [Abi97] are the well-known BibTeX files used for bibliographies in LaTeX documents. Obviously, these contain some structure, like the denomination of "authors", "year", etc. However, much of this information is optional. Thus, a completely structured data model would either be very inconvenient or very large to cover all of the possibilities.

[Abi97] proposes the following main aspects of semistructured data:

- *The structure is irregular.* Much of the information is heterogeneous or even missing.
- *The structure is implicit.* Often, the structure of the data only becomes meaningful together with the application.
- *The structure is partial.* The data may e.g. contain flowing text or bitmaps which are unstructured while other parts are well structured.
- *Indicative structure vs. constraining structure.* In completely structured databases like relational databases, the structure is always enforced to keep the data consistent. Together with the world wide web, it becomes increasingly important to be able to avoid strict typing.
- *A priori schema vs. a posteriori data guides.* Traditional databases with complete structure require to define a schema before even introducing any data. Semistructured data on the contrary allows evolving schemas.

[Abi97] furthermore suggest two different models for representing SSD. A lightweight model (OEM - [CGMH$^+$94]) and a heavyweight model (ODMG). Both are useful for certain kinds of applications.

For the purpose of this text it is sufficient to know that both models use directed graphs. In many cases, however, it suffices to restrict the model to trees. This is also done in this thesis.

Note that nowadays the most prominent representative of semistructured data is XML (eXtensible Markup Language) [xml00b]. Nonetheless there are other kinds of semistructured data, notably XML's precursor SGML [ISO86], but also data formats like BibTeX [Pat88] or LaTeX [Lam86].

1.3 Motivation

1.3.1 Example 1: Course of Studies

Consider the following example of a course of studies at the Munich University:

In the first 4 terms, some courses are optional while others are required. Thus there are **and** and **or** connections between courses.

Terms	Courses		
	Computer Sciences	Mathematics	Seminars
1	CS I	Algebra I **and** Analysis I	
2	CS II **and** Hardware Basics	Algebra II	
3	CS III	Graph Theory **and** Applied Analysis	Programming **or** System **or** Hardware **or** Logics
4	CS IV **and** Advanced Algorithms	Stochastic **or** Numerical Mathematics	

While this table reminds of a standard database item like e.g. a (non-first normal form) relation, the **and** and **or** connections show a very different semantics. Obviously, a data model is needed using which grouping constructs like the **and** and **or** connectives can be expressed and used during localization and data retrieval in general. In the following we will see that other grouping constructs are also desirable.

Let us first consider a similar example at a more abstract level:

Imagine that you want to store an information like $a \wedge (b \vee c)$. In a relational database model, this would be achieved by transforming this to its conjunctive normal form $(a \wedge b) \vee (a \wedge c)$ and then storing each of the conjuncts (a, b) and (a, c) in a relation R (informally, the $\wedge$ is between the columns, or attributes, while the $\vee$ is between the rows, or tuples, of the table). This would result in the relation $R = \{(a, b), (a, c)\}$.

However, this representation has two drawbacks:

- *redundancy*: The information about a is stored several times. This is inefficient in both, space and computation, and error prone for updates.

- *information loss*: The fact that b and c are **and** connected to the common item a might be important from a semantic viewpoint. This information has to be recomputed (i.e. the conversion to the conjunctive normal form has to be reversed).

With the SSD data model things are even worse: While the relational model at least has the connections **and** and **or** built-in (i.e. the **and** connections between the columns and the **or** connections between the rows), in the SSD data model it is not possible to express such information in an application independent manner.

As semi-structured data, the previous course of studies example could be expressed as follows (an XML syntax is used here to ease the understanding for the reader; any other SSD syntax would be possible).

```
<course_of_studies>
  ...
  <term>
    <number>4</number>
    <computer_sciences>
      <course>
        CS IV
      </course>
      <course>
        Advanced Algorithms
      </course>
    </computer_sciences>
    <mathematics>
      <course>
        Stochastic
      </course>
      <course>
        Numerical Mathematics
      </course>
    </mathematics>
    <seminars>
      <course>
        Programming Course
      </course>
      <course>
        System Course
      </course>
      ...
    </seminars>
  </term>
</course_of_studies>
```

In this excerpt, some information is missing: It is not expressed which courses are optional and which are required. A common solution would be to provide this information in an application dependent query interface.

However, this approach would not be portable since every application would have its own data format. This would be unfortunate because it is the idea of SSD to be application *in*dependent. Note that such semantic groupings occur frequently in data exchange (e.g. in e-commerce catalogs, bioinformatics databases [Kröar],etc.).

The proposal of this work is to add general constructs to the SSD data model so as to allow the grouping of elements according to certain properties ("grouping facets"), thus trying to overcome the above mentioned deficiencies of the relational and the standard semi-structured model.

With grouping facets the introductory example can be represented as follows. Again, the XML syntax has been retained. Also note that grouping can be expressed through other entities than through elements (see Chapter 2 for a discussion on the topic).

```
<course_of_studies>
  ...
  <term>
    <number>4</number>
    <computer_sciences>
      <AND>
        <course>
          CS IV
        </course>
        <course>
          Advanced Algorithms
        </course>
      </AND>
    </computer_sciences>
    <mathematics>
      <OR>
        <course>
          Stochastic
        </course>
        <course>
          Numerical Mathematics
        </course>
      </OR>
    </mathematics>
    <seminars>
      <OR>
        <course>
          Programming Course
        </course>
        <course>
```

```
          System Course
        </course>
        ...
      </OR>
    </seminars>
  </term>
</course_of_studies>
```

1.3.2 Example 2: Addressbook

Now consider an example that is quite different than the previous example, but shares nonetheless the necessity to have some abstract grouping constructs.

Imagine an address book consisting of entries for persons/organizations. Each entry has a name, one or more postal addresses, any number of phone numbers (including zero) and any number of email addresses (including zero).

Again, an XML syntax is chosen for convenience. In this XML syntax a sample from this address book could look as follows:

```
<abook>
  <entry>
    <name>Sebastian Schaffert</name>
    <address>
      <street>Fruehlingstrasse 25</street>
      <city>83278 Traunstein</city>
    </address>
    <email>schaffer@informatik.uni-muenchen.de</email>
    <email>wastl@wastl.net</email>
  </entry>

  <entry>
    <name>Mustermann GmbH</name>
    <address>
      <street>Hauptstrasse 12</street>
      <city>88888 Irgendwo</city>
    </address>
    <address>
      <street>Marktplatz 10</street>
      <city>98765 Woanders</city>
    </address>
    <phone>555-12345</phone>
    <phone>566-54321</phone>
  </entry>
</abook>
```

As can be seen, there are entries that have several email addresses. What does this mean for someone that wants to send a message to this person? Should the message be sent to both email addresses, only to the first or only to the second? Or is it no use sending the message via email since the person only checks his email occasionally?

The second entry, a company, doesn't have an email address. The message should be sent anyway, but to which address? Or is it preferable for this company to call by phone?

In most cases, this kind of information belongs to each individual entry, to the data. Certainly, this could be implemented again in an application-dependent manner, but obviously this is not very desirable since the idea of semi-structured data is to carry such structural information in the data itself.

In the next section, grouping is investigated systematically and other grouping facets than **and** and **or** are suggested.

After that, a matching technique for localization in a context with grouping constructs is discussed.

1.4 Grouping Facets

Since our extension *groups* data items and adds additional information to the already-existing structure, it is called **grouping constructs**. The individual kind of grouping is called **grouping facets**. The following grouping facets are suggested:

- *connector*: for grouping items with the connectors AND, OR and XOR (the connector facet has one of the properties "AND", "OR" and "XOR").
- *order*: for specifying whether items are ordered or not (properties "ordered", "not ordered")
- *repetition*: for specifying whether items of the same type may be repeated or not (properties "repetition allowed" and "repetition not allowed").
- *selection*: for allowing a query to select/match a certain number of the items (property "n to m")
- *exclusion*: for excluding certain items (property "excluded")
- *depth*: for allowing a pattern to span several levels in a matched tree (property "n to m").[1]

Grouping facets can be of importance in three areas: database modeling, query patterns/schemas and answers to a query.

An informal discussion with extensive examples for each grouping facet can be found in Appendix 13.

[1]Allowing infinity as value for m allows to express the classical quantifiers "*", "+" and "?" as "n to m" facets

Not all of the mentioned grouping facets fit equally well to databases and to patterns/schemas. While e.g. the connector facet may be of relevance in both databases and patterns/schemas, the exclusion facet makes sense for patterns/schemas only.

In this paper we deliberately impose the following restrictions on grouping facets:

- the data model is currently limited to trees[2]
- only one grouping facet can be specified for a group of nodes
- the specified grouping facet always applies to all immediate children, not all ancestors

The rationale for these restrictions is the focus on the novel issue. An extension is possible (and desirable) in the future.

Note that XML-Schema [xml00a] and some of its precursors (XML-Data [xml98], DDML [ddm99], etc.) all had a few constructs similar to some of the above mentioned grouping constructs. However, they were only used for grouping in a schema, not in the data. Furthermore they were lacking a systematic treatment of grouping constructs.

[2]Extensions to DAGs and forests do not pose principal problems

Chapter 2

Representation of Grouping Constructs in tree based SSD models

There are two possible basic approaches to introduce grouping constructs in SSD tree representations: One is using attributes and the other is adding grouping nodes to the tree.

As the most common representation of tree based SSD models is the XML syntax nowadays, some of the considerations and advantages/disadvantages discussed in this section are closely related to some of the properties of XML. Other SSD models might require slightly different approaches.

These two representations both have their advantages and disadvantages and are discussed in the next sections.

Afterwards, some considerations about the expressive power of these approaches are made.

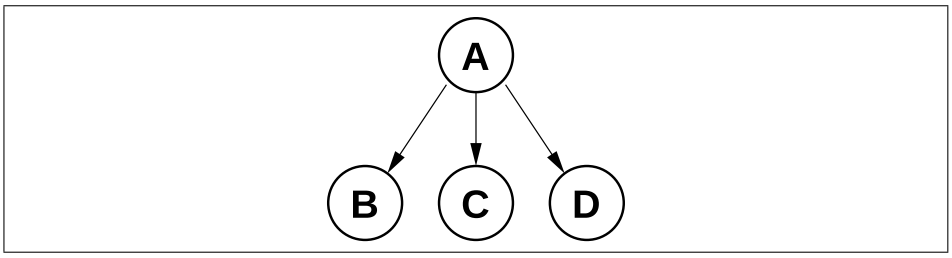
In this text, the attribute approach is chosen for the representation of grouping constructs, thus deliberately restricting the possible groupings. However, the ideas presented in this thesis are general enough so that they apply to both kinds of representations.

2.1 Representation through Attributes

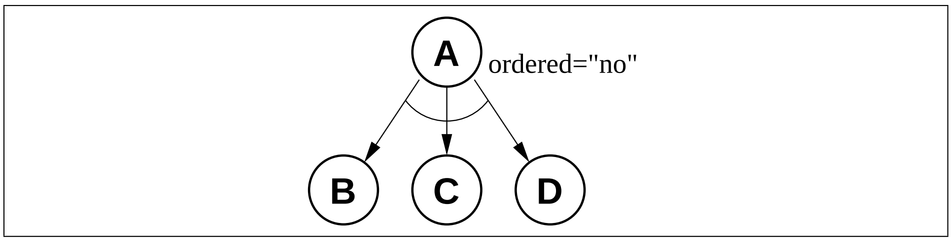
The first possibility is to represent the additional grouping information through attributes carried by the elements that specify the kind of grouping that is done.

For sibling relationships like the order and connector facets, the attributes can be carried by the parent element, specifying a relationship between all child elements. Adding these attributes to the siblings themselves isn't useful as this would require to use some reference to other sibling elements.

Example 2.1.1
Consider the following tree structure:



Now imagine that you want to provide the grouping information that the child elements should be considered unordered. Using attributes, this could be expressed like this:



This representation has several advantages:

- it is simple
- it doesn't modify the tree structure
- implementation of localization might be easier than with the other approach

However, there are disadvantages that are also worth considering:

- the attributes that are used require that the grammar of existing (XML-) applications has to be changed so that a document that uses the application can make use of them
- it is necessary that the special attributes become part of the (XML-) data types so that they may be used in any application
- the expressive power is less than in the second approach (see below)
- combining different facets will most likely require special handling in the localization algorithms

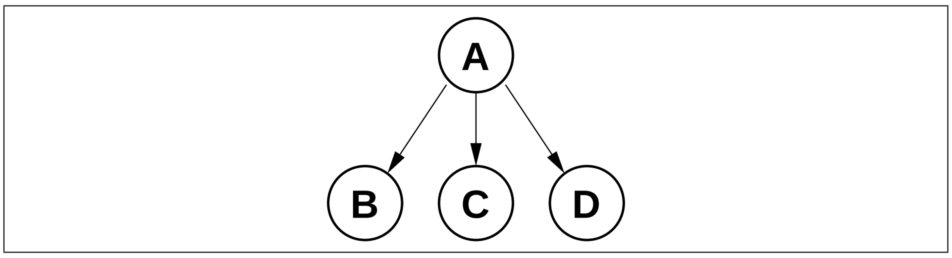
2.2 Representation through Additional Nodes

A different approach would be to add "special" nodes to the data tree that are used just to express the grouping and carry the necessary information in attributes. Actually, this can be considered an extension of the attribute approach since the attributes are still the same, only it is now possible to use a structure similar to "brackets".

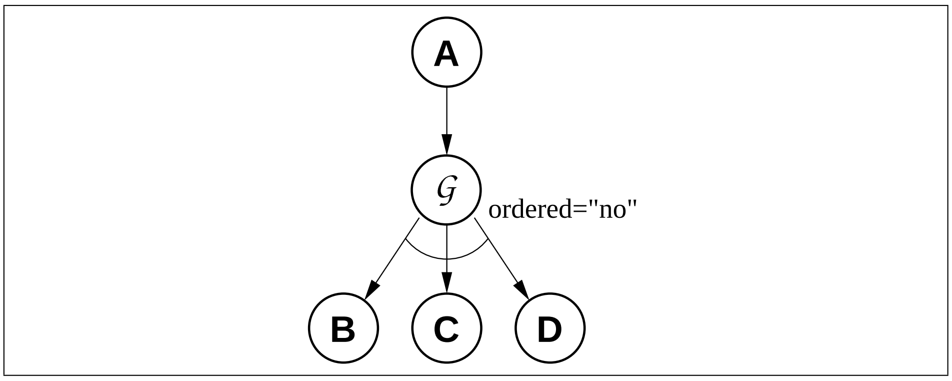
Sibling relationships can be expressed by adding the affected siblings to a specific grouping node and providing the grouping information in attributes of that node.

In parent-child relationships, the additional element can be placed between the (affected) children and the parent node.

Example 2.2.1
Again have a look at the previously used example:



The grouping that provides the information that the nodes are not ordered could be provided like this, the $\mathcal{G}$ representing the grouping node:



The advantages of this approach are that the expressive power is bigger (see below) and it is probably easier to implement the combination of grouping facets.

However, the disadvantages are also complementary to the advantages of the attribute representation:

- the representation is more complicated
- it is not possible for a node to be in two different groups
- it modifies the tree structure and thus the document becomes less intuitively readable
- localization with grouping elements could become difficult

Note that in XML, the last problem can be overcome by using namespaces which can be ignored by applications.

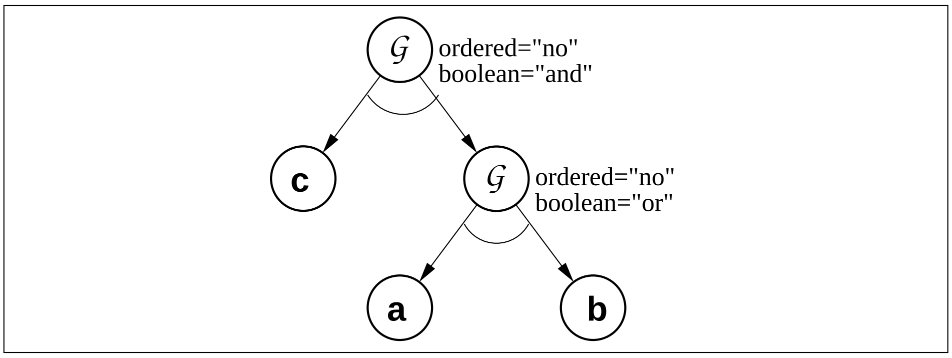
2.3 Considerations about Expressive Power

When comparing the two approaches, one important criterion is the expressive power of each representation. On first look, both seem to have the same possible applications.

However, on a closer look it is possible to recognize that the attribute approach doesn't allow to express some of the information that can be expressed with the element representation: With the latter it is also possible to only group a subset of the child nodes. This becomes important if it is intended to combine several of the above mentioned grouping facets.

Example 2.3.1
Imagine that you want to express a relation of $((a \lor b) \land c)$ between the child nodes of a node "root" (The braces already indicate that several groupings are necessary).

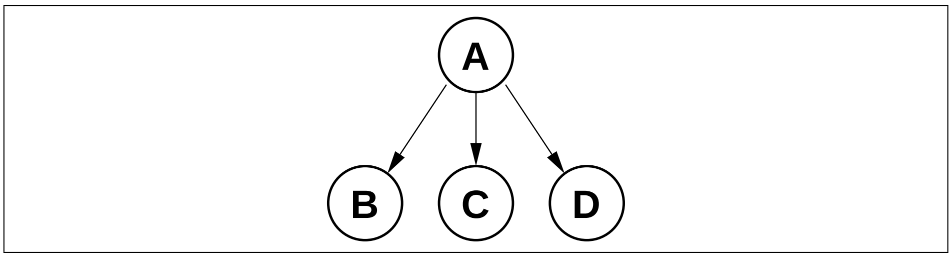
The appropriate tree structure using the node approach would look like this:



Such a representation obviously cannot be achieved when using the attribute approach.

However, the previous considerations only cover very simple cases of facet combination. For more sophisticated combinations, the element approach can become increasingly complicated since most of the facets have "different dimensions", i.e. the different groupings are very likely independent from each other.

Example 2.3.2
Imagine the following tree structure:



Now consider that you want to express that the elements B,C and D should be treated ordered and that there is an "XOR"-relationship between B and C while C and D are "AND"-related.

A representation with the element approach is no longer possible since this would require to add the child nodes to several group elements, thus breaking the tree structure.

Since such relationships require a graph structure but the data only provides a tree structure, it is obvious that it is not possible to express this additional information directly in the data with "easy" means.

As this would require the use of a graph structure and combination of grouping facets will not be treated in this thesis, solutions to this problem are not investigated any further.

Chapter 3

Data Model: Trees with Grouping Constructs

In this chapter, the semantics of grouping facets and the relationships between them will be defined.

This chapter provides a formalism for semi-structured databases with grouping structures. After introducing a syntactic representation, the semantics will be defined.

Note that in this data model, we simply use attributes to represent grouping constructs, not the additional nodes proposed in Chapter 2. However, for the concept this is of no relevance.

3.1 Syntactical Representation of Grouping Constructs

In this document, we assume that semi-structured databases are of a tree- or tree-like structure. The first definition will cover the **elementary data tree** which can be used to represent such tree-structured databases. This definition will then later be extended to an **enriched data tree** (**data tree with grouping**) also covering the grouping constructs.

The trees used here are always *node-labelled*, i.e. the nodes are carrying the labels. Another common representation is to use *edge-labelled* trees. As it is easy to convert one form into the other, it is just a matter of taste to choose the one or the other.

The definitions given here suffice for the scope of this paper. Some concepts like "tree", "forest" and "graph" are used in this thesis with their standard meaning in graph theory which can be found in many sources; some introductions are in [Jun94] and [Knu97] (pages 308-402). Tree matching has also been addressed extensively in [Kil92].

Note that the *elementary data tree* can be considered as a syntactic as well as a semantic definition. Syntactically, it just represents the definition for a tree structure. Semantically, it organizes data in a special way and therefore gives meaning to it.

In contrast, the *enriched data tree* itself only represents syntactical structure. The semantics for it are given in section 3.2.

In semi-structured databases like XML documents it is also possible to build graph structures that are not limited to trees. This is achieved by using pointers (like XLink and XPointer in XML; see [xpt00]).

In this approach, this possibility is left out consciously. There are several kinds of links and often it is not obvious how to interpret them (see e.g. [FS00], page 2). Since the focus of this document is to define a complex set of grouping structures, it is convenient to treat such pointers just as "syntactic sugar" and use the simple tree structure as a base.

Note however, that the data model and localization techniques suggested here are general enough so that they can also be extended to graph structures if this might be necessary.

3.1.1 Elementary Data Trees

We will first introduce *elementary data trees* for representing tree-structured databases. The next definitions are derived from definitions in [FS00] and [Wad00].

In the following, let **Vertices** denote a set of vertices, and **Labels** a set of string values used for labelling nodes. Furthermore, let $\mathbf{Lists}(X)$ denote the set of all permutation lists over some set X.

Definition 3.1.1 (elementary data tree)
An ***elementary data tree*** *DT is a tuple $(Nodes, name, children, root)$ such that*

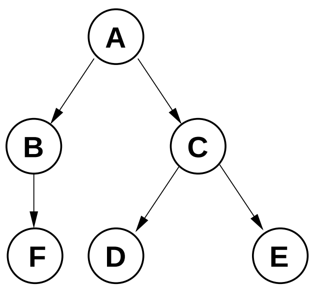
- *$Nodes \subset \mathbf{Vertices}$ is a (finite) set of vertices. Elements in $Nodes$ are called "node".*
- *$name : Nodes \rightarrow \mathbf{Labels}$ is a mapping from nodes to their element names*
- *$children : Nodes \rightarrow \mathbf{Lists}(Nodes)$ is a function mapping nodes to a list of their child nodes*
- *$root \in Nodes$ is a distinguished element called the root node with $\forall N \in Nodes : root \notin children(N)$.*
- *$\forall N, N' \in Nodes, N \neq N' : children(N) \cap children(N') = \emptyset$*

This is in most respects a quite usual definition for a tree with one exception: The child elements are always ordered. This is required because the data will always be ordered in the *syntax* (assuming a reader in the western hemisphere, i.e. from left to right, from top to bottom). The property that the *semantics* of the data is unordered will be achieved through the interpretation and matching methods that are used (i.e. “unorder” is a property of the semantics and not of the syntax of the data).

Notations. In this article, data trees will also be written as $A(B_1, \ldots, B_n)$ meaning a tree with root node A and subtrees $B_1, \ldots, B_n$ in the given order (like in a *list*) and $A\{B_1, \ldots, B_n\}$ denoting a tree with root node A and the set of subtrees $B_1, \ldots, B_n$ in any order (like in a *multiset*).

The following example illustrates definition 3.1.1.

Example 3.1.1
Have a look at the simple tree and its representation as an ***elementary data tree****.*



$Nodes = \{\_A, \_B, \_C, \_D, \_E, \_F\}$
$name(\_A) = "A", \ldots, name(\_F) = "F"$
$children(\_A) = [\_B, \_C]$
$children(\_C) = [\_D, \_E]$
$children(\_B) = [\_F]$
$root = \_A$

This example is very simple. In a real data tree nodes will share the same name or leaf nodes containing the same data. Nonetheless, each of them will be an individual node, thus the denotation with underscore.

Note that a node in graph theory denotes a single item that is possibly connected through edges with other nodes while in semi-structured databases a node usually denotes an element together with its context, i.e. a whole subtree. In this document, the notion of a node in graph theory is chosen, since this allows a much easier specification of the abstract syntax and the semantics of the grouping facets.

Terminology

Some terms that are frequently used in this thesis are introduced for elementary data trees in this paragraph. Most terminology is in analogy to [Knu97] and other sources about graph theory.

For each node $N \in Nodes$, if $children(N) = \emptyset$, N is called a **leaf node**. All other nodes (i.e. all nodes where $children(N) \neq \emptyset$) are called **inner nodes**.

Example 3.1.2
In example 3.1.1 the leaf nodes are $\{D, E, F\}$ *and the inner nodes are* $\{A, B, C\}$.

Also, each node has some “relatives”: Given some node N with $children(N) \neq \emptyset$. The nodes in $children(N)$ are called **children** of N. For any node $M \in$

$children(N)$, all other nodes in $children(N)$ are called **siblings**. The node N is called the **parent** of M.

Sometimes it will be necessary to talk about several trees. A **forest** is an unordered set of zero or more disjoint trees.

3.1.2 Grouping Facets

The **elementary data tree** will now be extended to an **enriched data tree** by adding the possibility to add **grouping facets**.

First, it will be necessary to define the possible grouping facets and the values they can take. For an informal description of these grouping facets please refer to Appendix 13.

Definition 3.1.2 (set of grouping facets)
*For the following sections, let **G** be a set of the following grouping facets:*

- ***connector facet*** *- children are grouped with the boolean relationships AND, OR and XOR.*
- ***order facet*** *- specifies whether child elements are ordered or not ("ordered" and "unordered").*
- ***repetition facet*** *- restricts whether repetition of elements is allowed or not ("allowed", "not allowed")*
- ***selection facet*** *- specify how many of the children have to match; values*
  - *"n to m", n, m positive integers ≥ 0*
  - *"exactly n"*
  - *"some"*
  - *"any"*
  - *"all"*
  - *"none"*

  where for any node with n child elements

  - *"exactly i" is an abbreviation for "i to i"*
  - *"some" is an abbreviation for "1 to n"*
  - *"any" is an abbreviation for "1 to 1"*
  - *"all" is an abbreviation for "n to n"*
  - *"none" is an abbreviation for "0 to 0"*
- ***exclusion facet*** *- the child elements are excluded*

The *depth facet* is left out consciously. It will be introduced together with variables in 6.3 as the changes that are required are less if it is already the possible to represent variables (which cover things that are yet "unknown" like the depth in the depth facet).

3.1.3 Data Trees with Grouping

Applying those grouping facets to (elementary) data trees leads to the following definition:

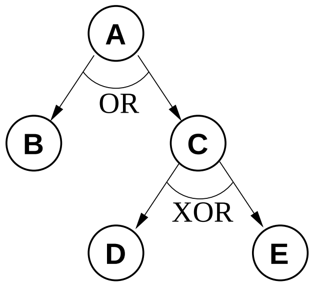
Definition 3.1.3 (data tree with grouping)
A "data tree with grouping facets" $(Nodes, name, children, grouping, root)$ *is a data tree such that:*

- $(Nodes, name, children, grouping, root)$ *is an elementary data tree*
- *grouping* : $Nodes \rightarrow \boldsymbol{G}$ *is a function mapping each node to a single grouping facet*

For shortness, data trees with grouping will also be called **enriched data trees** (which reflects the additional value compared to elementary data trees).

As the combination of grouping facets is not a trivial task, this definition limits the number of grouping facets that can be attached to a node to at most one. The combination of grouping facets is a possible topic for future research.

Example 3.1.3
The following data tree contains the grouping facets "XOR" and "OR".



$Nodes = \{\_A, \_B, \_C, \_D, \_E\}$
$name(\_A) = "A", \ldots, name(\_E) = "E"$
$children(\_A) = [\_B, \_C]$
$children(\_C) = [\_D, \_E]$
$grouping(\_A) = ["OR"]$
$grouping(\_C) = ["XOR"]$
$root = \_A$

3.2 Semantics of Data Trees with Grouping

The topic of this section will be to first define an interpretation of a data tree with grouping facets and then to provide the notion of a model for a given pattern.

3.2.1 Interpretation of Data Trees with Grouping

Grouping facets can be seen as defining for a single node and its children a set of possible interpretations (in form of different elementary data trees).

Since some of the grouping facets (depth facet) are defined much easier together with variables they are left out here.

enriched subtree $N_{\mathcal{G}}$	represented through subtrees
$N_{\epsilon}(M_1, \ldots, M_n)$	all $N\{M_1, \ldots, M_n\}$
$N_{AND}(M_1, \ldots, M_n)$	all $N\{M_1, \ldots, M_n\}$
$N_{OR}(M_1, \ldots, M_n)$	all $N\{P_1, \ldots, P_i\}$ with $\{P_1, \ldots, P_i\} \subseteq \{M_1, \ldots, M_n\} \backslash \emptyset$
$N_{XOR}(M_1, \ldots, M_n)$	all $N\{M_i\}$, $1 \leq i \leq n$
$N_{ordered}(M_1, \ldots, M_n)$	$N(M_1, \ldots, M_n)$
$N_{unordered}(M_1, \ldots, M_n)$	all $N\{M_1, \ldots, M_n\}$
$N_{repeat}(M_1, \ldots, M_n)$	$N\{M_1, \ldots, M_n\}, \ldots, N\{M_1, \ldots, M_1, \ldots, M_n, \ldots, M_n\}$
$N_{i\ to\ k}(M_1, \ldots, M_n)$	all $N\{P_1, \ldots, P_j\}$ with $\{P_1, \ldots, P_j\} \subseteq \{M_1, \ldots, M_n\}$,$i \leq j \leq k$
$N_{exclude}(M_1, \ldots M_n)$	all $N\{L_1, \ldots, L_k\}$ with $\{L_1, \ldots, L_k\} \cap \{M_1, \ldots, M_n\} = \emptyset$

Table 3.1: Interpretation of grouping facets

The table is read as follows: The left column shows an enriched subtree that contains a grouping facet. The right column gives the possible interpretations where the node N no longer contains the grouping facet $\mathcal{G}$ (but the child nodes M_i still have theirs).
Note that the subtrees in the left column are always ordered, while in the interpretation column they may be considered ordered or not. This reflects the property that unorder is only relevant in the semantics and the matching. To get the *real* interpretation of an unordered tree, it is necessary to consider *all* permutations of child nodes.

Note that the definitions presented here are not yet as precise as they could be. This is due to the introductory character of this thesis. A more formal approach is suggested in the chapter about "Ongoing Work", section 11.1.

The interpretation of a data tree with grouping constructs is given in two steps. The first definition shows how a single subtree with a given grouping facet at the root node represents a whole "forest" (i.e. several trees) without this grouping facet. In the second step, an algorithm is presented that generates the forest of elementary trees for a data tree with grouping, i.e. all grouping facets will be removed recursively and step by step.

Definition 3.2.1 (Interpretation of a Node with Grouping)
A given subtree $N(M_1, \ldots, M_n)$ with root node N and some grouping facet $\mathcal{G} \in grouping(N)$ represents the forest of subtrees with root node N and without $\mathcal{G}$, denoted as $\mathbf{I}_{\mathcal{G}}(N)$, as given in table 3.1.

$\mathbf{I}_{\mathcal{G}}(N)$ *is called the* interpretation *of the node N with grouping facet $\mathcal{G}$.*

Having defined the interpretation of grouping facets for individual nodes it is now possible to give an algorithm that generates all of the possible interpretations for a whole data tree with grouping facets.

Definition 3.2.2 (Interpretation of a Data Tree with Grouping)
Let EDT be an enriched data tree with grouping and $Nodes$ its set of nodes. The set of interpretations for the tree EDT, $\mathcal{I}(EDT)$, is generated as follows:

let N be the root node of EDT.

- *if N is a leaf node of EDT then the interpretation is just the node N ($\mathcal{I}(EDT) = \{N\}$). End.*
- *if N is an inner node of EDT*

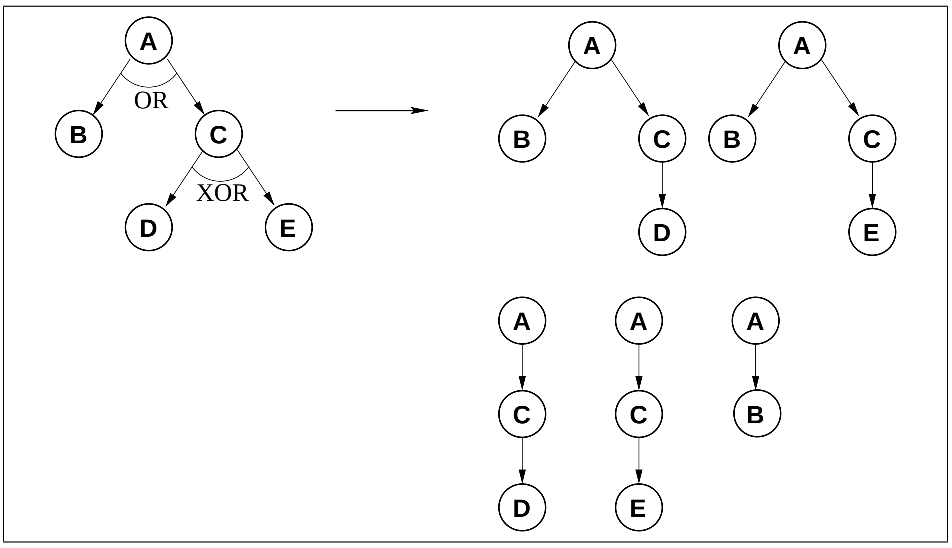
  1. *for each node $M \in children(N)$, generate the set of interpretations $\mathcal{I}(M)$ for the subtree with root M.*
  2. *generate the set of interpretations $\mathbf{I}(N)$ for N following definition 3.2.1. Initialize the working set WS with $\mathbf{I}(N)$.*
  3. *for all interpretations I in $\mathcal{I}(M)$: if there is an interpretation J in WS where $M \in children(N)$ (wrt J), add a new interpretation to WS that equals J where the subtree with root M in J is replaced with I.*
  4. *$\mathcal{I}(EDT)$ is then the working set WS restricted to elementary data trees.*

Informally, the set of interpretations is generated by just recursively applying the rules from definition 3.2.1 on the tree EDT beginning with the root node. The only thing that should be stated explicitly is that rule 3 combines each interpretation of the root node with all interpretations of the child nodes, which will generally produce a very large set (space complexity $O(n) \approx exp$).

Definition 3.2.3 (Model of a data tree with grouping)
Each of the elementary data trees $DT \in \mathcal{I}(EDT)$ is called a ***Model*** *of EDT, written $DT \models EDT$.*

An implementation that generates the elementary data trees for trees with the grouping facets AND, OR, XOR and ϵ is given in form of a Haskell program in appendix 14.1. For pragmatic reasons, this implementation doesn't generate all possible permutations of the child elements.

Example 3.2.1
A simple data tree with grouping facets and its set of elementary presentations is shown in the picture. Each of the elementary data trees is a model for the enriched tree.



One can easily see that the number of possible interpretations of a tree grows exponentially with the number of grouping facets in the tree because it is necessary to combine each of the grouping facets of the parent node with the ones of the child trees.

This also shows the expressive power of grouping facets: It is possible to express information that would usually require a large number of elementary data trees in one single data tree with grouping facets.

3.2.2 Databases and Patterns

Defining semantics for semi-structured data with grouping facets will be of little use if there is no possibility to use them for querying. Thus it is useful to talk about databases and patterns.

Semi-structured databases and patterns are just simple data trees. The difference comes from the usage. While a **database** will usually consist of one ore more data trees that *contain* some (useful) data, a **pattern** is used as a query to such a database that either matches (and possibly binds variables) with the database or not. Hence a **pattern** is just a declarative way to pose a query.

The usage also suggests a *different way of interpreting grouping facets*: In a pattern with grouping facets it is often sensible to allow additional child elements on the elementary data trees that are not part of the pattern, since usually a query is much smaller than the database that is queried. This is a very liberal strategy that implies that everything that is not explicitly excluded may or may not be present.

Notations. Matching that allows additional child elements on the database side will be called **liberal matching** while matching that requires that the pattern matches the database exactly will be called **strict matching**.

However, there is a conflict between the intention of some grouping facets and the liberal matching. An example is the "AND"-facet which may be inter-

enriched subtree $N_{\mathcal{G}}$	represented through forest of N without $\mathcal{G}$
$N_\epsilon(M_1,\dots,M_n)$	all $N\{M_1,\dots,M_n,R_1,\dots,R_m\}$
$N_{AND}(M_1,\dots,M_n)$	all $N\{M_1,\dots,M_n,R_1,\dots,R_m\}$
$N_{OR}(M_1,\dots,M_i)$	all $N\{P_1,\dots,P_i,R_1,\dots,R_m\}$ with $\{P_1,\dots,P_i\}\subseteq\{M_1,\dots,M_n\}\backslash\emptyset$
$N_{XOR}(M_1,\dots,M_n)$	all $N\{M_i,R_1,\dots,R_m\}$, $\{M_1,\dots,M_n\}\backslash\{M_i\}\cap\{R_1,\dots,R_m\}=\emptyset$, $1\le i\le n$
$N_{ordered}(M_1,\dots,M_n)$	$N\{M_1,\dots,M_n,R_1,\dots,R_m\}$ where the M_i have to appear in the given order
$N_{unordered}(M_1,\dots,M_n)$	all $N\{M_1,\dots,M_n,R_1,\dots,R_m\}$
$N_{repeat}(M_1,\dots,M_n)$	$N(M_1,\dots,M_n,R_1,\dots,R_m),\dots,$ $N(M_1,\dots,M_1,\dots,M_n,\dots,M_n,R_1,\dots,R_m)$
$N_{i\ to\ k}(M_1,\dots,M_n)$	all $N\{P_1,\dots,P_j,R_1,\dots,R_m\}$ with $\{P_1,\dots,P_j\}\subseteq\{M_1,\dots,M_n\}$, $i\le j\le k$, $\{R_1,\dots,R_m\}\cap\{M_1,\dots,M_n\}\backslash\{P_1,\dots,P_j\}=\emptyset$
$N_{exclude}(M_1,\dots M_n)$	all $N\{R_1,\dots,R_m\}$ with $\{R_1,\dots,R_m\}\cap\{M_1,\dots,M_n\}=\emptyset$

Table 3.2: Liberal interpretation of patterns with grouping facets

$R_1,\dots,R_m$ is a set of arbitrary subtrees ("rest"). As can be seen in the rules for `XOR` and `i to k`, the interpretations for *liberal* matching also require a "not-list", i.e. a list of children that are explicitly forbidden. This is not required for *strict* matching since then everything that is not there is forbidden.

preted as to restrict that the pattern must contain all of the children to match successfully.

To solve this problem, we can assume for now that the user is able to chose for each node whether to use strict or liberal matching. A more convenient solution will be presented in sections 4.1 and 5.2.3 where the problem will be addressed in terms of results of a query.

Table 3.2 extends the rules given in table 3.1 for liberal matching. (Remember that $N\{A_1,\dots,A_2\}$ represents the node N with the child trees A_i without considering order, i.e. the children on the left side of the table may be mixed with the additional children on the right side).

Generating a set of *all* interpretations for a pattern with liberal matching is obviously not possible since it will result in an infinite number of models. However, given a database and such a pattern, it is a useful and calculable extension. A relationship between patterns and databases will be discussed in the next section.

3.2.3 Matching Patterns with Databases

Matching

The intention of a relationship between databases and patterns is that there should be the possibility to decide whether a given pattern "matches" with a given database or not.

Obviously this is not sufficient for querying, since there should be more associated to the result than "true" or "false". Chapter 4 will be dedicated to the topic on how to produce richer results out of the pattern matching.

At this point, pattern matching illustrates the semantics of data trees with grouping facets. It is sufficient here to just consider the two cases (i.e. results) "it matches" and "it does not match".

"Matching" for elementary databases is related to the model relationship:

Definition 3.2.4 (Matching for elementary databases)
A (liberal) pattern P ***matches*** *with an elementary database DB if the database is a model of it, written $DB \models P$.*

If the pattern is not intended to match liberally, there would be an easy way to calculate whether a pattern matches with a database or not: The database would have to be in the set of possible interpretations for the pattern, which can be generated using the algorithm presented in appendix 14.1.

However, this approach is neither efficient nor does it suffice for *liberal* patterns, since as stated above, the set of interpretations for the pattern is infinite.

Simulation for Graphs

Therefore, it is more convenient to use a definition for the model relationship that respects the structure of both the pattern and the database. The definition will use a technique called **simulation** which is, among others, being used in the book [ABS00]. The advantage is that simulation is a topic that has been extensively studied in other areas of computer sciences, so there is a large number of results and algorithms that might be useful for grouping facets as well (see e.g. [HHK96]).

Informally, the simulation just "walks down" the two graphs in parallel, checking whether for a node in the one graph there exists a corresponding node in the second graph while respecting the edges.

Notation. Given a binary relation $\mathcal{R}$ over the Cartesian product of two sets, a pair $(x, y) \in \mathcal{R}$ is also written as $x\mathcal{R}y$. Given a directed, labeled graph (V, E), each edge label l induces a binary relation $[l]$ on V, V, written $x[l]y$ for each pair (x, y).

Definition 3.2.5 (Simulation [ABS00])
Given Graphs $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$ and a set of labels L, a binary relation $\mathcal{R}$ on V_1, V_2 is a ***simulation*** *if it satisfies*

$\forall l \in L \forall x_1, y_1 \in V_1 \forall x_2 \in V_2 (x_1[l]y_1 \wedge x_1 \mathcal{R} x_2 \Rightarrow \exists y_2 \in V_2 (y_1 \mathcal{R} y_2 \cap x_2[l]y_2))$

On closer look on the definition, one can observe that this just describes a mapping between two (labelled) graphs that ensures that the same structure is kept.

Actually, simulation in [ABS00] is used for schema verification. This is not a disadvantage since a pattern could also be viewed as a schema for a database.

In the next paragraphs, this definition will be modified and extended to *data trees*.

Simulation for Elementary Data Trees

The simulation definition will be modified to fit with elementary data trees first. This definition will then be extended to reflect data trees with grouping facets.

A simulation for data trees has to take into consideration the differences between the labelled, directed graphs used in definition 3.2.5 and the unlabelled data trees.

While in the original definition, the "matching criterion" is the name of the label in the two trees, in data trees the matching criterion should be the name of the node (which is irrelevant for the former). Since this doesn't generate any edge-relation, it will also be necessary to include the *children*-function of a node.

Furthermore, as data trees are trees, the simulation for data trees is *rooted*, i.e. the roots have to be in the simulation ([ABS00]).

Also, since we want to also implement the simulation later, it is useful to have a constructive instead of a declarative definition.

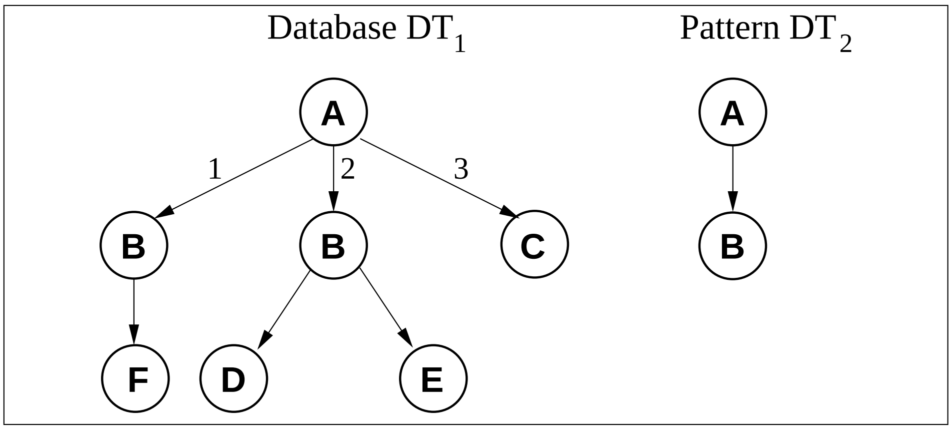
Definition 3.2.6 (Elementary Simulation)
Given two elementary data trees DT and DT' with the set of nodes $Nodes$ and $Nodes'$ respectively. A binary relation $\mathcal{R} \subseteq Nodes \times Nodes'$ on DT and DT' is called an ***elementary simulation*** *if it can be generated as follows:*

- *let $n \in Nodes$, $n' \in Nodes'$ be the root nodes of DT and DT' respectively. $(n, n') \in \mathcal{R}$ if $name(n) = name(n')$*
- $\forall n_1, n_2 \in Nodes\ \forall n_1' \in Nodes'\ (n_1 \mathcal{R} n_1' \wedge n_2 \in children(n_1))\ \exists n_2' \in Nodes'\ (n_2' \in children(n_1') \wedge name(n_2) = name(n_2')) \Rightarrow n_2 \mathcal{R} n_2'$

If $\mathcal{R}$ is an elementary simulation on the two data trees DT and DT', we shall write $DT \xrightarrow{\textbf{sim}}_{\mathcal{R}} DT'$.

The first rule provides a starting point for the simulation. Without it, the simulation relation would be empty, as the second rule always requires at least one element in the simulation to add further elements to it. It is possible to use a different pair than the two root nodes as the starting point and thus providing non-rooted simulations. In this paper, however, this possibility is not pursued any further (see also section 4.3).

Example 3.2.2
The next figure shows two elementary data trees DT_1 and DT_2.



The branches at the first level in the database DT_1 are numbered so they can be differentiated easier. The two B will be denoted B_1 and B_2 according to the branch number, but the name is still B.

Using Definition 3.2.6, three simulations $\mathcal{R}, \mathcal{S}$ and $\mathcal{T}$ between DT_2 and DT_1 can be constructed: $\mathcal{R} = \{(A, A), (B, B_1)\}$, $\mathcal{S} = \{(A, A), (B, B_2)\}$ and $\mathcal{T} = \{(A, A), (B, B_1), (B, B_2)\}$.

One might already observe that this definition allows *liberal* matching, i.e. the nodes in the data tree DT' might have additional child elements that don't have a corresponding node in DT. This is also reflected through the arrow in the $\xrightarrow{\textbf{sim}}_{\mathcal{R}}$ notation.

Actually, *strict* matching would be achieved through *bisimulation* which enforces the simulation relation into both directions (i.e. from DT to DT' and vice versa). Bisimulation is a topic that is relevant in e.g. process algebra and schema verification. It will not be discussed extensively in this thesis as the use for matching is very restricted.

Obviously (liberal) pattern matching between an elementary database and an elementary pattern can be achieved thus, DT being the pattern and DT' the database. An implementation of this matching would traverse the pattern recursively from the root to the leaves, on the way trying to establish the simulation relationship $\mathcal{R}$.

A sample implementation in Haskell that performs this simulation for a pattern and a database in form of XML documents can be found in appendix 14.2.

Algorithms for tree matching will be discussed in ???.

Naïve Matching: Simulation for Data Trees with Grouping

For data trees with grouping, we will first present a simulation approach that seems straightforward for the task. In Section 5.2.1, we will see that this approach needs further refinement.

The extension of the elementary simulation for data trees with grouping will have to take into consideration the interpretations $\mathcal{I}_{\mathcal{G}}$ for the data trees.

Definition 3.2.7 (Grouping Simulation)
Given two enriched data trees DT and DT' with grouping facets and their set of nodes $Nodes$ and $Nodes'$ respectively. An elementary simulation $\mathcal{R} \subseteq Nodes \times Nodes'$ is a ***grouping simulation*** *on DT and DT' if it satisfies*

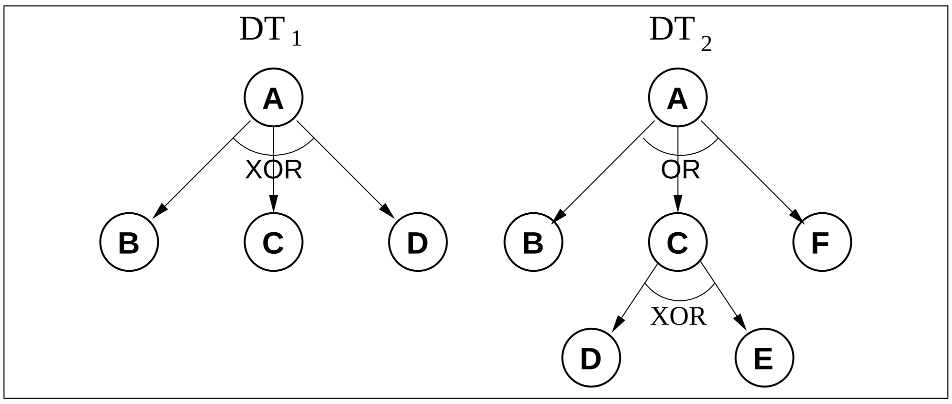
$$\exists\ I \in \mathcal{I}_{\mathcal{G}}(DT)\ \exists\ I' \in \mathcal{I}_{\mathcal{G}}(DT')\ (I \xrightarrow{\textbf{sim}}_{\mathcal{R}} I' \Rightarrow DT \xrightarrow{\textbf{sim}}_{\mathcal{R}} DT')$$

If $\mathcal{R}$ is a grouping simulation on DT and DT with grouping, then we shall write $DT \xrightarrow{\textbf{sim}^g}_{\mathcal{R}} DT'$ instead of $DT \xrightarrow{\textbf{sim}}_{\mathcal{R}} DT'$.

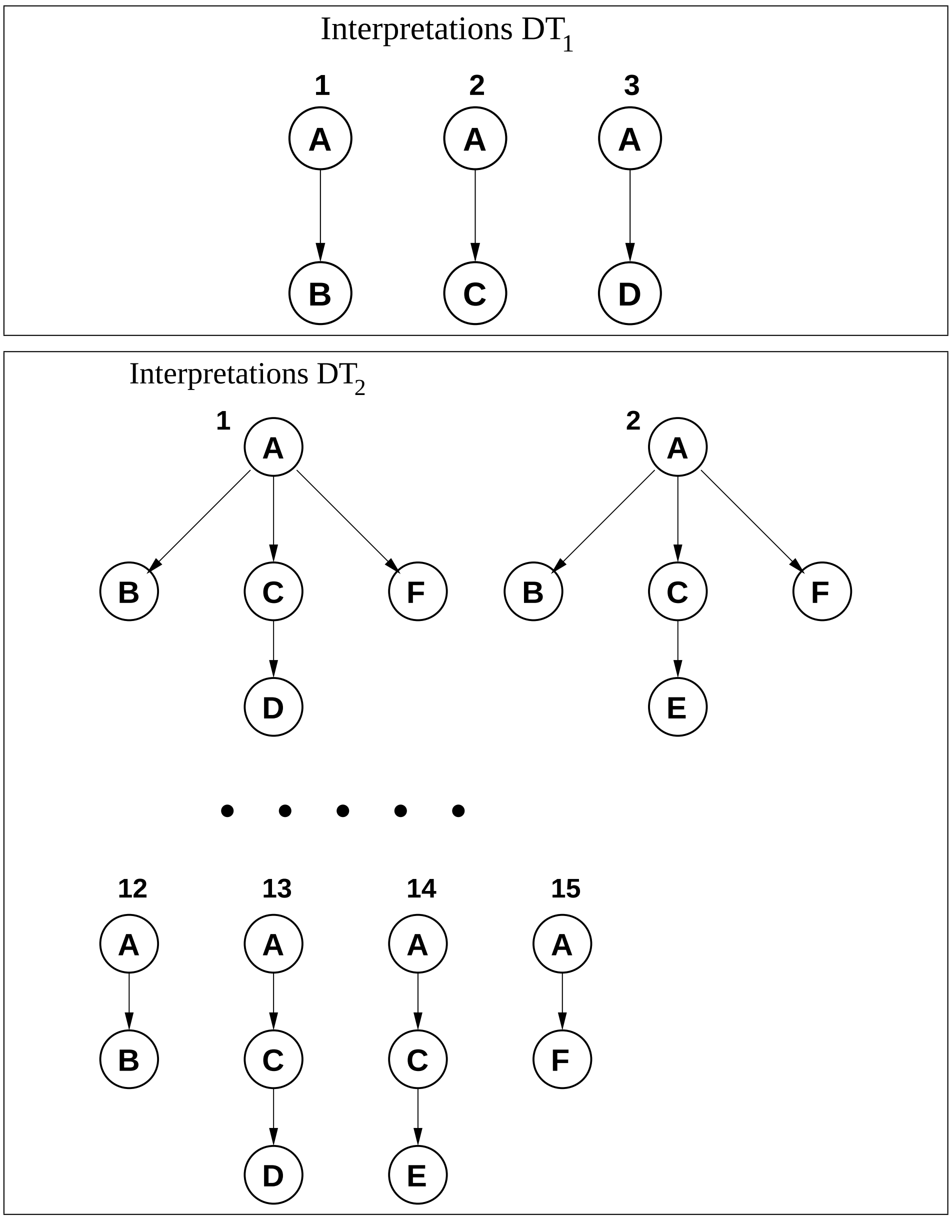
Informally speaking, the grouping simulation first generates the forest of elementary data trees for each of the two enriched data trees and then tries to find an elementary simulation between at least one elementary data tree of the first forest and one of the other forest.

The following example illustrates the application of Definition 3.2.7 on data trees with grouping.

Example 3.2.3
In the following figure we give two data trees DT_1 and DT_2. DT_1 can be seen as the pattern and DT_2 as the database.



Following definition 3.2.7 (grouping simulation), the interpretations for the two data trees are first generated. Of the 15 interpretations of DT_2, only the first two and the last 4 are given in this figure.



A grouping simulation on DT_1 and DT_2 corresponds to an elementary simulation on one interpretation of DT_1 and one interpretation of DT_2. In this example, there are three elementary simulations on interpretations of DT_1 and DT_2, $1 \xrightarrow{\mathbf{sim}}_{\mathcal{R}} 12$, $2 \xrightarrow{\mathbf{sim}}_{\mathcal{S}} 13$ and $2 \xrightarrow{\mathbf{sim}}_{\mathcal{T}} 14$. Note that the semantics of the XOR facet forbids elementary simulations such as $1 \xrightarrow{\mathbf{sim}}_{\mathcal{R}} 1$

It is important to note that in practice it is not necessary that all of the interpretations for each of the data trees is generated. Instead, this can be done step-by-step when recursively traversing the data trees beginning with the root (like in lazy evaluation of functional programming). Thus it is possible to avoid generating the interpretations for a node in many cases. This is comparable to a branch and bound search.

This may in many cases reduce the execution time of a simulation algorithm (although it doesn't reduce the complexity, of course).

An implementation for simulation with grouping facets is again presented in form of a Haskell program in appendix ???.

In section 5.2.1 we will show a much more efficient way to generate the matching result for data trees with grouping by just comparing the individual facets.

Strict and Liberal Matching on the Interpretation Level

In the previous sections, strict and liberal matching was used at the node level, i.e. to determine whether there might be additional child elements in the database or not.

However, there is also a different application of strict and liberal matching, when grouping constructs are used: In the previous definition, it was sufficient that there is at least one interpretation of the pattern that has an elementary simulation with at least one interpretation of the database. This could be called a *liberal* matching between interpretations.

Some applications might want to handle this differently: Either of the two $\exists$ could be replaced with a $\forall$, thus creating something like a *strict* matching between the interpretations.

Part II

Matching and Querying

Chapter 4

Matching and Querying

The matching with grouping facets that has been discussed so far has been used to show how the grouping facets may be used while the actual result wasn't investigated any further than "the pattern matches" or "the pattern does not match".

However, for database querying the main issue is to return some value (after all, the person posing the query will in most cases assume that his pattern already matches with the structure of the database).

The solutions presented in the next sections show how to use the simulation-based pattern matching to produce query results.

Querying is not covered completely, there is much more to it than just pattern matching. A full-fledged query language should provide things like the possibility to join results, simple data manipulation and selection/projection of data items. All this will not be discussed here.

Nonetheless, pattern matching is a first building stone for querying. Several characteristics are presented in this chapter. In the first step, variables like used in many query languages are introduced. Two properties are worked out and their application to data trees is investigated.

After that, a possibility to generate results out of simulation-based matching is suggested.

At first, however, we will again have a look upon patterns and databases and their relationships in terms of querying.

4.1 Patterns Redefined

It is now possible to refine the definition of patterns that was used in the previous sections with regards to querying.

4.1.1 Patterns as building stones for querying

Patterns in querying are used as the first step of fetching the data out of a database. If a pattern matches with the database, this will in general return the fragment of the database that matched, if it does not match, the query fails.

In the case of a match, the data will be further processed using commonly known querying techniques like joins, arithmetic operations, structural transformation, etc, but also again applying a refined pattern to the query result. For semistructured data, such techniques are covered for example in the works of the XML Query working group ([xqw]) and in languages like the very recent XQuery ([xqu01]), XQL ([Rob99]) or Quilt([RCF00]). Structural transformation is done e.g. in XSL ([xsl00]) where input trees can be transformed to different result trees.

Each of the mentioned query languages is based on some sort of localization language: XQL and XSL use XPath ([xpa99]) to locate data in the tree while SQL is based on the notion of tuples in the tuple calculus where the position or name of the attribute is relevant (these tuples are actually just “flat” patterns).

Hence the patterns are a way to *localize* the data in the database and return some reference to it. The data can then be manipulated in further steps. So the pattern can be seen as the *first building stone* of the query.

4.1.2 A new View on Patterns and Databases

In this context, patterns also have a slightly changed meaning than in the previous chapter.

First, the result from a pattern matching will no longer be “true” or “false”, but a fraction of the database. Thus, while the information in the pattern is mainly relevant for the matching process, the information contained in the database (including grouping facets) is mainly relevant in the result. Discussion about this will be the topic of the sections 6 and 5.

This also requires a slightly different view on the matching process which has the advantage that liberal matching can be used in all cases. This view will be explored in section 5.2.3.

4.2 A declarative Localization Language for SSD

Most data retrieval systems have some sort of localization language. For the most commonly used representation of semistructured data, XML, the localization language is XPath [xpa99]. XPath, however, is a *navigational* language as it requires to give the exact path to the data that is to be localized.

On the other hand, a declarative localization language (like the unification in Prolog [SS94]) would allow to specify the whole graph of possible choices, thus allowing to provide a context for the localization that is not restricted to the path. Also, it would be possible to retrieve all data items that are in some way "connected", not only one at a time.

Informally, using a tree pattern it is specified how the result should look like instead of how it can be found [Kil92].

Example 4.2.1
Imagine an XML document that just contains an address entry that should be displayed in a suitable HTML document.

```
<address>
  <given>Sebastian</given>
  <name>Schaffert</name>
  <street>Frühlingstrasse 25</street>
  <city>83278 Traunstein</city>
</address>
```

A typical XSLT stylesheet using XPath would probably look like this:

```
...
<xsl:template match="/">
  <HTML>
    <HEAD>...</HEAD>
    <BODY>
      <TABLE>
        <TR>
          <TD>Given Name:</TD>
          <TD><xsl:apply-templates select="/address/given"/></TD>
        </TR>
        <TR>
          <TD>Last name:</TD>
          <TD><xsl:apply-templates select="/address/name"/></TD>
        </TR>
        ...
      </TABLE>
    </BODY>
  </HTML>
</xsl:template>
...
```

Now consider the following program in Prolog, that is equivalent in terms of the transformation:

```
document(html(head(...),
     body(table(tr(td('Given Name:'),td(X)),
                tr(td('Last name:'),td(Y)),...)))) :-
     address(given(X),name(Y),Z1,Z2).
address(given(Sebastian),
        name(Schaffert),
        street(Frühlingstrasse 25),
        city(83278 Traunstein)).
```

A query for `document(X)` *would bind the wanted document to the variable* `X`*. This example can easily be extended to use more complicated structures found in the XSL transformation language.*

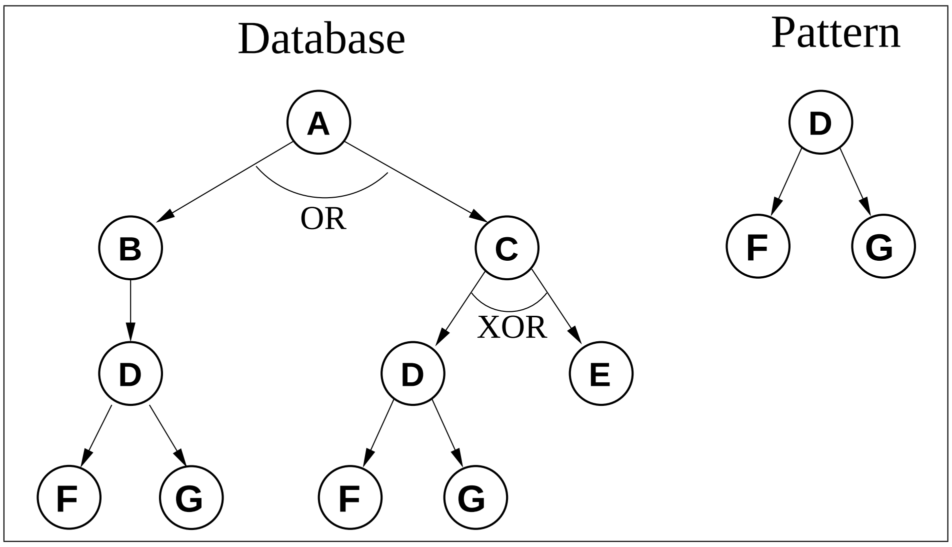
Using the presented tree-patterns and grouping constructs, we get such a *declarative localization language* for semistructured data. The intention of the next chapters is to examine how it is possible to use the simulation-based pattern matching for localization in semistructured data with and without grouping.

4.3 Rooted vs. Unrooted Matching

In Chapter 3.2, the simulation between patterns and databases was always *rooted*, i.e. the root nodes of the database and pattern were inserted into the simulation as a starting pair when the names matched.

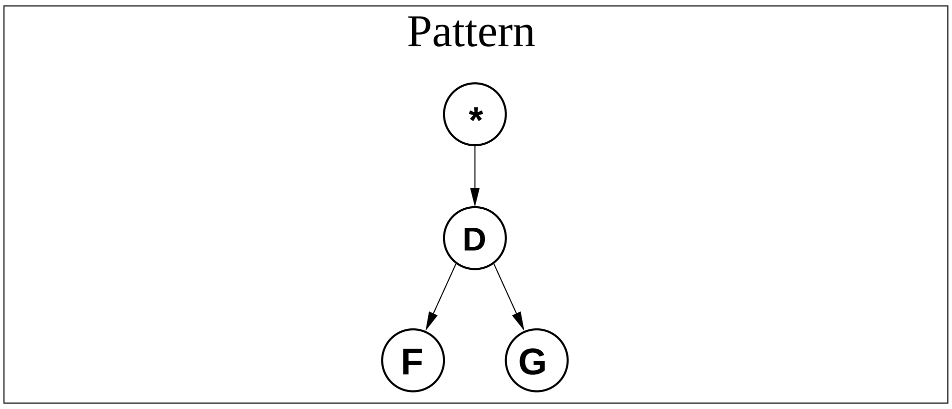
This is not the most general approach. One could also imagine that the pattern does not necessarily start at the root of the database but matches with some subtrees in the database, which would be desirable for many applications.

Example 4.3.1
The following figure shows a database and a non-rooted pattern for it that could match with two subtrees of the database:



It is possible to reflect this property in the rooted matching by using the depth facet. As we will see later (section 6.3), calculating the depth facet is not trivial and very time consuming. Furthermore, generating the result out of such a simulation produces results with much overhead (in form of the whole ancestors of the matched subtrees) which might reduce the effect of the selection.

Example 4.3.2
*Using the depth facet with a *-node, it is possible to express the previously used pattern like in the following figure:*



On the other hand, finding appropriate starting pairs for the non-rooted simulation is not very difficult and may have performance advantages compared to the calculation of the depth facet.

This topic is left open for further discussion here. While rooted simulation is used in this paper, it is not difficult to modify the simulation definitions to also include non-rooted simulations.

Chapter 5

Answer Semantics

Results for a matching can be of various forms. In this chapter, the topic of answer semantics is introduced in two steps.

First, only elementary data trees are considered. The easiest approach is to use the information that is already contained in the simulation relation. Since this has some serious drawbacks, we also introduce the notion of a "maximal simulation" for generating more convenient answers for matching on elementary data trees.

After that, we extend the results from the elementary matching to data trees with grouping. A more efficient approach to matching is introduced, allowing the matching to be calculated by just comparing the facets of database and pattern. This also allows to compute a combined answer for the otherwise many possible simulations. After that, some topics that are relevant for localization in practice are treated.

In Chapter 6, the answer semantics will be extended by introducing variables, thus providing even more capabilities.

5.1 Answer Semantics for Elementary Data Trees

5.1.1 Simulation as Result

So far, simulation has only been used to test the model relationship between two data trees (one denoted "the pattern" and the other "the database"). The contents of the *set* defined by the simulation were of no interest.

On closer examination one can notice that this set contains the information which parts of the database matched with the pattern. Actually, the fragment matched by the pattern can be considered the (or "one") *result* of a pattern matching, as this is the data the user intended to retrieve for further processing.

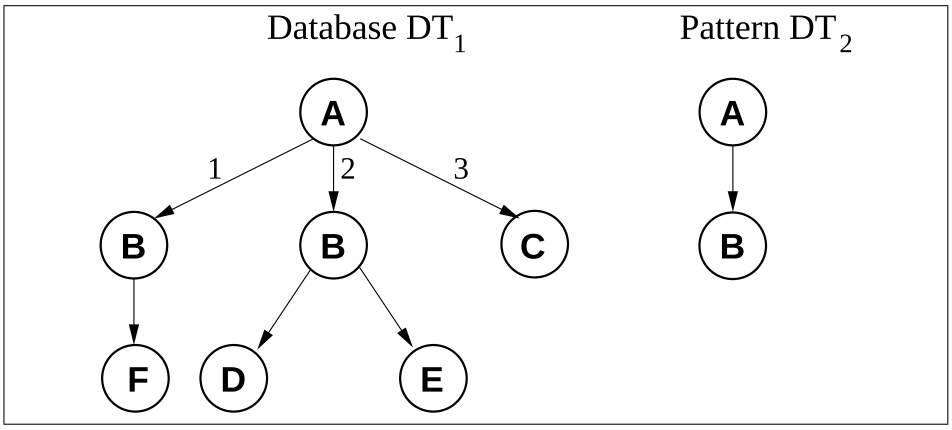
We will see later that there is also some more data to retrieve (section 5.2.2), but for the moment it suffices to assume that the simulation defines the full result.

In fact, this result comes natural since it is also in a model relationship with the database.

As we are now interested in the actual content of the simulation, it does no longer suffice to find only one simulation between a pattern and a database. Now it is necessary to find all of the simulations at every position where more than one choice of a matching partner exists. Such positions are very common in enriched data trees due to the manifolded interpretations of grouping facets.

An important question is how to present the results of these simulations to the user. Certainly it is possible to return the set of elementary data trees (from the interpretation of the pattern) that have simulations with the database.

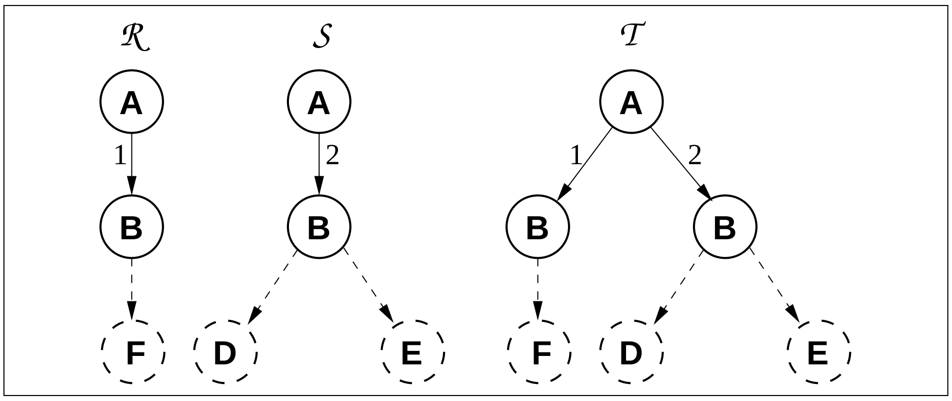
Example 5.1.1
Consider the two elementary data trees given in the next figure.



The branches at the first level in the database DT_1 are numbered so they can be differentiated easier. The two B will be denoted B_1 and B_2 according to the branch number, but the name is still B.

There are three simulations $\mathcal{R}, \mathcal{S}$ and $\mathcal{T}$ between DT_2 and DT_1: $\mathcal{R} = \{(A, A), (B, B_1)\}$, $\mathcal{S} = \{(A, A), (B, B_2)\}$ and $\mathcal{T} = \{(A, A), (B, B_1), (B, B_2)\}$.

The possible results generated from these simulations are shown in this figure:



Some nodes are shown dashed. These are not part of the actual simulation and would thus not be contained in the result. They are shown here to underline difference between the trees. Discussion about including such nodes in the *results is done in Section 5.2.2.*

5.1.2 Maximal Simulation

The technique shown in the previous example is very simple but not very convenient, since there are several answers for one query. Hence it would be beneficial to have a combined representation of the results of a matching which covers all of the possible simulations for a pattern and a database.

[ABS00] uses a **maximal simulation** which is essentially the simulation that "contains" all other simulations.

The following proposition (which is called "fact" in [ABS00], page 136) is the base for the *maximal simulation*:

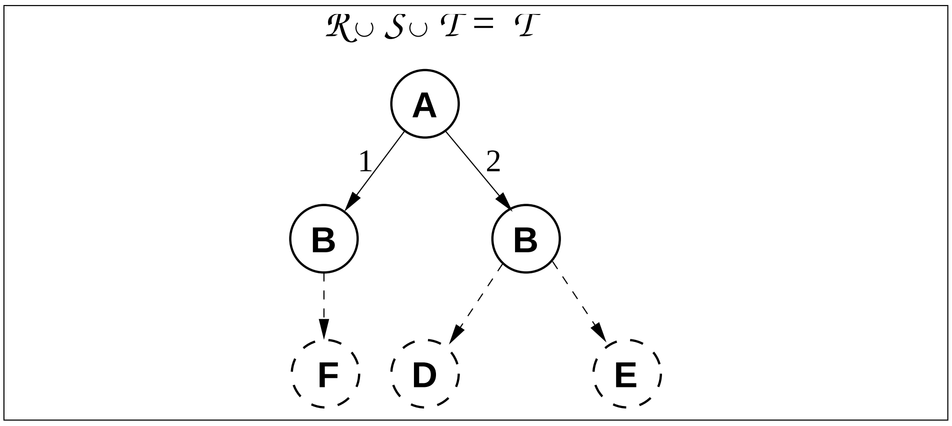
Proposition 5.1.1
If $G_1 \overset{\textbf{sim}}{\longrightarrow}_{\mathcal{R}_1} G_2$ *and* $G_1 \overset{\textbf{sim}}{\longrightarrow}_{\mathcal{R}_2} G_2$ *then* $G_1 \overset{\textbf{sim}}{\longrightarrow}_{\mathcal{R}_1 \cup \mathcal{R}_2} G_2$*.*

The **maximal simulation** is the union over all simulations between two data trees.

Obviously, the maximal simulation supersumes all other possible simulations, in other words at every point in the matching process where it is possible to match the pattern with several options in the database, all of them are chosen.

The maximal simulation therefore generates the result that covers all matchings. The following example shows this for the sample data that has been used previously:

Example 5.1.2
For the database and pattern in the previous example, the result tree generated from the maximal simulation is the tree generated by simulation $\mathcal{T}$ which supersumes the simulations $\mathcal{R}$ and $\mathcal{S}$.



$\mathcal{R} \cup \mathcal{S} \cup \mathcal{T} = \mathcal{T}$
A
1
2
B
B
F
D
E

According to [ABS00] there are several efficient algorithms for calculating the maximal simulation.

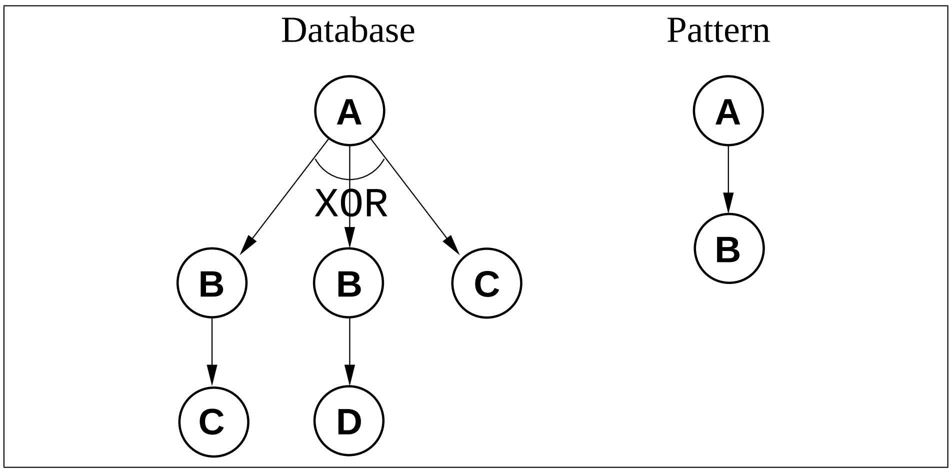
5.2 Answer Semantics for Data Trees with Grouping

5.2.1 Combining Results with Grouping Constructs

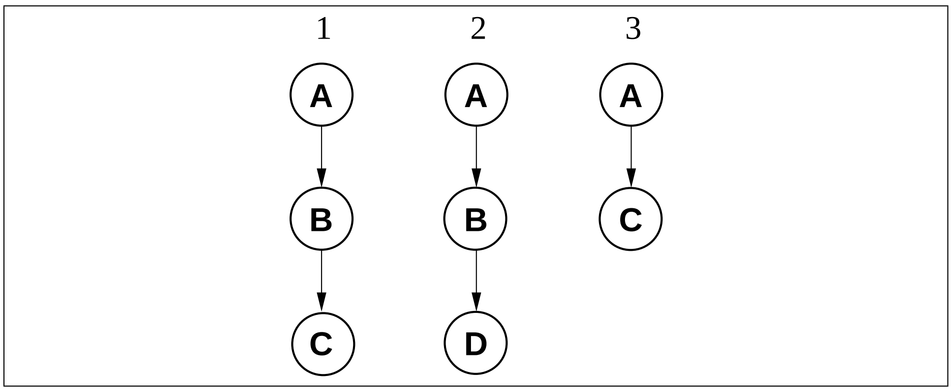
The maximal simulation still has some problems if used as a result.

First, with grouping facets it is still possible that there are several maximal simulations between a pattern and a database. The reason for this is that the interpretation function from 3.2 is used in the simulation definition. This interpretation can generate elementary data trees that don't contain each other.

Example 5.2.1
Again, a database and a pattern for it:

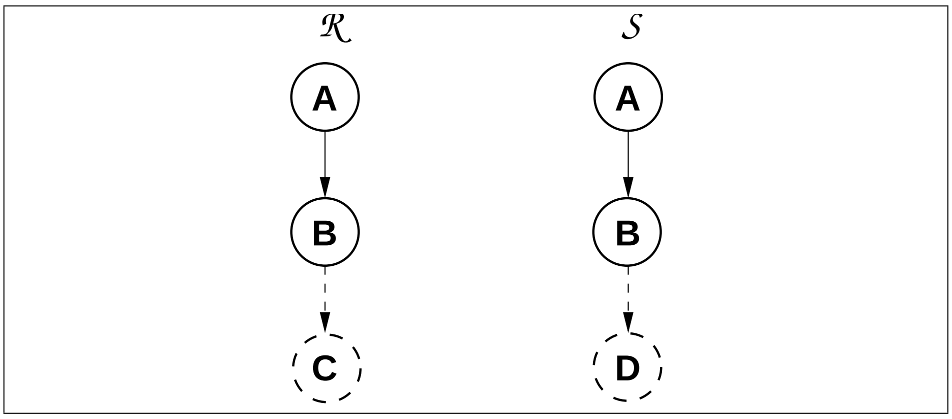


The three interpretations for the database are given in the next figure:



There are ***two*** *maximal simulations for this database and pattern, one between the pattern and interpretation 1 and the other between the pattern and interpretation 2 of the database.*

This will generate the following results:



The reasons for this problem are obvious: When a data tree with grouping is first transformed in a forest of elementary data trees, each of the elementary data trees may contain different child nodes so that the union of all simulations would not be a simulation any more.

The second problem is that the information about the other (smaller) possible results is supersumed by the maximal simulation and therefore lost.

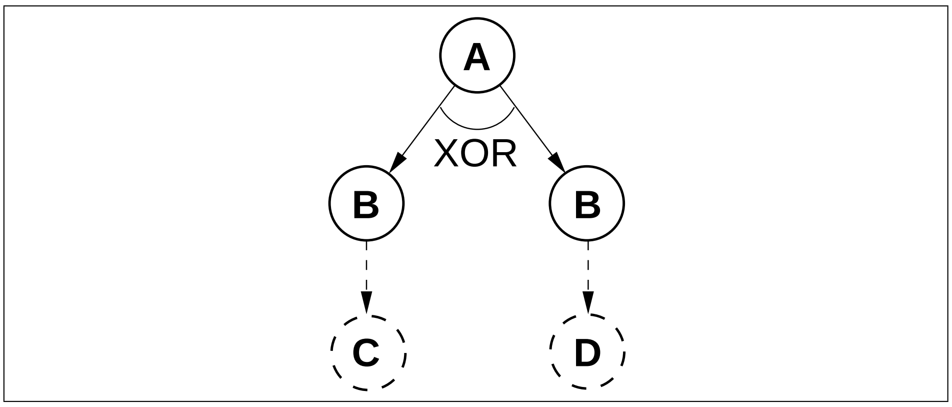
A solution that allows to overcome both of the mentioned problems is **grouping inheritance**. Based on the maximal simulation and the relationships given in table 5.1, page 49, the grouping constructs are carried over to the result, thus creating a *combined result*.

This combined result actually allows to aggregate all of the possible maximal simulations into one result.

The relationships in the table are currently limited so as no properties can be combined that come from different facets (e.g. from connector and order facet). The issue of facet combination will be a topic of further research (see Section 11.2), as it has many implications that have to be investigated. Also, only the relationships for the connector, order and selection facet are given, but it is possible to provide them for most of the other facets using a similar approach, but possibly with some problems resulting from the exclusion of certain nodes (e.g. the exclusion facet would require that there is *no* simulation to or from the child nodes).

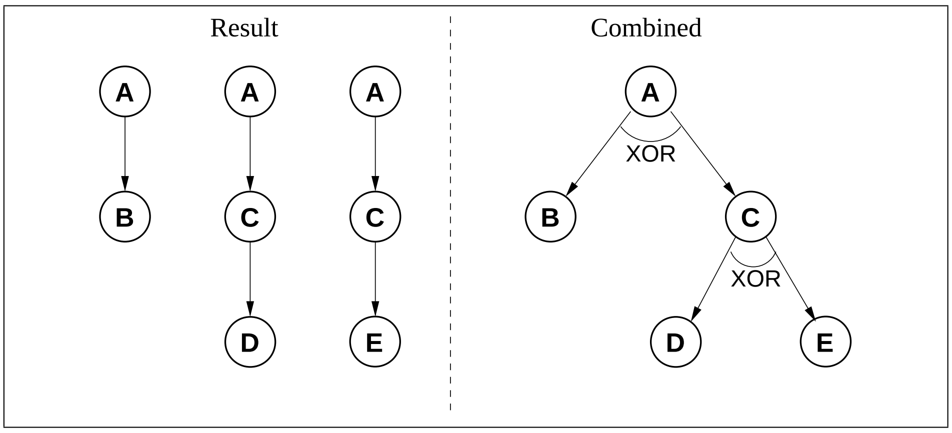
The idea behind this approach is that it would thus be possible to generate a result for the pattern matching between data trees with grouping by restricting the matching to the simple case of matching between elementary data trees and then just using the relations between grouping facets given in table 5.1. This can be seen as a "matching on a semantic level".

Example 5.2.2
In example 5.2.1, inheriting the grouping facets will result in the following result tree:



Obviously, such matching is much more efficient than first generating the possible interpretations for the data trees like in Definition 3.2.7.

Example 5.2.3
Recall again example 3.2.3 on page 31 that has been used to illustrate the application of the grouping simulation. On the left side you can see the three maximal simulations and on the right side the result obtained by grouping inheritance.



5.2.2 Extending the Results from the Simulation

A problem of the results generated by maximal simulations has not yet been addressed: They only contain the nodes that directly matched with the pattern.

Usually, this information is already known, however, since it is contained in the pattern. The real interest of the query is in the elements that are not contained in the pattern, i.e. siblings and other children of the current data (In the previous examples, these nodes were shown dashed).

The problem is illustrated by the following example (given for simplicity in an XML syntax):

Example 5.2.4
The following fragment of an XML document (which is just an elementary data tree in this form) represents an address book with several entries:

```
<addressbook>
  ...
  <entry>
    <name>Schaffert</name>
    <given>Sebastian</given>
    <address>
       ...
    </address>
  </entry>

  <entry>
    <name>Olteanu</name>
    <given>Dan</given>
    <address>
       ...
    </address>
  </entry>
  ...
</addressbook>
```

A pattern to this database that selects all of the entries that belong to someone with last name "Schaffert" could look like this:

```
<addressbook>
  <entry>
    <name>Schaffert</name>
  </entry>
</addressbook>
```

Now the problem is that the result generated from the pattern matching is just the pattern as there is only one simulation for this pattern and database (since both are elementary data trees and it is assumed that only one entry has a name "Schaffert"). It does not contain the address part of the entry, which would be the information one is usually interested in.

So it would be reasonable for the result to not only include the nodes that match but also the siblings and children.

However, this is not practical since it would always return the whole database (and thus negate the intention of the pattern matching) because all of the children of the root node are also contained.

A solution to this problem would be to let the user specify at which position the children should also be included. For this thesis it suffices to just have the option to include all or only the matched children of a node, but it is possible to have further refinements that specify more accurately to which children the property applies, in breadth (e.g. "all children after" or "the n-th child") as well as in depth (only the immediate children or all of the descendent).

Example 5.2.5
In an implementation, the pattern for the previous example could be refined as follows, assuming that the "result" attribute defines whether all or only the matched children should be retrieved.

```
<addressbook>
  <entry result="all-children">
    <name>Schaffert</name>
  </entry>
</addressbook>
```

The result could be the whole entry from the database that matches.

An implementation of this property could also maintain a pointer to the database for each of the nodes in the result. This would allow the user to specify which of the nodes to expand at a later point, after further processing the result that has been obtained from the first matching. An approach following this idea is the introduction of variables covered in Chapter 6.

Due to the time constraints of this thesis and the fact that the solutions suggested here are not difficult to implement but add to the complexity of the matching, the algorithms and implementation of the simulation-based matching presented in this paper do not support this additional property.

5.2.3 Strict vs. Liberal Matching

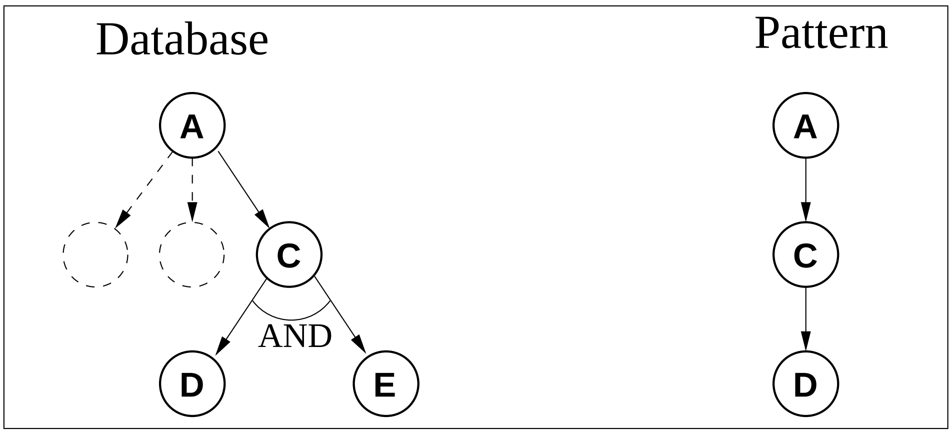
With the inheritance of grouping facets and the extended results from the last section, another problem can be solved: As mentioned before, in some cases it is desirable to have strict matching instead of liberal matching. An example is the "AND" grouping facet that specifies that all of the children have to be in the matching partner while liberal matching would allow additional child elements in the database that are not in the pattern.

With the two techniques introduced in the last section, the problem can be solved. Liberal matching can be used in all cases while the restriction is inherited to the result of the pattern matching.

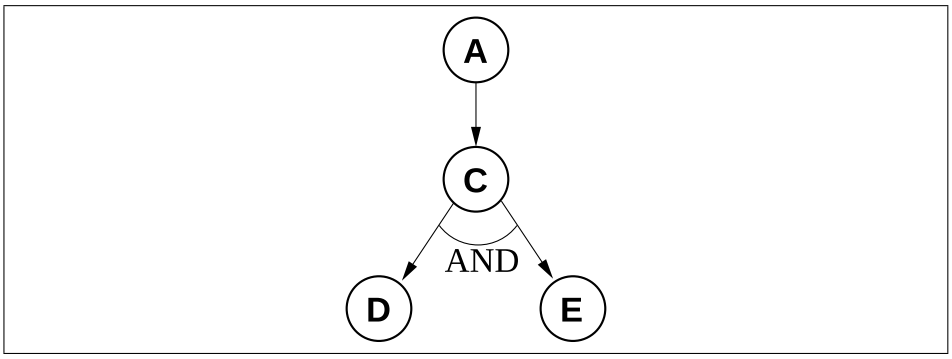
This is not obvious and needs further explanation. First it is necessary to recall once again what a user intends to say when he specifies a grouping facet in a data tree. He provides additional data about how the data should be interpreted.

Nothing is lost in liberal matching, it is just possible that a result contains more of a database than with strict matching. Since the grouping facet is carried on to the result, the information about the data is still available.

Example 5.2.6
Consider the following database and pattern for it where the node "C" in the pattern has a property of "all-children" like introduced above.



With liberal matching, the pattern obviously matches with the database. The result using grouping facet inheritance and the "all-children" property of the node "C" would be:



In the result the information from the database is still preserved, it is not really relevant in the matching process.

Connector Grouping Facet in the		
database	pattern	combined result
ϵ	ϵ	ϵ
AND	ϵ	AND
OR	ϵ	OR
XOR	ϵ	XOR
ϵ	AND	AND
AND	AND	AND
OR	AND	AND
XOR	AND	—[*]
ϵ	OR	OR
AND	OR	AND
OR	OR	OR
XOR	OR	XOR
ϵ	XOR	XOR
AND	XOR	—[*]
OR	XOR	XOR
XOR	XOR	XOR
unordered	ϵ	unordered
ordered	ϵ	ordered, if children in pattern appear in the same order as in the database, — otherwise
ϵ	unordered	unordered
unordered	unordered	unordered
ordered	unordered	ordered, if children in pattern appear in the same order as in the database, — otherwise
ϵ	ordered	ordered
unordered	ordered	ordered
ordered	ordered	ordered, if children in pattern appear in the same order as in the database, — otherwise
i to k	l to m	— if result contains less than $max(i,l)$ children
i to k	l to m	— if $l < k$ or $m < i$
i to k	l to m	$max(i,l)$ to $min(k,m)$

Table 5.1: Combining Results with Grouping Constructs
For a database with the grouping facet in the left column and a pattern with the grouping facet in the center column, the result based on the maximal simulation between the data trees without considering the grouping will inherit the grouping facet in the right column.
A dash (—) indicates, that the inheritance fails and there is no grouping simulation.

[*]AND and XOR will not generate a match if the number of elements is larger than 1

Chapter 6

Variables

In most pattern matching languages (like functional and logic programming languages) it is possible to use variables to act as place-holders for some data that should be matched. In this chapter we will discuss what are actually the properties of these place-holders and how they might be introduced into data trees and our simulation-based matching.

Furthermore, a way to represent the depth facet based on such place-holders is presented.

6.1 Variables in Data Trees

6.1.1 Two Properties of Variables

Variables in declarative languages (as used in e.g. functional and logic programming) actually have *two* properties: One is the property to bind to certain values (from now on called **assignment property**) and the other is the **joker property**, which allows them to match with any other value (possibly restricted by type).

While these often occur together, this is no necessity: On the one hand one might know the node name of a node and is interested in the subtree it matches with (see Section 5.2.2). On the other hand, it might be that one wants to match with any node and is not really interested in what is matched.

Thus, it is reasonable to separate these two properties and introduce each of them individually into data trees. Actually, the "joker" property can be represented similar to a grouping facet with an interpretation, while the "assignment" property only becomes relevant in the simulation (i.e. the matching algorithm).

The name "variable" or "variable node" in this document is used to identify the "assignment" property, since this is the property which usually represents the meaning of variables. The "joker" property will be called "joker".

6.1.2 Representing the "joker" property

The "joker" property can be represented as a special node that has its own semantics and interpretation function. These can be defined similar to the grouping facets (see section 6.2).

For data trees, this special node will be denoted as "?" with $name(?) = "?"$. A formal representation of data trees with variables is given in Section 6.1.4.

6.1.3 Representing the "assignment" property

The impact of the "assignment" property on the data tree syntax is not very big: it is necessary to somehow identify the nodes that can be bound. This is achieved by simply adding a partial function "*vname*":

Definition 6.1.1 (Variable Names)
The notion of data trees defined in 3.1.1 and 3.1.3 is extended by the (partial) function $vname : Nodes \to \textbf{Labels}$ *which assigns to all nodes that should bind a value a* ***variable name*** *by which the assignment can be identified.*

With this definition, it is possible to identify variable assignments (and thus the common notion of "variables") by a name. Since it doesn't have an impact on the semantics of the data (at least as long as each variable name only occurs once), further discussion of this topic will be delayed until the discussion of variable assignments (section 6.4).

6.1.4 Adding Variables to Data Trees

To summarize the new properties, here is a new definition for "data trees with variables".

Definition 6.1.2 (Data Tree with Variables)
A data tree with variables *DB is an "enriched data tree" with the following modifications:*

- $Nodes \subset \textbf{Vertices} \cup \{?\}$ *is a (finite) set of vertices.*
- $vname : Nodes \to \textbf{Labels}$ *is a partial mapping from nodes to their variable names. It is possible that several variable nodes share the same name, i.e.* $|image(vname(V))| \geq 1$.

6.2 Interpretation of a Data Tree with Variables

Variable assignment only becomes relevant in the definition of the simulation. This will be discussed in the sections about the results of a matching (sections 5 and 6.4).

enriched subtree $N_{\mathcal{G}}$	represented through the forest
$?(M_1, \ldots, M_n)$	all $N\{M_1, \ldots, M_n, R_1, \ldots, R_m\}$ with $N \in$ **Vertices**

Table 6.1: Interpretation of data trees with variables

The only change in the semantics is because of the introduction of the "joker" node (?-Node).

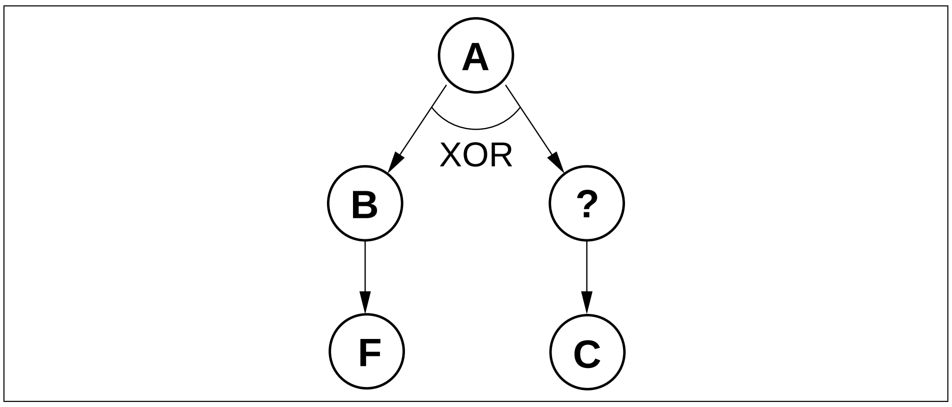
Informally, the ?-Node "matches" with all imaginable nodes, i.e. all (sub-) trees with any root node and the same children are models for it.

Representation of this property can easily be achieved by extending the notion of the interpretation function that is already used for grouping facets. In fact, the ?-Node can be seen as a pseudo-grouping-facet.

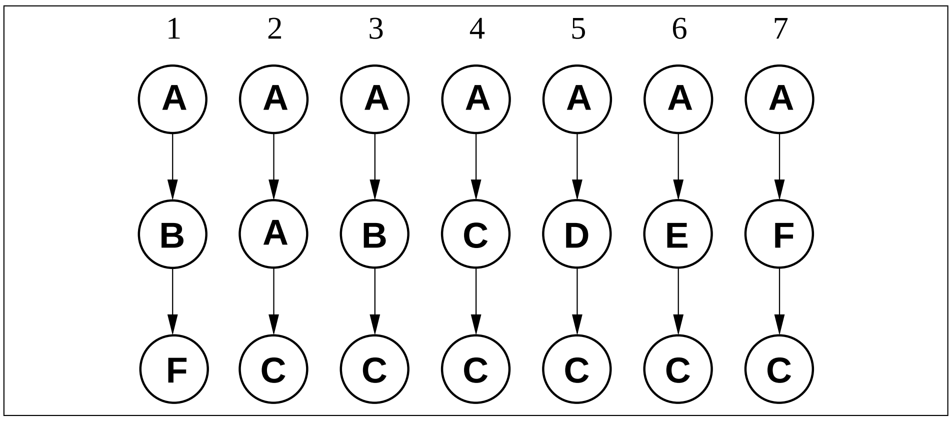
Definition 6.2.1 (Interpretation of a Data Tree with Variables)
The (liberal) interpretation function $\mathbf{I}_{\mathcal{G}}(N)$ defined in 3.2.1 and in table 3.2 for data trees with grouping facets is extended by the rule for variable nodes given in table 6.1.

Obviously, this definition doesn't require any change to the simulation definition used for enriched data trees. This is also the main advantage of this approach.

Example 6.2.1
Consider the enriched data tree with the joker node in the next example.



Assume that there is a universe consisting of a set of nodes **Vertices** $= \{"A", "B", "C", "D", "E", "F"\}$. *The following interpretations exist for the data tree:*



As can be seen in the previous example, the set of interpretations for a data tree with variables can get very large. In an implementation of the simulation, it is not necessary to generate all imaginable interpretations from **Vertices**. It is sufficient to take into consideration the nodes in the matching partner (i.e. its set of nodes). Furthermore the tree structures of both data trees can be used to limit the set of candidates.

6.3 Representing the Depth Facet: A Simple Approach

This section will show *one* possible way to represent the depth facet within the framework provided so far. The depth facet will be interpreted on the base of the ?-node.

The presentation of the algorithms later in this paper will refrain from implementing the depth facet because it is very time-consuming and doesn't fit into the time frame of this thesis. Efficient ways of calculating variable depth in trees are e.g. given in [MSB01] where it is possible to provide regular expressions over edges.

Recall that the depth facet allows to specify at what depth the children of a node may occur. A representation of this property is possible based on the ?-node, repeated the number of times that are specified as upper and lower bounds.

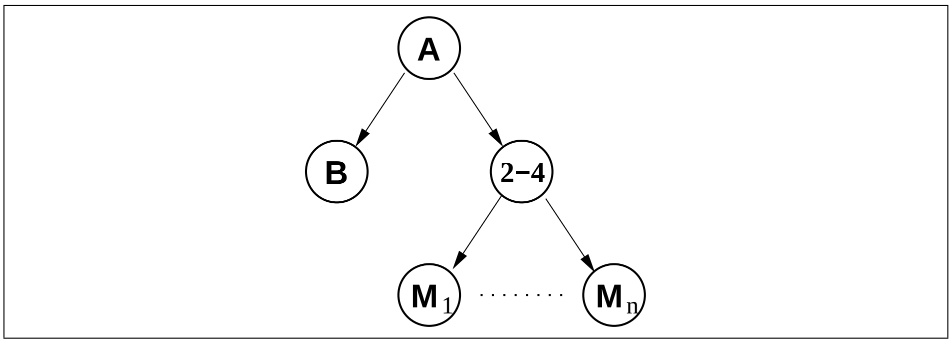
The syntactical representation of the depth facet could be done through the following extension:

Definition 6.3.1 (Depth Facet)
*The **depth facet** can be represented through the following nodes:*

- $*$*-node - the node spans an arbitrary depth including zero*
- $+$*-node - the node spans an arbitrary depth excluding zero*
- $m - n$*-node - the node spans an arbitrary depth between m and n*

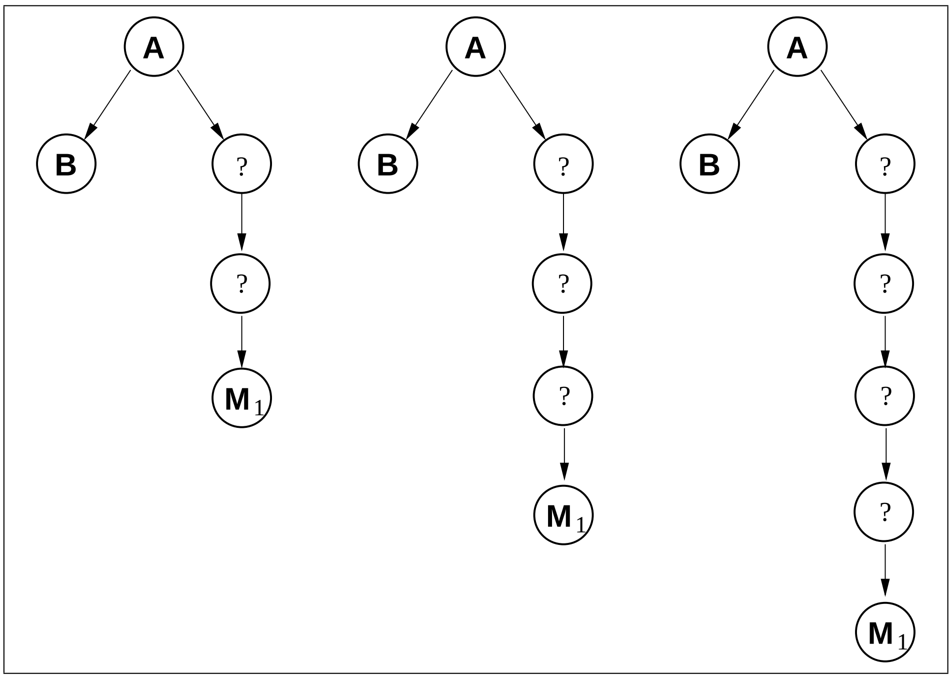
An informal definition of the semantics is already given in this definition. For the next step we should first consider what should actually be expressed by the depth facet. This is best achieved by using an example.

Example 6.3.1
Assume a tree like the one given in the following figure:



What data trees should match with these fragments?

The case is relatively simple, when $n = 1$. The fragment will then match with any N with a descendent M_1 at a level of 2 to 4 below A. The interpretation of the fragment thus is the set of the following enriched data trees:



This also shows the relation between the ?-node and the depth facet.

It becomes more complicated if $n > 1$, because this means that any of the child nodes M_i can occur at any level between 2 and 4 below the parent node N. A possibility would be to reduce the child elements to 1 by creating a new fragment for each of the child elements, i.e. if $n = 2$ this would generate the fragments $A(\text{"}2-4\text{"}(M_1))$ and $A(\text{"}2-4\text{"}(M_2))$ which would then again be interpreted as above.

As shown in the above example, the depth facet will produce a very large set of interpretations (infinite because of the ?-node, but still very large if only generated for a "small universe".

An implementation of the depth facet is possible by generating the set of interpretations based on the nodes and structure of the two trees matched. However, using this approach is very inefficient and it is desirable to investigate further approaches of representing and implementing the depth facet.

6.4 Variable Assignments

Variable assignments bind a certain variable (or: variable name) to a certain value. In Section 6.4.1, we discuss the possible options for the values of such a binding.

After that, the simulation based matching is extended to support variables. There are several issues that have to be addressed there, most notably the differentiation between singleton variables (variables that occur at most once) and multiple occurrences of the same variable. We will see that while matching with singleton variables is relatively simple, the case is considerably more difficult with multiple occurrences.

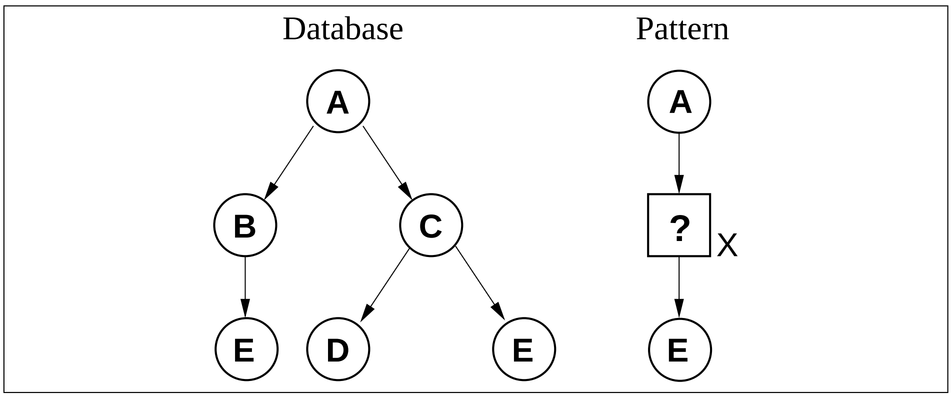
6.4.1 Types of Variables

To define variables, it is important to know what exactly variables should be bound to (i.e. what "type" a variable has). The types that come to mind are "node", "subtree" and "path".

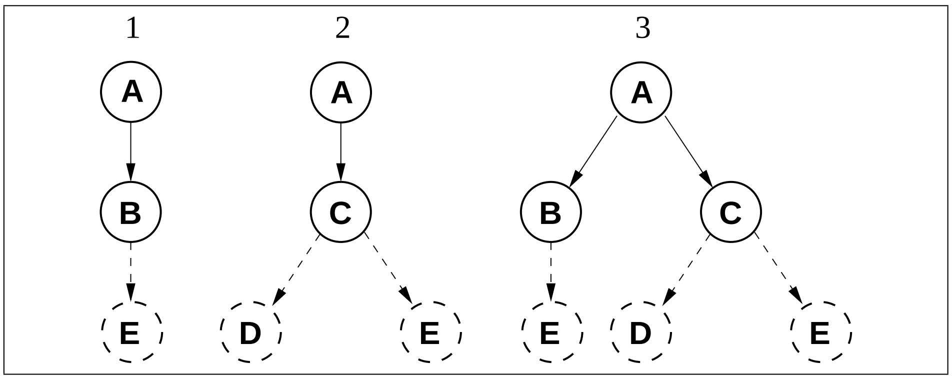
The three types will be introduced shortly in the following together with their advantages and disadvantages.

First, we introduce an example that will be used to demonstrate the different aspects of the variable types.

Example 6.4.1
The following figure shows a simple database and a pattern containing a variable. In this and the following figures, variables will always be denoted as a box instead of a circle. The variable name is given on the right lower corner of the box, in this case "X".



The following three simulations exist between the pattern and the database. The nodes that are not part of the simulation are again shown dashed.



Node Type

The node type is the basic notion of a variable. The variable will be assigned exactly the node it matched with.

Example 6.4.2
In the running example of this section, X will be assigned the values "B" and "C".

The advantage of this approach is that it requires just minimal changes to the pattern matching introduced so far. There would just have to be an extra set containing mappings from variables to single nodes.

The disadvantage, as can be seen easily in the example, is that the information that is gathered thus is only very limited, since a variable will only match with a certain node (which is carrying the information), tearing the information bit out of its context.

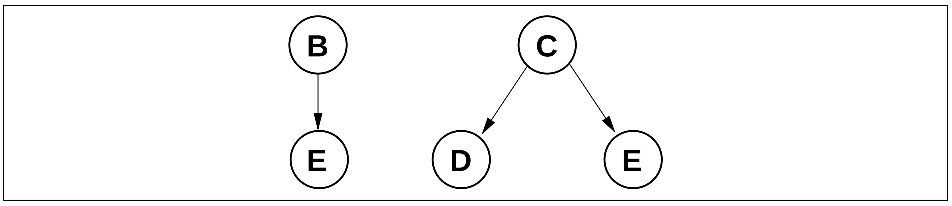
Example 6.4.3
In our example, the information about the child node "D" of "C" is missing.

Subtree Type

The second possibility is to match a variable with a whole subtree (i.e. with a mapping to the root node of a subtree and preserving the structure beneath it). All child nodes of the variable node can then be viewed as additional constraints to the matching that are nonetheless contained in the result.

A more formal description of a subtree is given in Section 6.4.2.

Example 6.4.4
The following assignments for X are possible when using a subtree-oriented approach for matching:



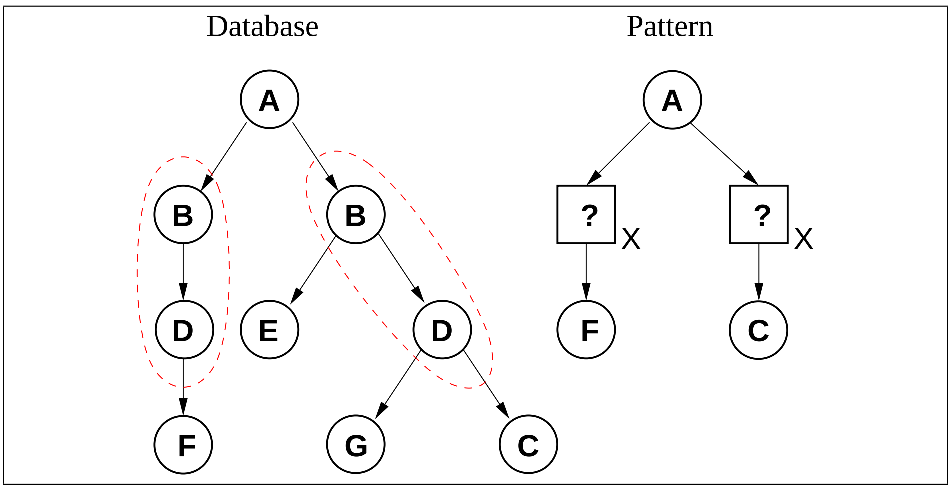
In this approach, the additional context of the matched node is preserved. However, the approach is more difficult to handle in terms of implementation than the node-only approach.

Note that it is also possible to solve the problem of fetching the context of certain nodes from Section 5.2.2.

Path Type

One other disadvantage of the subtree approach comes with multiple occurrences of a variable: In some cases it might be desirable, that a variable only gets assigned one node or a certain path, that should be repeated at another position.

Example 6.4.5
In the following figure, the variable X should bind to a path only that is repeated at a different position in the database, but has different child elements.



The disadvantages are mostly shared with the node type, but implementing this approach will probably lead to some difficulties. Also, a representation of this assignment will probably be not very convenient.

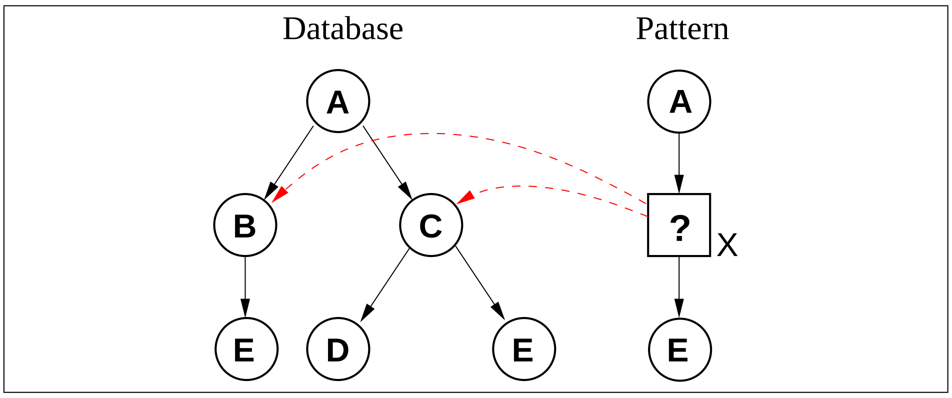
For the variable assignment we choose a combination of the node and the subtree type, as it is easy to introduce into data trees and intuitive to the user. This combination will be introduced in the next section. It might be possible to cover the path type with a special equality relationship for multiple occurrences. The need of equality relationships will be discussed for multiple variables in Section 6.4.5.

Combining Node and Subtree Type

Since the node type and the subtree type have complementary advantages and disadvantages, it would be best to find a solution that takes the advantages of both.

The idea is very simple: Let the variable "point" to the node, but keep the node in its context, i.e. in the database. This is similar to a "copy-by-reference" instead of a "copy-by-value" of the parameters for functions in common programming languages.

Example 6.4.6
Again consider the example used previously in this section. After matching the pattern with the database, the variable "X" could have two "pointers" into the database to the nodes "B" and "C". These are denoted as dashed, red arrows in the next figure.



This approach has the simplicity of the node type and the information of the subtree approach.

Thus, it will suffice in practice to interpret variables as pointers to the database. Fetching the required data out of that (which also includes greedy and non-greedy subtrees) is then up to the user.

Localization with variables is then just adding pointers to certain parts of the database (which arguably covers the idea of localization very well).

Changing the simulation definition to cope with such variables appears to be rather simple. If every variable occurs only once in a pattern and database, this is true. But for multiple occurrences, it will be necessary to actually compare assignments. Therefore, we still need to introduce the notion of subtrees even if the actual matching only assigns pointers.

6.4.2 Subtrees

As discussed above, variables will point to "subtrees". In this section a notion for subtrees of a data tree is given. This is necessary to be able to compare subtrees, which will be needed in Section 6.4.5.

A subtree will generally extend from a given root node to some or all of the leaf nodes. This can be achieved by using the transitive closure of the mapping $children()$ as the maximal node set for a given root and inheriting the underlying tree structure.

First we introduce the notion of the *descendants* of a node:

Definition 6.4.1 (Descendants)
For an arbitrary Node N in a data tree DT, a node N' is called a ***descendant*** *of N if it satisfies:*

- *either $N' \in children(N)$*
- *or $\exists M : M$ is a descendant of N and $N' \in children(M)$*

The set of all descendents of a node N with respect to a data tree DT shall be written $descendants_{DT}(N)$.

Having defined the notion of descendants, it is now easy to describe subtrees:

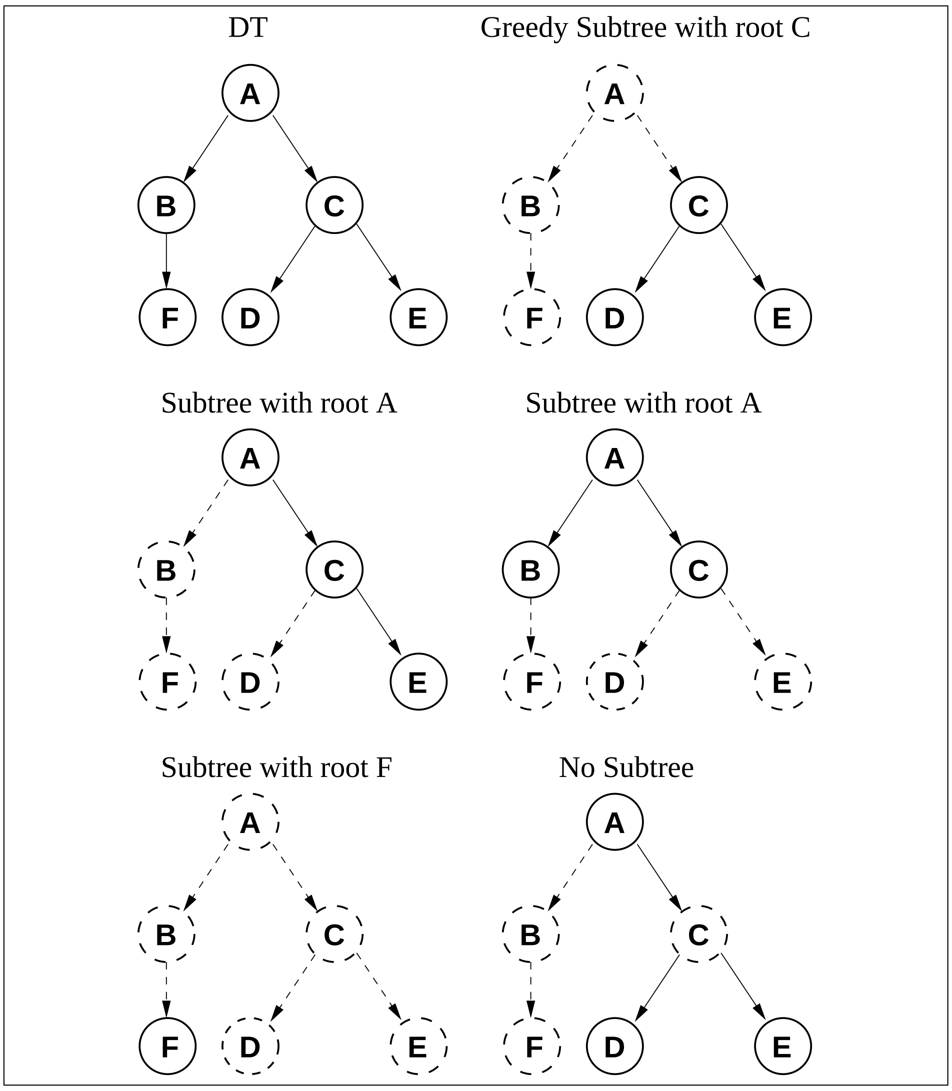
Definition 6.4.2 (Subtree)
Let $DT = (Nodes, name, children, root)$ be an (enriched or elementary) data tree, $N \in Nodes$ an arbitrary node in DT. A ***subtree with root*** *N $(Nodes', name, children, N)$ over DT is defined as a data tree with root node N and $Nodes' \subseteq descendants_{DT}(N) \cup \{N\}$ with the restriction that for every node $\neq N$ in $Nodes'$ the parent node must also be contained in $Nodes'$.*

A subtree is called ***greedy****, if it spans to all of the leafs in the underlying tree structure (i.e. $Nodes' = descendants_{DT}(N) \cup \{N\}$). The greedy subtree with root node N over DT will be denoted as $subtree_{DT}(N)$.*

Thus a subtree with root N can contain any children of N in the underlying structure if it at least contains the immediate parents of each node.

A "greedy" subtree is the complete subtree for a node up to the leafs. As already noted above, variables will be bound to greedy subtrees.

Example 6.4.7
The following figure shows a data tree DT and some subtrees in it. Also, a tree that is not a subtree is shown. The nodes that are in the subtree are shown solid, all others dashed.



Note that with the differentiation between greedy and non-greedy subtrees it is possible to describe the problem with extended results (results containing more than the nodes in the pattern) in Section 5.2.2 more accurately.

6.4.3 Singleton and Multiple Variables

When investigating pattern matching with variables, it is necessary to consider whether each variable only occurs once or may occur several times in the pattern. The implications of this decision are big: While matching with only one occurrence only requires simple assignment, multiple occurrences of the same variable require that each of the assignments of a variable have the same value. The latter requires the implementation of backtracking over the possible variable assignments and also influences the simulation-based pattern matching to a big extent.

Programming languages that use pattern matching handle it differently: Most functional languages like Haskell allow only one occurrence of a variable in a pattern (and none in the database!). See e.g. [Tho99], page 122. Logical programming languages, on the other hand, allow variables to occur an arbitrary number of times as long as certain conditions are satisfied ("occurs check"). See e.g. [SS94], page 87ff. In logic programming, pattern matching with multiple occurrences of variables is also called *unification*.

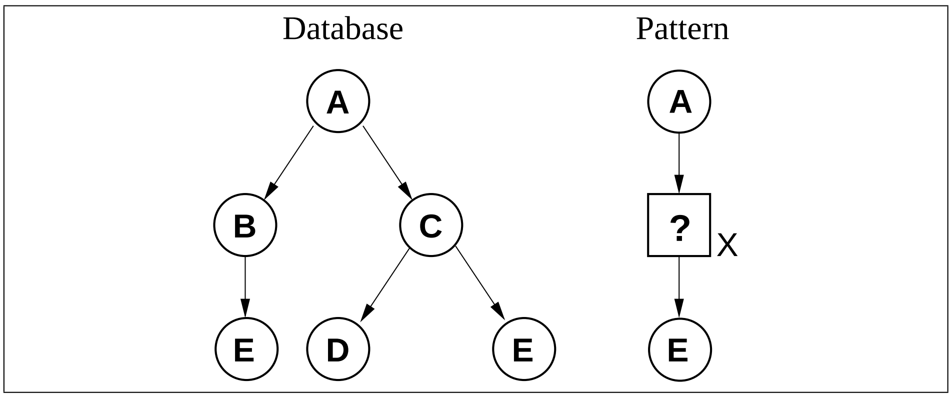
In this paper, we will first only investigate the case where a variable may only occur once and only in the pattern. Such variables will also be called **singleton variables**.

Some proposals are then made on how to handle multiple occurrences of variables (called **multiple variables**).

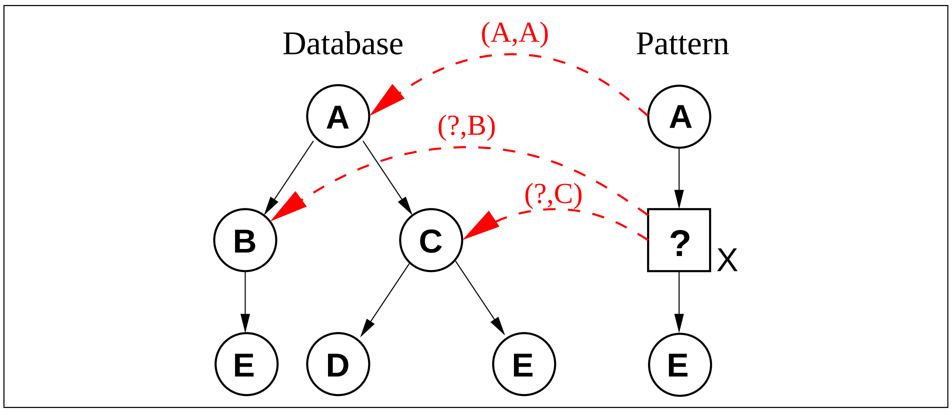
6.4.4 Variable Assignments for Singleton Variables

As discussed in Section 6.4.1, variables will be interpreted as references to the database. Handling this in an implementation is very simple, as the required relationship is already given in the simulation between a database and a pattern.

Example 6.4.8
Consider the example used in Section 6.4.1. For convenience, it is given again in the next figure.



In the next figure, the maximal simulation $\{(A, A), (?, B), (?, C)\}$ *between the pattern and the database is shown with dashed, red arrows.*



As can be seen in the example, the pointers from the variable named "X" into the database already exist – they just have to be taken from the simulation.

After this informal investigation, it is now possible to define the notion of a *variable assignment*.

Definition 6.4.3 (Variable Assignment)
Let DT be an (enriched) data tree (database) and P be an (enriched) data tree (pattern) with singleton variables. Furthermore, let $N \in P$ be a node with $|vname(N)| > 0$ (i.e. a variable).

If there is a (grouping) simulation $\mathcal{R}$ between P and DT ($P \xrightarrow{\textbf{sim}^g}_{\mathcal{R}} DT$), then the ***variable assignment*** *for $vname(N)$ with respect to $\mathcal{R}$, written $\sigma_{\mathcal{R}}(vname(N))$, is the subset of $\mathcal{R}$ limited to all tuples containing N ($\sigma_{\mathcal{R}}(vname(N)) = \{(M, M')|(M, M') \in \mathcal{R} \wedge M = N\}$).*

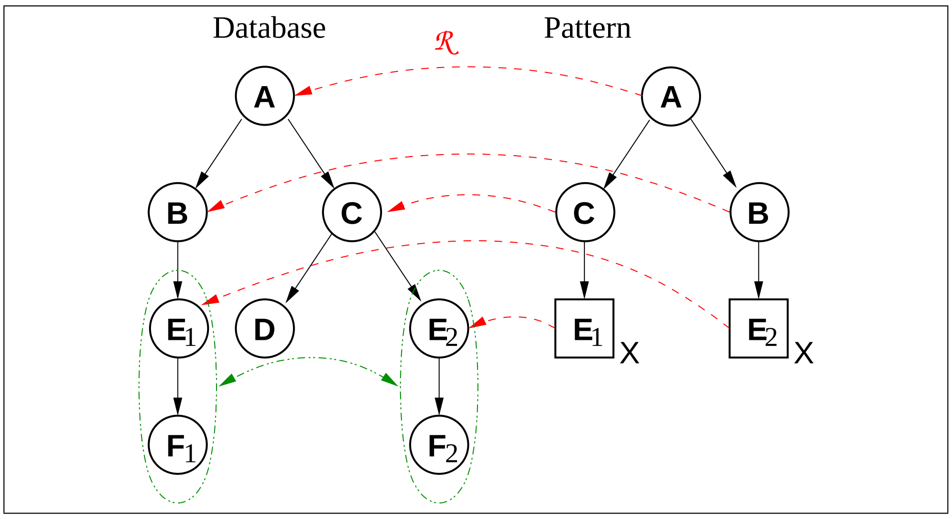
Example 6.4.9
In example 6.4.8, the variable assignment for "X" in the maximal simulation $\mathcal{R}$ is $\sigma_{\mathcal{R}}("X") = \{(?, B), (?, C)\}$.

6.4.5 Variable Assignments for Multiple Variables

Since the topic of variable assignments with multiple occurrences of a variable is a very complex topic (see *unification* in [SS94], pages 87ff.), we only provide a sketch here on how to deal with such variables in our data trees. It is not very difficult to provide a clear formalism, but developing algorithms and implementations involves more work.

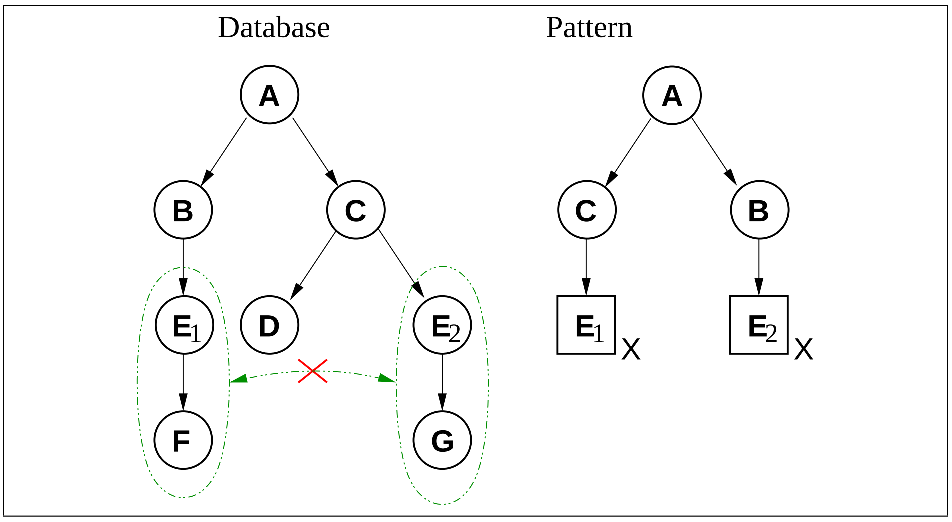
The main issue with multiple occurrences of a variable is that it is necessary to ensure that every occurrence of the same variable gets the same assignment. This not only involves the necessity to have a backtracking over the assignments for each variable, but it also restricts the set of possible simulations between two data trees.

Example 6.4.10
The following example shows a database and a pattern that contains the variable named "X" two times. The numbers on the E and F are only used for better identification. The name of the nodes are still "E" and "F",



The only simulation between the Pattern and the database is $\mathcal{R} = \{(A, A), (B, B), (C, C), (E_1, E_2), (E_2, E_1)\}$. *One could imagine a variable assignment for "X" like* $\{(E_1, E_2), (E_2, E_1)\}$ *because the two assignments match.*

Now consider the slightly modified database in the next figure:



Now, a variable assignment is not possible, because the two assignments won't match.

Furthermore, note that there is still the simulation $\mathcal{R}$ *between the pattern and the database, if we apply the simulation definition used so far. However, this is not desirable, because this simulation does not provide a variable assignment for "X".*

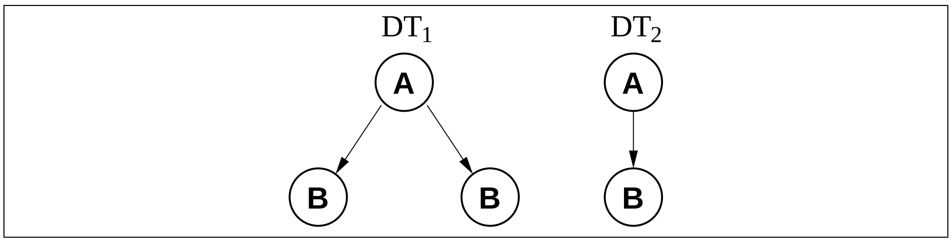
As can be seen in the example, some sort of *equality relationship* will be needed to support multiple occurrences of variables. There are several ways to define equality, we have already used one of them throughout the document: Simulation. Informally, we can say that if there are two or more occurrences of a variable, it is necessary that there is a *bisimulation* between the subtrees where the assignments of the occurrences point to.

Definition 6.4.4 (Bisimulation)
A relation $\mathcal{R}$ between two elementary data trees DT_1 and DT_2 is called an ***elementary bisimulation****, written $DT_1 \overset{\textbf{sim}}{\longleftrightarrow}_{\mathcal{R}} DT_2$, if $DT_1 \overset{\textbf{sim}}{\longrightarrow}_{\mathcal{R}} DT_2$ and $DT_2 \overset{\textbf{sim}}{\longrightarrow}_{\mathcal{R}} DT_1$.*

A relation $\mathcal{R}$ between two data trees with grouping DT_1 and DT_2 is called a ***grouping bisimulation****, written $DT_1 \overset{\textbf{sim}^g}{\longleftrightarrow}_{\mathcal{R}} DT_2$, if $DT_1 \overset{\textbf{sim}^g}{\longrightarrow}_{\mathcal{R}} DT_2$ and $DT_2 \overset{\textbf{sim}^g}{\longrightarrow}_{\mathcal{R}} DT_1$.*

Note that the grouping bisimulation as it is defined here doesn't ensure complete structural equality (*isomorphism*. In each direction it is sufficient that there *exists* an interpretation that has a simulation with *some* interpretation of the other subtree. This may in many cases not represent the intention of the user. However, it is not difficult to provide a different equality relationship that is stricter than this one. Especially, it will have no influence on the form of the simulation with multiple variables.

Example 6.4.11
Consider the two elementary data trees DT_1 and DT_2 in the next figure.



The two trees are bisimular (for each of the nodes in both trees exists a corresponding node in the other tree), but obviously not isomorph, as the structure is different.

Other equality relationships between trees are imaginable. Isomorphism on graphs is one of them, ensuring complete structural equality.

Now that we have an equality relationship, we can define a simulation with multiple variables. To do so, we will have to merge the ideas from Definitions 3.2.6 and 6.4.3 and then limit the possible simulations by the bisimulation.

For simplicity, we will first show the idea on elementary data trees.

Definition 6.4.5 (Elementary Simulation with Multiple Variables)
Let DT_1 be an elementary data tree with variables and DT_2 be an elementary data trees without variables, $Nodes_1$ and $Nodes_2$ be their respective set of nodes. A simulation $\mathcal{R}$ between them ($DT_1 \overset{\textbf{sim}}{\longrightarrow}_{\mathcal{R}} DT_2$) is a ***elementary simulation with multiple variables*** *if it satisfies:*

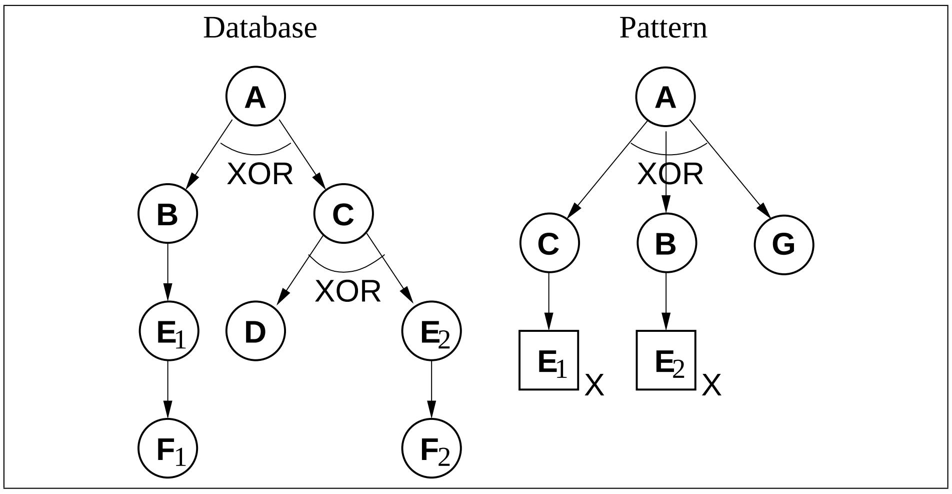
$\forall N_1, N_1' \in Nodes_1 \forall N_2, N_2' \in Nodes_2 : vname(N_1) = vname(N_1') \wedge (N_1, N_2) \in \mathcal{R} \wedge (N_1', N_2') \in \mathcal{R} \Rightarrow \exists \mathcal{S} subtree_{DT_2}(N_2) \overset{\textbf{sim}}{\longleftrightarrow}_{\mathcal{S}} subtree_{DT_2}(N_2')$

This definition explicitly limits the variables to occur only in the pattern. An extension to the database is possible, but can be complicated if the same variable occurs in the pattern and in the database (think e.g. of liberal matching).

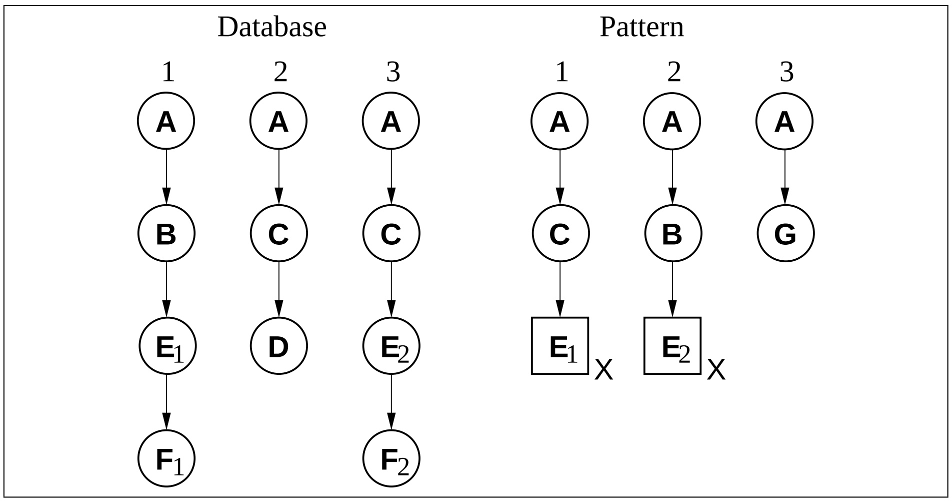
Example 6.4.12
In the previous example (Example 6.4.10), the bisimulation between the two assignments of "X" is highlighted with dashed circles.

An extension to data trees with grouping can be done in several ways. The first would be to just take Definition 3.2.7 and then use an elementary simulation with multiple variables instead of the elementary simulation between the interpretations of the enriched trees.

Example 6.4.13
Consider the following data trees with grouping facets (these are from the above example with slight modifications.



The interpretations that can be generated for the database and pattern are given in the following figure:



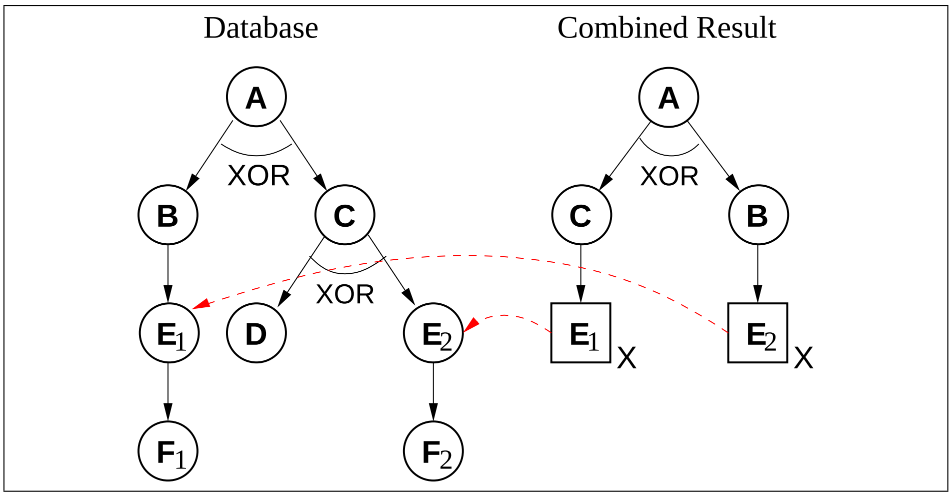
Several problems arise. First, there is no single maximal simulation between the forest from the database and the forest from the pattern, there are actually *two of them. Which of the variable assignments is the right one?*

Furthermore, in the above example, the two occurrences of variables are assigned in different simulations so actually its only a singleton variable assignment for each of the simulations.

Obviously, this approach also shares the disadvantages of the grouping simulation, namely that the interpretations have to be generated. Furthermore, variable assignments will no longer point directly to the database, but to the individual interpretations of the database.

A second possibility is to use the combined result from Section 5.2.1 and its underlying maximal elementary simulation. The variables would be carried to the result and then point back into the database. This would allow assignments that point immediately into the database, however, the semantics may be different and it is not yet investigated how to solve the bisimulation in that case (a grouping bisimulation in our current definition just needs to find one interpretation on either side, but the intention will often be to have all interpretations).

Example 6.4.14
In the same example as before, a variable assignment can be realized like in the next figure, using the combined result from database and pattern.



The final choice and formalism is left open at this point. Further investigations of this topic are planned for future research.

Part III

Algorithms

In this part, the basic algorithms for data trees with grouping are given: Generating the interpretations for a data tree with grouping is the first, followed by the algorithms for elementary and grouping simulation. Last but not least, an algorithm for calculating the maximal simulation is presented.

The algorithms are given in a Pascal-like pseudo-code, which uses some constructs that are not necessarily part of Pascal but are common in other programming languages. Also, some more abstract data types are used in their intuitive way (e.g. "Sets"), which should be obvious to the reader.

Most of the algorithms are first introduced on a lengthy example which adds to the understanding of the pseudo-code presented afterwards and also shows why the algorithm generates what it intends to.

After presenting the pseudo-code, each step is commented as seems appropriate.

We refrain from presenting a proof of correctness for the algorithms. Most of the steps are derived immediately from the definitions of the corresponding concepts and should be illustrated by the examples.

Chapter 7

Data Trees

In this chapter we will shortly resume the definitions from chapter 3 for further use in this part.

First again have a look at elementary and enriched data trees. Then we repeat (with slight modifications) the algorithm presented in Definition 3.2.1 to generate the set of elementary representations for a data tree with grouping.

7.1 Basic Data Structures

For easier reference, the definitions for elementary and enriched data trees are shortly resumed here.

An *elementary data tree DT* is a tuple $(Nodes, name, children, root)$, where $Nodes$ is the set of nodes, $name$ is a function mapping nodes to their names, $children$ a function mapping to the set of child nodes and $root$ is the root node of DT.

An *enriched data tree* or *data tree with grouping* is a tuple $(Nodes, name, grouping, children, root)$ such that $(Nodes, name, children, root)$ is an elementary data tree and $grouping$ is a function mapping each node to a set of grouping facets.

7.2 Interpretation

In Definition 3.2.1, an algorithm is presented that produces the possible interpretations for an enriched data tree. It is repeated here for convenience.

Let EDT be an enriched data tree with grouping and $Nodes$ its set of nodes. The set of interpretations for the tree EDT, $\mathcal{I}(EDT)$, is generated as follows:

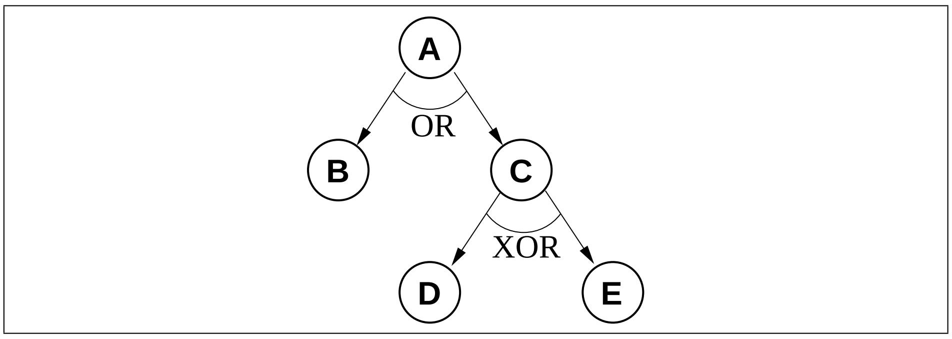
let N be the root node of EDT.

- `if` N is a leaf node of EDT `then` the interpretation is just the node N ($\mathcal{I}(EDT) = \{N\}$). End.

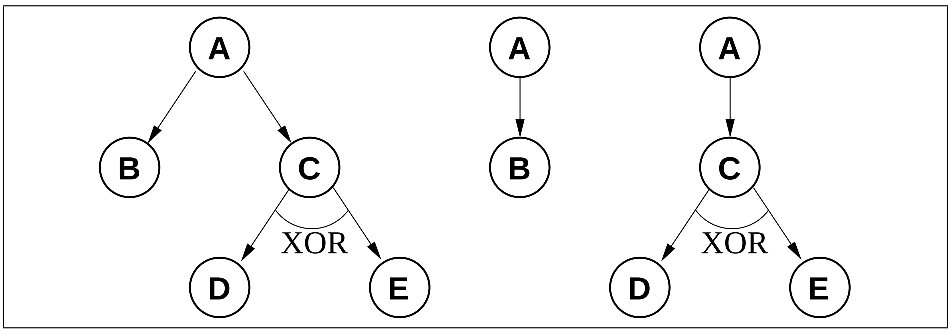
- `if` N is an inner node of EDT `then do`
    1. for each node $M \in children(N)$, generate the set of interpretations $\mathcal{I}(M)$ for the subtree with root M.
    2. generate the set of interpretations $\mathbf{I}(N)$ for N following definition 3.2.1. Initialize the working set WS with $\mathbf{I}(N)$.
    3. for all interpretations I in $\mathcal{I}(M)$: if there is an interpretation J in WS where $M \in children(N)$ (wrt J), add a new interpretation to WS that equals J where the subtree with root M in J is replaced with I.
    4. $\mathcal{I}(EDT)$ is then the working set WS restricted to elementary data trees.

So, as can be seen in the second item, it will first be necessary to be able to generate the interpretations of a single node without recursing to the children. This can be achieved using the table in definition 3.2.1.

Example 7.2.1
Consider the following tree with grouping:



For the node "A", the algorithm will return a set of interpretations like given in the next figure (Note that the child nodes still have their grouping facets):



Thus, it is necessary to duplicate the node "A" several times and attach a different set of child nodes to it. The complete table for this is given in Definition 3.2.1.

Algorithm 7.2.1
The algorithm for generating the interpretations of a single node consists of

a function called `nodeinterpretations`*. This function takes the node as an argument and returns the set of interpretations for this node.*

It is assumed that there exists a function named `removegrouping` *that simply eliminates the grouping of the node. Also, we assume a function called* `grouping` *which returns the grouping facet of the node. Furthermore, we need a function called* `clone` *which just duplicates a node without the references to the children, and a procedure* `addChild` *that adds a child node to the node.*

```
(00) FUNCTION nodeinterpretations(VAR node: Node): Set;
(01) VAR S: Set; g: Grouping; child, nnode: Node;
(02) BEGIN
(03)   G := grouping(node);
(04)   CASE G OF
(05)     AND: S := { removegrouping(node) };
(06)     XOR: S := {};
(07)       FOREACH child IN children(node) DO
(08)       BEGIN
(09)         nnode := clone(node);
(10)         addChild(nnode,child);
(11)         S := S UNION { nnode };
(12)       END;
(13)   ...
(14)   ESAC
(15)   RETURN S;
(16) END;
```

For space reasons, the algorithm is only given for the “AND” and “XOR” case, which is sufficient to show the idea.

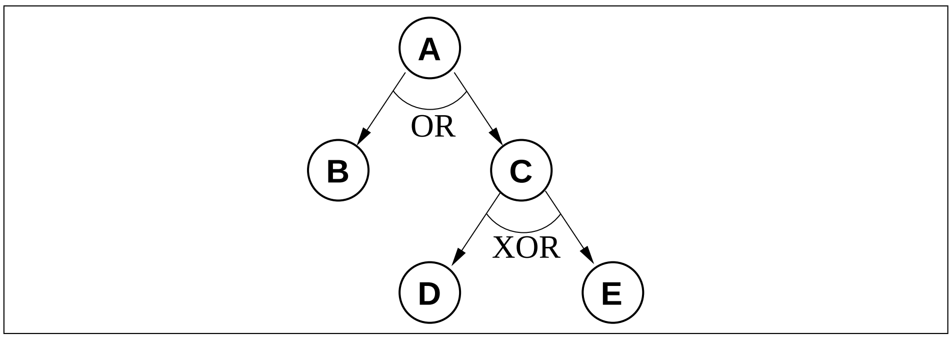
In line `03`, the grouping of the node is fetched and then used in the case selection beginning with line `04`. The `AND` case is simple, we just return the node together with its children, but without grouping (line `05`). The case is more difficult for `XOR`: It is necessary to generate new nodes that equal the given node and then attach to each of them exactly one child node of the original node (lines `06` to `12`).

One can imagine, that other facets like “OR” or “ordered”/“unordered” need even more code structures to generate all of the interpretations.

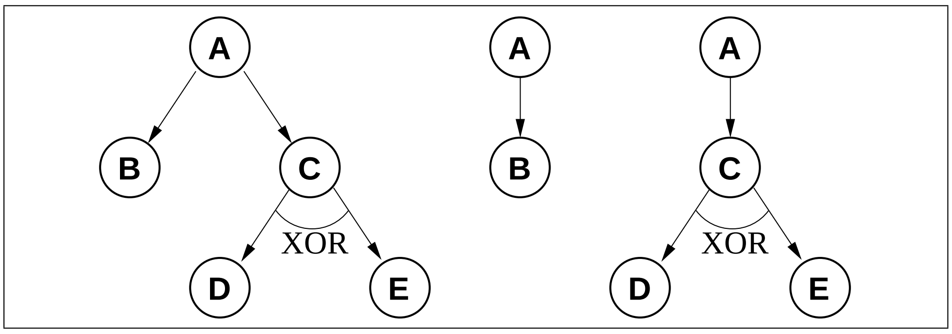
The algorithm for generating the individual interpretations for a node will now be used in an algorithm that generates the elementary representations for a whole data tree with grouping.

Again, the algorithm is first demonstrated on a simple example.

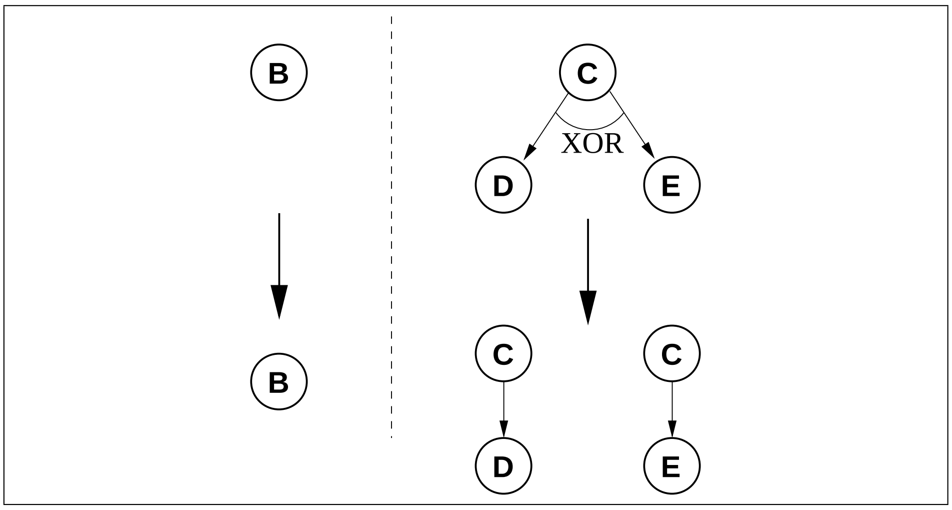
Example 7.2.2
Consider again the tree that has already been used for the interpretations of an node.



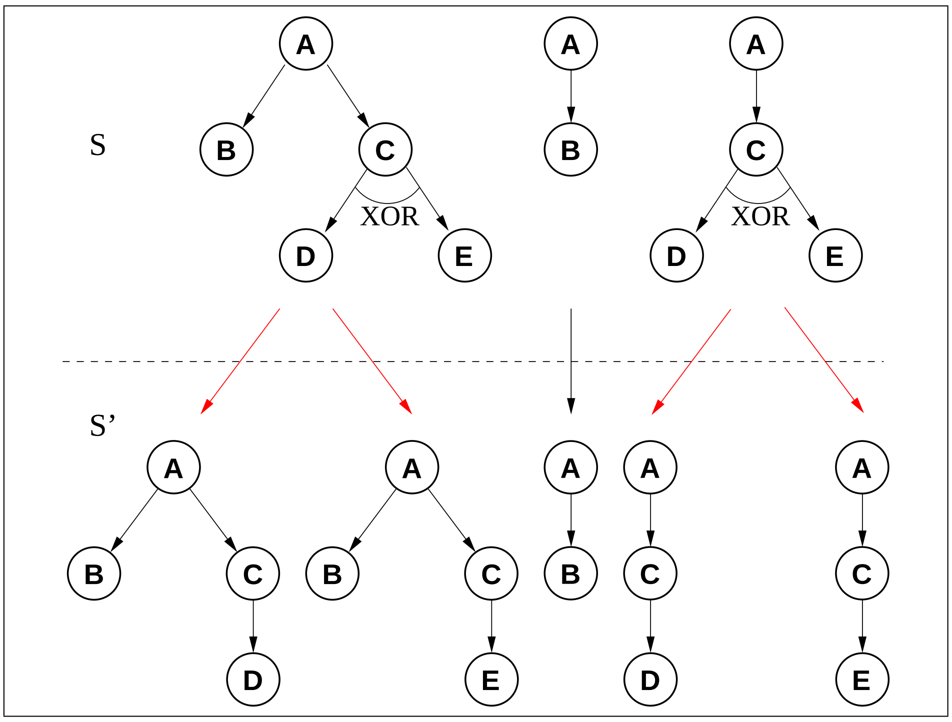
The following set of node interpretations is first generated from the root node "A". This will be the working set $\mathcal{S}$ for this node.



In the next step, we (recursively) generate the interpretations for all of the child nodes of the root node "A". Since there is only one grouping left (below the "C"), we only need the knowledge of the node interpretation.



Finally, all occurrences of children of "A" that have new interpretations are replaced in the set S with their new values, splitting up entries if necessary. The new set is called S' here.



The algorithm can now be presented in the pseudo-code language, performing the same steps as in the example:

Algorithm 7.2.2
The algorithm consists of a function that takes as an argument the root node of the data tree with grouping and returns a set of elementary data trees, the interpretations for this tree.

```
(00) FUNCTION interpretations(VAR root: Node) : Set
(01) VAR R,S: Set; node, interp1, interp2: Node;

(02) BEGIN
(03)   IF children(root) = {} THEN RETURN { root };

(04)   R := nodeinterpretations(root);

(05)   FOREACH node IN children(root) DO
(06)   BEGIN
(07)     S := interpretations(node);

(08)     FOREACH interp1 IN R DO
(09)     BEGIN

(10)       IF node IN children(interp1) THEN
(11)       BEGIN
(12)         <remove interp1 from R>
```

```
(13)          FOREACH interp2 IN S DO
(14)          BEGIN
(15)            <add interp1 to R where node is replaced with interp2>
(16)          END;
(17)        END;
(18)      END;
(19)     END;

(20)     RETURN R;
(21)   END;
```

Line `03` shows the recursion stop. If the set of children is empty, there is only a single interpretation, namely the node itself.

In line `04`, the set of node-interpretations is generated according to Algorithm 7.2.1 and stored in the variable `R`. The loop beginning in line `05` generates the interpretations for each of the subtrees (recursion case) and then replaces for each interpretation in `R` the child element that has just been treated with its elementary representations (lines `08` to `18`). In line `12` the current interpretation is removed from `R` and then again added according to the number of elementary child nodes for the current child in lines `13` to `16`.

Chapter 8

Simulation

In this chapter we will introduce some basic algorithms that demonstrate how to calculate the simulation relation between two data trees for elementary data trees and data trees with grouping.

8.1 Elementary Simulation

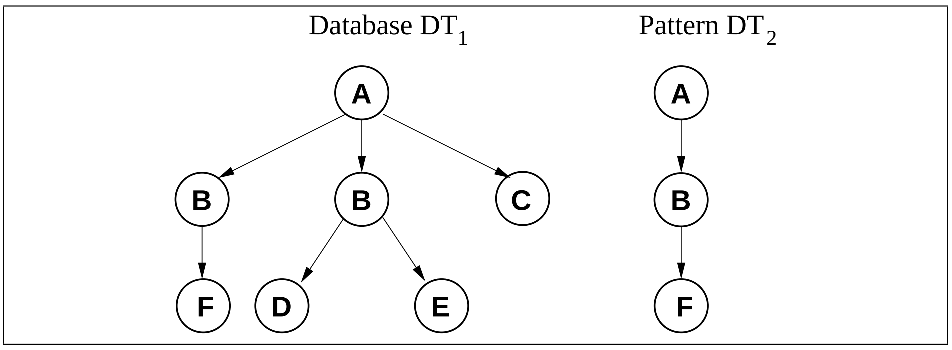
For the elementary simulation, we need the following data structures:

- two elementary data trees DT_1 and DT_2 with their sets of nodes $Nodes_1$ and $Nodes_2$ and root nodes $root_1$ and $root_2$.
- a set of pairs $\mathcal{R}$ denoting the simulation between the two data trees

We want to calculate the simulation between DT_1 and DT_2, i.e. $DT_1 \xrightarrow{\textbf{sim}}_{\mathcal{R}} DT_2$. To achieve this, a "divide et impera" (divide and conquer) approach is chosen.

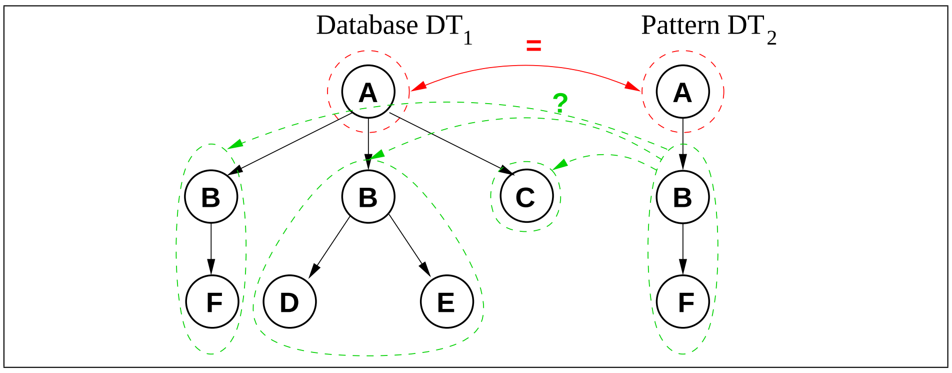
We will first introduce the method informally with an example. The principle is the following: For a given node N in DT_1, if the name of the node is equal with the/a corresponding node in DT_2, and there is a simulation for all of the subtrees of child nodes in DT_1 on subtrees of child nodes in DT_2, then there is also a simulation for this node.

Example 8.1.1
Consider the database and pattern in the following figure.



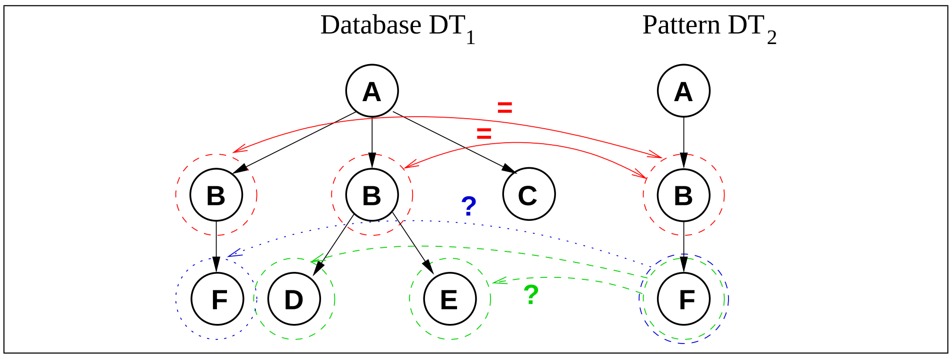
The simulation algorithm will first begin with the root nodes of the two trees ("rooted simulation"). As already described, it will be necessary that the names of the two tested nodes are equal and there is a simulation between all child subtrees of the pattern and some child subtrees of the database.

In the next figure, the equality between the two root nodes is shown with a red arrow (and red dashed circles), while the simulation for the subtrees of the children that is still to be found is shown in green.

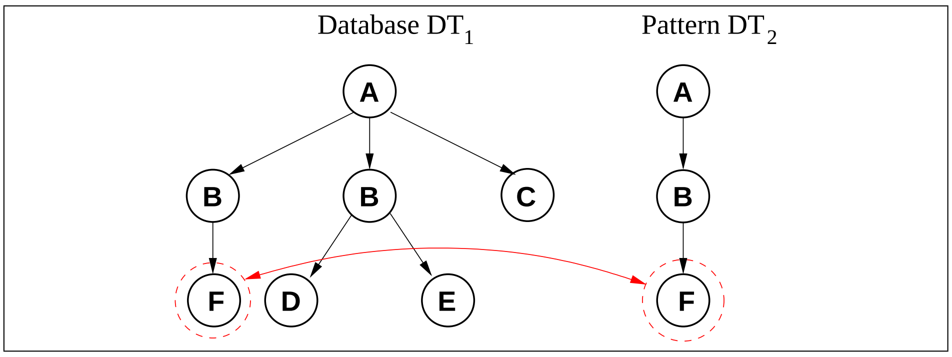


After having established that the root nodes are equal, we try to find a simulation for all of the children in the pattern (in our example only one).

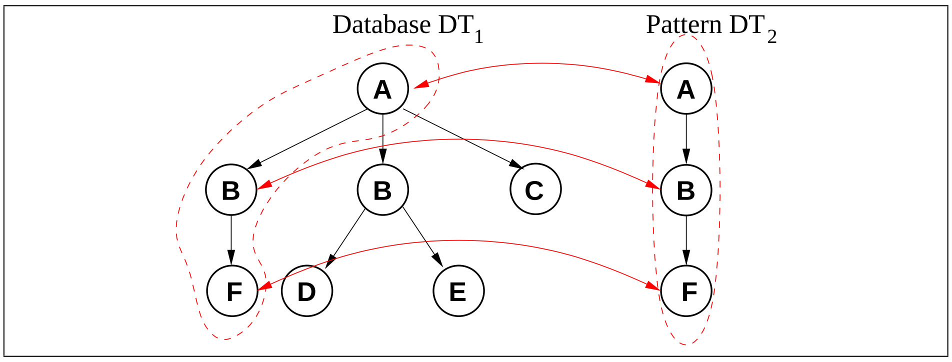
There are two child subtrees in the database that have the same root node as the subtree in the pattern (denoted with red). These are candidates and will provide a simulation if there is also a simulation for the children (denoted with blue/dotted and green/dashed). There has to be a simulation between the child subtree with root "F" in the pattern and one of the children marked green so that the middle "B" will get into the simulation. Similarly, there has to be a simulation on "F" in the pattern and (one of) the child subtrees marked blue.



The last step is simple, as the two subtrees consist of a root node only. There is only one possibility to fulfill the conditions in the example:



Thus, the result of the calculation will be the simulation given in the next figure:



Now that we have introduced the algorithm on a simple example, we will formalize it in a pseudo-code notation that somewhat resembles the Pascal line of programming languages. `R` is the set that will contain the simulation after processing. `root1` and `root2` are the node sets of DT_1 and DT_2 respectively.

Algorithm 8.1.1 (Simulation for Elementary Data Trees)

```
function simulation(root1:Node, root2:Node, VAR R:Relation)
                                                  :boolean
VAR S,T: set; flag: boolean;
BEGIN
  IF name(root1) <> name(root2) THEN RETURN false;

  T := {};
  FOREACH node1 IN children(root1) DO
  BEGIN
    flag := false;
    FOREACH node2 IN children(root2) DO
    BEGIN
      S := {};
      IF simulation(node1,node2,S) THEN
      BEGIN
        flag := true;
        T := T UNION S;
        break;
```

```
(17)        END;
(18)      END;
(19)      IF NOT flag THEN RETURN false;
(20)    END;

(21)    R := R UNION { (root1,root2) } UNION T;
(22)    RETURN true;
(23)  END;

(24)  BEGIN
(25)    set R := {};
(26)    IF simulation(root1,root2,R)
(27)       THEN write("Simulation: ",R);
(28)       ELSE write("No simulation");
(29)  END;
```

A reader not familiar with the C/Java branch of programming languages may wonder about the `break` statement used in line `16`. It just leaves the current embracing loop.

As this algorithm is not very pleasant to look on, the most important steps are described in the next few paragraphs.

The function `simulation` declared in line `00` takes three arguments: the root nodes of the two trees that should be tested and the set that will contain the relation. The latter is a reference as the function will add tuples to it. The return value of the function is a boolean value that tells whether the simulation was successful or not.

Line `04` tests whether the two root nodes are equal. If not, the function immediately returns `false`. If yes, for all of the child nodes of `root1` (line `06`) we test whether there is a simulation on some other child node of `root2` (lines `07` to `20`). If the test is successful, we have found a simulation that satisfies the condition for the selected child node. We add the found simulation to our current simulation and break the loop (lines `12` to `17`. In line `19`, the variable `flag` indicates whether we found a corresponding node for `node1` in the children of `root2`. If not, there is no simulation and the function returns false.

If line `21` is reached, a simulation for all of the child nodes of `root1` has already been found. Thus, the pair `(root1,root2)` can be added to the simulation and the function can return success. Obviously, if `root1` does not have child nodes and the test from line `04` was successful, we immediately reach this point.

Lines `24` to `29` make the initial call for the function `simulation`, beginning with the root nodes of DT_1 and DT_2.

8.1.1 Correctness

In the next paragraphs it will be shown that Algorithm 8.1.1 really provides what we defined as a "simulation" before.

For the proof, it is first necessary to introduce a short lemma which shows a relation between the simulation on two data trees and the simulations on their subtrees (see definition 6.4.2):

Lemma 8.1.1
Given any two data trees DT and DT' with root nodes r and r' respectively.

1. *If there is a simulation $\mathcal{G}$ between DT and DT' (written $DT \xrightarrow{\textbf{sim}}_{\mathcal{G}} DT'$), the following holds: $\forall n \in children(r)\ \exists n' \in children(r')$ such that $subtree_{DT}(n) \xrightarrow{\textbf{sim}}_{\mathcal{G}} subtree_{DT'}(n')$*
2. *If there exists a simulation $\mathcal{G}$ such that $\forall n \in children(r)\ \exists n' \in children(r')$ such that $subtree_{DT}(n) \xrightarrow{\textbf{sim}}_{\mathcal{G}} subtree_{DT'}(n')$ is satisfied and $(r, r') \in \mathcal{G}$, $\mathcal{G}$ is also a simulation between DT and DT'.*

Proof. 1. Assume that DT consists of only the node r. The condition holds trivially as r does not have any child nodes and $(r, r') \in \mathcal{G}$. Now assume that r has more than zero children. If the root nodes r and r' are removed from DT and DT', $\mathcal{G} \backslash \{(r, r')\}$ is still a simulation on the remaining forest. Thus the condition is satisfied. 2. Assume that DT consists of only the node r. Again, the condition is satisified trivially, as $(r, r') \in \mathcal{G}$. Assume that r has more than zero children. $\mathcal{G}$ is a simulation between the forests that remain when removing r and r' from DT and DT'. Thus, adding (r, r') suffices to generate a simulation between DT and DT'.□

With this lemma, it is now possible to proof the correctness and completeness of Algorithm 8.1.1.

Proposition 8.1.1
For any two elementary data trees DT_1 and DT_2, if and only if there is at least one elementary simulation between DT_1 and DT_2, algorithm 8.1.1 succeeds (i.e. has a return value of `true`*) and returns one elementary simulation between DT_1 and DT_2.*

Proof Sketch. Proposition 8.1.1 can be proven using a structural induction over the structure of DT_1. First, it is to be shown that if there is a simulation between DT_1 and DT_2 and DT_1 only consists of one node, then Algorithm 8.1.1 succeeds and returns the correct simulation (trivial). The induction step, when DT_1 consists of more than one element and there is a simulation between DT_1 and DT_2, can be proven by first investigating the equality of the root nodes (which is provided by the fact that there is a simulation), thus avoiding the unsuccessful return in line `04`. Then the problem is reduced to the child nodes in DT_1, where it is to be shown that if there is a simulation between DT_1 and

DT_2, then there is also a simulation between each of the child trees in DT_1 and one of the child trees in DT_2 (Lemma 8.1.1). This will avoid the unsuccessful return in line 18. With this result, it is possible to reduce the problem to the induction hypothesis (line 11). In the other direction, it is necessary to show that if Algorithm 8.1.1 succeeds, there is also a simulation between DT_1 and DT_2. This can again be achieved using structural induction: If DT_1 consists of only one node, Algorithm 8.1.1 only succeeds when the root nodes of DT_1 and DT_2 are equal (line 4) and thus there is a simulation. The induction step again compares the root nodes and reduces the problem to the child trees; if there is a simulation of each of the child nodes in DT_1 then there is also a simulation between DT_1 and DT_2 if the root nodes are equal (Lemma 8.1.1.□

8.2 Grouping Simulation

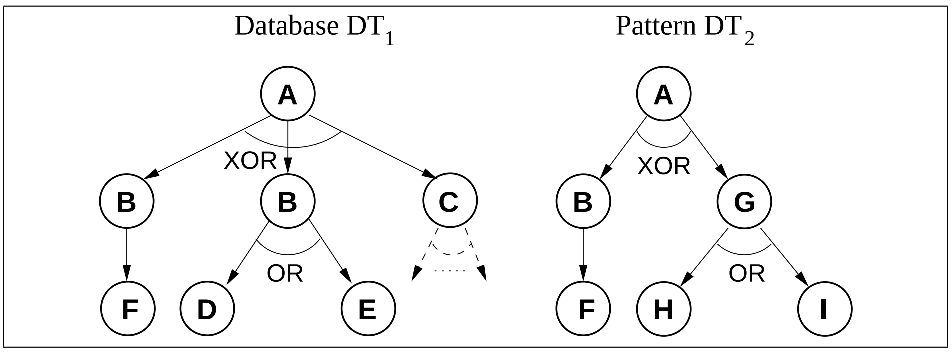
A very naïve approach for calculating the grouping simulation would be to just combine the algorithms for generating the interpretation and the algorithm for calculating the elementary simulation (Algorithm 8.1.1).

However, calculating the simulation thus is not very efficient, as it would first be necessary to calculate all of the interpretations of both data trees. This has already been mentioned in Section 3.2.

We improve this approach such that it takes into consideration whether it is necessary in terms of the simulation to compute the interpretation at a certain node or not. The interpretations for a certain node are only computed if they are needed in the current step of the calculation.

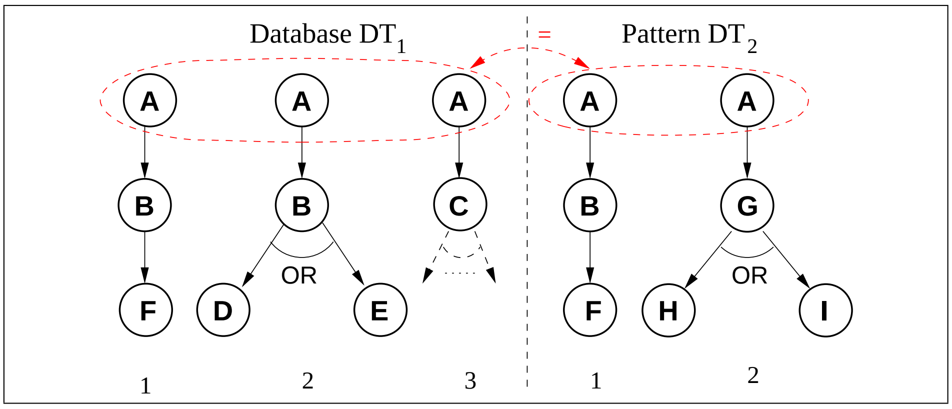
With such an improvement it is possible to restrict the number of generated interpretations in most cases, especially with a relatively large database compared to a relatively small pattern. This is comparable to a branch-and-bound search: While the complexity of the problem isn't reduced, it may save a lot of processing time in practice.

Example 8.2.1
Again, we illustrate the steps that are performed on a simple example. In the following figure we have a database and pattern with grouping constructs. Please note that there is some undefined part with grouping below the node "C" in the database.

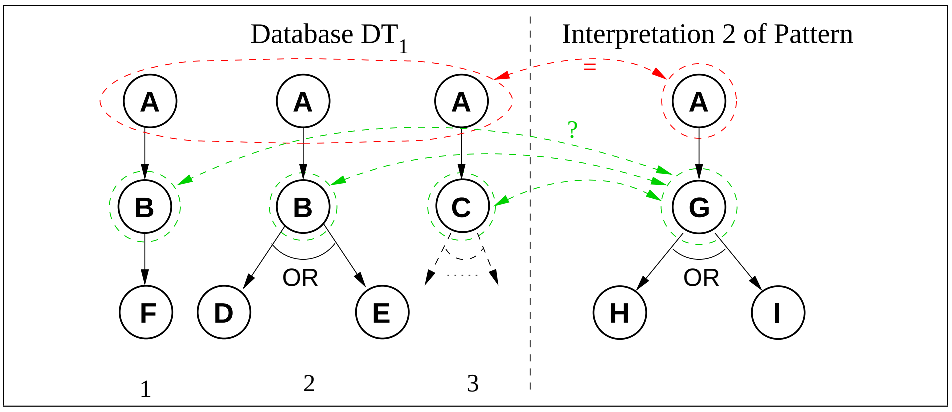


Obviously, generating the interpretations for these two trees would be very time- and space-consuming. The improved approach, however, works differently. For every node that is processed, we first check whether it is even a candidate, i.e. whether the name is equal to the one currently tested. If yes, the interpretations are generated. If no, this is not necessary.

In the example, the first step would require to generate the interpretations for the root node. This can be done by using the Algorithm 7.2.1 for generating the interpretations for a single node presented before:



In the next step, we try to find a simulation from one of the party generated interpretations of the pattern to one of the partly generated interpretations of the database. Obviously, in the figure it would not even be necessary to compute any further since one can see the simulation between interpretation 1 of the pattern and interpretation 1 of the database. However, we assume that it would first be necessary to find a simulation between interpretation 2 of the pattern and some of the interpretations of the database. The parts that are marked red already have completed the algorithm while the green parts are still to be tested:



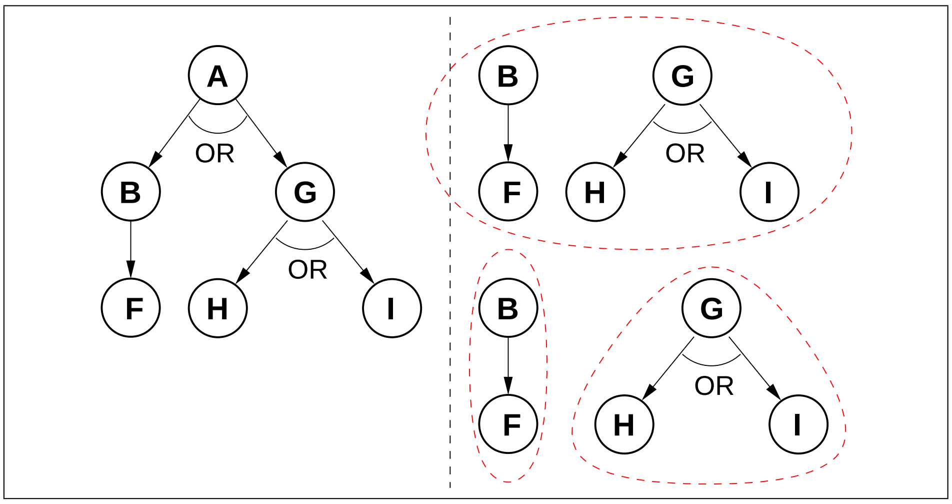
As one can see, there is no possibility to fulfill the equality relationship between the node names with any of the three interpretations in the database. Therefore, it is not necessary to investigate the interpretation 2 of the pattern any further, so the "OR"-facet will not be expanded.

Similarly, if the interpretation 1 of the pattern should find a corresponding *interpretation in the database, it is not necessary to investigate the node "C" any further. Thus, the simulation algorithm doesn't care about what is below the "C" and whether it contains grouping constructs or not.*

In contrast, if we would have generated the interpretations for the two data trees beforehand, it would have mattered what is below the "C" and below the "G".

In the following algorithm we assume that there exists a function `interpretations(Node)` that generates all of the interpretations for a given node, but does not recurse to the subtrees. The result of this function is a set containing combinations of child elements of the parameter node.

Example 8.2.2
In the following figure, the result of applying the interpretation function on the tree on the left side is shown as sets in the red, dashed circles.



If `nodeinterpretations` is applied to a node without grouping, it will just contain one set of all child nodes.

In order to save space we will use a more compact notation for the `FOREACH` statement. `FOREACH a IN A, b IN B` should construct all possible combinations of elements in A and B.

Algorithm 8.2.1 (Simulation for Data Trees with Grouping)

```
(00)  function gsimulation(root1:Node, root2:Node, VAR R:Relation)
(01)                                                    :boolean
(02)  VAR S,T,I,J: set; flag_node, flag_interp: boolean;
(03)  BEGIN

(04)    IF name(root1) <> name(root2) THEN RETURN false;

(05)    IF children(root1) = {} THEN
(06)    BEGIN
```

```
(07)      R := R UNION {(root1,root2)};
(08)      RETURN true;
(09)    END;

(10)    I := nodeinterpretations(root1);
(11)    J := nodeinterpretations(root2);

(12)    FOREACH interp1 IN I, interp2 IN J DO
(13)    BEGIN
(14)      T := {};
(15)      flag_interp := true;
(16)      FOREACH node1 IN children(interp1) DO
(17)      BEGIN
(18)        flag_node := false;
(19)        FOREACH node2 IN children(interp2) DO
(20)        BEGIN
(21)          S := {};
(22)          IF gsimulation(node1,node2,S) THEN
(23)          BEGIN
(24)            flag_node := true;
(25)            T := T UNION S;
(26)            break;
(27)          END;
(28)        END;
(29)        flag_interp := flag_interp && flag_node;
(30)      END;

(31)      IF flag_interp THEN
(32)      BEGIN
(33)        R := R UNION {(root1,root2)} UNION T;
(34)        RETURN true;
(35)      END;

(36)    END;
(37)    RETURN false;
(38)  END;

(39)  BEGIN
(40)    set R := {};
(41)    IF simulation(root1,root2,R)
(42)       THEN write("Simulation: ",R);
(43)       ELSE write("No simulation");
(44)  END;
```

Again, the algorithm needs some explanation. Lines 00 to 04 stay unchanged except for the name of the procedure. Lines 05 to 09 are the recursion stop, `root1` does not have any more child nodes. Lines `09` and `10` generate the interpretations for the root nodes of the two trees, but *only if the names match* and without recursing to the children.

The loop that begins in line `12` tries all possible combinations of interpretations from the two data trees. Since we don't know beforehand which of the interpretations provide a simulation, we use a temporary set `T` to store simulation results (line `14`) from the children.

The boolean variable `flag_interp` indicates that we assume that the interpretations match at first. This is possibly modified later (line `29`), when we find out that one of the nodes from the first interpretation doesn't have a partner in the second interpretation. Similarly, we use the variable `flag_node` that will be set to `true` if a partner has been found for the node.

For each combination, all nodes are chosen stepwise from the interpretation of the first tree (line `16`). This is similar to choosing the children of `root1` in Algorithm 8.1.1, line `06`. For each node, we try to find a corresponding node in the interpretation of the second tree. This is again done by recursively calling the `gsimulation` function in line `22`.

If there is a simulation for all of the nodes in the first interpretations to some nodes in the second interpretation, the variable `flag_interp` will have the value `true` and we will add the temporary set `T` and the pair `(root1,root2)` to the simulation `R`. After that, we can return success (lines `31-35`).

If the function didn't return true for at least one of the combinations, we can safely return the value `false`, as there is no simulation between interpretations of the two trees.

8.2.1 Correctness

Proposition 8.2.1
For any two data trees DT and DT' with grouping. If and only if there is at least one grouping simulation $\mathcal{G}$ between DT and DT' ($DT \xrightarrow{\mathbf{sim}^g}_{\mathcal{G}} DT'$), Algorithm 8.2.1 succeeds (i.e. has a return value of `true`*) and returns a grouping simulation.*

Proof. Let us first assume that there is a grouping simulation between DT and DT', i.e. there exists an interpretation I in the interpretations of DT such that there exists an interpretation I' in the interpretations for DT' that there is an elementary simulation between I and I'. This also means that the two root nodes of DT and DT' are equal since each of the interpretations is rooted. Thus line 04 will not return false. For the rest, it suffices to show that the variable `flag_interp` does not get `false` for at least one combination of interpretations, since this would break the loop and thus the algorithm will return failure. Without loss of generality we can assume that the loop in line 12 will at some point select the combination of the interpretations I and I'. Lines 16-30 effectively do the elementary simulation between the two interpretations, which has already been proven in Proposition 8.1.1, just that failure would set the variable `flag_interp` to `false`. The assumption however was, that there is an elementary simulation between I and I', thus the variable will keep it's initial value of `true` and the algorithm will return success.

Let us now assume that there is no grouping simulation between DT and DT', i.e. there exists no pair of combinations from interpretations of DT and DT' such that there is an elementary simulation between the two. Either the two roots of DT and DT' are unequal. Then the algorithm will return failure in line 04. Or some nodes deeper in the tree don't match, which is covered by the recursion in line 22, so that each node will be "root" at some point in the recursion.$\square$

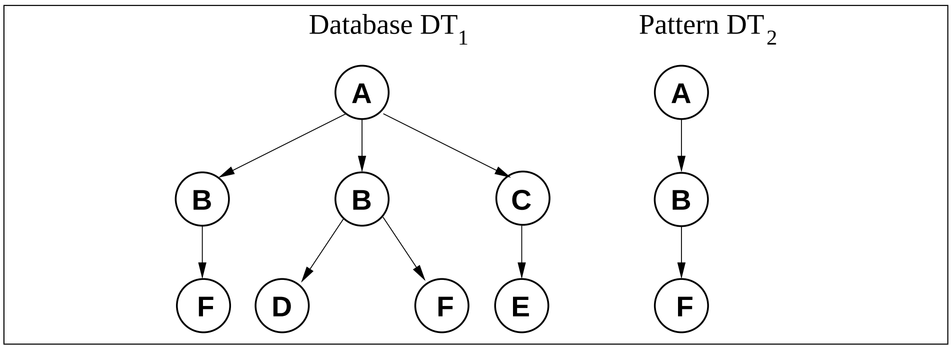
Chapter 9

Localization

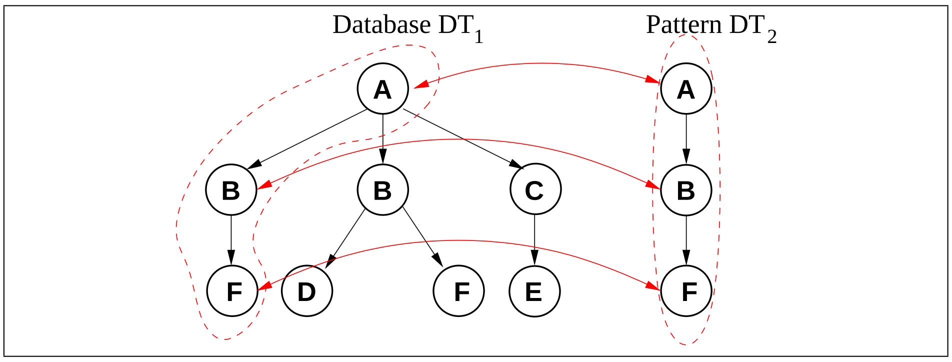
9.1 Maximal Simulation

Calculating the maximal simulation between two data trees actually only requires minor changes to Algorithm 8.1.1. Where it is sufficient in Algorithm 8.1.1 to find for each child of the first root node the "first" matching child in the children of the second root node, we just need to retrieve "all" matching children from that set.

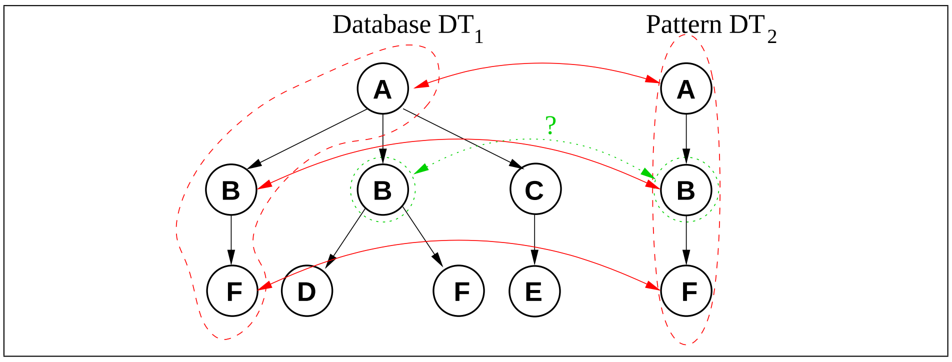
Example 9.1.1
Consider the two trees given in the next figure.



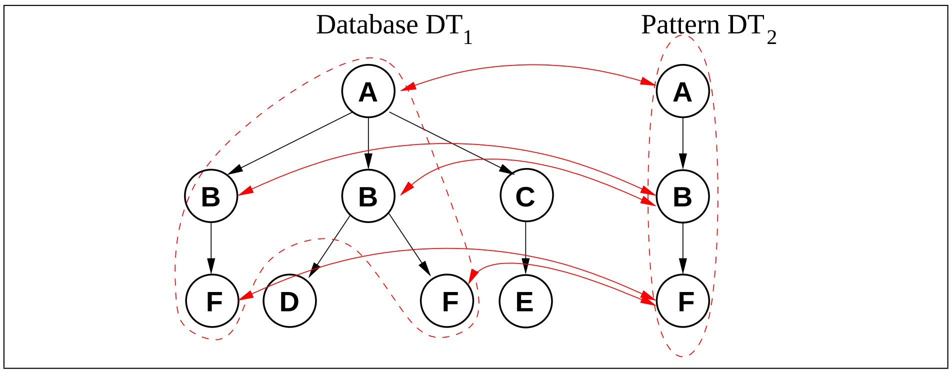
Calculating the elementary *simulation would result in the following matching:*



Thus, in order to calculate the maximal simulation, it would be necessary to *not only choose the first alternative for matching B, but also to investigate the other possible match:*



Since the second branch also matches, it can be added to the simulation, thus creating the maximal simulation between DT_1 and DT_2.



The modified algorithm for the maximal simulation just requires to remove the `break` statement in line `16`. This will provide that the inner loop checks all of the children of `root2` instead of stopping after the first match.

The boolean variable `flag` will indicate whether the simulation was successful or not and allow to abort in the latter case.

Algorithm 9.1.1 (Maximal Simulation for Elementary Data Trees)

```
(00)  function maxsimulation(root1:Node, root2:Node, VAR R:Relation)
(01)                                                       :boolean
(02)  VAR S,T: set; flag: boolean;
(03)  BEGIN
(04)    IF name(root1) <> name(root2) THEN RETURN false;

(05)    T := {};
(06)    FOREACH node1 IN children(root1) DO
(07)    BEGIN
(08)      flag := false;
(09)      FOREACH node2 IN children(root2) DO
(10)      BEGIN
(11)        S := {};
```

```
(12)        IF maxsimulation(node1,node2,S) THEN
(13)        BEGIN
(14)          flag := true;
(15)          T := T UNION S;
(16)          (* break; *)
(17)        END;
(18)      END;
(19)      IF NOT flag THEN RETURN false;
(20)    END;

(21)    R := R UNION { (root1,root2) } UNION T;
(22)    RETURN true;
(23)  END;

(24)  BEGIN
(25)    set R := {};
(26)    IF simulation(root1,root2,R)
(27)       THEN write("Simulation: ",R);
(28)       ELSE write("No simulation");
(29)  END;
```

9.1.1 Correctness

Again, it would be handy to proof the correctness of this algorithm using a simple proposition.

Proposition 9.1.1
For every two data trees DT and DT'. If there is a simulation between DT and DT', Algorithm 9.1.1 always finds the maximal simulation between DT and DT'.

Since there is at most one maximal simulation, completeness is trivial. For the proof it thus suffices to show correctness.

Proof. First it is shown that Algorithm 9.1.1 always finds a simulation between the two data trees if there is at least one. Since Algorithm 9.1.1 is just a slightly modified version of Algorithm 8.1.1, the result from Proposition 8.1.1 can be used. It suffices to show that the missing `break`-statement doesn't influence the result. This is trivial, since the boolean variable `flag` in line 14 will never modify it's value after switching to `true` and the set `T` just gets additional tuples and does not loose anything.

Second, it is necessary to show that Algorithm 9.1.1 always finds the *maximal* simulation between the two data trees. Assume there is only one simulation between DT and DT'. This will also be the maximal simulation and thus Algorithm 9.1.1 succeeds. If there is more than one simulation, then the "branching" would occur with the `break` statement in line 16. Without this statement, the algorithm tries all branches (lines 06 and 09) and thus `T` will contain the union over all possible simulations, which is the maximal simulation.□

Part IV

Conclusion

Chapter 10

Related Work

Parts of this work have been accepted for the WebH2001 conference in September 2001 [BOS01].

A different approach to localization queries in SSD is used by XPath [xpa99]. The difference between XPath and our localization approach is that the localization is done by a path instead of a tree and the result usually is a set of nodes instead of a combined answer.

Inspirations for the topic have originated from the paper [MSB01], where matching for elementary data trees with *aggregated answers* has been proposed. However, our work goes beyond and presents an enriched SSD data model based on adding grouping constructs, i.e. aggregated trees also for databases and patterns.

A collection of tree matching problems, called tree inclusion problems, has been addressed in [Kil92], where the ordered/unordered node-labeled tree model has been used. [Kil92] provides also an extension of tree inclusion problems by logical variables used to extract substructures of the pattern instances and to express equality constraints on them.

The work presented here is also related to semantic modeling in general, see e.g. [HK87] and [Tha00].

Chapter 11

Prospects

The data model we described in this paper is by no means complete. Many issues are still ongoing work . This section provides a quick overview over this ongoing work. Also, possible research directions for the future are shown.

11.1 Cleaner Definition of Interpretation

The current definition for the interpretation of a data tree with grouping (Definitions 3.2.1 and 3.2.2 is very vague and unprecise. A more formal approach would be useful.

A cleaner definition could be inductively defined as follows:

First, a basic case is formulated that covers the trivial and the inductive case:

$$\mathcal{I}(N()) = \{N()\}$$
$$\mathcal{I}(N(T_1, \dots, T_n)) = \{N(T'_1, \dots, T'_n) | T'_i \in \mathcal{I}(T_i), 1 \leq i \leq n\}$$

The difference between $N\{\dots\}$ and $N(\dots)$ could be solved with the following two interpretations. Note however, that the semantics here is slightly different than in the table presented earlier.

$$\mathcal{I}(N_{\mathcal{G}}\{\}) = \{\mathcal{I}(N_{\mathcal{G}}()\}$$
$$\mathcal{I}(N_{\mathcal{G}}\{T_1, \dots, T_n\}) = \{\mathcal{I}(N_{\mathcal{G}}(T_{\pi(1)}, \dots, T_{\pi(n)})) \mid \pi \text{ permutation of } \{1, \dots, n\}\}$$

For the connector facet, the interpretations might look as follows:

$$\mathcal{I}(N_{AND}()) = \dots$$
$$\mathcal{I}(N_{AND}(T_1, \dots, T_n)) = \mathcal{I}(N(T_1, \dots, T_n))$$
$$\mathcal{I}(N_{OR}()) = \dots$$
$$\mathcal{I}(N_{OR}(T_1, \dots, T_n)) = \bigcup\{\mathcal{I}(N(P_1, \dots, P_k)) \mid (P_1, \dots, P_k) \subseteq (T_1, \dots, T_n), 1 \leq k \leq n\}$$

$$\mathcal{I}(N_{XOR}()) = \ldots$$
$$\mathcal{I}(N_{XOR}(T_1, \ldots, T_n)) = \{N(T_i') \mid T_i' \in \mathcal{I}(T_i), 1 \leq i \leq n\}$$

Note however, that there are difficulties in this formalism: It is not clear how to treat grouping facets on leaf nodes.

This formalism has not been chosen for the thesis as it will require more research for some of the mentioned problems and also it will be necessary to develop such rules for all of the possible grouping facets.

11.2 Combining Grouping Facets

A topic that has not been addressed in this paper is the combination of several grouping facets for the same group of nodes.

At first glance, the issue may seem trivial, but it is not: Combining facets can give very different meanings to a set of nodes (consider e.g. the AND-connector and the depth facet).

For example, in some cases it might be relevant in which order the grouping facets of a node are interpreted (the two choices are not confluent). Other combinations might not make sense at all, how should they be interpreted?

Therefore, refining the semantics presented in Section 3 so as to accommodate multiple grouping is worth investigating.

11.3 Arbitrary Graph Structures

In this paper we restricted the model to tree structured databases. However, it would also be desirable to extend this to databases having an arbitrary graph structure.

In some areas, the extension to graph structures is quite simple. The simulation definitions, for example, have been used for real graphs in other fields already. Other areas, like the matching with variables and especially the depth facet, may need further modifications.

11.4 Non-Rooted Matching

In this paper, matching algorithms always begin with the root of the database and the pattern.

However, in many applications it might be desirable to match some pattern with some substructure of the database. As already discussed in Section 4.3, such matching can be achieved using the depth facet. But as we have seen in Section 6.3, the representation and calculation of the depth facet is neither convenient nor efficient, so there might be other means to calculate such matching.

While the simulation technique allows the use of non-rooted matching, research is necessary in the field of answer semantics and implementation.

11.5 Combined Answer

The table given in section 5.2 for the combined answer only covers three of the grouping facets. An extension to the other grouping facets is desirable.

Furthermore, a deeper discussion of each of the combinations, also with respect to implementations, will be necessary.

11.6 Improved Treatment of Depth Facet

The interpretation and treatment of the depth facet has only been introduced shortly in this paper. There already exist more efficient means of representing such structures.

Inspirations can come from the fields of formal languages and especially regular expressions that have been investigating on the topic of efficient calculation of dynamic-length structures (see e.g. [MSB01]). Furthermore, similar topics have already been investigated in graph theory like [Kil92].

11.7 More Investigation on Variables

Variables have only been covered very shallow. Many possibilities can be investigated in future works, especially with multiple occurrences. Different, more convenient equality relationships can be introduced and examined.

Possibly there is also a way to have a "combined answer" for variables like the one that has been proposed for matching without variables.

Also, algorithms for matching with variables have not been given, as they can pose serious problems. Ideas to overcome this problem can come from "unification" that is e.g. used in logic programming (see [SS94]). Much research has already been done in this area.

11.8 Efficient Algorithms

The algorithms presented in this paper show the idea of matching with grouping constructs, but they are not necessarily the most efficient algorithms for calculating the results.

Tree matching in general has for example been investigated in [Kil92]. Further approaches could use indexing structures over the children of a node, similar to path indices used in theorem proving (see [EO93] and [Niv99]), which is similar in terms of tree/graph matching.

11.9 Implementations

One of the main issues is to bring the presented ideas into an algorithmic form. Currently finished is an implementation of matching with and without grouping, but generating results like presented in Section 6 is not yet possible.

Implementations of the interpretation and elementary simulation have been given in form of a Haskell program in Appendix 14. Grouping simulation has been implemented in Java, but is left out of this thesis due to space reasons.

11.10 Grouping Constructs for Query Optimization

Query languages for semi-structured data are currently very new. Due to the richness of the concept compared to the well-investigated relational databases, query optimization is at the moment very limited.

Grouping Constructs could provide an approach to optimize a query to some data due to the fact that many elementary data trees are aggregated to one data tree with grouping.

One could imagine algorithms that generate results in an aggregated answer, thus not only representing the results in a more compact way but also potentially providing a more efficient way of calculating the results.

Part V

Appendix

Bibliography

[Abi97] Serge Abiteboul. Querying semi-structured data. Technical report, INRIA-Rocquencourt, 1997.

[ABS00] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web. From Relations to Semistructured Data and XML.* Morgan Kaufmann Publishers, San Francisco, CA, 2000.

[BDS95] P. Buneman, S. Davidson, and D. Suciu. Programming constructs for unstructured data. In *DBLP*, 1995.

[BOS01] François Bry, Dan Olteanu, and Sebastian Schaffert. Towards grouping constructs for semistructured data. In *WebH2001 – International Workshop on Electronic Business Hubs (to appear)*, http://www.comp.nus.edu.sg/ icom/misc/WORKSHOP/index.html, 2001.

[CGL99] Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. Modeling and querying semi-structured data. *Network and Information Systems*, 2(2):253–273, 1999.

[CGMH$^+$94] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogenous information sources. In *Information Processing Society of Japan*, 1994.

[dam00] Defense Advanced Research Projects Agency. *The DARPA Agent Markup Language (DAML)*, 2000.

[ddm99] W3C, http://www.w3.org/TR/NOTE-ddml. *Document Definition Markup Language (DDML) Specification, Version 1.0*, Jan. 1999.

[EO93] N. Eisinger and H. J. Ohlbach. Deduction systems based on resolution. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming - Vol 1: Logical Foundations. Oxford, Clarendon Press*, pages 183–271. 1993.

[FKS00] Wenfei Fan, Gabriel M. Kuper, and Jérôme Siméon. *A Unified Constraint Model for XML.* Temple University, Bell Laboratories, 2000.

[FS00] Wenfei Fan and Jérôme Siméon. *Integrity Constraints for XML.* Temple University, Bell Laboratories, 2000.

[HHK96] Monika R. Henzinger, Thomas A. Henzinger, and Peter W. Kopke. Computing simulations on finite and infinite graphs, July 1996.

[HK87] Richard Hull and Roger King. Semantic database modeling: Survey, applications, and research issues. *ACM Computing Surveys*, 19(3):201–260, September 1987.

[ISO86] International Standards Organization (ISO). *ISO 8879. Information processing – text and office systems – Standard Generalized Markup Language (SGML)*, 1986.

[JTM92] R. Durbin J. Thierry-Mieg. Syntactic definitions for the ACeDB data base manager. Technical report, MRC-LMB xx.92, MRC Laboratory for Molecular Biology, Cambridge, 1992.

[Jun94] Dieter Jungnickel. *Graphen, Netzwerke und Algorithmen.* BI Wissenschaftsverlag Mannheim, 1994.

[Kil92] Pekka Kilpeläinen. *Tree matching problems with application to structured text databases.* PhD thesis, Department of Computer Science, University of Helsinki, 1992.

[Knu97] Donald E. Knuth. *The Art of Computer Programming: Fundamental Algorithms.* Addison Wesley, 3rd edition, 1997.

[Kröar] Peer Kröger. Modeling of biological data. Master's thesis, Institute for Computer Sciences, University of Munich, http://www.pms.informatik.uni-muenchen.de/lehre/projekt-diplom-arbeit/biological-data.html, 2001, to appear.

[Lam86] Leslie Lamport. *LaTeX: A Document Preparation System.* Addison-Wesley, 1986.

[MSB01] Holger Meuss, Klaus Schulz, and François Bry. Towards aggregated answers for semistructured data. In *International Conference on Database Theory*, 2001.

[Niv99] Hans De Nivelle. Datastructures for resolution, 1999.

[oil02] On-To-Knowledge IST Programme, http://www.ontoknowledge.org/oil/. *Ontology Inference Layer (OIL)*, 1999-2002.

[Pat88] Oren Patashnik. BibTeXing. Documentation for general BibTeX users, 8 February 1988.

[RCF00] Jonathan Robie, Don Chamberlin, and Daniela Florescu. *QUILT: an XML query language*. http://www.almaden.ibm.com /cs/people/chamberlin/quilt_euro.html, March 2000.

[Rob99] Jonathan Robie. *XQL: XML Query Language*. http://metalab.unc.edu/xql/xql-proposal.xml, August 1999.

[SS94] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. MIT Press, Cambridge, Massachusetts, second edition, 1994.

[Tha00] Bernhard Thalheim. *Entity-Relantionship Modeling. Foundations of Database Technology*. Springer, 2000.

[Tho99] Simon Thompson. *Haskell: The Art of Functional Programming*. Addison-Wesley, second edition, 1999.

[Wad00] Philip Wadler. *A formal semantics of patterns in XSLT*. Bell Labs, Lucent Technologies, March 2000.

[xml98] W3C, http://www.w3.org/TR/1998/NOTE-XML-data-0105/. *XML-Data*, Jan. 1998.

[xml00a] W3 Consortium, http://www.w3.org/TR/2000/WD-xmlschema-0-20000407/. *XML Schema*, 4 2000.

[xml00b] W3 Consortium, http://www.w3.org/XML/. *XML Specification*, 2000.

[xpa99] W3 Consortium, http://www.w3.org/TR/xpath. *XML Path Language (XPath)*, 1999.

[xpt00] W3 Consortium, http://www.w3.org/TR/xptr. *XML Pointer Language (XPointer)*, 2000.

[xqu01] W3 Consortium, http://www.w3.org/TR/xquery/. *XQuery: A Query Language for XML*, Feb 2001.

[xqw] *XML Query working group*. http://www.w3.org/XML/Query.

[xsl00] W3 Consortium, http://www.w3.org/Style/XSL/. *Extensible Stylesheet Language (XSL)*, 2000.

Chapter 13

Application Examples of Individual Grouping Facets

This chapter contains some informal discussion of grouping constructs that has been developed prior to the formal discussion in this thesis.

Nonetheless it might be useful to the reader as there are examples for each of the grouping facets. Furthermore, the examples are mostly closer to real-world applications than the "abstract" examples presented so far.

Note that there are additional facets as well as additional properties for some facets which are not part of the previous discussion. This is mainly due to the fact that they are either redundant or cannot be expressed properly with the formalism that is proposed in Part 1.

13.1 Sibling Relationships

Two nodes are "siblings", if they share the same immediate parent node in the tree structure.

Sibling relationships specify facets of a group of data items at the same level in the tree structure that models the data, i.e. between "siblings".

The facets covered here include "connector", "order" and the "repetition" facet.

13.1.1 Connector Facet

The Connector Facet describes connections between data items that are given a "simple" connection information. Typically this will be some sort of boolean connection like "and", "or" and "xor", but it is also conceivable that the information is a user defined qualifier.

Example 13.1.1
Consider that you want to buy a car. Most likely you will first have to assemble the "special" equipment that you want to have like air conditioning, radio, etc.

Now consider there is an electronic database containing all of the special equipment that can be used. This database should contain some constraints to the data, such that the user can only select a sensible configuration of the equipment:

Since it is reasonable to assume that a car only can have one radio, all available radio models would be "xor"-related. On the other hand, it will be possible to have both a radio and air conditioning, so these would be "and"-connected.

Also note that the relationship is not necessarily between only two data items but also between the data items of a set of arbitrary length.

For some data types it will also be useful to already specify relational characteristics in the schema, not only in the data itself. Hence it will be possible to specify a default connection between elements (that may optionally be overridden by individual data items, see "Overriding" below).

Example 13.1.2
Imagine that you want to give a schema for a database containing cars that clients have requested using the previously mentioned database for special equipment.

It would be possible and sensible to specify that a car can only have one radio in advance since a car with more than one (or no) radio is not very typical. Thus we will specify in the schema that the default for the connection between radios is "xor".

Summary

- the connection facet is applicable in both the data itself and in the schema for a set of databases
- possible connections include boolean operations like "and", "or" and "xor" and user defined qualifiers

13.1.2 Ordering Facet

In XML datasets, all child nodes are always considered "ordered", i.e. elements with the same children but in different order are not considered equal. Although the reason for this is obvious as XML originated from SGML where a dataset always was a text document, this is not useful in many XML applications that are just used as a database, e.g. in many electronic commerce applications.

In these applications it would be more convenient to be able to specify exactly whether the child nodes of an element are to be considered ordered or not.

Example 13.1.3
Imagine that you have an addressbook with several address entries each having between zero and several email addresses.

While for many entries that have several email addresses the order in which they are returned in a query is not important, it might be useful for some entries where the first email address should be preferred over the others. Thus order will be necessary in the latter case while it isn't in the first example.

Again, this facet is not only applicable in the data itself but also in schemas where the structural information for a whole set of possible applications is specified.

Example 13.1.4
Again consider that you have an electronic addressbook consisting of address entries. It is easy to see that the order between the entries is not important and that this is an information that will be valid for every instance of an electronic addressbook, at least if some query mechanism is provided (non-electronic addressbooks are usually ordered alphabetically, but this is just an "index" to improve the query to the data items, so we can safely assume that addressbooks do not require to be ordered).

When you compare this "addressbook database" to a (XML) document representing for example a book with chapters, sections, paragraphs and so on, it is obvious that order is important in the book and that this statement is valid for (almost) all books.

Additionally, it may sometimes be necessary to specify the criteria by which elements should be ordered. A simple and obvious example is if some (generic) application wants to insert a data item beneath a node. If the children of the node are ordered, the application could insert the data item at the right place by using the sorting criteria specified in the node and applying some sort algorithm on it (which is then up to the application).

One special sort criterion can be a user defined preference of some items over others, as mentioned in the example given above. Indeed, a preference is just a order between elements.

Summary

- the ordering facet is applicable in the data as well as in the schema for some database
- possible values are "ordered" or "unordered"
- it will be useful to specify the criteria by which a list of data items is sorted
- preference can be expressed by using the ordering facet

13.1.3 Repetition Facet

In databases it is common that entries of the same type are repeated and thus provide the same kind of information for a set of entries. This is a very simple property in "flat" relational databases. However, if the database is structured as a tree, this kind of repetition allows to represent much more information but at the same time gets more complicated.

To allow to express this information more efficiently, a third sibling facet called "repetition" is introduced. It specifies how often the nodes of a given type are allowed to be repeated beneath the parent node.

It is useful to give both a lower and an upper boundary which can take integer values between 0 and infinity. This will allow an exact specification of the repetition and provide a reasonable integrity constraint.

The most obvious application of this repetition facet is in a schema (in fact something similar is already part of XML-Schema). For each data type declaration it is possible to specify exactly how often it may occur beneath each parent.

Example 13.1.5
In the addressbook example it might be reasonable to limit the number of email addresses to at least one and at most three addresses so that it is ensured that every person can be reached by email and the choice is not too big.

This information can be specified in the schema for the addressbook database.

However, there is also a (not-so-obvious) application of repetition in the data itself: The repetition facet can be used as an integrity constraint for some specific dataset, specifying that a certain node can only have a constrained number of occurrences of some data type as child nodes.

Example 13.1.6
Imagine that you have a database consisting of car models that you want to equip with some special parts from another database (the one given above).

While it is reasonable to assume that most cars will only take one battery, there are some models that will use between one and two batteries (e.g. camping vans). Therefore, it should be possible to specify such an information for each individual model and not for the whole database.

Summary

- the repetition facet is mainly applicable in schemas, however there are also cases where they can be used efficiently in the data sets
- for the repetition facet there should be at least two attributes, one giving the minimum number of occurrences, the other giving the maximum number
- these attributes should take integer values between 0 and infinity

13.2 Parent-Child Relationships

There are also facets that give additional information about a parent-child relationship, i.e. the connection is between data items at different levels in the tree structure. Usually this provides some sort of restriction for either the parent or the child element.

Of these, the "dependency", the "selection", the "exclusion" and the "depth" facet will be introduced.

13.2.1 Dependency Facet

The "dependency" facet allows to express a dependency of some sort between the parent element and the child elements, e.g. a partial or hereditary relationship. Possible values of this facet are "part-of", "is-a", "has" or any user defined qualifier. Again, the three predefined values will cover most of the cases but it is also possible for a user to specify his own qualifier.

Since all of these values depend on the direction of the relationship i.e. whether the relation is from the parent to child or from the child to the parent, it will be necessary to explicitly specify the direction.

Example 13.2.1
If you again consider the database that contains special equipment for cars, it could be sensible to have an element "radio" that contains all of the possible models as children:

```
      Radio
   /    |    \
Sony Grundig Kenwood
```

However, if you have a look at the pure structure of this example and don't know what a Radio actually is (and a computer usually doesn't), it can mean anything! Hence it is useful to model the additional information that the relation is a "is-a" relation from the child to the parent.

```
      Radio { relation="is-a", "child->parent" }
   /    |    \
Sony Grundig Kenwood
```

Using this facet can be reasonable in the data itself as well as in the schema for a set of documents as an integrity constraint and a selection criterion.

Summary

- the dependency facet specifies a dependency-relationship between parent and child elements

- it is applicable in schemas as well as in the data itself
- possible values are "part-of", "is-a", "has" and user defined qualifiers
- the direction of the relationship has to be given explicitly

13.2.2 Selection Facet

The selection facet allows to specify that in case of a query a subset or sublist from the child elements of the element carrying this facet should be selected.

Possible values are

- "all" (which is usually the default), all elements should be selected from the child nodes
- "exactly n", exactly n elements should be selected, n is a positive integer number
- "between n and m", between n and m elements should be selected
- "some", any number but at least one should be selected
- "none", none of the elements may be selected

Example 13.2.2
Imagine again that you have a database of car equipments. One possibility would be to classify cars into "predefined" classes for a fixed price that allow you to select e.g. 3 special options (like air conditioning, radio with compact disc and airbag) from a set of e.g. 10 possibilities.

These values certainly overlap ("exactly n" equals "between n and n", ...), but are chosen because of the additional expressive power that is gained for the reader and the implementation doesn't become more complicated with the redundant information.

The selection facet is applicable in both the data (where for a given set of nodes the limits for selecting a subset of them can be specified) and in a schema where this information can be provided for all occurrences of the given data type. In most cases this facet will be used as a selection criterion and most likely no as an integrity constraint.

Furthermore it is also possible to use this facet at a "higher" level in the schema language by applying it to the structure of the schema document.

Example 13.2.3
Imagine that you want to create a schema for an element "a" that has at least two of a given set of subnodes. You could realize this by using the selection facet for the structure of the schema language, adding the information that of the element a's possible child nodes at least two have to occur.

Note that it is possible to express all of the logical relations mentioned in the "connector facet" by using the subset constructs: an "or"-relation can be an "some"-selection, an "and"-relation can be an "all"-selection, "xor" is "exactly 1" and a "not"-relation is a "none"-selection. Still, this is a "parent-child" relation because it also specifies a property of the parent by limiting the child elements, i.e. the parent is the focus here, while in the logical relations, the siblings are of interest.

Summary

- the "selection facet" allows to specify that a subset of the children with certain restrictions should be selected
- it is applicable in the data as well as in the schema, and in the latter it may be used as a "general" feature of all elements of a type and as a structure to describe the data in the schema
- possible values are "all", "exactly n", "between n and m", "some" and "none"

13.2.3 Exclusion Facet

A problem of databases is that it is unstated whether the fact that an entry is missing means that it it really not there or whether you just don't know and it might be inserted as a fact at a later point (non-monotony of negation).

Hence it will often be useful to explicitly state that an entry should be negated. In the database itself this can be used as an integrity constraint for a parent element, specifying that a data item of a specific kind must not be inserted, thus avoiding the problem of non-monotony.

Example 13.2.4
Again consider an addressbook. Perhaps there is an entry of a person (perhaps your grandmother) where you know that he/she will never have an email address, so you might want to say that it is incorrect to add one there.

The exclusion facet also has a useful application in the schema for a query to express that a certain node should not be among the child nodes of returned nodes.

Example 13.2.5
Imagine that you want to query your addressbook for all people that don't have a telephone number in the entry. Using the exclusion facet, you could say that there should not be a node of type "phone" below a node of type "entry".

In the schema that specifies the structure of a database, it will probably be used less frequently, since it doesn't make sense to explicitly forbid the existence of certain child elements as this information is already given if the said element does not exist in the grammar.

Summary

- the “exclusion facet” specifies that a (set of) specific nodes(s) may not appear beneath the parent node
- it is applicable as an integrity constraint in the data and in a schema for queries; application in the schema that specifies the structure doesn’t seem useful

13.2.4 Depth Facet

As a last group facet, “depth” is introduced. With this facet, the upper and lower limits of a group of data item’s depth below the parent (in the sense of the levels of the data tree) can be expressed. Thus it is possible to skip a variable number of levels between the parent and the child element.

Possible attributes are the upper and the lower limit, each taking positive integer values. The lower limit will then represent a structure like “at least n levels below the parent” and by analogy the upper limit will represent “at most n levels below the parent”.

It is obvious that this facet is only reasonably used in schemas that specify the structure or a query to the database since the information is at a meta level that describes the structure of the database and not the relationship between the items themselves.

Example 13.2.6
Consider that you have a tree structure like the one given below:

```
  P
 / \
A   B
```

Now imagine that you want to specify that the node A should be at any depth, while the node B should be either immediately below the parent or at most 3 levels below it. To provide this information, it will be necessary to name the connections between the nodes such that the connection P-A carries the information that it can span any number of levels while the connection P-B can span any number between 1 and 3.

Summary

- the depth facet represents a variable number of skipped levels between a parent and its children
- it is applicable in schemas only

13.3 Combining Facets

In many applications it will make sense to combine several of the mentioned grouping facets in one grouping structure and thus providing even more detailed information about the data items.

Example 13.3.1
It would for example be possible to combine the selection and ordering facets and in this way creating an integrity constraint that requires queries to not only return a subset but an ordered sublist.

However, there are also some implications that have to be considered: Which facets fit together (selection and ordering would obviously fit) and which do not?

Chapter 14

Sample Implementations and Data

In this appendix, sample implementations for the interpretation- and elementary simulation algorithm are given. Both are written in the functional programming language Haskell, which allows a very compact and declarative presentation. Note that the implementation does not follow the algorithms presented earlier very closely, due to the fact that we have to follow functional programming paradigms.

An implementation for elementary and grouping simulation has also been done in Java, but is not included here due to the space restrictions of this paper.

14.1 Interpretation Algorithm

14.1.1 The implementation

Here an implementation for the algorithm for generating the set of possible interpretations for a data tree with grouping facets will be presented. The algorithm is implemented as a Haskell program that takes an XML document with grouping facets in the form of attributes as input and generates an XML document with a root node `<result>` and all possible elementary data trees as children of this root.

–

```
-- This Haskell programme provides an algorithm for generating
-- the set of possible interpretations for a data tree with
-- grouping facets

--
-- Usage: interpretXML <file.xml>
-- will send the interpretation of file.xml to STDOUT
```

```haskell
import Prelude

import Xml2Haskell
import XmlTypes
import XmlCombinators
import XmlLib


-- Helper function: create the superset of a set
superset :: [a] -> [[a]]
superset [] = [[]]
superset (x:xs) = [x] : [ x:l | l <- superset(xs), length(l) > 0 ]
  ++ superset(xs)


-- Do some (recursive) combinations of lists (cross-product of
-- all lists in the list).
-- The input is a list of lists, the output a list of lists
-- containing all possible combinations of single elements of
-- the input lists
recCombine :: [[a]] -> [[a]]
recCombine [] = [[]]
recCombine (x:xs) = [ l:lr | l <- x, lr <- (recCombine xs) ]


-- Filter function to remove a grouping attribute from a list of
-- attributes
removeGrouping :: String -> [Attribute] -> [Attribute]
removeGrouping s attr =
    filter (\n -> not (n== ("grouping",AttValue [Left s]))) attr

-- Converts a data tree with grouping facets into a set of
-- elementary data trees

-- This set is then the set of possible interpretations for
-- the data tree.

-- Each of the elementary trees is a model for the data tree.
-- The function is an implementation of CFilter from HaXml
interpretXml :: CFilter

-- For elements that don't have child nodes, the grouping facet
-- has no relevance
interpretXml (CElem (Elem n attr [])) = [(CElem (Elem n attr []))]


-- The main function
```

```
interpretXml (CElem (Elem n attr children))

-- in the AND case the result is just one node, but interpretXml
-- has to be applied recursively

    | ("grouping",AttValue [Left "AND"]) `elem` attr =
[ (CElem (Elem n (removeGrouping "AND" attr) elem)) |
  elem <- recCombine (map interpretXml children)]


-- For XOR, a separate node is generated for each child node, at
-- the same time applying interpretXml recursively to each child

    | ("grouping",AttValue [Left "XOR"]) `elem` attr =
[ (CElem (Elem n (removeGrouping "XOR" attr) [l3])) |
  l2 <- (map interpretXml children), l3 <- l2]


-- OR is the most difficult case because there are a lot of
-- possible combinations. This is achieved by using the superset
-- function

    | ("grouping",AttValue [Left "OR"]) `elem` attr =
[ (CElem (Elem n (removeGrouping "OR" attr) elem)) |
  l2 <- (superset children), length(l2) > 0,
  elem <- (recCombine (map interpretXml l2)) ]


-- The default case is like the AND case

    | otherwise = [ (CElem (Elem n attr elem)) |
    elem <- recCombine (map interpretXml children)]

-- process the input XML file with grouping facets and produce a
-- result set of elementary data trees without grouping facets
-- (in form of an XML document with root node <result>)
main = processXmlWith (mkElem "result" [interpretXml])

-- Compilation:
-- ghc -package text -package lang -o interpretXML interpretXML.hs

--
```

14.1.2 Sample data

The following small XML document has been used as sample data to illustrate the execution of the interpretation algorithm.

```
<?xml version='1.0'?>

<!DOCTYPE A SYSTEM "test.dtd">

<A grouping="OR">
  <B/>
  <C grouping="XOR">
    <D/>
    <E/>
  </C>
</A>
```

14.1.3 Result for Sample Data

This is the result document generated from the example data.

```
<?xml version='1.0'?>
<result>
  <A>
    <B/>
  </A>

  <A>
    <B/>
    <C>
      <D/>
    </C>
  </A>

  <A>
    <B/>
    <C>
      <E/>
    </C>
  </A>

  <A>
    <C>
      <D/>
    </C>
  </A>
  <A>
    <C>
      <E/>
    </C>
  </A>
</result>
```

14.2 Simple Matching Algorithm

14.2.1 The implementation

The simple matching algorithm performs a "simulation" between two elementary data trees, one of them the pattern and the other a database. Since the simulation is "one-way", the matching is *liberal*, i.e. the database may contain more children than specified in the pattern for a successful matching.

–

```
-- This Haskell programme provides an algorithm for the simple
-- pattern matching between elementary databases and elementary
-- patterns

--
-- Usage: matching <database.xml> <pattern.xml>
-- will return "yes" if the pattern matches the database

import Prelude

import Xml2Haskell
import XmlTypes
import XmlCombinators
import XmlParse
import XmlLib

import IO

-- This is a simple recursive implementation of the simulation
-- algorithm for elementary data trees.
-- First argument: Database
-- Second argument: Pattern

-- Attributes and order are not considered, and if elements with
-- the same name and childs occur more than once below a single parent
-- they are treated as only one
matches :: Content -> Content -> Bool

matches (CElem (Elem a _ children1)) (CElem (Elem b _ children2))

-- if the names of the elements are different, they don't match
        | not ( a==b ) = False

-- if the element in the pattern has no children but the tagname
-- equals the one in the database, it matches (the database might
-- have additional children!)
```

```
        | length (children2) == 0 = True

-- else it returns true, if there exists a pair of a child node of
-- the database and a child node of the pattern that satisfies the
-- condition for each of the child nodes of the pattern
-- (looks complicated but isn't! :-) )
        | otherwise = and [ or [ matches c1 c2 |
                                 c1 <- children1 ] |
                            c2 <- children2]

-- Helper function
getContent :: Document -> Content
getContent (Document _ _ e) = (CElem e)

main = do
       dbfp <- readFile "db.xml"
       patfp <- readFile "pattern.xml"
       putStrLn (show (matches
                       (getContent(xmlParse dbfp))
                       (getContent(xmlParse patfp))
                      )
                )

--
```