UNIVERSITY OF LIVERPOOL

Hyperset Approach to Semi-structured Databases and the Experimental Implementation of the Query Language Delta

Thesis submitted in accordance with the
requirements of the University of Liverpool
for the degree of Doctor in Philosophy by

Richard Molyneux

Thesis Supervisors: Dr. Vladimir Sazonov
Dr. Alexei Lisitsa

External examiner: Dr. Ulrich Berger
Internal examiner: Dr. Grant Malcolm

Department of Computer Science
The University of Liverpool
January, 2009

Abstract

This thesis presents practical suggestions towards the implementation of the hyperset approach to semi-structured databases and the associated query language Δ (Delta). This work can be characterised as part of a top-down approach to semi-structured databases, from theory to practice.

Over the last decade the rise of the World-Wide Web has lead to the suggestion for a shift from structured relational databases to semi-structured databases, which can query distributed and heterogeneous data having unfixed/non-rigid structure in contrast to ordinary relational databases. In principle, the World-Wide Web can be considered as a large distributed semi-structured database where arbitrary hyperlinking between Web pages can be interpreted as graph edges (inspiring the synonym 'Web-like' for 'semi-structured' databases also called here WDB). In fact, most approaches to semi-structured databases are based on graphs, whereas the hyperset approach presented here represents such graphs as systems of set equations. This is more than just a style of notation, but rather a style of thought and the corresponding mathematical background leads to considerable differences with other approaches to semi-structured databases. The hyperset approach to such databases and to querying them has clear semantics based on the well established tradition of set theory and logic, and, in particular, on non-well-founded set theory because semi-structured data allow arbitrary graphs and hence cycles.

The main original part of this work consisted in implementation of the hyperset Δ-query language to semi-structured databases, including worked example queries. In fact, the goal was to demonstrate the practical details of this approach and language. The required development of an extended, practical version of the language based on the existing theoretical version, and the corresponding operational semantics. Here we present detailed description of the most essential steps of the implementation. Another crucial problem for this approach was to demonstrate how to deal in reality with the concept of the equality relation between (hyper)sets, which is computationally realised

by the bisimulation relation. In fact, this expensive procedure, especially in the case of distributed semi-structured data, required some additional theoretical considerations and practical suggestions for efficient implementation. To this end the "local/global" strategy for computing the bisimulation relation over distributed semi-structured data was developed and its efficiency was experimentally confirmed.

Finally, the XML-WDB format for representing any distributed WDB as system of set equations was developed so that arbitrary XML elements can participate and, hence, queried by the Δ-language.

The query system with the syntax of the language and several example queries from this thesis is available online at

`http://www.csc.liv.ac.uk/~molyneux/t/`

Keywords: Semi-structured, Web-like, distributed databases, hypersets, bisimulation, query language Δ (Delta)

Dedication

This thesis is dedicated to my loving grandparents.

Acknowledgement

The research presented in this thesis was undertaken at the Department of Computer Science under the supervision of Dr. Vladimir Sazonov and Dr. Alexei Lisitsa.

This work was inspired by the research of my primary supervisor Dr. Vladimir Sazonov, his encouragement and dedication was invaluable in developing those ideas presented here. Additionally, I am grateful to the help and support given by Dr. Alexei Lisitsa and Prof. Michael Fisher. This work was made possible by the scholarship awarded to me by the Department of Computer Science.

I wish to thank my parents whose love and support has been the foundation of all my achievements. Also, to my brothers and sister for their encouragement and support.

Contents

Chapter 1

Introduction

Before the emergence of the database culture in the late 1960's data processing involved the ad hoc manipulation of data on tape or disk. The complexity of developing and managing such systems inspired new research into the principles of data organisation. Three models were suggested during the late 1960's and early 1970's: i) the hierarchical model [72], ii) the network model [70] proposed by the Data Base Task Group, and iii) Codd's relational model [16].

The hierarchical and network models are closely related to the notion of *object-orientation* as is argued in [73] and are, in fact, based on the idea of object identity, i.e. an object whose meaning is determined not only by records of values of its fields (or attributes) but also by a pointer or address of this object within files or memory. Note that, two objects are identical if they have the same address or pointer, whereas two objects are equivalent if they share the same fields. Links $T_1 \rightarrow T_2$ denoting many-to-one relationships between record types constitute a graph in the case of the network model, and a forest (consisting of trees) in the case of the hierarchical model. Physically, each such graph or tree edge is represented by real relationships between OIDs of records of types T_1 and T_2.

On the other hand, the great success of Codd's relational model, which can be considered as a value-oriented approach, was based on taking the most fundamental concepts of logic and set theory as its foundation. Thus, any relation is a set of tuples, with each tuple also being represented[1] by a set of a special kind (a set of attribute labelled values). In fact, this approach assumes an abstract view on data values where the concept of object identity is not needed. (Note that the concept of object identity may play a role in implementation but not in the abstract model itself.) The relational model was further extended by object-orientation during the early 1990's [32], thus again absorbing the idea of object identity and additionally allowing complex data values with possibly nested structure and the idea of abstract data type with encapsulated methods.

[1] under our interpretation

However, object-relational databases are still restricted by an imposed relational schema, that is they have a rigid structure. Note that complex, nested structures considered in this approach are somewhat related with the idea of semi-structured databases discussed in this thesis, but the latter approach does not assume in general a rigid structure. Moreover, the hyperset approach to semi-structured databases presented in this thesis is crucially based on the value-oriented rather than the object-oriented view

From relations to semi-structured or Web-like data

From the second half of the 1990s a new idea of semi-structured databases emerged (see [1] as a general reference). In the age of the Internet and the World-Wide Web (WWW), allowing accessibility of remote and heterogeneous databases, the relational paradigm has become too narrow and restrictive. Indeed, the structure of the data over the WWW is typically non-fixed or non-uniform. The idea of graph representation of data was introduced with the interpretation of graph edges like hyperlinks on the Web. Due to this analogy such graph-like semi-structured databases can also be reasonably called Web-like databases (WDB) [41].

An important example of the graph approach (in its pure form) is the system Lore [46] and the corresponding query language Lorel [2], which considers graph vertices as object identities (OIDs) with equality between vertices understood as essentially literal coincidence of OIDs irrespectively of their information content (presented by outgoing edges according to our hyperset approach). In fact, this is typical for most semi-structured database approaches [2, 8, 13, 14, 15, 18, 19, 22, 26, 27, 31, 33, 46, 51], except in the case of the query language UnQL [11] (as discussed briefly below).

On the other hand, because of this idea of browsing by "picturing" the informational content (data value) of a graph vertex, considering such graphs merely as a binary (or ternary, if taking labels on edges into account) relation is not fully adequate in this context. Thus, we view the notion of semi-structured data as more than just a relation, that is more than just a graph where vertices are (uniquely presented by) object identities. In our hyperset theoretic approach, which is value-oriented, it becomes more appropriate to consider those target vertices of outgoing edges from any given vertex v as children or even as *elements* of v with v understood as a *set* of its elements. It is the latter view on graph vertices which makes it value oriented. In fact, similar terminology is used in Extensible Markup Language (XML), which is a widely adopted approach to semi-structured data. However, this is only a superficial similarity with the set theoretic approach. XML only allows to syntactically represent semi-structured data whereas treating such data as sets requires an additional level of abstraction (supported by an appropriate technique such as some set theoretic query language) which is more than just using the rudiments of set theoretic terminology.

XML documents, in fact, represent ordered tree structured data rather than arbitrary graph structured data, however, using the attributes `id` and `ref` allows one to imitate in XML arbitrary graphs as well. Considering the ordering of data in XML documents as an essential feature is related mainly with numerous software implementations which are deliberately sensitive to the order of such data. But, XML documents can also be treated as unordered, as we do in this thesis. Note that XML plays only an auxiliary role in our approach as a particular way of representing semi-structured data (XML-WDB format). Our main terminology and abstract data model is based on (hyper)set theory.

The graph model and set theoretic model

The interpretation of graph vertices as sets of their "children" leads us again to a set theoretic idea of representation of data, semi-structured data, a far going generalisation of the relational (value-oriented) approach. It is also worth noting that in the foundations of mathematics the previous century was marked by the triumph of the set theoretic approach for representing mathematical data as well as the style of mathematical language and reasoning. Mathematical logicians also developed generalised computability theory over abstract sets (of sets of sets, etc.) in the form of admissible set theory [6]. In computer science, the set theoretic programming language SETL [62, 63] was created, quite naturally, for the case of finite sets only. Also some theoretical considerations on computability and query languages over hereditarily finite sets were done in [20, 21, 43, 56, 57, 59, 61] with the perspective of a generalised set-theoretically presented databases – in fact semi-structured – even before the term "semi-structured databases" had arisen. Moreover, the set theoretic approach is closely related with a special version of the graph approach when graphs are considered up to bisimulation (see below).

The first mathematical result relating both the set and graph approaches was Mostowski's Collapsing Lemma, allowing the interpretation of graph vertices as sets of sets corresponding to children of these vertices. This, however, worked properly only for well-founded graphs and sets (which in the finite case, especially interesting for database applications, means the absence of cycles). But arbitrary graphs with cycles can also be "collapsed" into sets (interrelated by the membership relation) in the more general non-well-founded set theory also called hyperset theory [3, 5]. Here, for example the set $\Omega = \{\Omega\}$ consisting of itself is quite natural and meaningful, and corresponds to the simplest graph cycle $\circlearrowleft$.

These two trends, from abstract set theory to more concrete graph model of semi-structured data (which is closer to implementation), and vice versa were called in [61] top-down and bottom-up approaches. They meet most closely in the work on UnQL query language [11] which is devoted to a specific graph model approach to semi-structured data considered up to

bisimulation. The latter concept is also the key one in the works [41, 43, 56, 57, 61] (serving as the theoretical background for this thesis) for interpreting graph vertices as a system of (hyper)sets belonging one to another according to the graph edges. Nevertheless, [11] is still rather a graph approach than hyperset one according to the special, however related to, but not a genuine set theoretical way in which [11] treats graphs (see Section 11.3 and [61]). The main difference is that graphs considered in [11] have multiple "input" and "output" vertices, whereas graphs as considered in our hyperset approach have only one "input" corresponding to the set itself (and possibly one "output" corresponding to the empty set if it is contained in the transitive closure of this set). In fact, working with these "inputs" and "outputs" (used for appending one graph to another, etc.) is conceptually rather graph-theoretical than set-theoretical.

Hyperset approach to semi-structured or Web-like databases

As discussed above, the hyperset approach to semi-structured databases interprets graph structured data as abstract hypersets. Moreover, for the purposes of implementation, such graphs are represented as systems of set equations e.g. $\Omega = \{\Omega\}$ for the graph  . In fact, arbitrary finite graphs can be rewritten into systems of set equations and vice versa, where graph vertices (or object identities) represent set names. Moreover, elements of sets in these set equations should be labelled according to labelling of graph edges, and, in fact, these labels are the carriers of atomic information in the hyperset approach to semi-structured databases. Furthermore, graph structure or, respectively, set-element nesting organises such atomic data, just like relational tables in the relational or nested relational approaches. The notion of equality between sets can be represented in graph terms by the bisimulation relation on vertices or set names whose idea consists, roughly speaking, in (recursively) ignoring the order and repetition. Thus, any two graph vertices or set names denote the same set if they are bisimilar, that is contain the same (recursively, up to bisimulation) elements. In fact, the bisimulation relation is very important in our approach being a fundamental concept underlying hyperset theory.

Hyperset query language Δ

The associated Δ-query language is based on set theory and predicate logic, being an extension of the basic or rudimentary operations [30, 39] – the *core* fragment of Δ. The set theoretic operators of the Δ-language, like in the relational calculus, have clear and well-understood semantics. In fact, the expressive power of Δ (the core fragment plus transitive closure, decoration and recursion set theoretic operations) was shown in [57] and [43, 58] to capture all polynomial time computable operations over hereditarily finite sets and, respectively, hypersets. Also, another version of the language was shown in [40, 42] to capture exactly

all LogSpace computable operations over hereditarily finite sets (without cycles). Therefore, in principle, the Δ-query language can be reasonably considered as computationally viable and worthy of implementation.

Some earlier preliminary work on the implementation of the Δ-query language to WDB was done earlier by Yuri Serdyuk in [66], as well as in some practical attempt towards a new implementation based on multiple distributed agents working cooperately over the Internet [35] (taking into account the earlier theoretical work [60]). More recently the implementation work leading to this thesis was done in [49]. However, the latter implementation was insufficiently perfect. This antecedent work subsequently inspired the proposal for further research and the development of a sufficiently detailed implementation, that is, the point of the work done here. Note that some details of the implementation described here were published in [50].

Implementation of the hyperset approach

The goal of this work was to demonstrate how the hyperset approach to semi-structured or Web-like databases could be implemented, with the aim of presenting this approach in a practical rather than theoretical context and making it accessible to a more practically oriented audience. In particular, the practical characteristic of this work assumes representation of hyperset data as files distributed over the World-Wide Web and the implementation of the hyperset query language Δ allowing queries over such distributed data. Importantly, the implemented language should preserve the original high level, declarative character[2] and retain its set theoretic style. Further, this approach should demonstrate the power of the set theoretic style of thought towards semi-structured databases. Note that the query system (which is implemented in Java) and the example queries described in this thesis can be found at

`http://www.csc.liv.ac.uk/~molyneux/t/`

Efficiency issues

Another goal consisted in the subsequent investigation of theoretical considerations arising from this experimental implementation, specifically the problem of efficient implementation of the equality or the bisimulation relation – which crucially underlies this hyperset theoretic approach. Moreover, our proposed solution was restricted to making the bisimulation relation efficient only in context of distributed WDB which may require numerous and particularly expensive downloads of files from the World-Wide Web. However, this work does not consider the problem of efficiency in the non-distributed case, especially taking into account the previous

[2] Recall that, for example, Prolog initially intended to be a logical, declarative programming language, eventually has both declarative and imperative features. This mixture of ideologies was the result of making this language more efficient.

works on efficient bisimulation algorithms that, on the other hand, do not consider distribution [24, 25]. Note that, many other aspects of efficiency of the implementation (such as indexing, hashing and other physical data organisation techniques [73]) as well as various other questions which should be resolved for creating a sufficiently realistic database management system were inevitably postponed here. In fact, the primary aim of this work was the correct and meaningful implementation of a non-trivial and user friendly version of the Δ-language.

Organisation of the thesis

Details of the implementation are rather technical, thus it makes sense to firstly explain the intuitive (or high level) meaning of the hyperset approach and demonstrate example queries of the implemented Δ-query language. Secondly, technical details of the implementation appear towards the end of the thesis detailing the lower level aspects of our approach. Note that, the material presented in this thesis follows an intuitive perception of this approach towards semi-structured databases rather than a strict logical dependency.

The thesis is organised into four parts:

Part I, "Hyperset approach to querying Web-like databases", gives an overview of the implemented hyperset approach to semi-structured or Web-like databases and the associated query language Δ, including worked example queries. The point of this part is to introduce this approach on an intuitive level before discussing the technical details of implementation.

Part II, "Local/global approach to optimise bisimulation and querying", is concerned with the problem of efficient implementation of the equality or bisimulation relation. Here two joint strategies were suggested for resolving this problem: i) implementation of an Internet service for resolving bisimulation questions, and ii) the computation of bisimulation approximations on fragments of distributed Web-like databases to aid the computation of global bisimulation. The viability of these suggestions as solutions is supported by empirical testing.

Part III, "Implementation issues", presents the technical details of the implementation of the hyperset approach towards semi-structured or Web-like databases. We start by detailing query execution (which we feel is potentially more important for readers) followed with query parsing and contextual analysis, although query execution is, in fact, formally dependent on the latter syntactical considerations. Finally, XML representation of WDB systems of set equation has a quite isolated role in our approach and is presented at the end of this technical material, but this discussion is actually quite self-contained and can be read independently of the rest of this thesis.

Part IV, “Evaluation”, concludes with comparative analysis with other known approaches towards semi-structured databases, and finishes with some future prospects and closing remarks.

Part I

Hyperset approach to querying Web-like databases

Chapter 2

Semi-structured or Web-like databases

The term *semi-structured data* denotes data which has a characteristically unfixed or non-rigid structure, thus semi-structured data is considered as "schemaless" or "self-describing"[1] having no complete structural description or schema [1]. However, typically semi-structured data is similar to structured data e.g. relational data (as described below) but without strictly imposed structure. More specifically our approach to semi-structured databases is based on (hyper)set theory [3, 5].

2.1 Set theoretic view of structured and semi-structured data

2.1.1 Structured relational data

Structured data has a fixed and rigid structure such as relational data [17] described by relational schema $R(A_1, A_2, ..., A_n)$, where R is *relation* name and A_i are *attributes* (constrained by the domain D_i). In the relational model, relations are naturally represented as tables with attributes as named columns of a table. For example, the `Stud` relation shown in Figure 2.1 has the attributes `forename`, `surname`, `DOB` (date of birth) and `department`.

Stud:

forename	surname	DOB	department
Jack	Jones	30/6/1986	DeptChemistry
Sarah	Smith	27/11/1988	DeptBiology

Figure 2.1: Relational table of students.

[1] The consideration of semi-structured data as "self-describing" is somewhat misleading as it might be wrongly thought to suggest clear semantic description of such data. In particular, when considering the graph representation of semi-structured data, labels have only an informal meaning dependant on subjective interpretation of language, e.g. the imprecise term "location" could have many interpretations – address, map coordinates, URI, anatomical, etc.

The relational approach is essentially based on set theory, as well as on logic. For example, the `Stud` relation (above) can be represented as set of student *tuples* (rows or records),

```
Stud  = { st1, st2, ... }
```

or, better, as

```
Stud  = { student:st1, student:st2, ... }
```

where each student tuple is represented as a set of labelled atomic values, with labels being *attribute names*, and *attribute values* as atomic values (strings of symbols between quotation marks to distinguish them from set names and attribute names),

```
st1  = { forename:"Jack", surname:"Jones",
         DOB:"30/6/1986", department:"DeptChemistry" }

st2  = { forename:"Sarah", surname:"Smith",
         DOB:"27/11/1988", department:"DeptBiology" }.
```

Let us consider the relational database `Univ` as the following set of (labelled) relations,

```
Univ = { departments:Dept, students:Stud, lecturers:Lect,
         modules:Mod, courses:Course, ... }.
```

The relations `Dept`, `Lect`, `Mod` and `Course` will not be further described, they are plausible example relations, like `Stud`, that could belong to a University database. Here the labels (or attributes) `departments`, `students`, `lecturers`, etc., give an informal description of what the sets `Dept`, `Stud`, `Lect`, etc., are about. These sets could be denoted differently, say as `D`, `S`, `L`, etc. Thus, strictly speaking the denotation of sets does not necessarily carry informational content. Hence the important role of labels (attributes e.g. `forename`) and atomic values (e.g. `"Jack"`), which are the proper carriers of basic information.

2.1.2 Relaxation of structural restrictions on relational data

Relational data with the given schema $R(A_1, A_2, ..., A_n)$ has a rigid structure with mandatory attributes A_i for associated tuple components. It is also known of the more general approaches to *nested* relational databases [52, 54, 71] where attribute values could be relations. Say, in the above example we could reconsider `DeptChemistry` as a set (instead of an atomic value) by omitting the quotation marks around `DeptChemistry` and adding the corresponding set equation further detailing the chemistry department:

```
DeptChemistry  = { name:"Department_of_Chemistry",
                   lecturers:ChemLect,
                   modules:ChemMod,
                   ... }.
```

Moreover, we could relax the requirement on students tuples to have a value for each attribute `forename`, `surname`, `age` and `department`. For example, the DOB of a student could be absent by some reason, but some other information could be present, such as

```
email:"jones@liv.ac.uk"
```

or,

```
sex:"male".
```

Thus, relaxation of traditional structural restrictions on relational databases leads naturally to semi-structured databases, in fact, to the set theoretic approach where such data are considered as *arbitrary* set of (labelled) sets of sets, etc., to any depth, represented by set equations like above.

2.1.3 Semi-structured data

For simplicity, we consider semi-structured data as systems of *flat* set equations where a set equation consists of set name s_i equated to a bracket expression $B_i(\bar{s})$ like those considered in the above example. In vector form this can be summarised as

$$\bar{s} = \bar{B}(\bar{s}).$$

Flat bracket expression $\{l_1 : s_{i_1}, \ldots, l_n : s_{i_n}\}$ is thought of as a set of labelled elements. In the flat (non-nested form) only set names s_i from the list of all set names $\bar{s} = s_1, s_2, ..., s_n$, may participate as elements. Labels l_j can be considered as analogous to attributes in the relational approach, however, element labelling is optional with the default label being the empty label $\square$ (or `null`) which can be considered as invisible, such as the absence of labelling in the `Stud` set above. Formally our general approach does not consider atomic values such as `"Jack"`, `"Jones"`, etc., from the example above. However, any atomic value can be simulated as a set consisting of one labelled empty set [41, 57, 61], such as

```
"Jack" = {'Jack':{}}.
```

Strictly speaking, we should use single quotation marks for labels (often omitted for simplicity) and double quotation marks for atomic values. Of course, we can still use the denotation for atomic data like `"Jack"`, but it should be understood as above.

2.1.4 Syntactical and conceptual set nesting

In the case where nesting is allowed (like the participation of `{}` in the above definition of atomic values, and also in more complicated cases) any set name s_i can be substituted with the corresponding nested bracket expression B_i, and vice versa. For example, the `Stud` set

equation could be rewritten with the nested right-hand side (and adding the `student` attribute) as follows,

```
Stud = {
        student:{ forename:"Jack", surname:"Jones",
                  DOB:"30/6/1986", department:"DeptChemistry" },

        student:{ forename:"Sarah", surname:"Smith",
                  DOB:"27/11/1988", department:"DeptBiology" }
       }.
```

Here the nesting of data inside the `Stud` set equation proves useful in avoiding the introduction of new set names, and thus eliminating `st1` and `st2`. Moveover, this demonstrates that set names in set equations play an auxiliary role, and can even be readily renamed in an analogous way to renaming variables in any ordinary algebraic equations. Thus the real information of such semi-structured data is carried by labels and set/element nesting. More generally, we could allow (and, in fact, will consider later) arbitrary nesting in the right-hand sides of set equations $\bar{s} = \bar{B}(\bar{s})$. This can be evidently "unnested" or "flattened" by introducing new (fresh) set names and appropriate set equations. So, our restriction for non-nested systems of set equations (i.e. with non-nested right-hand sides) is not essential, but can simplify some considerations.

In fact, the notion of non-nested or flat system of set equations is only syntactical and, conceptually, flat systems of set equations allow arbitrary nesting with the participation of set names (corresponding to set equation) as elements

2.2 Hyperset theoretic view of semi-structured data

In the above approach to semi-structured data via systems of set equations $\bar{s} = \bar{B}(\bar{s})$ there was, in fact, no restriction on the form of these equations. Thus allowing not only arbitrarily nested, but also cycling data like in the simplest example of a set consisting of itself

$$\Omega = \{\Omega\}.$$

Mathematically, such kind of sets are considered as non-traditional, although they have already been deeply investigated in *hyperset theory*, as represented in the books [3, 5]. From the point of view of semi-structured data there is nothing strange in such sets. Imagine that we have a relational table where some cells can represent other relational tables, etc. Such nesting can be implemented so that "clicking" on such a cell leads to the corresponding nested relational table shown instead of the original table. There is no technical or conceptual problem to have such

a situation that after several such "clicks" we will arrive back to the original table we started "clicking" with – like in the World-Wide Web by successive "clicking" we can possibly return to the Web page we started with. Moreover, from the informational or database point of view this can be quite meaningful.

For example, let us consider the University database where formally the student set `st1` has the chemistry department set `DeptChemistry` as the member, and (possibly many) students are members of the `ChemStud` set of enrolled chemistry students, as described by mutually recursive set definitions,

```
st1  = { forename:"Jack", surname:"Jones",
         DOB:"30/6/1986", department:DeptChemistry }

DeptChemistry = { ..., enrolled:ChemStud, ... }

ChemStud = { student:st1, ... }
```

with `ChemStud` a subset of the set `Stud` of all university students. Any set (name) s_i can be defined by referring to other set (names) as elements, etc., so that eventually we could possibly come to the original set s_i – thus, arbitrary cycling is allowed.

There is more to say about the hyperset approach to semi-structured data on the conceptual level, in particular, on the concept of equality between sets (possibly denoted by different set names) but we will postpone this discussion to Section 2.4.1. On the current very preliminary level of consideration sets are thought simply as syntactical bracket expressions, or as represented by formal systems of set equations. In fact, we need an abstract concept of hypersets amongst which we could find a (unique) solution to any given system of set equations.

2.3 Graph or Web-like view

2.3.1 Graph representation of systems of set equations

Representation of semi-structured databases by systems of set equations presents a clear and mathematically well-understood[2] conceptual view of semi-structured data as (hyper)sets. But it also makes sense to consider visualisation of systems of set equations by the equivalent representation as (finite) labelled directed graphs. In fact, it is important for all considerations of this work that any given system of set equations can be considered as a labelled directed graph.

[2] taking into account Section 2.4

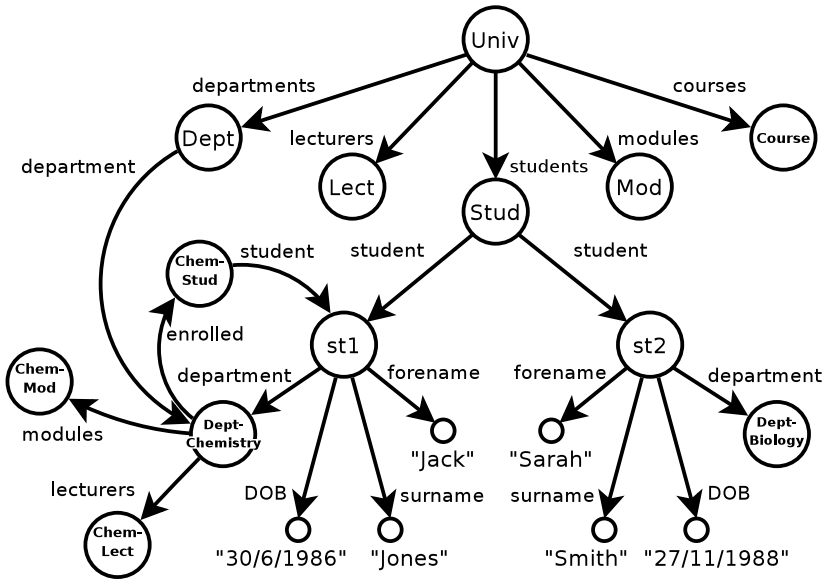


Figure 2.2: Semi-structured database `Univ` represented as directed graph.

In fact, most approaches to semi-structured databases typically consider them as labelled directed graphs, that is, semi-structured data is modelled as (finite) directed graph $G = \langle N, E \rangle$ with L-labelled edges, where L is an infinite set of possible labels ($l_1, l_2, \ldots$, etc., and the empty label $\Box$), N is a finite set of nodes ($s_1, s_2, \ldots$, etc.), and E is a finite set of edges with each edge $s_i \xrightarrow{l_k} s_j$ being formally an ordered triple of the form $\langle s_i, s_j, l_k \rangle$. For example, the University database considered in Section 2.1 has the corresponding representation by directed graph shown in Figure 2.2.

The membership of labelled element $label : s_2$ to the set s_1 ($label : s_2 \in s_1$) corresponds to the labelled edge $s_1 \xrightarrow{label} s_2$ (and vice versa), where set names s_i serve as (the unique names of) graph nodes. In general, each set equation $s_i = \{l_1 : s_{i_1}, \ldots, l_n : s_{i_n}\}$ from the system generates a fork of labelled edges $s_i \xrightarrow{l_1} s_{i_1}, \ldots, s_i \xrightarrow{l_n} s_{i_n}$ outgoing from s_i, as depicted in Figure 2.3. All those forks generated from every set equation give the corresponding representation as graph. Vice versa, any graph with labelled edges is evidently visualising a system of set equations, with one equation for each node so that each node is thought as a (hyper)set. Thus, graphs and (formal) systems of set equations are essentially equivalent concepts.

2.3.2 Graphs or systems of set equations as Web-like databases

The World-Wide Web (WWW) can, in principle, be considered as a large semi-structured database, consisting of an arbitrarily organised collection of hyperlinked HTML documents. Each HTML document has a corresponding URL (WWW address), and contains textual data

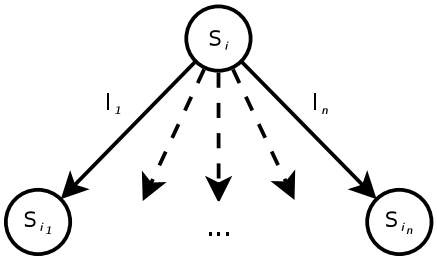


Figure 2.3: Forking of labelled edges generated by the set equation $s_i = \{l_1 : s_{i_1}, \ldots, l_n : s_{i_n}\}$.

with markup tags denoting visualisation and hyperlink information. The following fragment of HTML code is an example of a hyperlink,

```
<a href="http://www.liv.ac.uk/">University of Liverpool</a>
```

what in our symbolism of labelled elements can be represented as

`University of Liverpool` : `http://www.liv.ac.uk/`

and visually (in Web browser) this hyperlink would appear as "clickable" fragment of text

University of Liverpool

with the URL hidden. Hiding of URLs corresponds to the idea mentioned above that set names (names of graph nodes) actually do not matter from the point of view of the proper information. Only labels on edges or the "clickable" links (and other text and visual content) on Web pages carry information, plus, of course, the graphical structure. That is, URLs play a different role than proper information in the WWW. In Figure 2.4 we consider browsing between hyperlinked HTML documents by "clicking" on such links. It is evident from this example that hyperlinked HTML documents can express arbitrary relationships, for example the cycle when browsing by "clicking" on the links, `Departments`, `Medicine`, `University of Liverpool`, and so on.

Thus, any hyperlink can be denoted by the labelled edge $url_i \xrightarrow{label} url_j$, suggesting the intuitive understanding of hyperlinking as arbitrary labelled directed graph. Therefore, systems of set equations or equivalently labelled direct graphs, can be more generally named by the analogy *Web-like Databases* (WDB) [19, 41, 60, 61]. Furthermore, our approach also considers WDB as Web-like with distribution over the Internet (in a similar manner to hyperlinks), however, it is intended to be smaller, simpler and better organised than the WWW. Such WDB graphs can, in principle, be quite arbitrary but in real applications it is assumed to be

governed by some organisation or company, and possibly not allowed to be arbitrarily extended by anybody in the world (like typical databases). Additionally, WDB (or semi-structured data) can also have a schema restricting the shape of the WDB, but not necessarily so rigid like in the case of relational databases, see for example [9, 41, 57]. However, we will not go further into these details.

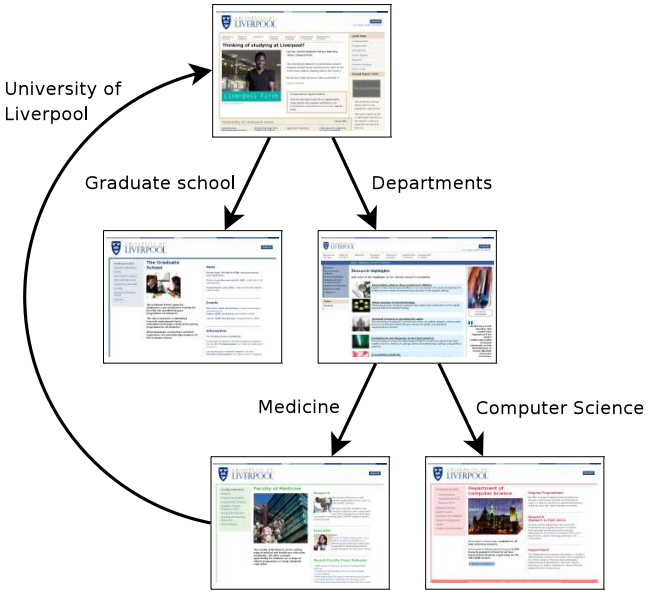


Figure 2.4: Browsing of hyperlinked HTML documents on the University of Liverpool website.

2.3.3 Distributed WDB

Any WDB represented as a system of set equations $\bar{s} = \bar{B}(\bar{s})$ can be quite big, and naturally divided into subsystems of set equations. Each subsystem corresponds to a XML-WDB file (see Chapter 10 for details of the XML-WDB representation) containing only some of the equations (desirably closely interrelated by a subject matter). Moreover, these files could be distributed between various servers over the world, like HTML files on the World-Wide Web. It may happen that set equations defined in some WDB file may involve set names defined by equations in other (non-local) WDB files.

Furthermore, when considering the real application of WDB distribution proves useful in the creation and management of (potentally large) databases, such as the plausible distribution of the University WDB. Let us consider that in the case of the University WDB, set equations might be distributed between many WDB files, let us say by department. Therefore, the WDB file `http://www.liv.ac.uk/ChemistryDepartment.xml` could contain the following subsystem of set equations[3]:

[3]This is still not very realistic situation to assume that the file `ChemistryDepartment.xml` contains all set

```
DeptChemistry = { ..., enrolled:ChemStud, ... }
ChemStud = { student:st1, ... }
```

Likewise, the WDB file `http://www.liv.ac.uk/BiologyDepartment.xml` could contain the subsystem of set equations:

```
DeptBiology = { ..., enrolled:BiolStud, ... }
BiolStud = { student:st2, ... }
```

Moreover, there could also be the WDB file `Students.xml` containing the set equations `st1 = {...}` and `st2 = {...}`. Thus, the set names `st1`, `st2`, etc. participating, respectively, in `ChemistryDepartment.xml` and `BiologyDepartment.xml` would now be described as sets in another file. In this case, we should consider the full versions of the simple set names, `st1`, `st2`, etc., described in `http://www.liv.ac.uk/Students.xml`, as discussed below.

2.3.3.1 Full versus simple set names

Taking into account the above example, any given set name should be considered as a *full set name*, consisting of WDB file URL and *simple set name* (with the simple set name described within the WDB file). For example, in the distributed University WDB considered above, the full set name of the biology student `st2` would be

```
http://www.liv.ac.uk/Students.xml#st2
```

with the WDB file URL and simple set name delimited by # symbol. However, in practice it suffices to use simple set names in the left-hand side of set equations, and also for those occurrences of set names appearing in the right-hand side of set equation definitions if they are defined in the same WDB file. In particular, the author of a WDB file can freely use any simple set name (as such or as part of full set names) without the danger of clashes with simple names participating in the other WDB files.

However, there is one subtle point: if a simple set name `set_name` occurs twice in some WDB file, once as a simple set name and again as part of a full set name `url#set_name` (with `url` referring to some different WDB file). Then in the latter case it refers to another file where the corresponding equation is defined, even if the current file already contains the equation `set_name = {...}`. Thus, these two occurrences are actually different set names because their corresponding full set names are indeed different. Of course, each set name must be defined either in the same or some other WDB file. Otherwise it is considered as syntactical error. Thus, it is necessary to download some WDB files whose URLs appear in full set names of the given file to confirm the existence of defining equations of the referenced set names.

equations related with this department (on students, lecturers, etc.). These set equations should be further divided into natural fragments (WDB files).

2.4 Hyperset data considered abstractly

The notion of WDB as a system of set equations presents a low level, syntactical understanding of semi-structured data. However, conceptually (and semantically) WDB is understood as consisting of *abstract* hypersets (like relational database consists of abstract relations). The hyperset approach considers WDB as an arbitrary finite system of set equations, each set equation consisting of set name equated to corresponding bracket expression. But the intended meaning of such a syntactical expression is a set of labelled elements, *not* an ordered sequence. Therefore according to this (hyper)set theoretic approach ordering and repetition of elements in a bracket expression should be completely ignored. That is, ignoring ordering and repetitions has some both *operational* and *conceptual* consequences.

This can possibly lead to equality between different set names s_i and s_j denoted as $s_i = s_j$ and meaning that s_i and s_j denote the same abstract hyperset, or strictly denoted as $s_i \approx s_j$ (to avoid possible misunderstanding of $s_i = s_j$ as the assertion that these set names are identical, and to stress on the particularly important role of this concept of equality). In fact, $\approx$ is the well known concept in the context of graphs called *bisimulation relation* between graph nodes or, in our case, between set names [3, 5, 61]. As the role of this relation is crucial for the hyperset approach to semi-structured databases, this approach is therefore more than pure graph theoretic, as considered in the approaches to semi-structured databases as graphs e.g. in [1, 2, 11, 18, 19, 36, 46] or as XML tree-like data e.g. in [23, 33]. Note that, however, [11] is also heavily based on the bisimulation relation, it is rather a graph than a hyperset approach as was argued in [61].

2.4.1 Bisimulation – preliminary considerations

In general, the bisimulation relation between set names (graph nodes) of a WDB, i.e. a system of set equations, and the corresponding recursive algorithm is based on the idea that any two sets are equal if for each (labelled) element of the first set there exists an equal (bisimilar) element in the second set (and vice versa). Bisimilar set names are said to denote the same abstract (hyper)set. The bisimulation relation will be further described in Chapter 4, with formal theoretical definition, and practical considerations for its implementation. We consider that this hyperset approach to WDB is worth implementing as it suggests a clear and mathematically well-understood view on querying such semi-structured data.

A WDB is called *strongly extensional* [3] or non-redundant, if different set names (nodes) are non-bisimilar i.e. denote different hypersets. In the case of strongly extensional WDB, equality between set names (nodes) trivially becomes the syntactical identity relationship. Otherwise, even the simplest queries like $x = y$ or $x \in y$ can be quite expensive to evaluate,

especially in the case of distributed WDB. Therefore, we devote Part II to some approach of dealing with this problem practically.

2.4.1.1 Example

Consider the set equations below, where trivially $x \approx x'$ holds because our (hyper)set approach ignores the ordering and repetition of elements:

$$x = \{y, z\}$$
$$x' = \{z, y, z\}.$$

However, set names (or graph nodes) may be equal (bisimilar) for some "deeper" reason than for x and x' above. Let us consider the above example extended with the (recursive) definitions of the sets z, y and y':

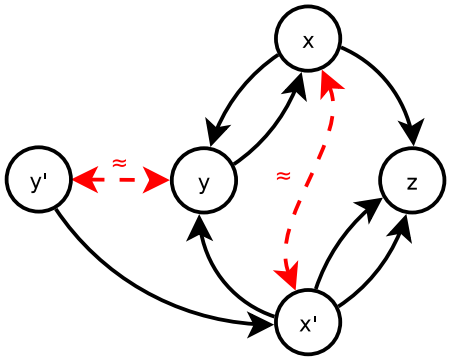
$$z = \{\}$$
$$y = \{x\}$$
$$y' = \{x'\}.$$

The sets y and y' both contain one element of syntactically differing set names (x and x' respectively), thus suggesting that y and y' might not be equal. However, the bisimulation relation defines two sets as equal if for each element of the first set there exists an equal (or bisimilar) element in the second set, and vice versa. In the case above we already know that $x \approx x'$ holds, and according to this informal definition of bisimulation all of the elements of y are bisimilar to the elements of y', and vice versa. Therefore we can deduce that, in fact, $y \approx y'$ holds.

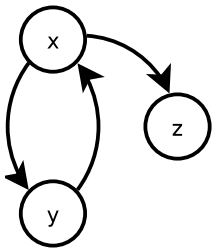
Let us now consider the strongly extensional version of this system of set equations obtained by eliminating the redundant set names x' and y', and omitting repetitions. Thus, after "collapsing" the bisimilar nodes x' to x and y' to y, and omitting element repetitions, the resulting system of set equations is

$$x = \{y, z\}$$
$$y = \{x\}$$
$$z = \{\}.$$

Thus, the elimination of redundancies (in the above system of set equations) is visualised by Figure 2.5.



(a) Redundant version, with red dashed edges relating bisimilar nodes (or sets)



(b) Non-redundant (strongly extensional) version

Figure 2.5: Graphical representation of a trivial WDB (cf. corresponding set equations above).

2.4.2 Redundancies in WDB

The above example, although artificial, demonstrates that bisimilarity between set names introduces redundancies into WDB. However, the crucial question in implementing the hyperset approach to WDB is whether the bisimulation relation ($\approx$) can be computed in any reasonable and practical way. Some possible approaches and views are outlined below.

In principle, the occurrence of bisimilar nodes in a realistic WDB (i.e. redundancies) should be infrequent. Therefore, such rare redundancies can be eliminated by supporting WDB in a *strongly extensional* state, with redundancies detected or even eliminated instantly as soon as they might potentially appear. Trivially, after eliminating redundancies equality between sets (i.e. bisimulation relation between set names or graph nodes) becomes the identity relation. However, eliminating redundancies is more expensive than only detecting them i.e. just computing bisimulation relation on the WDB. Thus, supporting WDB in strongly extensional form may be reasonable option when WDB is not large.

WDB should not be assumed to be just another version of WWW, freely extensible by anybody in the world. That is, an appropriate discipline of working with WDB could make the problem of bisimulation practically resolvable. Let us now consider several ways by which redundancies can appear.

2.4.2.1 Redundancies arising during query execution

Execution of queries leads to the temporary extension WDB$'$ of the original WDB (as detailed later in Section 3.3), with the addition of new set names and set equations locally. Such extensions WDB$'$ may potentially give rise to new redundancies, so that equality subqueries applied to these newly generated sets becomes non-trivial. Note that the set names in original

WDB do not refer to new ones in WDB$'$, thus WDB remains self-contained. Therefore, the new bisimulation relation ($\approx'$) on WDB$'$ restricted to those set names in WDB coincides with the identity relation on WDB. Moreover, the algorithm of query execution could be amended in such a way that as soon as new (auxiliary) set names are generated (like res in Section 3.3) any possible redundancies will be eliminated immediately. It should also be taken into account that the extensions WDB$'$ arising during query execution have several specific types, and are sufficiently simple and small, thus making the process of detecting/eliminating redundancies easier, see also [40, 42], but we will not go into the details here.

2.4.2.2 Redundancies which can appear during a local update

Local updates of WDB files are more problematic because previously non-bisimilar nodes outside this file may become bisimilar due to possible links (or paths) to the local nodes with changed/added meaning. The appropriate (more efficient than the standard) strategy of detecting/removing all such redundancies is not so straightforward and needs to be developed yet. However, taking into account the locality of changes, this task does not seem to be unrealistic.

2.4.2.3 Deliberate redundancies

Deliberate redundancies in WDB can also appear with the same aim as mirroring in WWW. But, if there is a requirement to officially registered such mirroring in the WDB, then such deliberate redundancies should most plausibly be dealt with in a quite feasible way.

2.4.2.4 Local versus global bisimulation

Unlike the other considerations above, we will consider the "local/global" approach and its implementation for supporting bisimulation relation on WDB (in background time) in more detail (see Part II). Now we present only some general introductory comments on this idea.

Assume that all WDB nodes are divided into classes L_i according to their sites (WDB servers) or even files. There is a quite natural definition of local (i.e. computed locally) lower and upper approximations ($\approx^L_-, \approx^L_+$) to the global bisimulation relation ($\approx$) on the whole WDB:

$$n_1 \approx^L_- n_2 \Rightarrow n_1 \approx n_2 \Rightarrow n_1 \approx^L_+ n_2$$

These approximations can help to compute and to permanently support global bisimulation in a distributed way in background time. Moreover, we could require *local independence* ($\approx^L_- = \approx^L_+$, and hence $= \approx\restriction L$) and additionally *local non-redundancy* ($\approx^L_- = \approx^L_+ = =^L$).

2.4.3 Bisimulation invariance

The hyperset approach assumes considering WDB (graphs or systems of set equations) up to bisimulation. Therefore, it is an important requirement for set theoretic operations and relations to be *bisimulation invariant*, that is to preserve the bisimulation relation. Although not fully proven here, it can be shown [58] that all definable queries q of the hyperset Δ-query language[4] (see Chapter 3) are bisimulation invariant:

$$\bar{x} \approx \bar{y} \Longrightarrow q(\bar{x}) \approx q(\bar{y}) \quad \text{(for set valued queries)}$$
$$\bar{x} \approx \bar{y} \Longrightarrow q(\bar{x}) \Leftrightarrow q(\bar{y}) \quad \text{(for boolean queries).}$$

For example, in the case of the set theoretic operation union we have:

$$x_1 \approx y_1 \;\&\; x_2 \approx y_2 \Rightarrow (x_1 \cup x_2) \approx (y_1 \cup y_2).$$

This actually means that we work with (abstract) hypersets rather than just with graph nodes or set names, however the operational semantics of the language Δ is based on the syntactical manipulations of set equations [61]. The point is that the semantics of the language Δ respects bisimulation and completely agrees with the hyperset theory [3, 5].

In particular, $x_1 \cup x_2$ is defined as a new set name, say u, with corresponding new set equation $u = \{\ldots,\ldots\}$, where the first "..." is the content of the right-hand side of the equation $x_1 = \{\ldots\}$ from the given WDB, and similarly for the second "..." and the equation $x_2 = \{\ldots\}$. The union $y_1 \cup y_2$ is computed in the same way from set equations for y_1 and y_2 giving rise to new set name, u', and the corresponding set equation $u' = \{\ldots,\ldots\}$. Then the conclusion of the above bisimulation invariance condition for $\cup$ actually means $u \approx u'$, and can evidentially be shown.

Note that the membership relation $x \in y$ for two sets (considering the unlabelled case for simplicity) is defined to be true if the set equation for y involves some set name x', where $y = \{\ldots, x', \ldots\}$ and, moreover, $x \approx x'$. Additionally, it can be shown that the membership relation is also bisimulation invariant:

$$x_1 \approx y_1 \;\&\; x_2 \approx y_2 \implies x_1 \in x_2 \iff y_1 \in y_2$$

For all other constructs of the Δ-language the operational semantics maybe more complicated, however, it follows that they also agree with this intuitive (abstract) set theoretical meaning. The syntax and semantics of the Δ-query language will be further detailed in Sections 3.1 and 3.2, with some further indications of the operational semantics in terms of set equations

[4] The operational meaning of Δ-queries are defined graph theoretically or in terms of set equations.

detailed in Section 3.3.

2.4.4 Anti-Foundation Axiom

Finally, we do not go into full mathematical details on hypersets, however, we could assert the following form of **Anti-Foundation Axiom** (AFA) [3, 5], which holds in the universe of abstract (in our case finite) hypersets:

> *Any system of set equations $\bar{s} = \bar{B}(\bar{s})$ has a unique abstract hyperset solution for set names $\bar{s}$ making these equations true.*

Therefore, set names of any WDB (as system of set equations) denote quite concrete, uniquely defined abstract hypersets. In this sense each set name (in a Δ-query) serves as a set constant (relative to the given WDB) denoting a unique hyperset. Note that, the Δ-language also has set variables which can be quantified unlike constants.

Strictly speaking all of this makes precise mathematical sense only in context of Chapter 4, which further details the bisimulation relation (with some additional mathematical considerations) beyond the general informal description of bisimulation relation so far.

Chapter 3

Query language Δ

3.1 The syntax

There has already been much theoretical considerations on (some versions of) the Δ (Delta) query language to hyperset/WDB databases [40, 41, 43, 57, 61]. The two main syntactical categories of Δ are:

- Δ-*terms* representing set valued operations over hypersets (*set queries*), and
- Δ-*formulas* representing truth valued operations (*boolean queries*).

Note that the denotation Δ bears partly from the well-known class Δ_0 of bounded formulas introduced by Levy, although Δ, as defined here, denotes a wider language. It is based on the *basic* or *rudimentary* set theoretic languages of Gandy [30] and Jensen [39]. Moreover, inclusion of set theoretic operators: transitive closure (TC), recursion (Rec) and, for the case of hypersets, decoration (Dec) (the latter due to Forti and Honsell [29] and Aczel [3]), allows to define in Δ exactly all polynomial time computable operations over hypersets represented as WDB, thus demonstrating and characterising theoretically its rich expressive power (assuming that a linear order on labels is given) [43, 56, 57, 58]. The operators of Δ are defined as follows:

$$\begin{aligned}
\langle \Delta\text{-term} \rangle ::= {} & \langle \text{set variable or constant} \rangle \mid \emptyset \mid \{l_1 : a_1, \ldots, l_n, a_n\} \mid \bigcup a \mid \mathsf{TC}(a) \mid \\
& \{l : t(x,l) \mid l : x \in a \;\&\; \varphi(x,l)\} \mid \mathsf{Rec}\; p.\{l : x \in a \mid \varphi(x,l,p)\} \mid \mathsf{Dec}(a,b) \\
\langle \Delta\text{-formula} \rangle ::= {} & a = b \mid l_1 = l_2 \mid l_1 < l_2 \mid l_1 \; R \; l_2 \mid l : a \in b \mid \varphi \;\&\; \psi \mid \varphi \vee \psi \mid \neg\varphi \mid \\
& \forall l : x \in a.\varphi(x,l) \mid \exists l : x \in a.\varphi(x,l)
\end{aligned}$$

The intuitive set theoretic semantics of the majority of the above constructs should be well-understood by anyone with the minimal mathematical background in set theory and logic. In the above constructs we denote: $a, b, \ldots$ as (set valued) Δ-terms; $x, y, z, \ldots$ as set variables;

l, l_i as label values or variables (depending on the context); $l : t(x, l)$ is any l-labelled Δ-term t possibly involving the label variable l and the set variable x; and φ, ψ as (boolean valued) Δ-formulas. Note that labels l_i participating in the Δ-term $\{l_1 : a_1, \ldots, l_n : a_n\}$ need not be unique, that is, multiple occurrences of labels are allowed. This means that we consider arbitrary sets of labelled elements rather than records or tuples of a relational table where l_i serve as names of fields (columns).

The binding label and set variables l, x, p of quantifiers, collect, and recursion constructs should not appear free in the bounding term a (denoting a finite set). Otherwise, these operators may become unbounded and thus, in general, non-computable. For example, let us consider the universal quantifier $\forall l : x \in \{\ldots, l : x, \ldots\}.\varphi(x, l)$ which becomes unbounded due to the quantified variables $l : x$ participating in the bounding term $\{\ldots, l : x, \ldots\}$. In fact, as $l : x \in \{\ldots, l : x, \ldots\}$ is always true the above quantified formula proves to be equivalent to unbounded one: $\forall l : x.\varphi(x, l)$.

3.2 Intuitive denotational semantics

Any Δ-query without free variables has either: i) (hyper)set value in the case of Δ-terms, or ii) boolean value in the case of Δ-formulas. Those participating set variables or set constants represent abstract hypersets (and thus correspond to set names in WDB), whereas participating label variables or label constants represent label values (corresponding to strings of symbols).

The intuitive meaning of Δ-queries is described by the *denotational semantics*, that is what any expression denotes[1]. For the purposes of implementation Δ-queries are also described by means of their *operational* or computational semantics (see Section 3.3) which must be coherent with our intuitive denotational semantics. Here we will also rely on intuition, without presenting any precise argument. In fact, the required coherence will be pretty much evident. So, we can concentrate on examples of queries and implementation aspects.

3.2.1 Boolean valued expressions — Δ-formulas

Equality $(=)$ and the *alphabetic ordering* $(<)$ between labels is understood standardly. In the theoretical Δ-language the relation R over labels is any easily computable relation over labels, however, in the implemented Δ-language described in this thesis we consider R as any of the following *substring* relations

[1] There is a deep mathematical theory of denotational semantics of programming languages based on Domain Theory [65, 68] (also see the contemporary reference [28]) to represent denotational values of a programming language expressions. The language Δ, where all computations evaluating queries are finite, does not require this theory which is based on the idea of potentially infinite computations (embodied in the so called "undefined" element $\perp$). Anyway, it makes sense to use the term denotational semantics, although we will describe this semantics on a very intuitive level by reference to the "domain" of sets and hypersets.

$$*l_1 = l_2 \mid l_1* = l_2 \mid *l_1* = l_2$$

where the wildcard $*$ represents any string of symbols. In principle we could include into the language more relations over labels, but in the implementation there are only $<$ and substring relations, and the user currently has no way to define more primitive relations over labels. It should be noted that equality between Δ-terms, $a = b$ or, for technical reasons, $a \approx b$, is understood as the equality of abstract hypersets denoted by these terms and, as such, is computed by the bisimulation algorithm discussed in Chapter 4. That is, when we discuss hypersets abstractly, we use $=$. But when considering bisimulation algorithm to determine whether two set names or graph nodes denote the same abstract hyperset, we use $\approx$. In the implemented version of the language we have only $=$ which, of course, involves calling the bisimulation algorithm, but this is hidden from the user who, therefore can think on hypersets abstractly. Moreover, bisimulation is implicitly involved in the (computational) meaning of the *membership* relation according to the equivalence

$$l{:}a \in b \iff \exists m{:}x \in b.(m{=}l \;\&\; x{\approx}a)$$

informally having the meaning: find an outgoing l-labelled edge from b which leads to some node x bisimilar to a. But, thinking abstractly, $l : a \in b$ says simply that a is an l-labelled element of b.

The *logical operators* ($\&, \vee, \neg$) have the usual meaning from propositional logic and can be used to form logical sentences from Δ-formulas. *Universal quantification* can be understood in terms of conjunction:

$$\forall l{:}x \in a.\varphi(x,l) \iff \bigwedge_{l_i:x_i \in a} \varphi(x_i, l_i)$$

and *existential quantification* in terms of disjunction:

$$\exists l{:}x \in a.\varphi(x,l) \iff \bigvee_{l_i:x_i \in a} \varphi(x_i, l_i)$$

assuming that $a = \{l_1 : x_1, \ldots, l_n : x_n\}$. It is evident from this definition that quantification occurs over those elements of the set denoted by a which satisfy the formula φ. That is, quantification is bounded by (elements of) the set a, with the Δ formula φ being called the scope of the quantifier.

Note that when a quantified formula participates as a subformula of a bigger formula or of a term the technical problem arises where exactly this (sub)formula is finished, that is what is the scope of the quantifier. In the implemented Δ-language (Appendix A.1) there is a discipline of using parentheses to find unambiguously the scope of quantifiers, both intuitively and by the implemented parser (and contextual analysis algorithm). Say, in

$$\forall l : x \in a \,.\, (\varphi \;\&\; \psi \;\&\; \chi)$$

the scope of the quantifier is the whole expression in the parentheses. But the general informal rule is: the scope of any quantifier is as small as possible. For example, in

$$(\forall l : x \in a \,.\, \varphi \;\&\; \psi \;\&\; \chi)$$

the multiple conjunctions requires some compulsory external parentheses (exactly as shown), and then the scope of the quantifier is either φ (excluding ψ and χ) or some initial part of φ, if syntactically meaningful at all. We will not give the formal definition which is usually widely known and intuitively evident. For the precise definition of the scope of quantifiers, declarations, etc. the reader should, first, inspect the relevant part of the Δ-language syntax in Appendix A.1 and, most importantly, read the Section 9.2 on contextual analysis which, in fact, served as a rigorous conceptual guidance for us to implement the language correctly.

3.2.2 Set valued expressions — Δ-terms

The set constant *empty set* ($\emptyset$) denotes the set $\{\}$ having no elements. In general, set values are represented symbolically by either: set constants, set variables or Δ-terms. Furthermore, "literal" set values can be introduced with the *enumeration* expression $\{l_1 : a_1, ..., l_n : a_n\}$ which can create new sets, possibly with nesting if some a_i are also enumeration expressions, however, a_i may also be arbitrary Δ-terms.

The *collection* operation $\{l : t(x,l) \mid l : x \in a \;\&\; \varphi(x,l)\}$ denotes the set of labelled elements $l\!:\!t(x,l)$ with $t(x,l)$ a Δ-term depending on the set and label variables l and x, where $l : x$ ranges over the set a, for which the Δ-formula $\varphi(x,l)$ holds. We can also consider the more special case of collection called the *separation* operation $\{l : x \in a \mid \varphi(x,l)\}$ which denotes the set of labelled elements $l\!:\!x$ in a for which $\varphi(x,l)$ holds.

The (unary) *union* operation $\bigcup a$ is understood as the (multiple) ordinary union over the elements of a. Let us assume $a = \{l_1\!:\!a_1, \ldots, l_n\!:\!a_n\}$ then

$$\bigcup a = a_1 \cup \ldots \cup a_n$$

with the ordinary union used in the right-hand side of equality. In particular, this also shows that the ordinary union is definable by means of the unary union and enumeration operators. This is only the simplest example of expressibility in Δ. As we mentioned, this language has, in fact, very high expressive power exactly corresponding to polynomial time computability over hereditarily-finite hypersets[2].

The *transitive closure* $\mathsf{TC}(a)$ denotes the set of (labelled) elements of elements, ... , of elements of a including a itself. This can also be written (not fully formally, say, due to ... present) as:

$$l{:}x \in \mathsf{TC}(a) \iff l{:}x \in x_0 \in \ldots \in x_n = a \vee (l = \Box \;\&\; x = a)$$

with x_i some intermediate elements in the membership chain, each belonging to the next x_{i+1} with some label l_i whose value is not important. In particular, we let $\Box : a \in \mathsf{TC}(a)$.

The above core constructs of the Δ-language extended with the two additional constructs recursion and decoration (introduced below) define all polynomial time computable operations and relations over hypersets (represented as WDB); see the precise formulations in [41, 43, 57].

3.2.2.1 Recursion operation

The *recursion* operator $\mathsf{Rec}\ p.\{l : x \in a \mid \varphi(x, l, p)\}$ defines a subset π of the set denoted by (the Δ-term) a, obtained as the result of stabilising (due to finiteness of a) the inflating sequence of subsets of a defined iteratively as:

$$\begin{aligned} p_0 &= \emptyset \\ p_1 &= p_0 \cup \{l{:}x \in a \mid \varphi(x, l, p_0)\} \\ p_2 &= p_1 \cup \{l{:}x \in a \mid \varphi(x, l, p_1)\} \\ &\ldots \\ p_{k+1} &= p_k \cup \{l{:}x \in a \mid \varphi(x, l, p_k)\}. \end{aligned}$$

Evidently, all $\emptyset = p_0 \subseteq p_1 \subseteq \ldots$ are subsets of a. As a is finite, $p_k = p_{k+1} = p_{k+2}, \ldots$ for some k, and this stabilised value, denoted above as π, is taken as the value of the recursion operator.

[2] Any hyperset set is hereditarily-finite if and only if it contains a finite number of elements, and these elements are also hereditarily-finite hypersets, etc. Moreover, it is required that the transitive closure of this hyperset is also finite.

3.2.2.2 Decoration operation

Recall that in Chapter 2 graph nodes were shown to denote (hyper)sets, and vice versa, arbitrary hereditarily-finite hyperset can be represented in this way.

Now, we shall consider finite graphs in set theoretic terms. Traditionally, this is done by defining a graph as a set of ordered pairs where ordered pairs represent graph edges, for example $\langle a, b\rangle$ denoting the edge $a \to b$. Here (the arbitrary sets) a and b, play the role of the source and target vertices of the edge $a \to b$. Thus, any set g of ordered pairs can be treated as a graph. Formally such ordered pairs are represented as the sets containing two elements labelled by fst and snd respectively, such as $\{fst : a, snd : b\}$. That is, we define $\langle a, b\rangle = \{fst : a, snd : b\}$. Any labelled ordered pair $l : \{fst : a, snd : b\}$ represents a labelled edge $a \xrightarrow{l} b$. In general, we can consider absolutely arbitrary hyperset g as representing a graph. Indeed, we can take into account only those elements of g which happen to be ordered pairs, and ignore the other non-pair elements. This will make the operation of decoration defined below applicable to the arbitrary hyperset g what is convenient. Otherwise the formulation of the language Δ would be more complicated. Also, the arbitrary set v may either participate as an element of the ordered pairs of g, i.e. serving as a g-vertex, or, otherwise, it is considered as an isolated vertex of the graph g. In this sense each set v serves as a g-vertex.

Definition 1. The abstract set theoretic *decoration operator* $\mathsf{Dec}(g, v) = d$ takes two arbitrary input sets g and v where the former represents a graph as a set of ordered pairs, and the latter represents some vertex v of this graph. It outputs a new (hyper)set d corresponding to the v-rooted graph g according to the first paragraph of this section.

Note that decoration is the only operator in Δ which allows for the construction of cyclic hypersets, like $\Omega = \{\Omega\}$, from the ordinary "uncycled" sets (of sets of sets,...) of finite depth. For example, consider the *trivial cyclic* graph g defined by the following system of set equations,

$$g = \{\ \{fst : a, snd : a\}\ \}$$
$$a = \{\}$$

The result of applying decoration to the graph g and the participating vertex a would be,

$$\Omega = \{\Omega\}$$

where Ω denotes the result $\mathsf{Dec}(g, a)$. Indeed this leads to the construction of the cyclic membership represented by the unique g-edge $a \to a$. In fact, here the Anti-Foundation Axiom

from Section 2.4.4 guarantees that Ω is a unique hyperset denoted by $\mathsf{Dec}(g, a)$ (and the same for arbitrary g and a).

This operator can also be reasonably called the *plan performance operator* [61] because its input(s) can be considered as a graphical plan for the construction of a hyperset with the output being the resulting abstract hyperset. Imagine that we have a plan of a Web site (i.e. of a system of hyperlinked Web pages) and that Dec is a tool (or query) which automatically creates all the required Web pages. See also Section 3.5.3 for a more involved example of using the decoration operation for defining a restructuring query.

3.3 Operational semantics

Consider any set or boolean query q which involves no free variables and whose participating set names (constants) are taken from the given WDB system of set equations. Resolving q consists in the following two macro steps:

- **Extending** this system by new equation $res = q$ with res a fresh (i.e. unused in WDB) set or boolean name, and
- **Simplifying** the extended system:

$$\text{WDB}_0 = \text{WDB} + (res = q)$$

  until it will contain only flat bracket expressions as the right-hand sides of the equations or the truth values *true* or *false* (if the left-hand side is boolean name).

After simplification is complete, these set equations will contain no complex set or boolean queries (like q above). In fact, the resulting version WDB_{res} of WDB will consist (alongside the old equations of the original WDB) of new set equations (new set names equated to flat bracket expressions) and boolean equations (boolean names equated to boolean values, *true* or *false*). This process of computation by *extension* and *simplification* was described in [61] as reduction steps

$$WDB_0 \rhd WDB_1 \rhd \ldots \rhd WDB_{res}$$

where WDB_0 is the initial state of WDB extended by the equation $res = q$, and WDB_{res} is the final step of reduction consisting of only flat set equations including the flattened version of set equation $res = q$ (or boolean equation, if q is a Δ-formula). Each reduction step represents simplification by applying rewrite rules which transform set equations involving complicated

Δ expressions into simpler, semantically equivalent, equations. Note that the rewrite rules described here are based on those in [61] but extended to the labelled case as considered in this thesis. In general, rewrite steps are denoted by the $\rhd$ symbol which means "transforms to". Firstly, let us assume participation of the set names s, p, r in the rewrite rules below, which correspond to the set equations

$$\begin{aligned} s &= \{l_1 : s_1, ..., l_a : s_a\}, \\ p &= \{m_1 : p_1, ..., m_b : p_b\}, \\ &\dots \\ r &= \{n_1 : r_1, ..., n_c : r_c\} \end{aligned}$$

existing either in the initial WDB or in the current reduction WDB_i. The operational semantics for the Δ operators (except for recursion, decoration, transitive closure, bisimulation and label relation operators) are described as the reduction rules

$$res = t(t_1, \dots, t_a) \rhd \begin{cases} res & = t(res_1, \dots, res_a), \\ res_1 & = t_1, \\ & \dots \\ res_a & = t_a. \end{cases}$$

$res = \{l : s, m : p, \dots, n : r\}$ – no further reduction required once $s, p \dots, r$, are set names,

$res = s \cup p \cup \dots \cup r \rhd res = \{l_1 : s_1, ..., l_a : s_a,\ m_1 : p_1, ..., m_b : p_b,\ \dots, n_1 : r_1, ..., n_c : r_c\}$,

$res = \bigcup s \rhd res = s_1 \cup \dots \cup s_a$,

$res = \mathsf{TC}(p)$ – operational semantics described in Section 8.1.5,

$res = \{l : x \in p \mid \varphi(l, x)\} \rhd res = \{m_{i_1} : p_{i_1}, \dots, m_{i_{b'}} : p_{i_{b'}}\}$

where $m_{i_j} : p_{i_j}$ are all those $m_i : p_i \in p$ for which $res_i = \varphi(m_i, p_i) \rhd res_i =$ **true**,

$res = \{t(l, x) \mid l : x \in p \ \&\ \varphi(l, x)\} \rhd res = \{t(m_{i_1} : p_{i_1}), \dots, t(m_{i_{b'}} : p_{i_{b'}})\}$

where $m_{i_j} : p_{i_j}$ are all those $m_i : p_i \in p$ for which $res_i = \varphi(m_i, p_i) \rhd res_i =$ **true**,

$res = \mathsf{Rec}\ p.\{l : x \in a \mid \varphi(l, x, p)\}$ – operational semantics described in Section 8.1.3,

$res = \mathsf{Dec}(a, b)$ – operational semantics described in Section 8.1.4,

$res = \forall l : x \in p\ .\ \varphi(l, x) \rhd res = \varphi(m_1, p_1)\ \&\ ...\ \&\ \varphi(m_n, p_n)$,

$res = \exists l : x \in p\ .\ \varphi(l, x) \rhd res = \varphi(m_1, p_1) \vee ... \vee \varphi(m_n, p_n)$,

$res =$ **true** $\&$ **true** $\rhd res =$ **true**,

$res = \textbf{false} \;\&\; \varphi \rhd res = \textbf{false}$,

$res = \varphi \;\&\; \textbf{false} \rhd res = \textbf{false}$,

$res = \varphi \vee \psi \rhd res = \neg(\neg\varphi \;\&\; \neg\psi)$,

$res = \neg\textbf{false} \rhd res = \textbf{true}$,

$res = \neg\textbf{true} \rhd res = \textbf{false}$,

$res = l : s \in p \rhd res = \exists m : x \in p \;.\; (s = x \;\&\; l = m)$,

$res = x = y \rhd x \approx y$ – operational semantics described in Section 4.2.1,

$res = l \; R \; m$ – operational semantics described in Section 3.2.1.

The implementation of Δ-query execution is based on this process of reduction except for the Δ-terms: recursion, decoration, transitive closure described in Section 8.1.3, Section 8.1.4 and Section 8.1.5 respectively; and the Δ-formulas: set equality (bisimulation) and label relation operators described in Section 4.2.1 and Section 3.2.1 respectively.

3.3.1 Examples of reduction

The above process of computation by *reduction* is quite natural as shown in the following examples.

3.3.1.1 Example elimination of complicated subterms

Let us consider the reduction of the query $q = \bigcup q_1$ containing the complex subquery q_1. In general, any complicated term $t(t_1, \ldots, t_n)$ can be simplified by invoking the splitting rule which transforms the equation $res = t(t_1, \ldots, t_n)$ to the resultant equations

$$\begin{aligned} res &= t(res_1, \ldots, res_n) \\ res_1 &= t_1 \\ &\ldots \\ res_n &= t_n \end{aligned}$$

Therefore, the complicated query $res = \bigcup q_1$ can be split into two subqueries, $res = \bigcup res_1$ and $res_1 = q_1$ where res_1 is a new set name.

3.3.1.2 Example reduction of union

In the case of our union query having the particular form $q = \bigcup\{l:s, m:p, n:r\}$ where s, p, r represent set names, it follows that the equation $res = q$ is reduced by the following steps:

1. Split the complicated equation $res = \bigcup\{l:s, m:p, n:r\}$ resulting in the equations:

$$res = \bigcup res_1$$
$$res_1 = \{l:s, m:p, n:r\}$$

   where s, p, r are set names, and hence do not require further splitting.

2. Reduce unary union $res = \bigcup res_1$ to multiple union resulting in the equation:

$$res = s \cup p \cup r$$

   with the unary union reduced to multiple unions over the elements of the set res_1 (the set names s, p, r).

3. Reduce multiple union $res = s \cup p \cup r$ to the bracket expression resulting in the equation:

$$res = \{l_1:s_1, ..., l_i:s_i,\ m_1:p_1, ..., m_j:p_j,\ n_1:r_1, ..., n_k:r_k\}$$

   assuming that the current extension of the original WDB already contains the simplified equations $s = \{l_1:s_1, ..., l_i:s_i\}$, $p = \{m_1:p_1, ..., m_j:p_j\}$ and $q = \{n_1:q_1, ..., n_k:r_k\}$. Here multiple union over the sets s, p, r is reduced to the bracket expression containing the elements of these sets.

In general, most of the Δ operators can be resolved using the above reduction rules except for recursion, decoration, transitive closure, bisimulation and label relation operators. In fact, there is no common framework for describing the operational semantics for all the Δ operators, with the latter exceptions described as lower-level algorithms in Chapters 4 and 8.

The main conclusion is that after reduction we will have the equation $res = \{...\}$ of the required form whose right-hand side should involve no complicated terms or formulas, only set names either from the original WDB or new set names introduced during reduction (like res_1 above) together with the corresponding equations of the required form. Thus, execution of a query extends the original WDB to WDB_{res} (simplification of WDB_0 above). This extension with the set name res as an "entrance point" to the result of the query can be considered as a temporary one until we need this result.

In principle, we could also consider *update queries* which would change the original WDB (not only extend it as above), but this is beyond the scope of this work.

3.4 Implemented Δ-query language

The implemented Δ-query language can express all operations definable in the original (as described above). For the purpose of writing queries the grammar of this language is expressed as BNF (see Appendix A.1) which the reader should take into consideration whilst reading the current section. (See Chapter 8 for technical details of the implementation of the Δ-query language.) Note that, not every computable set theoretic operation is definable within the Δ-language but everything which is polynomial time computable (and generic; cf. [41]) is already definable in the original language.

Additional features (not present in the theoretical version of the language) have also been included in the implemented language making the language more practically convenient, but not increasing its theoretical expressive power. These additions, however important practically, are just "syntactic sugaring" of the above theoretical version of Δ.

3.4.1 Queries with declarations

Like in many programming languages allowing procedure declarations and calls we also introduce in the language Δ query declarations and calls. Thus, a query once declared can be invoked as many times as we want by using its name with various parameters. Besides queries, we allow also constant declarations. Each declaration has its own scope especially delimited (unlike quantifiers) by the keywords `in` and `endlet` where the declared queries or constants can be used (called). For example, let us show how full set names[3] (which can be quite long and unmanageable) can be declared and then used as set constants. The following query declares the set constant `BibDB` as an abbreviation of the corresponding full set name:

```
set query
  let set constant BibDB be
    http://www.csc.liv.ac.uk/~molyneux/t/BibDB.xml#BibDB
  in QUERY( BibDB )
endlet;
```

Here `QUERY` denotes any subquery (according to the syntax in Appendix A.1) which may involve (possibly many times) the set constant `BibDB` declared once in the `let` declaration

[3] Recall that full set name consists of XML-WDB file URL extended by simple set name (delimited by # symbol).

at the beginning of the whole query. However in general `let` declarations of constants and queries can appear at any depth of a query.

Let us now consider the more useful case of the query declaration `getBooks`, which in the following example gives the set of all books in the bibliography database illustrated by the graph in Figure 3.1 in Section 3.5 below. We first declare the query `getBooks` with one set variable argument `input` and then call it with the argument value `BibDB`:

```
set query
  let set constant BibDB be
        http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f1.xml#BibDB,
      set query getBooks (set input) be
        separate {
            pub-type:pub in input
            where pub-type='book'
        }
  in call getBooks(BibDB)
endlet;
```

Here the keyword `call` means that we invoke the set query `getBooks` defined above. In general, any query can be declared once and invoked many times, e.g. `getBooks(BibDB1)`, `getBooks(BibDB2)`, etc., each time with various `<parameters>` which may be either any `<delta-term>` or `<label>` according to the BNF. Those relevant parts of the BNF for this set query are as follows,

```
<delta-term with declarations> ::=
     "let" <declarations> "in"  <delta-term> "endlet"

<set constant declaration> ::=
      "set constant" <set constant> ("be"|"=") <delta-term>

<set query declaration> ::=
     "set query" <set query name> "(" <variables> ")"
     ("be"|"=") <delta-term>

<set query call> ::=
     "call" <set query name> "(" <parameters> ")"
```

In general, there are also `<label constant declaration>` and `<boolean query declaration>` syntactical categories. Note that in the syntactic category `<delta-term with declarations>` the keyword `in` evidently does not play the role of the membership relation such as in the case of the other contexts of the Δ-language. Recursive calls are not

allowed in query declarations, that is the declared query name or constant should not occur in the scope of the declaration. For <recursion> (see the syntax in Appendix A.1) we have the special construct recursive separation already discussed above and illustrated below in Section 3.5.4.

3.4.2 Library

The library allows to create query or constant declarations independent of a query. Library commands allow creation and modification of user defined queries and constants. Predefined and also user defined queries and constants can then be used, i.e. called, (globally) in any query. For example, the following library command adds the set constant some-book for the appropriate full set name to the library:

```
library add set constant some_book =
  http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f1.xml#b1;
```

where the identifier some-book may now participate in any subsequent queries in the current query session[4]. Queries and constants can be modified or redeclared by rerunning the library add command. For example, the set constant some_book (above) could be redeclared as follows:

```
library add set constant some_book =
  http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f1.xml#b2;
```

Predefined and user defined[5] library queries/constants can be listed, in brief without the full declarations, with the command,

```
library list;
```

with result of this command (including predefined queries/constants) being,

```
Library command is well-formed and well-typed, but not
executable

Warning, library command successful but no query executed.

Warning, in the case of duplicate declaration names those
declarations at the bottom of the list have precedence.

List of library declaration(s):
```

[4] Query session is the period of time between opening the query system (for running queries and library commands) and closing it. When query system is restarted, only build in query and constant declarations (see the current list in the Appendix A.3) can be used.

[5] added in the current query session

```
set query Pair (set x,set y),
boolean query isPair (set p),
set query First (set p),
set query Second (set p),
set query CartProduct (set x,set y),
set query Square (set z),
set query LabelledPairs (set v),
set query Nodes (set g),
set query Children (set x,set g),
set query Regroup (set g),
set query CanGraph (set x),
set query Can (set x),
set query TCPure (set x),
set query HorizontalTC (set g),
set query TC_along_label (label l,set z),
set query SuccessorPairs (set L),
boolean query Precedes5 (set R,label l,set x,label m,set y),
set query StrictLinOrder_on_TC (set z),
set constant some_book,
set constant some_book
```

The order of query/constant declarations depends on the order in which the corresponding `library add` commands were executed. Note that, the duplicate declarations named `some_book` is the result of running above the `library add` commands, and those declarations appearing at the bottom of the list have precedence over those at the top of the list. Thus, the set constant `some_book` appearing globally in any query would, in fact, have the redeclared set name `http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f1.xml#b2`. However, there is one subtle point: if a query q is declared in the library which calls another library query q_1 (or constant), then q will invoke the latest declaration of q_1 *preceding* this declaration of q even if q_1 is redeclared again after q. Note that the modification or deletion of user defined declarations is not yet implemented, but it can be done easily.

Also, the full declarations of user defined and predefined queries/constants can be listed with the command,

```
library list verbose;
```

with the result being,

```
Library command is well-formed and well-typed, but not
executable

Warning, library command successful but no query executed.
```

```
List of library declaration(s):

  set query Pair (set x,set y) be
    { 'fst':x, 'snd':y },

  boolean query isPair (set p) be (
    exists l: x in p . (
      l='fst'
      and
      forall m:z in p . ( m='fst' => z=x )
    )
    and
    exists l:y in p . (
      l='snd'
      and
      forall m:z in p .( m='snd' => z=y )
    )
  ),

                  ...

  set constant some_book be
  http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f1.xml#b1

  set constant some_book be
  http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f1.xml#b2
```

Here the list of queries/constants follows as above, but including the full declaration for all other default library declarations (omitted here for brevity; see the full listing of predefined library declarations in Appendix A.3). Those relevant parts of the BNF for the library commands are as follows:

```
<library commands> ::= "add" <declarations> |
                       "list" [ "verbose" ]
```

Note that, only the predefined library declarations will remain in the library after finishing the query session. In principle the ability to work with several libraries (as well as user defined libraries) should also be implemented. The queries `Pair`, `isPair`, `First`, `Second` will be formally explained below; `CartProduct`, `Square` and `HorizontalTC` in Section 3.5.4; `LabelledPairs`, `CanGraph` and `Can` in Section 3.5.6; `TC_along_label` in Section 3.6; `SuccessorPairs`, `Precedes5`, `TCPure`, `StrictLinOrder_on_TC` in Section 3.7 and Appendix A.3; whereas `Nodes`, `Children` and `Regroup` in Section 8.1.4.1.

3.4.2.1 The queries `Pair`, `isPair`, `First` and `Second`

Thus, let us now define several auxiliary queries dealing with ordered pairs. According to the syntax in Appendix A.1 query declarations have the general form:

$$\texttt{set query}\ \ q(\bar{x}) = t(\bar{x}),$$
$$\texttt{boolean query}\ \ q(\bar{x}) = \varphi(\bar{x}).$$

Here q is either set or boolean query name, respectively, with query parameters defined by the list $\bar{x}$ of participating set or label variables.

3.4.2.1.1 `Pair`: Our first query defines the operation creating an ordered pair:

```
set query Pair(set x,set y) = {'fst':x,'snd':y}
```

where `'fst'` and `'snd'` are label values helping to distinguish the first element `x` from the second element `y` of the ordered pair, with `x`,`y` as set variables denoting any (hyper)sets. Recall that the order of elements in a set is ignored, playing no role. But, labels of elements such as `fst` and `snd` add the required structure.

3.4.2.1.2 `isPair`: Now we consider the boolean valued query `isPair(p)` which given a set `p` says whether it is an ordered pair `p={'fst':x,'snd':y}` for some sets `x` and `y`:

```
boolean query
isPair(set p) =
    (exists l:x in p .
      ( l='fst' and forall m:z in p . (m='fst' implies z = x) )
    and
     exists l:y in p .
      ( l='snd' and forall m:z in p . (m='snd' implies z = y) )
    )
```

Note that the equalities `z=x` and `z=y` in this query are actually based on the bisimulation relation. It follows that `isPair(p)` can hold even if the set equation `p={...}` contains syntactically more than two elements between braces. It is required that there exists only one element in `p` labelled by `'fst'` and one labelled by `'snd'` only up to bisimulation.

3.4.2.1.3 `First` and `Second`: Let us also define the set valued operations `First(p)` and `Second(p)` giving the first and the second elements of any pair p:

```
set query First(set p) =
    union separate {l:x in p where l='fst' }
```

```
set query Second(set p) =
    union separate {l:x in p where l='snd' }
```

Note that the union operation is necessary here. Indeed, assuming that the input is an ordered pair `p = {'fst':u,'snd':v}`, then we would get without union just singleton sets `{'fst' : u}` and `{'snd' : v}`, respectively, generated by the separation operator whereas we need their elements `u` and `v`, respectively. Therefore, we need to use the general set theoretic identity

$$\bigcup\{l : u\} = u$$

where u is any set. Of course, in the case of arbitrary set input `p` separation will not necessary generate a singleton set. Anyway, `First(p)` and `Second(p)` will give some set values so that these operations are always defined.

3.4.2.2 Implementation of the library

Although general implementation issues will be postponed till Part III, we can easily comment here how implementation of the library can be reduced to the general `let-endlet` construct of the language. Thus, let us assume that the library contains a list of declarations

$$d_1, d_2, \ldots, d_n$$

already added by the `add` command. Then any query q can use these declarations and thus can contain constants and query names which are not declared in q, but must be declared above in the library. In fact, any such query

`set query` q`;` or `boolean query` q`;`

is automatically transformed by the implemented query system, respectively, to the query

`set/boolean query let` $d_1, d_2, \ldots, d_n$ `in` q `endlet;` (3.1)

Then this query is checked to be well-formed and well-typed and then executed as it is discussed formally in Chapters 9 and 8. This way also the problem of dependency between library declarations $d_1, d_2, ..., d_n$, whose order may be essential[6], is resolved automatically. Also query declarations when added to the library are automatically checked simply by

[6] A declaration d_i can depend only on d_j with $j < i$. Even if d_i calls a constant or query name declared by d_k with $i < k$, appropriate (rightmost) d_j with $j < i$ should be really found and used. But this does not require any special or additional care for the library declarations because the contextual analysis algorithm in Section 9.2 will guarantee this automatically under translation (3.1).

transforming them to the usual query

$$\texttt{set query let } d_1, d_2, \ldots, d_n \texttt{ in } \{\} \texttt{ endlet;}$$

where the trivial version of $q = \{\}$ is used. Well-formedness and well-typedness of the latter query is considered, by definition, as well-formedness and well-typedness of the declarations in the library.

3.5 Example Δ-queries

Let us consider the following example queries based on the bibliographic WDB presented in [50] and similar to the example in [1]. This WDB is distributed (split into two fragments) as illustrated by the colouring of the graph in Figure 3.1. Each fragment is given by a subsystem of set equations represented practically as an XML-WDB file (see Chapter 10 for the technical details of the XML-WDB representation). These files can be examined in the Appendix A.2.

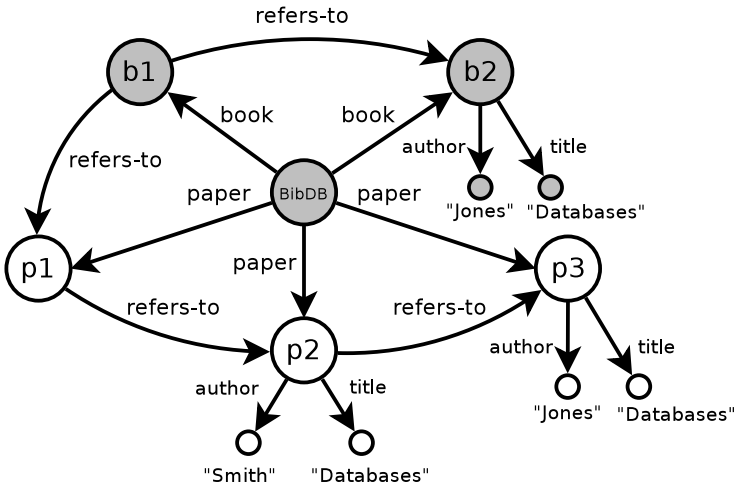


Figure 3.1: Example distributed WDB of a small bibliographic database, distributed into two fragments.

Let us consider the corresponding subsystems of set equations represented practically as XML-WDB files. Note that, full set names are denoted as the concatenation of URL, `#`, and simple set name; however, the URL and the delimiter `#` can be omitted for local set names. The subsystem of set equations represented by the XML-WDB file `http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f1.xml` is as follows:

```
BibDB = {
 'book':b1,
 'book':b2,
 'paper':http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f2.xml#p1,
 'paper':http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f2.xml#p2,
 'paper':http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f2.xml#p3
}

b1 = {
 'refers-to':http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f2.xml#b2,
 'refers-to':p1
}

b2 = {
 'author':"Jones",
 'title':"Databases"
}
```

The XML-WDB file `http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f2.xml` represents the subsystem

```
p1 = {
 'refers-to':p2
}

p2 = {
 'author':"Smith",
 'title':"Databases",
 'refers-to':p3
}

p3 = {
 'author':"Jones",
 'title':"Databases"
}
```

Recall that single quotation marks are used to denote labels such as `'author'`, whereas double quotation marks denote atomic values which are, strictly speaking, special singleton sets, e.g. `"Jones"` means `{'Jones':{}}`.

3.5.1 Example of a non-well-typed query

In our first example the query is non-well-typed because the identifiers `BibDB` and `b2` are formally undeclared within the following query, although intuitively corresponding to some graph nodes. The intended informal meaning of the query being: find all publications which refer to the book `b2`.

```
set query collect {
    pub-type:pub
    where pub-type:pub in BibDB
    and exists 'refers-to':ref in pub . ref=b2
};
```

The result of running this query is the error messages:

```
Query is well-formed, but not well-typed

Error at character 76,

occurrence of identifier name BibDB not declared:
  set query collect { pub-type:pub
  where pub-type:pub in BibDB <-------

  and exists 'refers-to':ref in pub .

Error at character 127,

occurrence of identifier name b2 not declared:
  and exists 'refers-to':ref in pub .
  ref=b2 <-------

  };
```

Here `well-typed` would intuitively mean that all identifiers and their types (*set* or *label*, etc.) in the query are appropriately described by declarations, quantifiers, etc., and used in other places of the query accordingly. But unfortunately the error messages show that it is not the case. The corrected version of this query is presented in Section 3.5.2, where the identifiers `BibDB` and `b2` are appropriately related to the WDB considered. We will pay much more attention to well-typedness of queries in Chapter 9 which is highly important for the correct implementation of Δ.

3.5.2 Example of valid and executable query

After correction of the above query we have:

```
set query
  let set constant BibDB be
        http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f1.xml#BibDB,
      set constant b2 be
        http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f1.xml#b2
  in collect { pub-type:pub
      where pub-type:pub in BibDB
      and exists 'refers-to':ref in pub . ref=b2
    }
  endlet;
```

Evidently the result of this query contains the book `b1` (which refers to `b2`) and, not so obviously, the paper `p2` which refers to `p3`, the latter being formally bisimilar to `b2` with the same `title` and `author` elements. The result of the modified query is,

```
Query is well-formed, well-typed and executable

Result = {
  'paper':http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f2.xml#p2,
  'book':http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f1.xml#b1
}

Finished in: 398 ms
```

This result might seem strange, but formally it is correct taking into account our hyperset theoretic approach to WDB. The question here is to the designer(s) of this bibliographic database who overlooked that essentially the *same* publication is presented in the database both as a book and as a paper. If these are really different publications then they should be represented in the database accordingly (as discussed in the considerations below). Note that the incoming edges labelled by `book` or `paper` do not count when determining bisimilarity of the nodes `p3` and `b2` — only outgoing edges play a role. Such fundamental flaws can be introduced accidentally when possibly many users create distributed WDB. Evidently, this WDB was poorly designed, therefore, better understanding of the structural design of WDB would make this process less error-prone. Anyway, even with the (traditional) relational approach database design is a crucial step.

3.5.2.1 Query semantics versus WDB design

If we really want to include only references to the book `b2` (without redesigning this WDB), then it might seem that the solution is to replace the equality `ref=b2` by the formula

```
(ref=b2 and 'book':ref in BibDB)
```

in the above query. However, this would not really help because in any case p3=b2 (these set names / graph nodes are bisimilar) in the above WDB. Equality of (hyper)sets is defined by their elements, elements of elements, etc., i.e. by outgoing edges, and not by incoming edges. So, after formally removing redundancies (say, omitting p3) we should have one joint node b2 with two incoming edges BibDB $\stackrel{\texttt{book}}{\longrightarrow}$ b2 and BibDB $\stackrel{\texttt{paper}}{\longrightarrow}$ b2 (besides two more incoming refers-to edges from b1 and p2 and the evident two outgoing edges). This is probably not what the designer(s) of this distributed WDB had in mind. Anyway, we will continue using this example as a good and simple illustration of the (hyper)set theoretic approach. In principle, we could imagine that the creators of this WDB really wanted to have publications classified both as a book and a paper. This is not a contradiction, as anything is possible in semi-structured data. In fact, the problem is only to decide what we really want and whether this intuition is reflected correctly by the given WDB design.

This example emphasises the real meaning of set theoretic versus pure graph approaches to semi-structured databases, and the role of removing redundancies on the level of the design. The right approach here should be based on a well-chosen discipline, for example:

(i) *Reconstruct* this database by replacing labels book and paper by publication and adding outgoing edges from each publication showing its type ('book' or 'paper'; see Figure 3.2 [7]), or alternatively

(ii) Enforce some WDB *schema* during the design of WDB e.g. requiring that there is only one book or paper edge from BibDB leading to any given publication considered up to bisimulation.

Here the term "up to bisimulation" means that if two children of BibDB are bisimilar then they, in fact, have identical labelling. But it is not our goal here to go into details of such kind of discipline and consider WDB schemas. In any case, we should be precise and accurate with the design of WDB, and in formulating both formal and intuitive versions of our queries. The mathematical ground of hyperset theory is quite solid and sufficient for that.

The main point is that any formal query has a unique (up to bisimulation) answer – in fact, either a hyperset or boolean value – and all the queries are *bisimulation invariant* and can be computed in polynomial time (with respect to the size of WDB). Vice versa, any P-time computable and bisimulation invariant (and also "generic" [41, 57]) query is definable in Δ. In fact, this also means that the language Δ has full P-time computable power of *restructuring*, not only simple retrieval of already existing elements in the WDB. For example the query

[7] Strictly speaking, Figure 3.2 reflects this idea only partially because it is devoted to illustrate a related but formally different example of restructuring query in the Δ-language. It still has a publication which is characterised as both book and a paper, however, this is more noticeable "locally" reducing accidental user error.

restructuring the BibDB database as is essentially described in (i) above could be written in Δ using the plan performance operator Dec.

3.5.3 Restructuring query

The ability to define queries arbitrarily restructuring any given data is the most essential requirement of any database query language. Here we will consider one simple example which could hopefully convince the reader that Δ has a very strong restructuring power.

Firstly, let us recall the informal meaning of the following useful query declarations in the default library (with the formal meaning fully described in Section 3.4.2.1) and introduce semi-formally one more query CanGraph to be formally defined in Section 3.5.6:

- Pair(x,y) – denoting the ordered pair $\langle x, y\rangle$, in fact the two element set of the form {'fst':x,'snd':y} allowing to distinguish between the first and second elements.
- First(p) – first element of p if p is an ordered pair.
- Second(p) – second element of p if p is an ordered pair.
- CanGraph(x) – denoting the set of labelled pairs $l : \langle u, v\rangle$ where $l : v \in u$ holds in the transitive closure TC(x).

Then the required restructuring query (described informally in (i) above) is defined as follows:

```
set query
  let set constant BibDB =
        http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f1.xml#BibDB,
      set constant restructuredBibDB be
        (U collect{
          'null':if (L='paper' or L='book')
                 then { 'publication':X,
                        'type':call Pair(call Second(X),{L:{}}),
                        L:call Pair({L:{}}, {}) }
                 else {L:X}
                 fi
           where L:X in call CanGraph(BibDB)
           }
        )
  in
    decorate ( restructuredBibDB, BibDB )
  endlet;
```

Here `CanGraph(BibDB)` is essentially the bibliography graph in Figure 3.1, but represented in the traditional set theoretic way as the set of labelled ordered pairs, each denoted in the query as `L:X` with `L` the label and `X` the ordered pair in question. The required restructuring in terms of ordered pairs consists in relabelling of labels `'book'` and `'paper'` as `'publication'`, and creating additional leaf edges with the publication type is done essentially by the following fragment

```
'null':if (L='paper' or L='book')
       then { 'publication':X,
              'type':call Pair(call Second(X),{L:{}}),
               L:call Pair({L:{}}, {})
             }
       else {L:X}
       fi
```

generating appropriate sets of labelled ordered pairs. Then these sets[8] are `collect`ed, and taking the union gives rise to the required restructured set of labelled ordered pairs denoted as `restructuredBibDB`. But abstractly, we need a hyperset rather than this graph (a set of pairs). Thus, finally, the decoration operation applied to the graph `restructuredBibDB` and the vertex `BibDB` generates the required abstract hyperset (as described in general in Section 3.2.2.2). The result of this query is,

```
Query is well-formed, well-typed and executable

Result = {
  'publication':res2,
  'publication':res0,
  'publication':res1,
  'publication':{
    'type':"book",
    'refers-to':res1,
    'refers-to':res2
  }
}

res0 = {
  'type':"paper",
  'author':"Smith",
  'title':"Databases",
  'refers-to':res1
}
```

[8] where the value of the label `'null'` is not important

```
res1 = {
  'type':"paper",
  'type':"book",
  'author':"Jones",
  'title':"Databases"
}

res2 = {
  'type':"paper",
  'refers-to':res0
}

Finished in: 1646 ms (query execution is 1643 ms, and
postprocessing time is 3 ms)
```

As we discussed formerly, atomic values, strictly speaking, denote corresponding singleton sets, for example `"Smith"`, denotes `{'Smith':{}}`. The (new) set names `res0`, `res1` and `res2` correspond, respectively, to the "restructured" publications `p2'`, `p3'/b2'` and `p1'`. Note that, the query system replaces some set names on the right-hand side by the corresponding bracket expression where suitable, thereby presenting the result in a "nested" form. For example the publication `b1'` is implicitly nested in the `Result` set equation.

This result can be more conveniently visualised by Figure 3.2 with the set name `Result` replaced by `BibDB'`, and new set names replaced by corresponding names revelant to the restructured publications (as was discussed above).

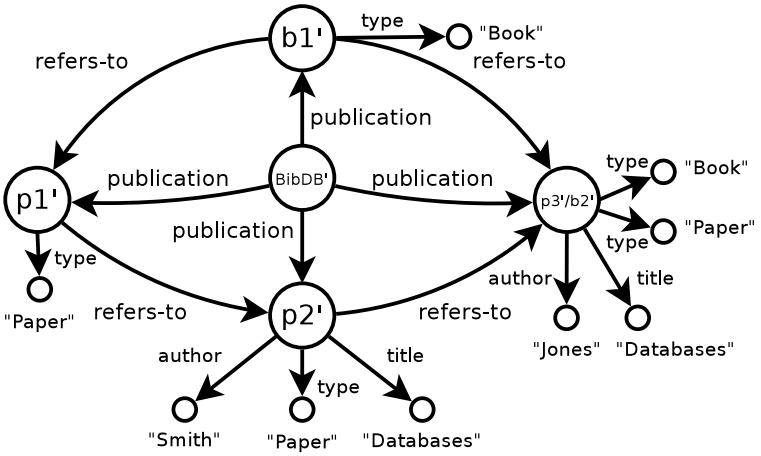


Figure 3.2: The result of the restructuring query.

Note that the publication `p3'/b2'`[9] has both the type `book` and `paper`, and that this unusual

[9] denoted by the new set name `res1` (see query result above)

feature is the result of the initial design of `BibDB` and not a failure of the above query. Anyway, in principle this graph suggests a potentially better (less semantically error prone) design for the bibliography database.

3.5.4 Horizontal transitive closure

Let us now consider the query which can generate the "horizontal" transitive closure[10] of any graph g (a set of ordered pairs). Consider the trivial example graph g represented as the nodes a, b, c with edges $\langle a, b\rangle$ and $\langle b, c\rangle$ depicted by solid black edges in Figure 3.3[11]. The result of applying horizontal transitive closure to the graph g is shown by the original edges (in solid black) and the additional edges $\langle a, c\rangle$, $\langle a, a\rangle$, $\langle b, b\rangle$ and $\langle c, c\rangle$ highlighted in Figure 3.3 as red dashed edges.

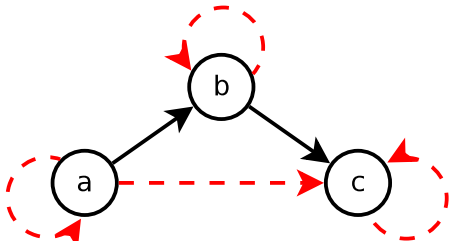


Figure 3.3: The result of "horizontal" transitive closure applied to the abstract graph g.

The result is also a graph denoted as g^* which extends g by new ordered pairs ($g \subseteq g^*$) such that for each edge $\langle x, y\rangle \in g^*$ there exists a path from x to y belonging to the original graph g, and vice versa. This can be recursively defined as follows:

$$\langle x, y\rangle \in g^* \iff x = y \vee \exists z.(\langle x, z\rangle \in g^* \wedge \langle z, y\rangle \in g)$$

or as

$$g^* = \{\langle x, y\rangle \in |g| \mid x = y \vee \exists z \in |g|.(\langle x, z\rangle \in g^* \wedge \langle z, y\rangle \in g)\} \tag{3.2}$$

where $|g|$ is the set of all g-nodes. It is assumed that g^* is the least set of pairs satisfying the above equivalence. This operation could prove useful complementing "vertical" transitive closure $\mathsf{TC}(x)$ in the original Δ-language, whose result is the set of elements of elements, etc. for any given set x (including x itself).

[10] This should not be mixed with the set theoretic meaning of the Δ-term operator transitive closure TC which can be understood intuitively as "vertical" transitive closure, that is $\mathsf{TC}(x)$ represents the set of (labelled) elements of element of elements, etc. of x (including x itself) as defined in Section 3.2.2. The point is that it is typically convenient to think of elements of a set as lying *under* this set – hence *vertical* view.

[11] We should not mix this graph, which is only a visual representation of a *set of ordered pairs*, with any other graphs depicted before and having rather a visual representation of a *system of set equations*.

Thus, let us implement g^* (denoted below as `HorizontalTC(g)`) in the following straightforward way based on the above formula (3.2). Firstly, let us add to the library the set query declaration `Nodes(g)` (formally described in Section 8.1.4.1), denoted above as $|g|$ and extracting from the set of ordered pairs `g` the set of elements participating in these ordered pairs.

`Nodes`:

```
set query Nodes (set g) =
    union separate { m : p in g  | call isPair ( p ) }
```

We will also need the ordinary and very important (not only for defining the horizontal transitive closure) set theoretic operations of

`CartProduct` and `Square`:

```
set query CartProduct(set X,set Y) =
      U collect {'null':collect {'null':call Pair(x,y)
                                 where l:y in Y
                                 }
                 where m:x in X
                 }

set query Square(set X) = call CartProduct(X,X)
```

Finally, the set query `HorizontalTC(g)` can be easily defined using the recursion operator as follows.

`HorizontalTC`:

```
set query HorizontalTC(set g) be
    recursion p {
        'null':pair in call Square(call Nodes(g)) where (
            call First(pair)=call Second(pair)
            or
            exists m:z in call Nodes(g) . (
                'null':call Pair(call First(pair),z) in p
                and
                'null':call Pair(z,call Second(pair)) in g
            )
        )
    }
```

Let us now execute `HorizontalTC` applied to the graph `g` (see above),

```
set query
  let set constant g be {
                  'null':call Pair("a","b"),
                  'null':call Pair("b","c")
                }
  in
  call HorizontalTC(g)
endlet;
```

and see that the result is as expected, although with many repetitions which witness that the implementation is currently not optimal. However, all the repetitions in the query result can be easily eliminated by *canonisation* (to be discussed in Section 3.5.6 below). First note that the canonisation set query declaration (`Can`) is already added to the default library

```
set query Can(set x) be decorate(call CanGraph(x),x)
```

and that the above query can be rewritten using `Can` as follows:

```
set query
  let set constant g be {
                  'null':call Pair("a","b"),
                  'null':call Pair("b","c")
                }
  in
  call Can(call HorizontalTC(g))
endlet;
```

Now, by running the amended query, we see that all repetitions have been eliminated.

3.5.5 Dealing with proper hypersets

The hyperset theoretic approach to WDB can represent and query semi-structured databases possibly involving arbitrary cycles (see Chapter 2). For example let us consider the WDB graph in Figure 3.4 with the cycle between the vertices a and b (edges $a \longrightarrow b$ and $b \longrightarrow a$).

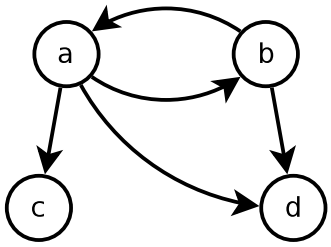


Figure 3.4: WDB graph with cycle.

It is easy to see that $a \approx b$ and $c \approx d$ are the only positive bisimulation facts, and hence a and b, and also c and d actually denote the same hypersets (the latter two denote $\emptyset$). The strongly extensional version of this WDB with all redundancies removed is shown in Figure 3.5.

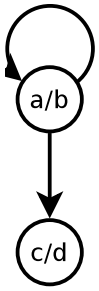


Figure 3.5: Strongly extensional version of the WDB in Figure 3.4.

Let us show how to define in Δ the hyperset denoted by the vertex a. It can be done with the help of decoration operation as follows:

```
set query let
   set constant g = {
         'null':call Pair("a","b"), 'null':call Pair("b","a"),
         'null':call Pair("a","c"), 'null':call Pair("a","d"),
         'null':call Pair("b","d")
   }
in
  decorate (g, "a")
endlet;
```

The result of this query exactly corresponds to the graph in Figure 3.4:

```
Query is well-formed, well-typed and executable

Result = {
  'null':{
    'null':Result,
    'null':{}
  },
  'null':{},
  'null':{}
}

Finished in: 20 ms (query execution is 20 ms, and
postprocessing time is 0 ms)
```

In the next section we will show how the strongly extensional result (corresponding to Figure 3.5) can be obtained. In fact, without using decoration it would be impossible to define this cyclic set `Result` corresponding to the vertex a. Further, let us consider the query to compute equality (bisimulation) between the sets denoting the vertices a and b as

```
boolean query let
   set constant g = {
         'null':call Pair("a","b"), 'null':call Pair("b","a"),
         'null':call Pair("a","c"), 'null':call Pair("a","d"),
         'null':call Pair("b","d")
   }
in
  decorate (g, "a") = decorate (g, "b")
endlet;
```

where the evident result `true` of this query corresponds to the intuitive observation that, in fact, `"a"` and `"b"` denote bisimilar graph g-nodes.

3.5.6 Query optimisation by removing redundancies

The following example demonstrates the general task of removing redundancies by a particular set query `Can` (for "canonisation") on the above graph in Figure 3.4 (in Section 3.5.5). Here we use our knowledge[12] on the implementation of the decoration operation (see Section 8.1.4) to remove the redundancies in the original graph (see the result of the set query above) by applying the decoration operator to the canonical form of this graph (as a set of pairs representing graph edges) and the participating vertex a.

First, let us define the set query declaration

`LabelledPairs:`

```
set query LabelledPairs (set v) be
    collect {
        l:{ 'fst':v , 'snd':u }
        where l:u in v
    }
```

with the result of `LabelledPairs(v)` being the set of labelled pairs $l : \langle v, u \rangle$ denoting labelled edges $v \xrightarrow{l} u$ corresponding to the set memberships `l:u` in the set `v`. This set query declaration participates in another important library set query

[12] This solution may not be so intuitively evident yet to those users who are unfamiliar with the set theoretic meaning of decoration and the details of *how* this operation was implemented (see Section 8.1.4). But running queries with `Can` can nevertheless clearly demonstrate its usefulness.

CanGraph:

```
set query CanGraph(set x) be
    union
        collect {
            'null':call LabelledPairs ( v )
            where m:v in TC(x)
        }
```

whose output is the set of labelled pairs $l : \langle u, v \rangle$ corresponding to those labelled elements $l : v \in u$ with u ranging over the elements of transitive closure $\mathsf{TC}(x)$. Here `'null'` is a label whose value is not important. Indeed, the `union` operation unifies the labelled pairs from `LabelledPairs(v)`. The third library query we need is the set query `Can(set x)` (invoking `CanGraph` above) which takes any set x and returns the same abstract set x, but in its strongly extensional form.

Can:

```
set query Can(set x) be
    decorate (call CanGraph(x), x)
```

In fact, we should always have `Can(x)=x` because `CanGraph(x)` is evidently the canonical graph whose node `x` represents the set `x` itself, and, in this sense, the set query `Can` does nothing. It follows also that `Can` and `decorate` are essentially inverse operations. Thus, `Can` changes nothing in the abstract set theoretical sense. But due to applying decoration to get `Can(x)` and taking into account both strong extensionality of `CanGraph(x)` and the way `decoration` used in `Can` is implemented in Section 8.1.4, the resulting system of set equations generated by `Can(x)` is always non-redundant (strongly extensional).

Therefore the result of `Can(a)` for the example in Figure 3.4 consists of one set equation for the node a/b of the graph shown in Figure 3.5. Indeed, running the query:

```
set query let
   set constant g = {
         'null':call Pair("a","b"), 'null':call Pair("b","a"),
         'null':call Pair("a","c"), 'null':call Pair("a","d"),
         'null':call Pair("b","d")
   }
in
  call Can ( decorate (g, "a") )
endlet;
```

gives the result:

```
Query is well-formed, well-typed and executable

Result = {
  'null':Result,
  'null':{}
}

Finished in: 35 ms (query execution is 35 ms, and
postprocessing time is 0 ms)
```

with the set `Result` denoting a/b. From the abstract hyperset view this is exactly the same result as without using `Can`, but represented in a better, non-redundant way.

Note that `Can` can be used for the more general purpose of query optimisation (not only for optimisation of query results by removing redundancies). Of course, using `Can(t)` instead of `t` will require some time to compute `TC(t)` and then decoration (which in fact requires computation of many bisimulation facts). But the benefit is that `Can(t)` will be represented without any redundancies at all, in contrast to the set `t` which could contain a large number of equal elements due to possible redundancies and thus would be much smaller after eliminating them. Then, for example, `Square(t)` (the Cartesian product of `t`) would also be represented without any unnecessary repetitions, and thus possibly much smaller. In particular, if we want to have recursion over this `Square` (like in the case of recursive definition of `HorizontalTC`), it would be computed much more efficiently, also with smaller number of iteration steps, assuming `Can(t)` instead of `t`.

In principle, we could extend the language by adding *literal equality* `eq(x,y)` for set names (object identities). This, of course, would change the set theoretic character of the language as queries using such equality will not necessarily be bisimulation invariant. But if we would use this equality only over the elements of sets represented as `Can(t)`, then this can work as an additional optimisation. In principle, the query system could recognise the expressions `Can(t)` and automatically replace bisimulation over this set by literal equality.

Finally, note that the above optimisation was given for the current implementation of the Δ-language so that users can exploit canonisation to optimise some queries. In principle, this optimisation could be build into the implementation, so that, any possible redundancies are removed during query execution. In fact, the query system, while executing a query, supports a list of currently known positive bisimulation facts (see Chapter 4) which can be used in background time to remove at least some redundancies in set equations stored in local memory.

3.6 Imitating path expressions

The ability to select nodes of a WDB graph to arbitrary depth can be elegantly achieved using path expressions. As shown in [61], the action of a rich class of path expressions is definable in the original Δ, itself having no path expressions at all, with the help of TC and Rec. In spite of this fact, an important goal for the future work is to implement the extension of Δ by such user friendly path expressions like in the following example query[13] (for simplicity only involving set constants for full set names from the bibliographic WDB):

```
set query
separate {
    pub-type:x in BibDB
    where exists path <b1>refers-to*<x>refers-to<b2> .
          'author':"Smith" in x
};
```

The result of this query would be:

```
Result = {
    paper:p2
}
```

Quantification goes over paths from `b1` to `b2` having an appropriate intermediate set (or node for a publication) `x` which is required to have the element `author:"Smith"`, but it appears that there does not exists such an explicit path. Nevertheless, the required path does exist, as shown in Figure 3.6 by the dashed edges labelled `refers-to` leading from `b1` to `p3`, where `p3` is equal (bisimilar) to `b2` (`p3` $\approx$ `b2`) as we already know. In strongly extensional graphs (where there are no bisimilar nodes) path expressions would be understood quite straightforwardly. Our hyperset approach leads to such kind of complications, but this is the compromise for having a natural language with clear semantics and strong (also precisely characterised) expressive power.

Note that the result of the above query would be the empty set if the Kleene star "$\star$" was removed from the path expression. Indeed, there are no paths of length two from b1 to b2, even up to bisimulation.

[13] The keyword `path` is added to aid reading.

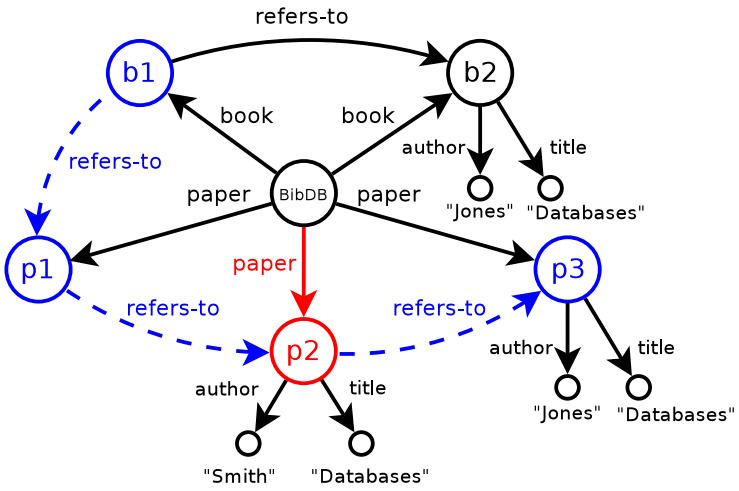


Figure 3.6: Visualisation of the path expression <b1>refers-to*<x>refers-to<b2> applied to the bibliographic WDB.

The action of the path expression <b1>refers-to*<x>refers-to<b2> can, in fact, be "rewritten" into Δ (in its present form) by the following steps. Firstly, consider the subexpression <x>refers-to<b2> denoting a path from the candidate publication x to b2 labelled by 'refers-to'. This can be expressed as the Δ-formula:

```
'refers-to':b2 in x
```

where b2 is set constant and x is set variable. Secondly, the subpath expression <b1>refers-to*<x> denotes set of candidate publications x which can be reached from b1 by navigating zero or more refers-to labelled edges. Thus, let us include in the library the general set query which will give the set of graph nodes (of a graph representing a hyperset z) reachable by navigating zero or more l-labelled edges.

TC_along_label:

```
set query TC_along_label(label l, set z) be
    recursion p { k:x in TC(z)
                  where (
                          ( x=z and k='null' )
                        or
                          ( k=l and exists m:y in p . l:x in y )
                        )
    };
```

Here p is a recursion set variable to representing the set T=TC_along_label(l,z) of nodes lying on potentially all the l-labelled paths outgoing from z. All elements of T are l-labelled, except possibly z. If l:z is in z then l:z will be added to T. But in any case 'null':z will appear in T at the first stage of iteration. Hence the query call

```
TC_along_label('refers-to', b1)
```

represents the path expression <b1>refers-to*<x> where 'refers-to' is label value and b1 is set constant.

Finally the path expression <b1>refers-to*<x>refers-to<b2>, understood as the set of all x lying on the paths matching this path expression, is expressed as:

```
set query
      separate {
          n:xx in call TC_along_label('refers-to', b1)
          where 'refers-to':b2 in xx
      };
```

Now, the fragment

```
exists path <b1>refers-to*<x>refers-to<b2> .
'author':"Smith" in x
```

of our path expression query can be rewritten as

```
exists m:y in separate
              {n:xx in call TC_along_label('refers-to',b1)
               where 'refers-to':b2 in xx
               } .
(x=y and 'author':"Smith" in x)
```

so that we can insert it in the full query

```
set query
let
set constant BibDB =
    http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f1.xml#BibDB,
set constant b1 =
    http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f1.xml#b1,
set constant b2 =
    http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f1.xml#b2
in
separate {
    pub-type:x in BibDB
    where
        exists m:y in separate {
            n:xx in call TC_along_label('refers-to',b1)
            where 'refers-to':b2 in xx
        } .
        ( x=y and 'author':"Smith" in x)
      }
endlet;
```

and run it to see the required result:

```
Query is well-formed, well-typed and executable

Result = {
  'paper':http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f2.xml#p2
}

Finished in: 5766 ms (query execution is 5764 ms, and
postprocessing time is 2 ms)
```

Despite this example of successfully imitating path expressions it would be more useful to also include path expressions directly within the implementation language. Although much more general path expressions can be imitated by Δ-queries in the current version [61], this imitation can be quite complicated in general and is not a particularly efficient way of implementing and executing queries with path expressions. Anyway, the Δ-language, as it is implemented now, is very expressive.

3.7 Linear ordering query

The query example considered in this section has mainly theoretical interest, although it might be useful in practice. The point is that we can define in Δ linear ordering on the transitive closure of any hyperset by using the lexicographical linear ordering we have on labels. In fact, the resulting linear ordering on hypersets is itself, in a sense, lexicographical. Having defined linear ordering, we can further define any ("generic" polynomial-time) computable operation over hypersets by simulating any given Turing Machine (as shown in descriptive complexity theory [34, 37, 55, 74]). This is the key point of the main result in [57] (for well-founded sets) and in [58, 41, 43] (for hypersets) on the expressive power of Δ coinciding with polynomial time computability over (hyper)sets. (We omit precise formulation which is more subtle in the case of hypersets having labelled elements; see [57, 41]).

Let us consider the set query declaration `StrictLinOrder_on_TC(set z)` (and other associated declarations) which can be found in Appendix A.3[14]. In fact, the rather complicated query `StrictLinOrder_on_TC` serves as additional witness demonstrating that everything is implemented correctly, and to check whether and where any optimisation of the implementation is required. Note that `StrictLinOrder_on_TC` invokes `Can` and without this canonisation the transitive closure

```
TCPure(BibDB)
```

[14] It is based on formula (22) and Theorem 2 in [43]. We leave this for the reader to realise how this query below is related with this formula and why it gives a strict linear ordering (see [43]).

participating in the query below (according to Appendix A.3) would have too many repetitions, and, hence, `Square` would have even more repetitions so that the recursion in the set query `StrictLinOrder_on_TC` over this `Square` would take many hours. Now let us run

```
set query
let
set constant BibDB =
    http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f1.xml#BibDB
in
    call SuccessorPairs(
        call StrictLinOrder_on_TC(BibDB)
    )
endlet;
```

Note that `SuccessorPairs` (defined in Appendix A.3) makes the result more concise. We see that our database `BibDB` becomes linear ordered (with corresponding simple set names from the bibliographic database substituted in the place of new set names generated by the query system):

```
Query is well-formed, well-typed and executable

Result = {
  'null':{'fst':{},          'snd':"Databases"},
  'null':{'fst':"Databases",'snd':"Jones"},
  'null':{'fst':"Jones",     'snd':"Smith"},
  'null':{'fst':"Smith",     'snd':BibDB},
  'null':{'fst':BibDB,       'snd':p1},
  'null':{'fst':p1,          'snd':b1},
  'null':{'fst':b1,          'snd':b2/p3},
  'null':{'fst':b2/p3,       'snd':p2}
}

p2 = {'author':"Smith",'title':"Databases",'refers-to':b2/p3}
b2/p3 = {'author':"Jones",'title':"Databases"}
p1 = {'refers-to':p2}
b1 = {'refers-to':b2/p3,'refers-to':p1}
BibDB = {'paper':p1,'paper':p2,'paper':b2/p3,'book':b1,
         'book':b2/p3}

Finished in: 270500 ms (~ 4 minutes and 30 seconds)
```

The correspondence of set names with those nodes in the graph in Figure 3.1 is explicitly shown in the above result. Thus, the resulting linear ordering on the transitive closure of `BibDB` is:

```
{}, "Databases", "Jones", "Smith", BibDB, p1, b1, b2/p3, p2.
```

Here it is important that recursion in `StrictLinOrder_on_TC` does not use bisimulation for comparison iteration steps (see Chapter 4). This crucially optimises recursion, and in particular the query `StrictLinOrder_on_TC` which also uses `Can` in its library declaration. Without the first optimisations this query would take about 30 minutes, and without also using `Can` even hours. Of course, several minutes for such a small database (with `TC(BibDB)` containing 9 sets) is also quite long, and thus the query system implementation needs to be further optimised. But the query is rather complicated (see Appendix A.3), and recursion actually uses $81 = 9^2$ steps of iteration if `Can` is involved. This means in the average about 3.3 seconds per iteration step.

Chapter 4

Bisimulation

Before discussing the theoretical and practical issues surrounding bisimulation, let us recall some relevant details of the hyperset approach to WDB. As previously described in Chapter 2 WDB is represented as a system of set equations $\bar{x} = \bar{b}(\bar{x})$ where $\bar{x}$ is a list of set names $x_1, \ldots, x_k$ and $\bar{b}(\bar{x})$ is the corresponding list of bracket expressions (for simplicity, "flat" ones). Visually equivalent representation can be done in the form of labelled directed graph, where labelled edges $x_i \stackrel{label}{\longrightarrow} x_j$ correspond to the set memberships $label : x_j \in x_i$ meaning that the equation for x_i has the form $x_i = \{\ldots, label : x_j, \ldots\}$. In this case we also call x_j a child of x_i. Note that, our usage of the membership symbol ($\in$) as relation between set names or graph nodes is non-traditional but very close to the traditional set theoretic membership relation between abstract (hyper)sets. Of course this analogy is very important for us and it is indeed highly natural, hence we decided not to introduce a new kind of membership symbol here. For the purposes of our description below labels can be ignored, as inclusion of labels will not affect the nature of our discussion. We will also apply the transitive closure operator $\mathsf{TC}(x)$ to a set name x. The essential point is that in this context $\mathsf{TC}(x)$ is understood as a set of set names (or graph nodes) rather than of abstract sets denoted by these names. Again, we do not bother with introducing a new denotation for such TC.

4.1 Hyperset equality and the problem of efficiency

One of the key points of our approach is the interpretation of WDB-graph nodes as set names $x_1, \ldots, x_k$ where different nodes x_i and x_j can, in principle, denote the same (hyper)set, $x_i = x_j$. This notion of equality between nodes is defined by the bisimulation relation denoted also as $x_i \approx x_j$ (to emphasise that set names can be syntactically different, but denote the same set) which can be computed by the appropriate recursive comparison of child nodes or set names. Thus, in outline, to check bisimulation of two nodes we need to check bisimulation between some children, grandchildren, and so on, of the given nodes, i.e. many nodes could be

involved. If the WDB is distributed amongst many WDB files and remote sites, downloading the relevant WDB files might be necessary in this process and will take significant time. There is also the analogous problem with the related transitive closure operator (TC) whose efficient implementation in the distributed case requires additional considerations not discussed here. So, in practice the equality relation for hypersets seems intractable, although theoretically it takes polynomial time with respect to the size of WDB. Nevertheless, we consider that the hyperset approach to WDB based on bisimulation relation is worth implementing because it suggests a very clear and mathematically well-understood view on semi-structured data and the querying of such data. Thus, the crucial question is whether the problem of bisimulation can be resolved in any reasonable and practical way. Some possible approaches and strategies related with the possible distributed nature of WDB and showing that the situation is manageable in principle are outlined below.

Although for the general database perspective we should consider graphs with labels on edges and hypersets with labelled elements, the majority of our considerations in this chapter will be devoted to the pure case without any labels. Extension to the labelled case is quite straightforward and is not explicitly considered, except in Definition 2 (b). Of course, our implementation of bisimulation relation considers the labelled case.

4.1.1 Bisimulation relation

Equality between set names (or graph nodes) of any WDB is determined by bisimulation relation defined according to [3] (see also [48, 53]).

Definition 2. (a) *Bisimulation relation* $\approx$ (or $\approx_{\mathrm{WDB}}$) on a WDB without labels (the pure case) is the largest one such that for all set names x, y the following implication holds:

$$x \approx y \Rightarrow \forall x' \in x \exists y' \in y(x' \approx y') \;\&\; \forall y' \in y \exists x' \in x(x' \approx y'). \tag{4.1}$$

(b) In the general labelled case, it should satisfy the implication

$$\begin{aligned} x \approx y \Rightarrow \forall l : x' \in x \exists m : y' \in y(l = m \wedge x' \approx y') \;\& \\ \forall m : y' \in y \exists l : x' \in x(l = m \wedge x' \approx y'). \end{aligned} \tag{4.2}$$

It is well-known that the largest such relation does exist. Indeed, the class $\mathcal{R}$ of relations R satisfying any of the above formulas (in place of $\approx$) is evidently closed under taking unions, so the union of all of them is the required largest one $\approx$. In fact, for $\approx$ the implication $\Rightarrow$ above can be replaced by $\Longleftrightarrow$. Moreover, the class $\mathcal{R}$ evidently contains the identity relation $=$ and is closed under taking compositions $R \circ S$ and inverse relations R^{-1}. It follows that the largest such relation $\approx$ is reflexive, transitive and symmetric, that is, an equivalence relation.

The bisimulation relation is completely coherent with hyperset theory as it is fully described in the books of Aczel [3], and Barwise and Moss [5] for the pure case, and this fact extends easily to the labelled case. It is by this reason that the bisimulation relation $\approx$ between set names can be considered as equality relation $=$ between corresponding abstract hypersets. So, we will not go into further general theoretical details concerning the bisimulation relation (except for the concept of local bisimulation in Chapter 6 below), paying the main attention to implementation aspects.

4.2 Computing bisimulation over WDB

Bisimulation relation is computed in our implementation by recursively deriving bisimulation facts. Both positive ($\approx$) and negative ($\not\approx$) bisimulation facts can be derived with the following rules:

$$x \approx y :- \forall x' \in x \exists y' \in y (x' \approx y') \ \& \ \forall y' \in y \exists x' \in x (x' \approx y'). \tag{4.3}$$

$$x \not\approx y :- \exists x' \in x \forall y' \in y (x' \not\approx y') \vee \exists y' \in y \forall x' \in x (x' \not\approx y'). \tag{4.4}$$

In principle, using the rule (4.3) for deriving positive facts is unnecessary. They will be obtained, anyway, at the moment of stabilisation in the derivation process by using only (4.4) (see below). Derivation of bisimulation facts using the above rules (4.3 and 4.4) occur after initial facts have been derived. The rules for deriving these initial facts are partial cases of the main rules (4.3 and 4.4):

$$x \approx y :- (x = \emptyset \ \& \ y = \emptyset) \tag{4.5}$$

$$x \not\approx y :- (x = \emptyset \ \& \ y \neq \emptyset) \vee (y = \emptyset \ \& \ x \neq \emptyset) \tag{4.6}$$

$$x \approx x \tag{4.7}$$

After the derivation of initial facts, rules 4.3 and 4.4 can be recursively applied. Since it is known that bisimulation is an equivalence relation, the following transitivity and symmetry rules can be used alongside the above rules:

$$x \approx z :- x \approx y \ \& \ y \approx z \tag{4.8}$$

$$x \approx y :- y \approx x \tag{4.9}$$

All these rules should be applied until stabilisation, the stage when no more new $x \approx y$ or $x \not\approx y$ facts can be derived by the above rules. Evidentially, stabilisation is inevitable because there are only finitely many set names in the original WDB, i.e. in the corresponding system of set equations. All remaining non-resolved bisimulation questions ($x \stackrel{?}{\approx} y$) can now be concluded as resolved positively as $x \approx y$.

4.2.1 Implemented algorithm for computing bisimulation over distributed WDB

The deeply recursive nature of the bisimulation algorithm seems to suggest that it maybe necessary to effectively compute the transitive closure of the two set names participating in any bisimulation question. For example in the case of the bisimulation question $x \stackrel{?}{\approx} y$, stabilisation is sufficient to establish only for the facts between set names in $\mathsf{TC}(x)$ and $\mathsf{TC}(y)$. In general, it may happen that the full transitive closures will be involved. However, in an optimistic approach, derivation rules (described in Section 4.2) may be applied to the partial transitive closures, with a "progressive" transitive closures computed as necessitated by the derivation rules to facilitate the resolution of a bisimulation question.

Bisimulation algorithm $Bis(x, y)$:

START with resolving the bisimulation question $x \stackrel{?}{\approx} y$.

1. **Create two (initially empty) lists Q and and Eq.** Q will consist of bisimulation questions $u \stackrel{?}{\approx} v$ or their possible answers, and Eq of (downloaded) set equations.

   Note: *During the computation, some bisimulation questions $u \stackrel{?}{\approx} v$ from the list Q can be resolved – replaced by either $u \approx v$ (positive) or $u \not\approx v$ (negative) facts. Thereby Q will contain both non-resolved questions, and positive or negative facts. The process will continue until Q will stabilise*[1].

2. **Initialise populating Q by inserting the bisimulation question $x \stackrel{?}{\approx} y$.**

3. **Acquire set equations** corresponding to those set names involved in all non-resolved bisimulation questions in Q by downloading appropriate WDB files containing these equations. That is, for the question $u \stackrel{?}{\approx} v$ in Q, download the uniquely defined WDB files (by full set names u, v) containing equations $u = \{\ldots\}$ and $v = \{\ldots\}$ (if they have not been downloaded yet).

   Add these equation into the (originally empty) list of set equations Eq (acquired from the WDB).

   Extend Q by all new bisimulation questions (more precisely, those not yet included in Q neither as questions nor as positive or negative answers) for all set names participating in Q plus set names in the right hand side of the (downloaded) set equations from Eq.

   Note: *Not all the downloaded equations (from the downloaded files) will likely participate in Eq and in the generation of transitive closure $\mathsf{TC}(x) \cup \mathsf{TC}(y)$ for the initial question $x \stackrel{?}{\approx} y$, and in this case they may be ignored when generating new questions*

[1] In the case of using the Oracle, as described later in Chapter 5, the questions already asked to the Oracle should be appropriately labelled to avoid asking them again.

(to be added in Q). But they could probably be useful in future computations and could save time on downloading if some equations to be downloaded as prescribed by the current stage have been already downloaded earlier. Thus, all downloaded equations (in fact, WDB files) should be saved in a cache of WDB (in memory) for possible future use. Therefore, before making the quite expensive step of downloading a WDB file the system should check whether it has already been downloaded. This WDB cache should be initialised when beginning general query execution and used by both the general query evaluation procedure and algorithm described here for evaluating bisimulation (or equality) subqueries $u \stackrel{?}{\approx} v$.

Similarly to the cache of WDB, the current versions of Q and Eq should not be discarded from the memory till the end of executing a given query, involving the subquery $x \stackrel{?}{\approx} y$ considered in the current algorithm, because some other bisimulation questions might be involved which could be easily answered with already known Q and Eq.

4. **Iteratively apply derivation rules (4.3) and (4.4)** (thereby resolving some questions in Q) until the initial bisimulation question $x \stackrel{?}{\approx} y$ becomes a resolved fact or, otherwise, until exhaustion by using the currently downloaded (probably incomplete) list Eq of set equations.

   Note: *Some enumerated in Q questions could still remain unresolved.*

5. **Recursive jump**:

   (a) **Is the initial bisimulation question $x \stackrel{?}{\approx} y$ now a resolved fact in Q?**

   Yes – The original bisimulation question has now been resolved (end of algorithm).

   No – Move to step 5b to continue trying to resolve initial bisimulation question and other non-resolved questions in Q.

   (b) **Are there set names u participating in non-resolved questions in Q for which set equations $u = \{\ldots\}$ have not yet been downloaded?**

   Yes – Then move to step 3 by which further facts may be derived once the relevant set equations have been downloaded.

   No – Then the full transitive closure $\mathsf{TC}(x) \cup \mathsf{TC}(y)$ of the initial bisimulation question $x \stackrel{?}{\approx} y$ has been completed, therefore there are no further possibilities to derive/resolve new facts, and stabilisation of the list Q has been achieved. Postulate all non-resolved bisimulation questions as **true** facts. In particular, the original

bisimulation question $x \overset{?}{\approx} y$ has now been resolved positively as $x \approx y$ (end of algorithm).

END with the bisimulation question $x \overset{?}{\approx} y$ resolved positively $x \approx y$ or negatively $x \not\approx y$.

The essential point of the above algorithm for computing bisimulation is that downloading of WDB files is done in a "lazy" way – only when no derivation step is possible. This strategy is chosen because downloading WDB files is the most expensive process of the general implemented bisimulation algorithm. Therefore only in the worst case downloading all the necessary set equations (generating the full transitive closure of the original bisimulation question) will be necessary. Usually this should save a lot of time and memory.

Part II

Local/global approach to optimise bisimulation and querying

Chapter 5

The Oracle

5.1 Computing bisimulation with the help of the Oracle

The concept of the Oracle for Web-like databases is somewhat similar to that of an Internet search engine, such as Google, where the Oracle will attempt to provide bisimulation facts to the Δ-query system when requested and thereby to facilitate the easier computation of set equality. Furthermore, the Oracle should work in background time independently (as well as by requests from the Δ-query system) to derive bisimulation facts.

We assume that to the bisimulation question $x \overset{?}{\approx} y$ the Oracle should give one of three answers *"Yes"*, *"No"* or *"Unknown"*[1]. In the latter case *"Unknown"* should consequently be replaced by the Oracle (after resolving the question itself, probably resulting in some delay) with either *"Yes"* or *"No"*. The answers *"Yes"* or *"No"* must be correct. In fact, asking the Oracle is a way to resolve bisimulation questions, just like applying derivation rules. However, it is likely that the Oracle only provides a partial bisimulation relation (depending on the current state of its work) because of possible updates to WDB forcing the Oracle to redo at least some of its work and the time required to compute bisimulation. Thus, those bisimulation questions answered *"Unknown"* should invoke an initial attempt by the query system to resolve the question locally, hence downloading WDB files with those set equations corresponding to the set names participating in the question(s), etc., as in the algorithm of Section 4.2.1 above. If during the process of local computation the Oracle will replace *"Unknown"* by *"Yes"* or *"No"* then this local attempt to resolve the bisimulation question will be automatically halted due to replacing this question by its answer, however, downloaded WDB files may prove to be useful in future derivation steps of other possible bisimulation questions and should not be discarded from the local cache.

[1]More precisely, to know which question is answered, full answers should be given: "$x \approx y$", "$x \not\approx y$" or "$x \overset{?}{\approx} y$".

For example, let us consider the Oracle attempting to resolve a bisimulation question posed by the Δ-query system as shown below:

Δ-query system: $x \overset{?}{\approx} y$ (is the set name x bisimilar to the set name y?).

Oracle: *"Unknown"* (based on the current state of knowledge of the Oracle).

The Oracle works towards resolving various bisimulation questions, in particular $x \overset{?}{\approx} y$.

500ms later...

Oracle: *"No"* ($x \not\approx y$ holds).

5.2 Imitating the Oracle for testing purposes

As the first attempt, an Oracle which is able to answer bisimulation questions can be simulated with a single file containing a list of bisimulation facts with the states *"Yes"* or *"No"*. Further, those bisimulation questions initially answered as *"Unknown"* can be also simulated as delayed answers of *"Yes"* and *"No"* by associating each bisimulation fact with number of milliseconds delay.

For the purposes of our preliminary implementation the trivial Oracle (simulated as a file instead of a special Internet server) was implemented as an XML file[2]. The trivial Oracle (XML file) contains all the necessary information to simulate the behaviour of the Oracle: bisimulation facts corresponding to all possible bisimulation questions. Also, to simulate those questions initially answered *"Unknown"* by the Oracle (such as in the example above) each bisimulation fact has an associated delay time. These XML files are generated by one of the programs belonging to our suite of tools from a given WDB in such a way that all *"Yes"*/*"No"* facts presented there are automatically true, that is the bisimulation relation is computed by this program and presented as an XML file. Furthermore, arbitrary delay times (useful for the purposes of testing) are added manually to those XML files generated by this program.

Each bisimulation fact (in the trivial Oracle) is represented as an XML tag with `set_names`, bisimulation `value` and `delay` times as mandatory attributes. For example, let us consider the bisimulation fact $y \not\approx z$ with no delay time represented in the trivial Oracle as,

```
<facts set_name="y">
  <fact set_name="z" value="no" delay="0" />
</facts>
```

[2] which should not be mixed with XML-WDB files used to represent set equations

where bisimulation facts are grouped, inside <facts> and <fact> tags, according to those set name participating in the WDB. The grouping of facts is a feature of the implementation used to generate these XML files. Let us consider the trivial Oracle for the bibliographic WDB (considered in Section 3.5) represented as the XML file:

```
<oracle>

  <facts set_name="http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f1.xml#BibDB">
    <fact delay="0"
        set_name="http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f1.xml#b1" value="no"/>
    <fact delay="0"
        set_name="http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f1.xml#b2" value="no"/>
    <fact delay="0"
        set_name="http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f2.xml#p1" value="no"/>
    <fact delay="0"
        set_name="http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f2.xml#p2" value="no"/>
    <fact delay="0"
        set_name="http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f2.xml#p3" value="no"/>
  </facts>

  <facts set_name="http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f1.xml#b1">
    <fact delay="0"
        set_name="http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f1.xml#b2" value="no"/>
    <fact delay="0"
        set_name="http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f2.xml#p1" value="no"/>
    <fact delay="0"
        set_name="http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f2.xml#p2" value="no"/>
    <fact delay="0"
        set_name="http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f2.xml#p3" value="no"/>
  </facts>

  <facts set_name="http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f1.xml#b2">
    <fact delay="0"
        set_name="http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f2.xml#p1" value="no"/>
    <fact delay="0"
        set_name="http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f2.xml#p2" value="no"/>
    <fact delay="0"
        set_name="http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f2.xml#p3" value="yes"/>
  </facts>

  <facts set_name="http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f2.xml#p1">
    <fact delay="0"
        set_name="http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f2.xml#p2" value="no"/>
    <fact delay="0"
        set_name="http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f2.xml#p3" value="no"/>
  </facts>

  <facts set_name="http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f2.xml#p2">
    <fact delay="0"
        set_name="http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f2.xml#p3" value="no"/>
  </facts>

  <facts set_name="http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f2.xml#p3">
  </facts>

</oracle>
```

Note that only one value "yes" appears above as it is already known concerning our bibliography database that only the set names b2 and p3 are bisimilar. Information encoded within the such an XML file simulates the responses of the Oracle, i.e. the responses to bisimulation questions. These responses, i.e. the desired bisimulation facts (possibly delayed with the immediate temporary answer *"Unknown"*) may assist the regular bisimulation algorithm. To simulate the Oracle, the bisimulation algorithm in Section 4.2.1 should be

extended replacing step 3 as follows:

3. **Acquiring set equations** $u = \{\ldots\}$ and $v = \{\ldots\}$ corresponding to all those unresolved questions $u \overset{?}{\approx} v$ in Q should now begin with asking the Oracle all these questions (which have not already been asked), and the necessary downloads should follow only in the case where the Oracle answers with *"Unknown"*.

   Note: *According to Footnote 1 (on page 73), the answer* "Unknown"*, in fact, means that the Oracle returns back to the query system the question "*$u \overset{?}{\approx} v$*", and similarly for the answers* "Yes" *and* "No" *in which case the full answers "*$x \approx y$*" and "*$x \not\approx y$*", respectively, should be returned. Otherwise, because of delays, the system will not know how to treat* "Yes"*,* "No" *and* "Unknown".

Evidentially, whilst resolving bisimulation questions (the modified version of) Step 2 will pose many bisimulation question to the Oracle, which will be answered either "Yes" ($u \approx v$) or "No" ($u \not\approx v$) possibly with delays. In fact, the behaviour of the modified bisimulation algorithm can be characterised as follows, depending on the Oracle's responses:

- **Bisimulation questions (**$u \overset{?}{\approx} v$**) to the Oracle directly answered *"Yes"* (**$u \approx v$**) or *"No"* (**$u \not\approx v$**):** In this case, the answer from the Oracle should immediately replace the unresolved question in Q, and the modified bisimulation algorithm will resume its work resolving other non-resolved bisimulation questions from Q.

- **Bisimulation questions (**$u \overset{?}{\approx} v$**) to the Oracle initially answered *"Unknown"* (**$u \overset{?}{\approx} v$**):** In this case, the modified bisimulation algorithm will, in fact, resume its work resolving $u \overset{?}{\approx} v$ and other non-resolved bisimulation questions from Q. Thus, the question will either be resolved locally or the Oracle will replace its answer *"Unknown"* ($u \overset{?}{\approx} v$) by either *"Yes"* ($u \approx v$) or *"No"* ($u \not\approx v$) possibly with some delay.

  Note that, if the Oracle answers the question positively or negatively before being resolved locally then this answer should replace the question in Q and the modified bisimulation algorithm should continue its work (taking into account the newly resolved question – it does not matter in which way the question is resolved, by the Oracle or by the query system)[3].

Note that, step 2 in the present modified form plays a crucial role in performance: resolution of bisimulation questions by the Oracle will save costly downloading of WDB files.

[3] A question answered *"Unknown"* does not require asking the Oracle again. In general, Oracle (as a special Internet server) should remember all questions and reply to the appropriate client accordingly when the answer will be ready.

5.3 Empirical testing of the trivial Oracle

In principle, with the help of the Oracle those Δ-queries which involve set equality (bisimulation) should be executed quicker. The aim of the following empirical testing is to measure the improvement in query performance with the help of the Oracle, in addition to demonstrating the effects of delayed answers to bisimulation questions (those initially answered *"Unknown"*) by the Oracle.[4]

The distributed bibliographic WDB considered in Section 3.5 (see Figure 3.1) is fragmented into two XML-WDB files, thus making computation of bisimulation more dependent on the time taken to download these files. The following example query (already considered in Section 3.5.2) involves set equality to determine which publications belonging to `BibDB` refer to the publication (possibly bisimilar to) `b2`. The requirement to compute bisimulation across the distributed bibliographic WDB makes this simple example particularly suitable for empirical testing of the Oracle:

```
set query
  let set constant BibDB be
        http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f1.xml#BibDB,
      set constant b2 be
        http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f2.xml#b2
  in collect { pub-type:pub
      where pub-type:pub in BibDB
      and exists 'refers-to':ref in pub . ref=b2
     }
  endlet;
```

The execution time of this example query under various experimental conditions can be seen in the Table 5.1. The results suggest a marked improvement in performance with help of the Oracle, and only a slight improvement in performance when the Oracle returns an answer after delay 50ms or 75ms. However, when the Oracle provided a greatly delayed answer ($\geq$ 100ms) query execution occurs with no real help by the Oracle, and bisimulation is computed locally without any real help from the Oracle. Thus, under this circumstance, query execution time increases, and the optimal approach appears to be query execution without invoking the Oracle. This result may be explained by the numerous (and seemingly futile) bisimulation questions posed to the Oracle (all of which are answered *"Unknown"* and never improved) which provide no real help.

[4]Even more optimal would be to postpone local resolution of bisimulation questions in favour of some other independent subqueries of the given query with the hope that the Oracle will give a definite answer before starting local resolution. There are many ways to optimise our implementation, but we can consider only a limited range of such possibilities.

In summary, these results were based on experiments with the trivial Oracle (simulated as an XML file instead of an Internet server). Additionally, the example WDB is too small and, crucially, only distributed into two fragments. In principle, invoking the help of the Oracle should improve query performance considerably when the WDB is distributed into a large number of fragments.

Strategy	Query execution time [ms]
Bisimulation algorithm without invoking the Oracle	588
with help of the Oracle (no delay time per question)	390
with help of the Oracle (50ms delay time per question)	500
with help of the Oracle (75ms delay time per question)	500
with help of the Oracle (100ms delay time per question)	608
with help of the Oracle (125ms delay time per question)	608

Table 5.1: Experimental results showing query execution time [ms] corresponding to each strategy for computing bisimulation.

In a more realistic situation, the Oracle should be implemented as an Internet service (called the bisimulation engine) for large distributed WDB, working in background time to derive all possible bisimulation facts on the current state of WDB. The goal of the bisimulation engine consists in answering bisimulation questions $x \stackrel{?}{\approx} y$ from the Δ-query system (possibly with a delay[5]). The Oracle should be based on the bisimulation algorithm described in Section 4.2.1 and, additionally, on the idea of local/global bisimulation considered in Chapter 6. We will consider implementation (still rather an imitation) of the Oracle in Chapter 7 and some further advanced experiments.

[5] In principle, the Oracle, when asked the question $x \stackrel{?}{\approx} y$, could change its regular behaviour, and try to resolve such questions (with appropriate strategy of priority) from one or more querying clients.

Chapter 6

Local/global bisimulation

Let a proper set[1] $L \subseteq$ *SNames* of "local" vertices (set names) in a graph WDB (a system of set equations) be given, where *SNames* is the set of all WDB vertices (set names). Let us also denote by $L' \supseteq L$ the set of all set names participating in the set equations for each set name in L both from left and right-hand sides. Considering the graph as a WDB distributed among many *sites*, L plays the role of (local) set names defined by set equations in some (local) WDB files of one of these sites. Then $L' \setminus L$ consists of non-local set names which, however, participate in the local WDB files, have defining equations in other (possibly remote) sites of the given WDB. Non-local (full) set names can be recognised by their URLs as different from the URL of the given site. Set names (or vertices) from L' can be reasonably called "almost local".

We will consider *derivation rules* of the form $xRy :- \ldots R \ldots$ for three relations over *SNames*:

$$\approx^L_- \subseteq \approx \subseteq \approx^L_+ \quad \text{or, rather, their negations} \quad \not\approx^L_+ \subseteq \not\approx \subseteq \not\approx^L_-$$

defined on the whole WDB graph (however, we will be mainly interested in the behaviour of $\approx^L_-$ and $\approx^L_+$ on L). We will usually omit the superscript L when it is clear from the context. In particular, this chapter deals mainly with one L, so no ambiguity can arise.

6.1 Defining the ordinary bisimulation relation $\approx$

Recall the derivation rule defining $\not\approx$:

$$x \not\approx y :- \exists x' \in x \forall y' \in y (x' \not\approx y') \vee \exists y' \in y \forall x' \in x (x' \not\approx y'). \tag{6.1}$$

If $u \not\approx v$ is underivable for some vertices/set names u, v then we assume $u \approx v$ to be true

[1] $L \neq \emptyset$ and $L \neq$ *SNames*

(indistinguishable sets are considered equal), and similarly in other cases below. Equivalently, $\not\approx$ is the least relation satisfying (6.1), and its positive version $\approx$ is the largest relation satisfying

$$x \approx y \Rightarrow \forall x' \in x \exists y' \in y(x' \approx y') \;\&\; \forall y' \in y \exists x' \in x(x' \approx y'). \tag{6.2}$$

The relation $\approx$ is called *bisimulation* relation which is also known to be an equivalence relation on the whole graph. Below are defined its upper and lower (relativised to L) approximations $\approx_+$ and $\approx_-$.

6.2 Defining the local upper approximation $\approx_+^L$ of $\approx$

Let us define the relation $\not\approx_+ \subseteq \mathit{SNames}^2$ by derivation rule

$$x \not\approx_+ y :- x, y \in L \;\&\; [\exists x' \in x \forall y' \in y(x' \not\approx_+ y') \vee \ldots]. \tag{6.3}$$

Here and below "..." represents the evident symmetrical disjunct (or conjunct). Thus the premise (i.e. the right-hand side) of (6.3) is a *restriction* of that of (6.1). It follows by induction on the length of derivation of the $\not\approx_+$-facts that,

$$\not\approx_+ \subseteq \not\approx, \quad \approx \subseteq \approx_+ \tag{6.4}$$

$$x \not\approx_+ y \Rightarrow x, y \in L \tag{6.5}$$

$$x \notin L \vee y \notin L \Rightarrow x \approx_+ y. \tag{6.6}$$

As $L \neq \mathit{SNames}$, the set of all vertices, it follows from (6.6) that $\approx_+$ can be an equivalence relation on the whole graph *only* if it is trivial, making all vertices equivalent. But we will show below that it is an equivalence relation locally, that is on L.

Let us also consider another, "more local" version of the rule (6.3)

$$x \not\approx_+ y :- x, y \in L \;\&\; [\exists x' \in x \forall y' \in y(x', y' \in L \;\&\; x' \not\approx_+ y') \vee \ldots]. \tag{6.7}$$

It defines the same relation $\not\approx_+$ because in both cases (6.5) holds implying that the right-hand side of (6.7) is equivalent to the right-hand side of (6.3). The advantage of (6.3) is its formal simplicity whereas that of (6.7) is its "local" computational meaning. From the point of view of distributed WDB with L one of its local sets of vertices/set names (corresponding to one of the sites of the distributed WDB), we can derive $x \not\approx_+ y$ for local x, y via (6.7) by looking at the content of local WDB files only. Indeed, participating URLs (full set names) $x' \in x$ and $y' \in y$, although likely non-local names ($\in L' \setminus L$), occur in the locally stored WDB files with local URLs x and $y \in L$. However, despite the possibility that x' and y' can be in general

non-local, we will need to use in (6.7) the facts of the kind $x' \not\approx_+ y'$ derived on the previous steps for local $x', y' \in L$ only. Therefore,

Note 1 (Local computability of $x \not\approx_+ y$)**.** *For deriving the facts $x \not\approx_+ y$ for $x, y \in L$ by means of the rule (6.3) or (6.7) we will need to use the previously derived facts $x' \not\approx_+ y'$ for set names x', y' from L only, and additionally we will need to use set names from a wider set L' (available, in fact, also locally)*[2]*. In this sense, the derivation of all facts $x \not\approx_+ y$ for $x, y \in L$ can be done locally and does not require downloading of any external WDB files. (In particular, facts of the form $x \not\approx_+ y$ or $x \approx_+ y$ for set names x or y in $L' \setminus L$ present no interest in such derivations.)*

The upper approximation $\approx_+$ (on the whole WDB graph) can be equivalently characterised as the largest relation satisfying any of the following (equivalent) implications for all graph vertices x, y:

$$\begin{aligned} x \approx_+ y \Rightarrow x \notin L \vee y \notin L \vee [\forall x' \in x \exists y' \in y (x' \approx_+ y') \;\&\; \ldots] \\ x \approx_+ y \;\&\; x, y \in L \Rightarrow [\forall x' \in x \exists y' \in y (x' \approx_+ y') \;\&\; \ldots] \end{aligned} \tag{6.8}$$

The set of relations $R \subseteq \mathit{SNames}^2$ satisfying (6.8), in place of $\approx_+$, evidently: **(i)** contains the identity relation $=$ and is closed under **(ii)** unions (thus the largest $\approx_+$ does exist), and **(iii)** taking inverse.

Evidently, any ordinary (global) bisimulation relation $R \subseteq \mathit{SNames}^2$ (that is, a relation satisfying (6.2)) satisfies (6.8) as well[3]. For any $R \subseteq L^2$ the converse also holds: if R satisfies (6.8) then it is actually a global bisimulation relation (and $R \subseteq \approx$). It is easy to check that **(iv)** relations $R \subseteq L^2$ satisfying (6.8) are closed under compositions.

It follows from **(i)** and **(iii)** that $\approx_+$ is reflexive and symmetric. Over L, the relation $\approx_+$ (that is the restriction $\approx_+ \restriction L$) is also transitive due to **(iv)**. Therefore, $\approx_+$ is an *equivalence relation*. (In general, as we noticed above, $\approx_+$ cannot be equivalence relation on the whole graph, due to (6.6).) Moreover, any $x \notin L$ is $\approx_+$ to all vertices (including itself).

6.3 Defining the local lower approximation $\approx_-^L$ of $\approx$

Consider the derivation rule for the relation $\not\approx_- \subseteq \mathit{SNames}^2$:

$$\begin{aligned} x \not\approx_- y \;\; :- \;\; & (x, y \notin L \;\&\; x \neq y) \vee (x \in L \;\&\; y \notin L) \vee (y \in L \;\&\; x \notin L) \vee \\ & [\exists x' \in x \forall y' \in y (x' \not\approx_- y') \vee \ldots]. \end{aligned}$$

[2] This is the case when $y = \emptyset$ but there exists according to (6.7) an x' in x which can be possibly in $L' \setminus L$ (or similarly for $x = \emptyset$). When $y = \emptyset$ then, of course, there are no suitable witnesses $y' \in y$ for which $x' \not\approx_+ y'$ hold. Therefore, only the existence of some x' in x plays a role here.

[3] This imples (6.4) again because $\approx_+$ is the largest relation satisfying (6.8).

The following is an equivalent simplified rule:

$$x \not\approx_- y \;\; :- \;\; ((x \notin L \vee y \notin L) \;\&\; x \neq y) \vee [\exists x' \in x \forall y' \in y(x' \not\approx_- y') \vee \ldots] \tag{6.9}$$

which can also be equivalently replaced by two rules:

$$x \not\approx_- y \;\; :- \;\; (x \notin L \vee y \notin L) \;\&\; x \neq y - \text{“a priori knowledge”}, \tag{6.10}$$

$$x \not\approx_- y \;\; :- \;\; \exists x' \in x \forall y' \in y(x' \not\approx_- y') \vee \ldots \; . \tag{6.11}$$

Thus, in contrast to (6.3), this is a *relaxation*, or, an *extension* of the rule (6.1) for $\not\approx$. It follows that

$$\not\approx \subseteq \not\approx_- (\approx_- \subseteq \approx),$$
$$x \approx_- x \text{ for all } x \in \textit{SNames} \text{ — reflexivity}.$$

The former is trivial, and the latter means that $x \not\approx_- x$ is not derivable. (Indeed, $x \not\approx_- x$ can be derivable only if $x' \not\approx_- x'$ is derivable for some $x' \in x$ on an earlier stage; thus, there cannot exists a first such derivable fact.) It is also evident that

$$\text{any } x \notin L \text{ is } \not\approx_- \text{ to all vertices different from } x,$$
$$x \approx_- y \;\&\; x \neq y \Rightarrow (x, y \in L).$$

The latter means that $\approx_-$ (which is an equivalence relation on *SNames* and hence on L as it is shown below) is non-trivial only on the local set names. Again, like for $\not\approx_+$, we can conclude from the above considerations that,

Note 2 (Local computability of $x \not\approx_- y$)**.** *We can compute the restriction of $\not\approx_-$ on L locally: to derive $x \not\approx_- y$ for $x, y \in L$ with $x \neq y$* (taking into account reflexivity of $\approx_-$) *by (6.9) we need to use only $x', y' \in L'$ (by $x' \in x$ and $y' \in y$) and already derived facts $x' \not\approx_- y'$ for $x', y' \in L, x \neq y$, as well as the facts $x' \not\approx_- y'$ for x' or $y' \in L' \setminus L$, $x' \neq y'$ following from the “a priori knowledge” (6.10).*

The lower approximation $\approx_-$ can be equivalently characterised as the largest relation satisfying

$$x \approx_- y \Rightarrow (x, y \in L \vee x = y) \;\&\; (\forall x' \in x \exists y' \in y(x' \approx_- y') \;\&\; \ldots).$$

Evidently, $=$ (substituted for $\approx_-$) satisfies this implication. Relations R satisfying this implication are also closed under unions and taking inverse and compositions. It follows that $\approx_-$ is reflexive, symmetric and transitive, and therefore an *equivalence relation over the whole*

WDB graph, and therefore *on its local part* L.

Finally, we summarise that both upper and lower approximations $\approx_+^L$ and $\approx_-^L$ to $\approx$ restricted to L are computable "locally". Each of them is defined in a trivial way outside of L, and the computation requires only knowledge at most on the L'-part of the graph. In fact, only edges from L to L' are needed, everything being available locally.

6.4 Using local approximations to aid computation of the global bisimulation

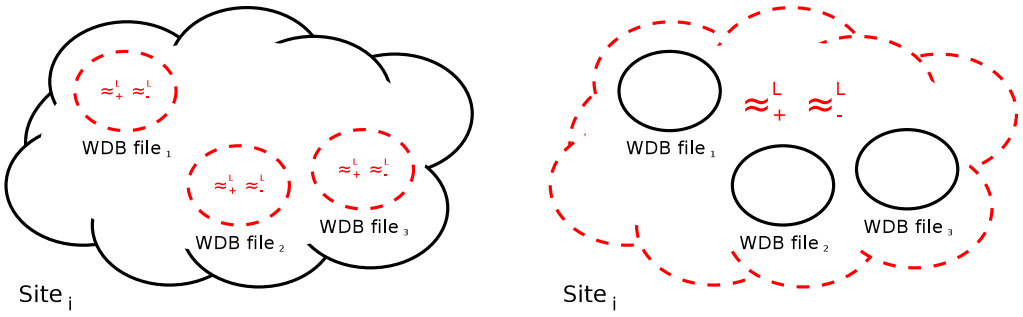
The point of previous considerations of this chapter was that, given any set L of "local" set names (or WDB graph vertices), we defined two (local to L) approximations $\approx_+^L$ and $\approx_-^L$ to the global bisimulation relation $\approx$. Now, assume that the set *SNames* of all set names (nodes) of a WDB is disjointly divided into a family of local sets L_i, for each "local" site $i \in I$ (so that *SNames* is the disjoint union *SNames* $= \bigcup_{i\in I} L_i$). Then we have many local approximations $\approx_+^{L_i}$ and $\approx_-^{L_i}$ to the global bisimulation relation $\approx$. As we discussed above, these relations can be easily computed locally by each site i using the derivation algorithms described in Notes 1 and 2, respectively.

Now the problem is how to compute the global bisimulation relation $\approx$ with the help of many its local approximations $\approx_+^{L_i}$ and $\approx_-^{L_i}$ in all sites i.

6.4.1 Granularity of sites

However, for simplicity of implementation and testing the above idea (and also because it is more problematic to create many sites with their servers) we will redefine the scope of i to a smaller granularity. Instead of taking i to be a site, consisting of many WDB files, we will consider that each i itself is a name of a single WDB file *file*$_i$. More precisely, i is considered as the URL of any such a file. This will not change the main idea of implementation of the Oracle on the basis of using local information for each i. That is, we reconsider our understanding of the term local – from being *local to a site* to *local to a file*[4] – as shown in Figure 6.1. Then L_i is just the set of all (full versions of) set names defined in file i (left-hand sides of all set equations in this file). Evidently, so defined sets L_i are disjoint and cover the class *SNames* of all (full) set names from the WDB considered. Recall that $\approx_+^{L_i}$ and $\approx_-^{L_i}$ are formally defined on the whole WDB (not only on L_i). Their restrictions to L_i are also equivalence relations (on L_i) denoted, for brevity and when it is clear from the context, also as $\approx_+^{L_i}$ and $\approx_-^{L_i}$.

[4] Moreover, this idea of locality to files (described below in detail) belonging to each such a site i is useful for computing i-th site's local upper and lower approximations of bisimulation as an intermediate step. Then these i-th approximations could be used in implementation of the global Oracle. That is, the idea of locality can be fruitfully



(a) Local to files

(b) Local to sites

Figure 6.1: Summary of a distributed WDB showing the difference between interpretation of local as: local to a file, or local to a site.

The relations $\approx_+^{L_i}$ and $\approx_-^{L_i}$ should be automatically computed, saved as file and maintained as the current local approximations for each WDB file i. In principle a suitable tool is necessary for editting (and maintaining) WDB, which would save a WDB file i and thereby generate the approximation relations $\approx_+^{L_i}$ and $\approx_-^{L_i}$ (file) automatically.

6.4.2 Local approximations giving rise to global bisimulation facts

We know that these approximations satisfy,

$$\approx_-^{L_i} \subseteq \approx \subseteq \approx_+^{L_i},$$

or, equivalently,

$$\not\approx_+^{L_i} \subseteq \not\approx \subseteq \not\approx_-^{L_i}.$$

It evidently follows that,

- *each positive local fact of the form* $x \approx_-^{L_i} y$ *is a positive fact about* $\approx$*, i.e. gives rise to the fact* $x \approx y$*, and*
- *each negative local fact of the form* $x \not\approx_+^{L_i} y$ *is a negative fact about* $\approx$*, i.e. gives rise to the fact* $x \not\approx y$*.*

Let $\approx^{L_i}$ (without subscripts $+$ or $-$) denote the set of positive and negative facts for set names in L_i on the global bisimulation relation $\approx$ obtained by these two clauses. This set of facts $\approx^{L_i}$ is called the *local simple approximation set* to $\approx$ for the file (or site) i. Let the *local Oracle* LO_i just answer *"Yes"* ("$x \approx y$"), *"No"* ("$x \not\approx y$") or *"Unknown"* to questions $x \overset{?}{\approx} y$ for $x, y \in L_i$ according to $\approx^{L_i}$.

used on various levels of granularity to optimise performance of the bisimulation engine (the Oracle).

In the case of i considered as a site (rather than a file) then LO_i can have delays when answering *"Yes"* ("$x \approx y$") or *"No"* ("$x \not\approx y$") because LO_i should rather compute $\approx^{L_i}$ itself and find out in $\approx^{L_i}$ answers to the questions asked which takes time. But, if i is understood just as a file saved together with all the necessary information on local approximations at the time of its creation then LO_i can submit the required answer and, additionally, all the other facts it knows at once (to save time on possible future communications).

Therefore, a centralised Internet server (for the given distributed WDB) working as the (global) Oracle or *Bisimulation Engine*, which derives positive and negative ($\approx$ and $\not\approx$) global bisimulation facts can do this by the algorithm of Section 4.2.1, in addition to asking (when required) various local Oracles LO_i concerning $\approx^{L_i}$. That is, the algorithm from Section 4.2.1 extended to exploit local simple approximations $\approx^{L_i}$ should, in the case of the question $x \stackrel{?}{\approx} y$ in Q with $x, y \in L_i$ from the same site/WDB file i[5], additionally ask the oracle LO_i whether it already knows the answer (as described in the above two items). If the answer is known, the algorithm should just use it. Otherwise (if LO_i does not know the answer or x, y do not belong to one L_i – that is, they are "remote" one from another), the global Oracle should work as described in Section 4.2.1 by downloading set equations, making derivation steps, etc. Thus, local approximations serve as auxiliary local Oracles LO_i helping the global Oracle.

6.4.3 Practical algorithm for computation of local approximations

The derivations rules for computing local approximations (described above by rules 6.3, 6.9 together with Notes 1, 2) can be implemented in a very similar way to the practical algorithm for computing the global bisimulation described in Section 4.2. Given a WDB file i as the input, the algorithm will generate *approximation files* i^A and i^{SA} containing local approximations $\approx^{L_i}_+$, $\approx^{L_i}_-$ and, respectively, local simple approximation set $\approx^{L_i}$ (all three approximations restricted to L_i). The derivation rules (6.3, 6.9) were formulated to compute the relations $\approx^{L_i}_+$ and $\approx^{L_i}_-$ over all set names (both local and non-local). According to Notes 1, 2 on local computability of local approximations the computation of restricted relations can be also restricted to local set names in L_i (or to slightly wider set L'_i). Additionally, the two clauses in Section 6.4.2 should be used.

Unlike the practical algorithm for computing global bisimulations, the computation of local approximations $\approx^{L_i}_+$, $\approx^{L_i}_-$, and $\approx^{L_i}$ (creation of approximation files i^A and i^{SA}) should be done after creating (and saving) WDB files i, therefore this operation does not require much attention towards optimisation.

[5] $x, y \in L_i$ can be trivially checked by comparing the full set names x, y with the URL i

Local simple approximation files, i^{SA}, are represented as XML files (quite similar to those of the imitated Oracle; see Section 5.2) containing global bisimulation facts derived locally on the fragment i ($\approx^{L_i}$). Each approximation fact is represented as an (XML) `fact` tag with boolean local approximation value and set name as mandatory attributes `value` and `set_name`. These approximation facts are grouped (inside `facts` tag) corresponding to all local set names in L_i[6].

For example, let us consider the simple approximation file i^{SA}, corresponding to the local simple approximation set $\approx^{L_i}$, for one particular fragment of the bibliographic WDB (see Section 3.5) `http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f1.xml`:

```
<simple-approximation>

  <facts set_name="http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f1.xml#BibDB">
    <fact set_name="http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f1.xml#b1" value="no"/>
    <fact set_name="http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f1.xml#b2" value="no"/>
  </facts>

  <facts set_name="http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f1.xml#b1">
    <fact set_name="http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f1.xml#b2" value="no"/>
  </facts>

  <facts set_name="http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f1.xml#b2">
  </facts>

</simple-approximation>
```

Note that all "no" values above correspond to negative bisimulation facts ($\not\approx$) resulting from the computation of the local simple approximation set $\approx^{L_i}$, where i is the WDB file mentioned above. Simple approximation files are predictably named based on the name of the corresponding WDB file i by concatenating the string "`approximation`" to the end of the WDB file name, for example the WDB file name "`BibDB-f1.xml`" will have corresponding simple approximation file with the name "`BibDB-f1.approximation.xml`".

[6] This is quite similar to the previous implemented tool to generate the (trivial) Oracle XML files.

Chapter 7

The Oracle based on the idea of local/global bisimulation

7.1 Description of the bisimulation engine (implementation of a more realistic Oracle)

Empirical evidence from the implementation of the imitated Oracle in Section 5.3 concluded that a centralised service providing answers to bisimulation question would increase query performance (for those queries exploiting set equality) – this service could be named *bisimulation engine*. The goal of such bisimulation engine would be:

- **Answer bisimulation queries** – Answers bisimulation questions communicating via standardised protocol (as discussed in Section 5.1).
- **Compute bisimulation** – Derive bisimulation facts in background time, and strategically prioritise bisimulation questions posed by the Δ-query system by temporary changing the fashion of the background time work in favour of resolving these questions[1].
- **Exploit local approximations** – Exploit those local approximations corresponding to WDB files to assist in the computation of bisimulation.
- **Maintain cache of set equations** – The Oracle (just like the Δ-query system) should maintain a cache of the downloaded set equations in the previous steps. These set equations may later prove useful in deriving new bisimulation facts with saving time on downloading of already known equations.

[1] Although due to limitations of time, the current implementation is more basic and does not adopt this strategy of prioritising. (See more in Section 7.1.1.

7.1.1 Strategies

In principle, the bisimulation engine should give strategic prioritisation to resolving those bisimulation questions posed by clients – favouring resolution of these bisimulation questions over background tasks (resolving all other bisimulation questions). Moreover, it is reasonable to make the query system adopt a "lazy" strategy while working on a query q. This strategy consists of sending bisimulation subqueries of q to the Oracle but not attempting to resolve them in the case of the Oracle's answer "Unknown" (according to the standard algorithm). Instead of such attempts, the query system could try to resolve other subqueries of the given query q until the resolution of the bisimulation question sent to the Oracle is absolutely necessary. The hope is that before this moment the bisimulation engine will have already given a definite answer.

However these useful features have not yet been implemented. In the current version, we have only a simplified imitation of bisimulation engine which resolves all possible bisimulation questions for the given WDB in some predefined standard order without any prioritisation and answers these questions in a definite way when it has derived the required information. Thus the Oracle, while doing its main job in background time, should only remember all the pairs (client, question) for questions asked by clients and send the definite answer to the corresponding client when it is ready.

7.1.2 Exploiting local approximations to aid in the computation of bisimulation

For implementation of the Oracle we use again the algorithm for computing the bisimulation relation, as described in Section 4.2.1. But, this algorithm will be extended to exploit local approximations by adding an additional step after acquiring set equations (step 3). This additional step (step 3') is detailed below:

3'. **Acquire local approximations** by (i) downloading the local approximation set $\approx^{L_i}$ (consisting of some positive and negative bisimulation facts) represented as the simple approximations file i^{SA} (cf. Section 6.4.3) for each WDB file i retrieved during step 3, and (ii) adding all the positive and negative bisimulation facts from i^{SA} to the list Q of questions and answers (replacing those questions in Q which were thereby answered positively or negatively).

Additionally, while computing global bisimulation by exploiting local approximations, the Oracle should always be ready to receive questions $u \stackrel{?}{\approx} v$ from various, possibly remote Δ-query systems and answer them immediately that the result is yet unknown (if it is so) and, when the result will become known either as $u \approx v$ or $u \not\approx v$, sending it back to the corresponding Δ-query system.

7.2 Empirical testing of the bisimulation engine

Preliminary results from testing of the simulated Oracle (described in Section 5.3) indicated that, in principle, an Internet Service providing answers to bisimulation questions would decrease query execution time for those queries involving set equality. However, these preliminary tests were idealised situations and did not describe the *relationship* between background work by the bisimulation engine and query performance. (In fact, the simulated Oracle did not work in background time, and only some intermediate result was represented.) Additionally, advantages of exploiting local approximations should be demonstrated.

Let us consider empirical testing of the bisimulation engine by measuring the performance of the query client executing (with the help of the bisimulation engine) set equality queries of the form $x \stackrel{?}{\approx} y$ where x, y belong to a some suitable large WDB. To simplify our considerations on measuring efficiency and to demonstrate some desirable effects we will consider rather artificial examples of WDB. As for WDB size, we will try to determine a threshold where the execution time becomes either unrealistically long or sufficiently reasonable. Note that, labels are ignored with just one (identical) label on all graph edges, as labels typically allow the bisimulation algorithm (see Section 4.2.1) to derive more negative facts and, thus, possibly terminating too early (before the transitive closure of both set names involved in any bisimulation question will be fully explored).

7.2.1 Determining the benefit of background work by the bisimulation engine on query performance

The aim of this experiment is to demonstrate the relationship between query execution time t by the query system, and background work by the bisimulation engine. Background work by the bisimulation engine is simulated by delay time d, summarised briefly as follows:

1. The bisimulation engine should begin working with the goal of resolving all possible questions $u \stackrel{?}{\approx} v$ for arbitrary set names of a given WDB. For the sake of the experiment, it should work uninterrupted (without being posed any questions by the query client) for the delay time d.

2. The query client should start executing the test query $x \stackrel{?}{\approx} y$ after the delay time d has expired, attempting resolution of the test question (and possibly other bisimulation questions which may arise during this process) with the help of the bisimulation engine. The bisimulation engine should continue its work, but now communicating with the query client.

Thus, the query execution time $t(d)$ by the query client (working with the bisimulation engine starting from the delay time d) depends on d, and it is this dependence which we want to investigate experimentally. Evidently, $t(d)$ should be a decreasing function: the later the client starts its work after the bisimulation engine, the more help it can provide, and the smaller should be the client's working time $t(d)$. Note that this is still an idealised experiment, in practice, there could be many query clients communicating with the bisimulation engine at arbitrary times.

Note 3. It should be noted that the current implementation of the hyperset language Δ does not use yet any bisimulation engine. These experiments were implemented separately and only to demonstrate some potential strategies for more efficient implementation of the most crucial concept of bisimulation relation underlying the hyperset approach.

In this experiment, the example WDB consists of 51 set names distributed over 10 WDB files, connected in chains as shown by the schematical graph in Figure 7.1. To increase the difficulty of computing bisimulation a copy WDB' of this WDB was made, changing only the URL part of full set names. Thus, the experiment is done over WDB + WDB'. Bisimulation between corresponding set names in WDB and WDB' under this circumstance is intuitively trivial (the answer being always "true"). However, it is a non-trivial task when calculated by our algorithm which has no advance knowledge that WDB and WDB' are essentially identical (isomorphic).

Further, our experimental procedure here was the measurement of execution time $t(d)$ by the query client executing the test query $x \stackrel{?}{\approx} x'$ where x, x' are corresponding set names in WDB and, its isomorphic copy, WDB'.

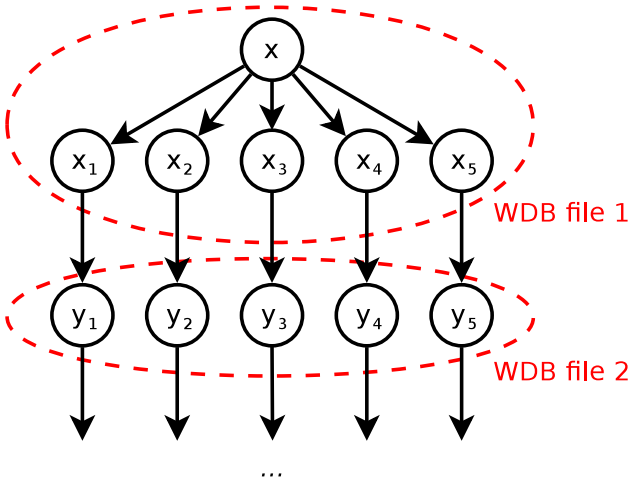


Figure 7.1: Schematical WDB graph divided into WDB files as shown by the red dashed ovals.

7.2.1.1 Experiment results

On examination of the results graph in Figure 7.2 the trend curve suggests an exponential decay relationship between partial work of the bisimulation engine and query performance. Moreover, this qualitative assessment by inspection of the graph is confirmed by examining the experimental values in Table 7.1, which demonstrate that $t(d)$ approximately halves as d increases by steps of 2500ms.

Therefore, query performance benefits considerably even when the bisimulation engine has been working (in the background) for relatively short periods of time (say, 5 seconds or more), with an exponential decrease in $t(d)$ as d increases. However, for sufficiently small delay time d, query performance suffers, as the bisimulation engine answers *"Unknown"* to nearly all posed bisimulation questions. Thus, in this case, the bisimulation engine provides no real help, and the query client is forced to start resolving the bisimulation question itself. This suggests that in this circumstance that local computation of bisimulation by the query system without invoking the help of the bisimulation engine would be more efficient, as shown by the threshold on the graph (dashed horizontal line). In fact, here query execution time $t(d)$ with the help of the bisimulation engine is smaller than without the help of the bisimulation engine when delay d is $>$ 2000ms.

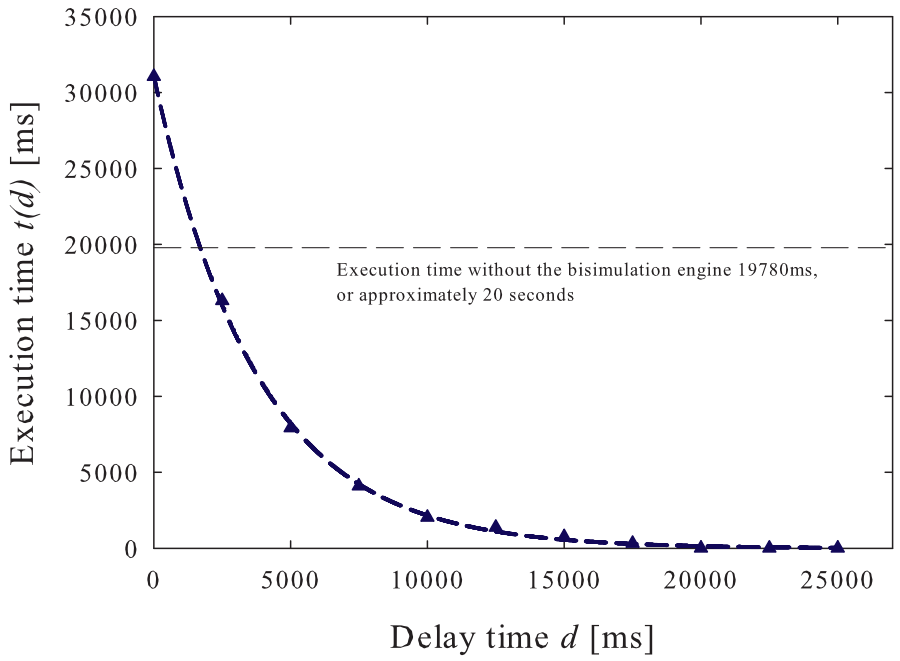


Figure 7.2: Graph of experimental results (cf. Table 7.1 below) showing the dependence of query execution time $t(d)$ [ms] on delay time d [ms]

Delay time d [ms]	Execution time $t(d)$ [ms]
0	31050
2500	16300
5000	7930
7500	4090
10000	2040
12500	1380
15000	770
17500	320
20000	10
22500	10
25000	10

Table 7.1: Experimental results showing dependence of query execution time $t(d)$ [ms] on delay time d [ms]

7.2.2 Determining the benefit of exploiting local approximations by the bisimulation engine on query performance

It seems plausible to expect that, in practice, each WDB file (or a group of closely related WDB files) should be sufficiently self-contained and have few links to the external files – relatively small dependence on the "external world". Therefore, we should expect that the set of locally derived bisimulation facts should be sufficiently large (the majority of questions $x \stackrel{?}{\approx} y$ for local set names should be resolved locally based on $\approx^L_+$ and $\approx^L_-$), and, hence, helpful for the work of bisimulation engine and improving its performance.

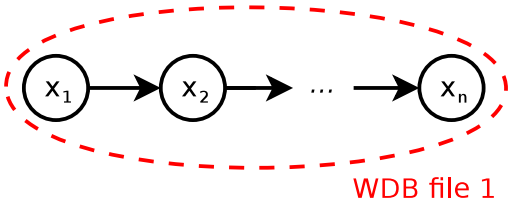


Figure 7.3: Schematical WDB graph consisting of one WDB file as shown by the red dashed oval.

Taking this into account, our alternative example WDB for testing consists of one WDB file containing a variable number n of set names (our experimental parameter as described below) connected in one chain, as shown by the schematical graph in Figure 7.3. Also, like the previous experiment, a copy WDB' of this WDB was made, changing only the URL part of full set name. Likewise, the experimental queries to follow are over WDB + WDB', that is over two

files. This example represents an extreme, idealised case when each of these two files is fully self-contained, i.e. has no links to the "external world". As we wrote above, in more realistic situations we should rather expect a relatively small number of such external links.

Recall that each of the WDB and WDB' files has a corresponding local approximations file, as described in Section 6.4.3, containing, respectively, local sets $\approx^L$ and $\approx^{L'}$ of (positive and negative) bisimulation facts which now will be available by demand to the bisimulation engine (as well as to the query system) which should considerably improve the performance. Thus, for our self-contained WDB file 1 (and similarly with its duplicate) the set of local set names is $L = \{x_1, \ldots, x_n\}$ and the corresponding local facts $\approx^L$ and $\not\approx^L$ obtained from the local approximations $\approx^L_+$ and $\approx^L_-$ trivially coincide with those global bisimulation facts $\approx$ and $\not\approx$ restricted to the set of names L.

The aim of the experiment is to determine the relationship between the size of WDB (input size based on the parameter n) and query performance time comparing the three strategies: **(i)** with the help of the bisimulation engine not exploiting local approximations; **(ii)** with the help of the bisimulation engine, exploiting local approximations; and **(iii)** without the help of the bisimulation engine[2]. Similarly to the previous experiment we measure query performance for the test query $x_1 \stackrel{?}{\approx} x'_1$ where x_1,x'_1 are corresponding set names of the example WDB and its copy WDB'. But now there is *no delay time* between the client and the bisimulation engine starting work. Delay time $d = 0$ is the "worst case" for the bisimulation engine, as proved by the previous experiment. (The case of variable d for a fixed n will be considered in another experiment later.)

7.2.2.1 Experiment results

The graph in Figure 7.4 suggests a sufficiently close to linear trend between query performance and WDB size when the bisimulation engine exploits local approximations. Moreover, this looks almost like a horizontal line, and query execution seems practically viable (~ 41 seconds for $n = 70$; see Table 7.2). On the other hand, with help of the bisimulation engine not exploiting local approximations, as well as without help of the bisimulation engine at all, query performance with sufficiently large WDB ($n = 70$) becomes intractable (more than one hour). In fact query performance improves at a threshold level of approximately $n = 27$ (see Table 7.2) with the bisimulation engine exploiting local approximations, with significant improvement in query performance for larger n compared to the bisimulation engine not exploiting local approximations or without using bisimulation engine at all.

[2] That is, without the help of the bisimulation engine the query client running the test query is forced to compute bisimulation itself.

Moreover, the absence of hyperlinks to other WDB files in our example WDB gives local approximations facts that coincide with those global bisimulation facts restricted to the set names in L or L'. Thus, computing bisimulation requires fewer derivation steps, dramatically decreasing the time required to compute bisimulation. Furthermore, these results suggest that local approximations are more useful when the WDB is divided into larger almost self-contained fragments. The latter is definitely the case when local is understood as *local to a site*. However, in the latter case, local approximations to $\approx$ could take some time to compute at each site. This situation is somewhat different from saving a WDB file with its local approximation set $\approx^L$. Thus more experimentation is required.

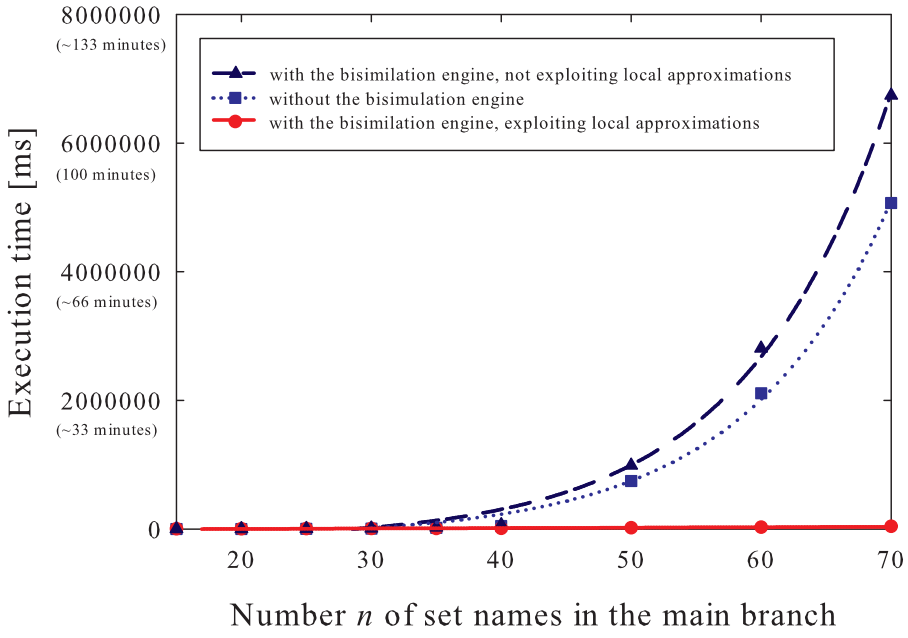


Figure 7.4: Graph of experimental results (cf. Table 7.2 below) showing the relationship between query execution time [ms] and size of WDB (based on the parameter n) – comparing the three strategies towards computing bisimulation

It might seem unexpectedly, but is actually quite natural that the results of this experiment also demonstrate that query performance is worse with the help of the bisimulation engine not exploiting local approximations compared to without the help of the bisimulation engine. In fact, this experiment was conducted with no delay time ($d = 0$), and we should recall the results of the experiments in Section 7.2.1 where a sufficiently small delay times decreased query performance with the help of the bisimulation engine (not exploiting local approximations) due to the additional expense of communication with the bisimulation engine.

Note that the WDB considered in this and the following experiments was artificially created to make computation of bisimulation more difficult. In real situations, in particular where labels are used, it should be possible to derive non-bisimilarity of vertices without the need to go so deeply. However, only realistic application of the Δ-query language can fully show its efficiency and where it should be improved.

	Query execution time (ms)		
Number of set names n	with bisimulation engine exploiting local approximations	without bisimulation engine	with bisimulation engine not exploiting local approximations
15	3422	1015	1340
20	4360	1781	2428
25	5500	3422	4585
30	7015	7781	10368
35	8547	19766	26309
40	10375	48422	64400
50	20063	746187 ($\sim$ 13 mins)	989750 ($\sim$ 16 mins)
60	27516	2113375 ($\sim$ 35 mins)	2810800 ($\sim$ 47 mins)
70	40983	5069797 ($\sim$ 84 mins)	6742890 ($\sim$ 112 mins)

Table 7.2: Experimental results showing query execution time [ms] against WDB size (based on the parameter n) – comparing the three strategies towards computing bisimulation.

7.2.3 Determining the benefits of background work by the bisimulation engine exploiting local approximations

Now let us consider the realistic case where the bisimulation engine is working in background time, comparing both strategies of working by the bisimulation engine: **(i)** with exploitation of local approximations, and **(ii)** without exploitation of local approximations. We shall adopt the same method of testing as previously in Section 7.2.1 with the aim to determine the relationship between query execution time against partial background work[3] by the bisimulation engine for both strategies.

The example WDB used in this experiment is based on notions described in Section 7.2.2 that WDB files (or groups of WDB files) should be relatively self contained with few external links. Thus, here the experimental WDB consists of one (main) WDB file with hyperlinks to two other (auxiliary) WDB files, describing 61 set names in total, as shown by the schematical graph in Figure 7.5. Note that, like those previous experiments in Section 7.2.1 and 7.2.2, the following experimental queries are over WDB and its identical copy WDB'.

[3]Recall that, in Section 7.2.1 the experimental parameter, delay time d, simulated partial background work by the bisimulation engine.

The aim of this experiment is to measure query execution time $t(d)$ by the query client with the help of the bisimulation engine for the test query $x \overset{?}{\approx} x'$ where x, x' are corresponding "root" set names of the example WDB and its copy WDB'. Our experimental parameter is the delay time d, as detailed in the previous experiment Section 7.2.1.

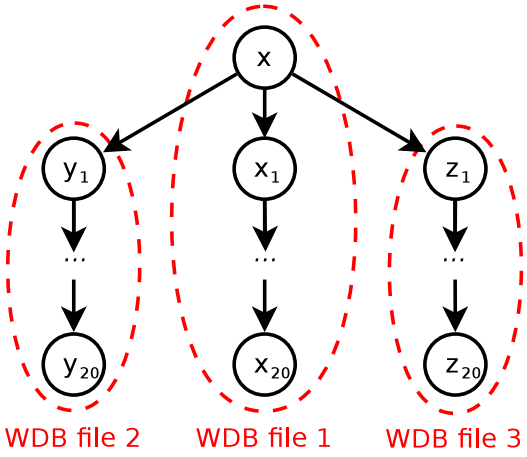


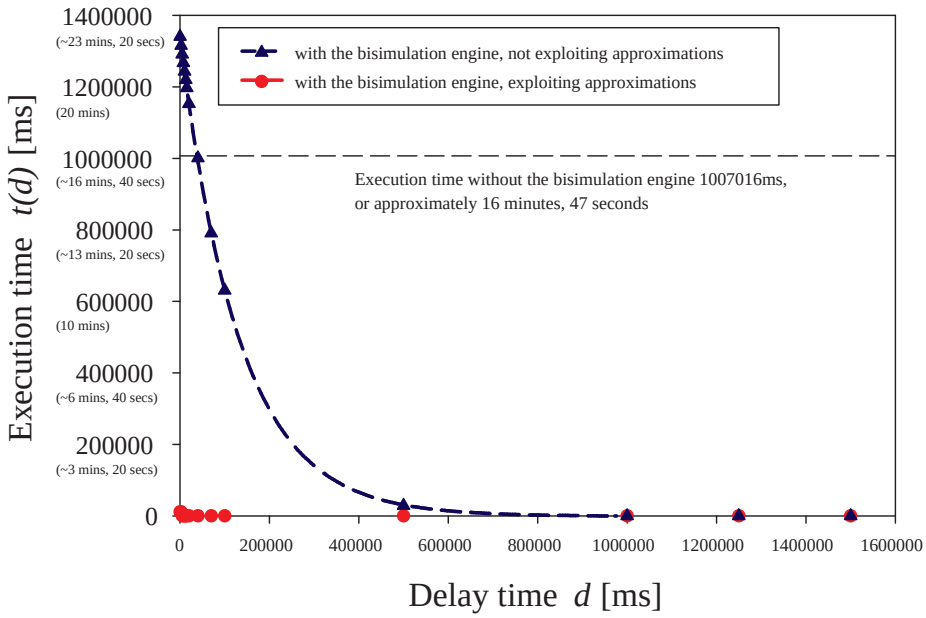
Figure 7.5: Schematical WDB graph divided into three WDB files as shown by the red dashed ovals.

7.2.3.1 Experiment results

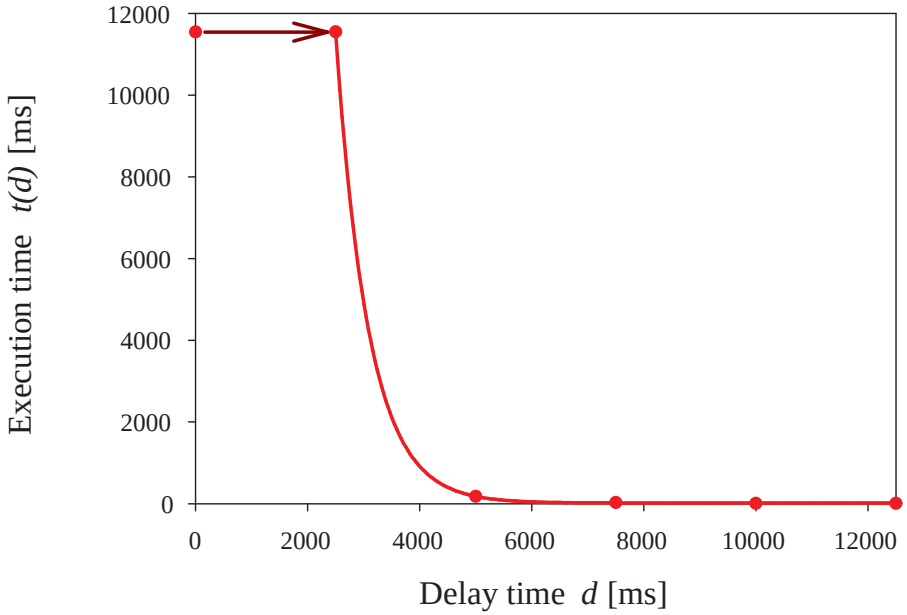
The results of the experiment in Table 7.3 extend previous results in Section 7.2.2 which suggested that exploitation of local approximation by the bisimulation engine increases query performance. However, comparing the influence of partial background work by the bisimulation engine, for both strategies of working, is somewhat difficult due to the difference in magnitude between the results (see Figure 7.6a). In fact, exploitation of local approximations (by the bisimulation engine) reduces query execution time from minutes to seconds, and hours to minutes.

Note that in the case of exploitation of local approximations, the process of derivation is preceded[4] by acquiring these approximations. The additional plot of data in Figure 7.6b shows threshold level, when d is small, that background work by the bisimulation engine does not improve query performance whilst (the initial required) local approximations are being downloaded, as shown by the brown arrow in Figure 7.6b. Furthermore, when exploiting local approximations, a sufficiently large number of locally derived bisimulation facts (on the stage of creating WDB files) actually means in this example that fewer real derivation steps are required.

[4] Downloading approximation files can occur at any stage whilst resolving some bisimulation question.



(a) Comparison between bisimulation engines with and without exploiting local approximations



(b) Bisimulation engine exploiting local approximations

Figure 7.6: Graphs of experimental results demonstrating the relationship between query execution time [ms] and background work by the bisimulation engine simulated by delay time d [ms]

7.3 Overall conclusion

In summary, here two strategies were suggested towards improving the performance of queries involving set equality (bisimulation), these strategies are: **(i)** implementation of an Internet service, bisimulation engine, answering bisimulation questions; and **(ii)** exploitation of local approximations (by the bisimulation engine) to facilitate the quicker computation of bisimulation. It was shown empirically that for an artificial WDB that both strategies, and most dramatically **(ii)**, improved query performance. In fact, the latter strategy demonstrates that querying of a medium sized example WDB could become practically viable.

Note that other recent research into the efficient computation of the bisimulation relation was not considered here, for example the bisimulation algorithm described by Dovier et al [24] (which was intended to optimise the theoretical semi-structured query language G-log [19]). However, the point of the approach presented here was to demonstrate some strategies for computing bisimulation in the case of distributed semi-structured data, unlike that by Dovier et al which did not consider distribution. There was not enough time to consider all possibilities for optimisation, and here we concentrated on those most novel and appropriate to our approach.

	Query execution time with help of the bisimulation engine $t(d)$ (ms)	
Delay time d [ms]	exploiting local approximations	not exploiting local approximations
0	11546	1340250 ($\sim$ 22 mins, 20 secs)
2500	11550	1315269 ($\sim$ 21 mins, 55 secs)
5000	180	1290715 ($\sim$ 21 mins, 31 secs)
7500	28	1266620 ($\sim$ 21 mins, 7 secs)
10000	10	1243000 ($\sim$ 20 mins, 43 secs)
12500	10	1219769 ($\sim$ 20 mins, 20 secs)
15000	10	1197025 ($\sim$ 19 mins, 57 secs)
20000	10	1152728 ($\sim$ 19 mins, 13 secs)
40000	10	1000520 ($\sim$ 17 mins)
70000	10	790760 ($\sim$ 13 mins)
100000	10	630772 ($\sim$ 11 mins)
500000 ($\sim$ 8 mins)	10	28765
1000000 ($\sim$ 17 mins)	10	118
1250000 ($\sim$ 21 mins)	10	10
1500000 (25 mins)	10	10

Table 7.3: Experimental results showing query execution time $t(d)$ [ms] against partial background work by the bisimulation engine simulated by delay time d [ms] – comparing both strategies towards computing bisimulation, with and without exploiting local approximations.

7.3.1 Claims and limitations

The main conclusion from the above experiments is that, although bisimulation (crucial to the hyperset approach to WDB and the Δ-query language) presents some difficulty in efficient and realistic implementation, this problem appears to be resolvable in principle. Moreover, this assertion is somewhat supported by the empirical testing of artificial WDB examples described in Sections 7.2.1–7.2.3. In particular, these artificial WDB were chosen to simulate some specific worst case structural features of WDB similarly to physicists conducting some very specific experiments allowing to understand the most fundamental laws of the nature instead of dealing with something complicated as in the real life. On the other hand, those artificial WDB example presented here are intrinsically limited by their small size[5] and have restricted structural features[6], and, in principle, further comprehensive tests should be done to further characterise the usefulness of those practical strategies towards computing bisimulation suggested here. Also, empirical testing of some particular real-world WDB of sufficiently big size is important, but in this case a lot of further work should be done on optimisation of query execution which was outside of the scope of this work but deserves further investigation. We only considered one essential aspect of efficiency for the current version of the query system related with the idea of local/global bisimulation. However, in principle, the experiments done here suggest that these strategies show potential and merit further investigation.

What has been demonstrated here is probably insufficient for a full-fledged implementation because in real-world circumstances using the Δ-query language could be much more complicated. Anyway, only further work and practical experimentation can reveal problems with the current implementation, which is, of course, not fully perfect. However, it shows that the hyperset approach to databases looks promising and deserves further not only theoretical but also practical considerations – and this was actually our main goal, as well as the goal to create a working implementation available to a wider range of users to realise practically what is the hyperset approach to WDB or semistructured databases.

[5] with the largest WDB considered here involving only 70 set names

[6] which should involve not only nested chains but also nested tree structures

Part III

Implementation issues

Overview of Part III

In this part we discuss the most essential issues of implementing the Δ-query language: (i) query execution (Chapter 8), (ii) syntactical aspects (Chapter 9), and (iii) XML representation of WDB (Chapter 10). These chapters can be read (almost) independently, however, logically their order should be the inverse. The chosen order rather reflects the importance of the material for the reader, who probably should be more interested in the principles of query execution than in the very technical details of implementation of the syntax (in particular related with the subtle points of well-formed vs. well-typed queries). But from the point of view of the actual implementation (including execution of queries) such syntactical aspects were very crucial and, in fact, such technical details serve as a guarantee that the whole implementation was done correctly. Indeed, the content of Chapter 9 arose to overcome the problems of ensuring well-formed/well-typed queries encountered during the first attempt at implementation [49]. Finally, Chapter 10 details the XML representation of WDB, and has quite a separate role. We think and work exclusively in terms of hypersets and set equations, and any WDB could be represented adequately and straightforwardly in the latter form. However, we have chosen XML form (XML-WDB format) as a representation of set equations to make our approach potentially more closely related to the existing practice of using XML for semistructured data. The reader should choose the level of details he/she needs from this chapter for understanding examples of XML-WDB files we use when running Δ-queries.

Chapter 8

$\triangle$ Query Execution

8.1 Implementation of Δ-query execution by reduction process

How to execute any Δ-query was explained mostly in Section 3.3 as operational semantics (based on the general abstract mathematical approach described in [61]) and continued in Section 4.2 on computing bisimulation. Here we will finalise the operational semantics by considering the clauses omitted in Section 3.3 in the style more close to that of implementation. Recall that in this approach, any Δ-term or Δ-formula query q should be equated, respectively, to a new set or boolean name res. Then this equation $res = q$ is reduced (in the context of all set equations of WDB) to an equation $res = V$,

$$res = q \rhd res = V, \tag{8.1}$$

where V is, respectively, either a

- set value – flat bracket expression $\{l_1 : v_1, \ldots, l_n : v_n\}$ where v_i are set names and l_i label values, or
- boolean value – **true** or **false**.

Note that this process of reduction can extend the original WDB by the auxiliary set equations $v_i = \{\ldots\}$ defining those set names v_i participating in V which were not the original set names in the WDB, and, possibly, many others participating in equations for v_i, and so on. Thus, strictly speaking, the reducibility statement (8.1) only partially reflects this process of reduction as the whole WDB extended by the equation $res = q$ can be involved. In the case of distributed WDB, over which some query q should be executed, this process of reduction also tacitly assumes downloading the (remote) WDB files with those required set equations participating in this process.

Implementation of the Δ-language should evidently follow the operational semantics in [61] or in Section 3.3. In this chapter, we will give implementation details on four important Δ-language constructs: separation, quantification, recursion, decoration and transitive closure. Equality (bisimulation) was already discussed in detail. Other cases are sufficiently evident or do not add much to the operational semantics and by this reason are omitted.

8.1.1 Separation construct

In the case of those queries which involve complex subqueries new equations will be created during the evaluation of the subquery (which was conceptually understood as the "splitting" rule; cf. Section 3.3).

Consider the reduction process for Δ-term separate $\{l\!:\!x \in t \mid \varphi(l,x)\}$:

$$res = \{l\!:\!x \in t \mid \varphi(l,x)\} \triangleright\ res = \{l_1\!:\!x_1, ..., l_n\!:\!x_n\}$$

where t is a set name with a flat set equation $t = \{l_1 : x_1, ..., l_m : x_m\}$ in the current version of WDB (possibly extended locally by the query system). In reality t could be a complicated Δ-term, but we may assume that the "splitting" rule from Section 3.3 has already been applied so that we have here just a set name. In fact, $l_1 : x_1, ..., l_n : x_n$ should be a sublist of $l_1\!:\!x_1, ..., l_m\!:\!x_m$ separated by the formula $\varphi(l,x)$ – for simplicity of denotation some initial sublist (so that $n \leq m$). Note that l, x are label and set variables whereas l_i, x_i are label values and set names participating in the current extended version of WDB. (See also the Δ-language syntax in Appendix A.1 on set names, and label and set variables.) The process of reduction is the quite evident iterative procedure,

Separation algorithm:

START with the current version of WDB and the separation term

$$\{l\!:\!x \in t \mid \varphi(l,x)\}$$

where t is set name, and WDB contains flat set equation $t = \{l_1\!:\!x_1, ..., l_m\!:\!x_m\}$.

1. **Extend current version of WDB** by the equation $res = \{l\!:\!x \in T \mid \varphi(l,x)\}$ where res is a new set name.

2. **Create the new (temporary) set equation** $res = \{\}$ (empty set) for the same set name res. (After populating the right-hand side by labelled set names, this equation will replace the above.)

3. **Iterate over the labelled elements** $l_i\!:\!x_i$ of t where $t = \{l_1\!:\!x_1, ..., l_m\!:\!x_m\}$.

(a) Call the corresponding reduction procedure for the Δ-formula $\varphi(l_i, x_i)$,

$$res_i = \varphi(l_i, x_i) \rhd res_i = \ldots,$$

for new set names res_i resulting in the boolean equations res_i = **true** or res_i = **false**.[1]

Does $res = \varphi(l_i, x_i) \rhd res_i$ = **true?**

Yes – Amend the equation for $res = \{\ldots\}$ initiated in the step 2 as $res = \{\ldots, l_i : x_i\}$ by adding the labelled element $l_i : x_i$. Move back to step 3 (iterate over next labelled element, if one exists).

No – Move back to step 3 (iterate over next labelled element, if one exists).

END with the (simplified) set equation $res = \{l_1 : x_1, ..., l_n : x_n\}$ (with res a subset of t).

8.1.2 Quantification

Consider, for example, the reduction process for the quantified formula $\exists l : x \in t.\varphi(l, x)$ where t is (for simplicity) a set name with a flat set equation $t = \{l_1 : x_1, ..., l_m : x_m\}$ (for l_i, x_i label values and set names, like above). It starts by replacing the bounded existential quantifier with the disjunction:

$$res = \exists l : x \in t.\varphi(l, x) \rhd res = \varphi(l_1, x_1) \vee ... \vee \varphi(l_m, x_m) \rhd \ldots.$$

By invoking the "splitting" rule it assumes the recursive subprocesses

$$res_i = \varphi(l_i, x_i) \rhd \ldots$$

(with new boolean names res_i) leading to truth values for res_i from which an appropriate truth value for res can evidently be obtained.

8.1.3 Recursive separation

Consider the recursion query:

[1] As the Δ-language is bounded (quantifiers and other variable binding constructs are bounded by appropriately restricting the range of variables explicitly required by the language syntax) any such reduction process will inevitably halt (in fact, in polynomial time). In the current case either **true** or **false** will be obtained.

$$\mathsf{Rec}\ p.\{l\!:\!x \in t \mid \varphi(x,l,p)\}$$

where, as above, t is considered as set name with a flat set equation $t = \{l_1 : x_1, ..., l_m : x_m\}$ for l_i, x_i label values and set names. To execute it, we should start by adding the set equation to the WDB with the new set name res,

$$res = \mathsf{Rec}\ p.\{l\!:\!x \in t \mid \varphi(x,l,p)\}.$$

The set name res denoting the result of the recursion query should represent a subset of t where only some of $l_i : x_i$ will participate. It is computed iteratively as an increasing sequence p_k of subsets of t:

$$\begin{aligned}
&p_0 = \{\}\ \text{ (empty set)}\\
&p_1 = p_0 \cup \{l\!:\!x \in t \mid \varphi(x,l,p_0)\} \rhd p_1 = P_1\\
&p_2 = p_1 \cup \{l\!:\!x \in t \mid \varphi(x,l,p_1)\} \rhd p_2 = P_2\\
&\qquad \ldots
\end{aligned}$$

This sequence of equations with new set names p_k (in fact, intermediate results) should be generated iteratively, with each new set equation generated after the previous one. Each of these complicated equations is reduced essentially by using the above process of reduction for the ordinary separation construct giving rise to a subset P_k of t. As these subsets are inflating, and t is finite, this process should be halted when $P_k = P_{k+1}$ (stabilisation). Note that checking equality between these sets does not require the computation of bisimulation as each iterative set p_k is an "explicit" subsets of t (elements of the bracket expression P_k are exactly, i.e. not up to bisimulation, some of $l_i : x_i$ from the right-hand side of the equation for t). Now, simplify the initial equation $res = \mathsf{Rec}\ p.\{...\}$ by replacing it with $res = P_k$:

$$res = \mathsf{Rec}\ p.\{l\!:\!x \in t \mid \varphi(x,l,p)\} \rhd res = P_k.$$

Note that the subprocesses of the above process

$$res_{ik} = \varphi(x_i, l_i, p_k) \rhd \ldots$$

(where φ can be quite complicated formula involving complicated subterms) may introduce

new set names with their corresponding set equations. Of course, they should also be considered as the part of the result of this computation (as soon as they are contained in the transitive closure of res). Thus, it has been demonstrated how to resolve the Δ-term recursive separation.

8.1.4 Decoration

Although the decoration operator can be explained sufficiently easily on the intuitive level (see [3] and Section 3.2.2.2), its implementation should be done particularly carefully and precisely. To resolve the query

$$\mathsf{Dec}(g, v)$$

over a WDB with g and v arbitrary set names, i.e. to simplify the equation

$$res = \mathsf{Dec}(g, v) \rhd res = \{\ldots\},$$

let us firstly consider some auxiliary queries which deserve to be included as library query declarations and, most importantly, add an intermediate conceptual level of abstraction in the description of the operational semantics for the decoration operator.

8.1.4.1 Auxiliary (library) queries useful for computing decoration

Let us now define several auxiliary queries dealing with representation of graphs as sets of ordered pairs.

8.1.4.1.1 `Nodes:` Now, consider a set name `g` with the flat[2] WDB-equation

```
g = { ..., l:p, ...}
```

with `l:p` any labelled set name appearing in the right-hand side (which can be a name of an ordered pair or just of an arbitrary set). The (abstract) set values `First(p)` and `Second(p)` are called *g-nodes*[3] so that

$$\texttt{First(p)} \xrightarrow{l} \texttt{Second(p)}$$

serves as an *g-edge*, and therefore the (absolutely arbitrary) set g plays the role of a *graph*. Alternatively, we could ignore those `p` in `g` which are not ordered pairs – the approach adopted

[2] Recall that the query system considers WDB as a flat system of set equations, and all set equations it eventually produces are also flat. (Only at the very last step of outputting the query result will the system produce set equations with reasonably nested right-hand sides.)

[3] Recall that `First(p)` and `Second(p)` are library queries defined in Section 3.4.2.1.3 and Appendix A.3.

below. Note that different set names may denote the same set, in particular, the same g-node, so that we will need to choose canonical g-node names in the algorithm considered below.

The set of g-nodes can be formally defined in Δ as library query declaration

```
set query Nodes (set g) =
    union separate { m : p in g  | call isPair ( p ) }
```

The set `Nodes(g)` (the union of two element sets `p` in `g`) contains exactly all g-nodes, but, strictly speaking, each g-node in this set (being an element of some p in g) has a label `fst` or `snd` and possibly appears twice, under both of these labels. However, this feature (which could be corrected by replacing these labels by the neutral "empty" label `null`) will play no role in the following considerations. On the other hand, preserving this information on the nodes in `Nodes(g)` might be useful in other examples of using this query declaration.

8.1.4.1.2 `Children`: We also need the concept of *g-children* of a node x in a graph g (as a set of ordered pairs), which is essentially the set of all outgoing edges from x in g. This can be defined set theoretically by the following library query declaration (with three occurrences of the `call` keyword omitted to simplify reading):

```
set query Children(set x,set g)=
    collect {l:Second(p)
        where l:p in g
              and ( isPair(p) and First(p)=x )
    }
```

Evidently, if the set `x` is not the value of `First(p)` for some pair `p` as required in this declaration then `Children(x,g)={}` (the empty set).

8.1.4.1.3 `Regroup`: Let us now define the set valued library operation `Regroup(g)` that can reorganise (without losing any essential information) any graph g into something closely similar to the system of set equations represented by this graph. (For simplicity we again omit all `call` keywords.) Pay attention to the use of the label `null` which can be considered here as the "empty" label (some label is formally necessary according to the BNF of the language):

```
set query Regroup(set g)=
          collect {'null':Pair(x, Children(x,g))
          where m:x in Nodes(g)
          }
```

Informally, each pair `Pair(x,Children(x,g))` collected in `Regroup(g)` is considered as *abstractly* representing a set equation, where:

- first element `x` of the pair (understood as the abstract set denoted by `x`) plays the role of a node of `g` or of an abstract set name – the left-hand side of the intended equation, and
- second element, set `Children(x,g)`, plays the role of the right-hand side of this equation – the evident bracket expression enumerating the labelled elements (`g`-nodes) of this set.

It is crucial here that the set of ordered pairs `Regroup(g)` is *functional* in the sense that for each (abstract set) `x` there exist at most one (abstract) pair `Pair(x,c)` in `Regroup(g)` with the first element `x` (and with `c` uniquely defined by `x` as `c=Children(x,g)`). In fact, `Regroup(g)` defines abstractly the correct system of set equations where each abstract set name (a set in `Nodes(g)`) has exactly one (abstract) equation with this name as the left-hand side. The operation `Regroup(g)` will make it easier extracting from `g` the required system of set equations, described in the main algorithm for computing decoration operation below.

8.1.4.1.4 An assumption. *Now, let us assume that the fragment of the Δ-language without decoration operation has already been implemented.* Then we can make calls to the above library queries applied to appropriate set name arguments in a given WDB, such as the set name `g` (representing a set of ordered pairs) in the call `Regroup(g)`. The latter call will be used in the implementation of decoration operator in the next section.

As usually, when executed by the query system, these library operations generate new set names and set equations and add them to the WDB. In particular, considering set names generated by the query system, the result of `Regroup(g)` is, informally, a set of ordered pairs of the form `{'fst':x,'snd':Children_x}` where `x` and `Children_x` (denoted as `c` in the algorithm below) are now set names[4]. Moreover, according to the natural implementation of the declaration for the query `Children(x,g)`, the right-hand side of the equation for each set name `Children_x`,

```
Children_x = { ..., l:y, ... },
```

contains labelled set names (in fact, `g`-node names) `l:y` for all (labelled) g-children of the g-node named by `x`. Note that the algorithm described in the next section operates with these g-node names.

8.1.4.2 Algorithm for computing decoration

We will show how the decoration operation `decorate(g,v)` can be implemented over a given WDB (with `g` and `v` any set names from the WDB) exploiting the above library query

[4] In further detail, when executing the query `Regroup(g)`, a new set name `r` and set equation `r=Regroup(g)` are generated. Then, the implemented reduction process (▷) executing this query will give rise to a flat equation `r={..., 'null':e, ...}` with each set name `e` in the right-hand side having the equation `e={'fst':x,'snd':Children_x}`.

declarations. This can be done as follows:

START with the current version of WDB and the term $\mathsf{Dec}(g, v)$ for a given set names g and v.

1. **Extend current version of WDB** by the equation $res = \mathsf{Dec}(g, v)$ where res is a new set name.

2. **Regroup** `g` **and canonise** `g`**-node names**.

   (a) Call the query `Regroup(g)`. This amounts to simplifying the extended system of set equations WDB + (`r=Regroup(g)`) for `r` a new set name, which results in some new (auxiliary) set names and flat set equations, including the flattened version `r={..., 'null':e, ...}` of `r=Regroup(g)`, and, for each `e` in `r`,
   `e={'fst':x,'snd':c}`, `c={..., l:y, m:z, ...}`.

   (b) Canonise `g`-node names:

      i. Extract `g`-*node names* (all `x`, `y`, `z`, `...`) from the result in (2a),

      ii. Compare which of them, considered as sets, are equal between themselves (bisimilar as set names, represent the same abstract `g`-node).

      iii. For each `g`-node name `u` find its canonical representative `Can_u` as the first in the lexicographical order `g`-node name bisimilar to `u`. (Thus, `u` is bisimilar to `Can_u`. Note that `Can_u` is not a new set name — just one of those extracted in the step 2(b)i.)

      iv. In the resulting set equations in (2a)
      `e={'fst':x,'snd':c}`, `c={..., l:y, m:z, ...}`
      (for each `e` in `r`) replace `g`-node names `x` and `y`,..., respectively, by `Can_x` and `Can_y`,..., thereby transforming these equations to
      `e={'fst':Can_x,'snd':c}`, `c={.., l:Can_y, m:Can_z,..}`, ...
      (The original versions of these equations should be deleted.)

      v. If for another pair of such equations (for `e'` in `r`),
      `e'={'fst':Can_x','snd':c'}`, `c'={..., l':Can_y', ...}`,
      set names `Can_x` and `Can_x'` in `e` and `e'`, respectively, coincide then omit one of these pairs (does not matter which), and repeat this until no such coincidence of canonical node names will exist.

      vi. Eliminate possible repetitions of labelled canonical node names `l:Can_y` in each `c` (which can arise, e.g. due to replacements in (2(b)iv) as `l:Can_y`

can literally coincide with some `m:Can_z` in `c` for different `g`-node names `y` and `z`).

From now on, these `Can_u` serve as *canonical* `g`*-node names*. Only these node names will be used below as uniquely representing `g`-nodes.

3. **Does a canonical `g`-node name bisimilar to `v` exist?** Find a canonical `g`-node name `w` bisimilar to set name `v` (or just coinciding with `v` if `v` is itself a canonical `g`-node name). Two answers are possible:

   No - The required canonical `g`-node name `w` bisimilar to `v` does not exist (and thus `v` can be treated as naming an isolated `g`-node):

   (a) Simplify the equation `res = decorate(g,v)` to `res = {}` (empty set). Then move to **END** of the algorithm.

   Yes - The required canonical `g`-node name `w` does exist (and thus `v` can be treated as naming a proper `g`-node):

   (a) Generate new set equations for duplicated canonical `g`-node names:

      i. For each set name `s` which is a canonical `g`-node name create a new duplicate set name `Dupl_s` (in particular, `Dupl_w`, `Dupl_Can_x`, etc.).

      ii. For the equations
      `e={'fst':Can_x,'snd':c}`, `c={...,l:Can_y,m:Can_z,...}`,
      obtained in (2(b)iv, 2(b)v, 2(b)vi) for each `e` in `r`, extend further the current extension of WDB by new set equations:
      `Dupl_Can_x = {..., l:Dupl_Can_y, m:Dupl_Can_z, ...}`,
      thereby constructing a system of set equations for duplicate names whose graph is isomorphic to the abstract graph `g`.

   In particular, this will add to the WDB the equation for `Dupl_w`:

   ```
   Dupl_w = W
   ```

   with the right-hand side a bracket expression `W` defined as described above (and involving only duplicated canonical `g`-node names).

   (b) Simplify the equation, `res=decorate(g,v)` by replacing it with the (flat) equation

   ```
   res = W.
   ```

   (End of algorithm.)

END with the (simplified) set equation $res = \{l_1 : x_1, ..., l_n : x_n\}$ (and the associated equations for set names in `W`, etc.).

In the case of the query, $res = \mathsf{Dec}(G, V)$ where G and V are Δ-terms and not just set names (as above), the "splitting" rule should be invoked first, which will result in three equations $g = G$, $v = V$ and $res = \mathsf{Dec}(g, v)$ for the new set names g and v. Then these equation should be simplified, in particular, by using the above algorithm for the decoration.

8.1.5 Transitive closure

Let us now consider implementation of the transitive closure operation $\mathsf{TC}(a)$, where a is considered as a set name with the flat equation $a = \{l_1 : x_1, ..., l_m : x_m\}$ for l_i, x_i label values and set names, as the following (recursive) algorithm:

START with the current version of WDB and the transitive closure term $\mathsf{TC}(a)$ where a is set name, and WDB contains flat set equation $a = \{l_1 : x_1, ..., l_m : x_m\}$.

1. **Extend current version of WDB** by the equation $res = \mathsf{TC}(a)$ where res is a new set name.

2. **Replace the original set equation $res = \mathsf{TC}(a)$ by the new (temporary) set equation** $res = \{'null' : a\}$ (singleton set) for the same set name res. (This will be further populated below.)

3. **Find the first labelled element** $m : z$ of $res = \{\ldots, m : z, \ldots\}$ such that $z \not\subseteq res$. (Elements for which $z \subseteq res$ should be marked and put at the end of the current bracket expression for res so that they will not be considered again and again. For efficiency, the bracket expression for res can be organised as a directed "loop" structure with some point of entrance. Each time when $z \subseteq res$ holds at the entrance point then this point in the loop will be marked and the entrance point shifted to the next one to repeat the inclusion test.)

   If it does not exist (the currently observed element and hence all $m : z$ are marked), go to the **END**.

   Else replace the current equation $res = \{\ldots, m : z, \ldots\}$ with the $m : z$ found (at the current entrance point) by

$$res = \{\ldots, m : z, \ldots\} \cup (z \setminus res)$$

   (inserting elements of $z \setminus res$ in the loop immediately after $m : z$, then marking $m : z$ as now $z \subseteq res$ for the extended res and shifting the entrance point from $m : z$ to the next point of so extended loop — the first element in $z \setminus res$).

   (Computing $z \setminus res$ can evidently also use the loop structure of res with marking ignored.)

Repeat 3.

END with the set equation for res**.**

Note that in fact $\mathsf{TC}(a) = \bigcup\{\{a\}, a, \bigcup a, \bigcup\bigcup a, ...\}$.

8.2 Representation of query output

Recall that the implemented query system works internally with (WDB represented as) a flat system of set equations, and produces query results in this flat form. The resulting set equations also use internally generated (local) set names having no mnemonics. It appears that some nesting in the outputted equations might be desirable which would simultaneously eliminate some internal set names by substituting them with bracket expressions. This substitution can be repeated giving rise to possibly deeply nested results. Consider, for example the result of the *restructuring query* from Section 3.5.3 obtained after some such automatic substitutions:

```
Query is well-formed, well-typed and executable

Result = {
  'publication':res2,
  'publication':res0,
  'publication':res1,
  'publication':{
    'type':"Book",
    'refers-to':res1,
    'refers-to':res2
  }
}

res0 = {
  'type':"Paper",
  'author':"Smith",
  'title':"Databases",
  'refers-to':res1
}

res1 = {
  'type':"Paper",
  'type':"Book",
  'author':"Jones",
  'title':"Databases"
}
```

```
res2 = {
  'type':"Paper",
  'refers-to':res0
}

Finished in: 1866 ms (query execution is 1864 ms, and
postprocessing time is 2 ms)

Comment(s):
Double quotation denotes atomic values like "atom" representing
singleton sets "atom" = {'atom':{}}, etc.
```

Note that, in this example further substitutions could be made to eliminate even those few local names `res0`, `res1`, `res2`, so that there would be just one deeply nested equation `result={...}`. However, this would be a rather inconvenient form as set names to be substituted occur several times, and identical subexpressions could be repeated many times making the query result difficult to grasp. Thus, the system makes such suitable nesting to avoid multiple substitutions in the whole system of equations. Additionally, nested bracket expressions like `{Paper:{}}` which imitate atomic values in our approach are replaced, quite naturally, by `"Paper"`. Note that in the later case there may be multiple substitutions and replacements of the same expression. Similarly, set names for the empty set are always replaced by `{}`. In this way query results become sufficiently readable. Lastly, in the case of cycles substitutions could be infinitely repeated. To avoid this, the system should only substitute those set names res_i with the corresponding bracket expression if $res_i \notin \mathsf{TC}(res_i)$ holds (in addition to the other rules for substitutions above). Also, the computation of transitive closure should be restricted to those new set names resulting from the execution of the query, thus, in principle, this can be done quickly on only local set names.

However, any such postprocessing of the query result can sometimes lead to unnatural looking output, for example in the above query result there is some undesirable extra nesting for one of the publications. In other cases (such as showing a graph as a set of ordered pairs) such nesting appears more reasonable. Also atomic values and explicitly shown empty sets `{}` are very natural. Of course it would be better if the user could choose the preferred form, or the result could be optionally visualised as a graph.

Chapter 9

Δ Query Syntax

9.1 Parsing (well-formed queries)

9.1.1 Implemented Δ-language grammar

The syntax of the implemented language was discussed in Chapter 3, with the full syntax appearing in Appendix A.1. The implemented language is described as *Extended Backus-Naur form* (EBNF or, shortened, BNF), defined as a set of production rules, with each production describing one syntactical category represented as a non-terminal. For example, the production rule

```
<query> ::=
    "boolean query" <delta-formula> | "set query" <delta-term>
```

defines the `<query>` syntactical category (also called *non-terminal*) by stipulating in general that a terminal can be substituted by a sequence of *terminals* such as `"boolean query"` and other non-terminals such as `<delta-formula>`. Here the symbol `|` allows to describe alternative productions. (There are also other ways in the BNF to describe more complicated alternations in production rules.) Continuing such substitutions by using production rules for `<delta-formula>`, etc., a sequence consisting only of terminals can be obtained. Further, as terminals are strings of symbols, the final concatenation is also a string of symbols which, properly speaking, is called *well-formed query*, provided it was generated starting from the non-terminal `<query>`. (Quite similarly we can consider well-formed *delta formulas*, *delta terms*, etc.) Thus, the BNF defines how to construct any query in Δ. In fact, each Δ-query, if well-formed, generates a parse tree (by using BNF-forks discussed below) which should be subsequently checked for well-typedness (see Section 9.2).

9.1.2 BNF forking

Firstly a general note on the BNF grammar. Each production rule from the BNF (except some auxiliary ones which can be eliminated as we will see below) can be represented as one, several, or even infinitely many alternative *forks* `F1,F2,...` each having the same label (syntactical category or non-terminal) on the root of the fork. For example, the rule

```
<A> ::= <B><C> | <B><D><E>
```

splits into two rules

```
<A> ::= <B><C>
<A> ::= <B><D><E>,
```

evidently corresponding to two forks with the branching degree two and three, whose roots are labelled by `<A>` and leafs labelled, respectively, as `<B>`, `<C>` and `<B>`, `<D>`, `<E>`. Let us analogously consider the production rule

```
<set constant declaration> ::= "set constant" <set constant>
                               ("be"|"=") <delta term>
```

which generates two unique forks depending on whether `"be"` or `"="` is used – each fork has a branching degree of four.

Thus whole BNF grammar can then be represented as a set of all such forks. In fact, the parse tree of a query is constructed of such forks. However, not all BNF production rules are so simple and literally split into forks as will be discussed below.

9.1.2.1 Recursion by Kleene operators

Recursive BNF rules using repetition by the Kleene star and plus ($\star$ and +) operators generates an infinite set of forks; $\star$ represents zero or more repetitions, and + represents one or more repetitions. For example the following rule represents a sequence of declarations:

```
<declarations> ::= <declaration> ( "," <declaration> )*
```

Each fork has a root labelled by `<declarations>` and any number of leaves labelled by `<declaration>`, separated by the terminal leaves labelled by `","`. Evidently, the branching of these forks have an arbitrary odd degree because of the separator `","` considered formally as a leaf. Analogously the following syntactic categories are also considered:

```
<variables>, <parameters>, <multiple union>, <conjunction>
<disjunction>, <quasi-implication>, <labelled terms>
```

9.1.2.2 Identifier forks

There is further simplification to the BNF forks and to parse trees by eliminating the "intermediate" <identifier> category playing rather an auxiliary role. Thus, we will replace corresponding production rules by those generating infinitely many simple (one child) forks:

```
<boolean query name> ::= ( (A-Z) | (a-z) | (0-9) | "_" | "-" )+
<set query name> ::= ( (A-Z) | (a-z) | (0-9) | "_" | "-" )+

<label variable> ::= ( (A-Z) | (a-z) | (0-9) | "_" | "-" )+
<label constant> ::= ( (A-Z) | (a-z) | (0-9) | "_" | "-" )+

<set variable> ::= ( (A-Z) | (a-z) | (0-9) | "_" | "-" )+
<set constant> ::= ( (A-Z) | (a-z) | (0-9) | "_" | "-" )+
```

There are infinitely many of such identifier forks because there are infinitely many sequences of *alphanumeric characters* (just those characters participating in the identifier forks) which can serve as a leaf label of a fork for each of the above syntactical categories.

Root nodes of these forks of the corresponding nodes in a parse tree are called *Identifier Nodes* (IN). In general, every occurrence of <identifier> in the right-hand sides of production rules in BNF is replaced by:

```
( (A-Z) | (a-z) | (0-9) | "_" | "-" )+
```

There is, however, restrictions on these alphanumeric strings: they should not coincide with keywords of Δ language.

9.1.2.3 Set name forks

Let us recall the production rules related with *full set names* represented by the syntactical category <set name>. This important category, including some additional auxiliary productions, appears as follows:

```
<set name> ::=     <URI> "#" <simple set name>

<URI> ::=          ( <web prefix> | <local prefix> ) <file path>

<web prefix> ::=   "http://" <host> "/" [ "~" <identifier> "/" ]
<local prefix> ::= "file://" ( (A-Z) | (a-z) ) ":/"

<host> ::=         <identifier> [ "." <host> ]
```

```
<file path> ::=    <identifier> ( "/" <file path> | <extension> )
<extension> ::=    ".xml"

<simple set name> ::= <identifier>
<identifier> ::= ( (A-Z) | (a-z) | (0-9) | "_" | "-" )+
```

Here all the syntactical categories, besides `<set name>`, play an auxiliary role. Therefore, by composing them, all these production rules will produce two kind of one child forks for set names

```
<set name> ::= "http://... " "#" ( (A-Z) | (a-z) | (0-9) | "_" | "-" )+
```

or

```
<set name> ::= "file://... " "#" ( (A-Z) | (a-z) | (0-9) | "_" | "-" )+
```

Here `"http://..."` and `"file://..."` represent any string of symbols allowed by the `<URI>` production rule. Therefore, the production rule `<set name>` generates an infinite number of (one child) forks with the root `<set name>` and the leaf a string of characters as defined in the above productions.

We will not consider other cases of defining BNF forks relying on the readers' intuition which should be based on the above examples. Assertions 1-3 from the next section should summarise and give more understanding on the way which BNF forks are defined.

9.1.2.4 Assertions on BNF forks

After defining the set of forks of the BNF, we can make the following assertions.

Assertion 1. *Only Identifier Nodes (IN) can have just one child leaf labelled by a sequence of alphanumeric characters.*

Proof. Inspection of the whole BNF (and the definitions above) show that only IN can have just one child leaf labelled by a sequence of alphanumeric characters. □

Note that `<set name>` forks, although one child, have leafs containing non-alphanumeric characters `":"`, `"/"` and `"#"`.

Assertion 2. *In fact, parsing of any given query generates a corresponding query parse tree constructed from these forks connected in the evident way. Here it is assumed that all keywords like "forall", "let", etc are included in the parse tree as terminals (except they are not allowed to be leafs of identifier forks).*

Assertion 3 (Uniqueness of forks[1])**.** *Two different forks can have coinciding leaf labels (in the natural order) only if each of them is an identifier fork (see above). That is, if one of the two forks F1 and F2 is not an identifier fork and both forks have the same leaves then (their roots coincide and) F1 = F2. Or equivalently, the syntactic category of any fork, except for identifier forks, can be determined according to the syntactic categories of its children.*

Proof. We should check all possible cases. Assuming that two forks F1 and F2 have the same leaves and one of them has the root labelled not as identifier fork, show that F1 = F2.

Example: If F1 or F2 has the root `<quantified formula>` then both have the same first leaf e.g. `<forall>` (or `<exists>`). Then, according to the BNF, another fork must also have the root `<quantified formula>` and therefore F1 = F2, as required.

Example: If F1 or F2 has the root `<forall>` then both have the same first leaf `"forall"` and the leaf `"in"` (or `"<-"`). Inspection of all BNF forks shows that any fork containing both these leafs must have the root `<forall>`. Therefore F1 = F2.

Example: If F1 or F2 has the root `<union>` then both have the same first leaf `"union"` (or `"U"`) and second leaf `<delta-term>`. Inspection of all BNF forks shows that any fork containing both these leafs must have the root `<union>`. Therefore F1 = F2.

All other cases follow as above. □

Note 4. Despite this Assertion which means a kind of unambiguity of parsing (actually only a conditional and partial unambiguity) we will see in Section 9.1.4 that parsing according to the BNF of Δ is actually quite ambiguous. This means that the same query can have parse trees of the same form, but with different labelling of nodes by syntactical categories. Later we will consider contextual analysis algorithm dealing with typing which will resolve this kind of ambiguity.

9.1.3 Query parsing

The parser for the BNF syntax of the language Delta can easily be implemented which can transform any query q into parse tree. The process of parsing q involves matching of BNF production rules (represented rather in the form of forks defined above) starting at the root production rule for `<top level command>` until all possibilities are exhausted. The output of parsing the query q is the query parse tree qt.

[1] This assertion will be used in the syntactical category renaming algorithm in Section 9.2.3.2

During the process of parsing, successful matching of production rules creates new nodes in the parse tree connected by fork edges from the previous node, except for the root production rule which itself has no parent node. Successful matching of terminals creates new nodes labelled by the sequence of matched characters.

9.1.3.1 Example query parse tree

Let us consider the simple example of query

```
boolean query
  let label constant l='Robert'
  in l='Rob*'
endlet;
```

and the corresponding query parse tree,

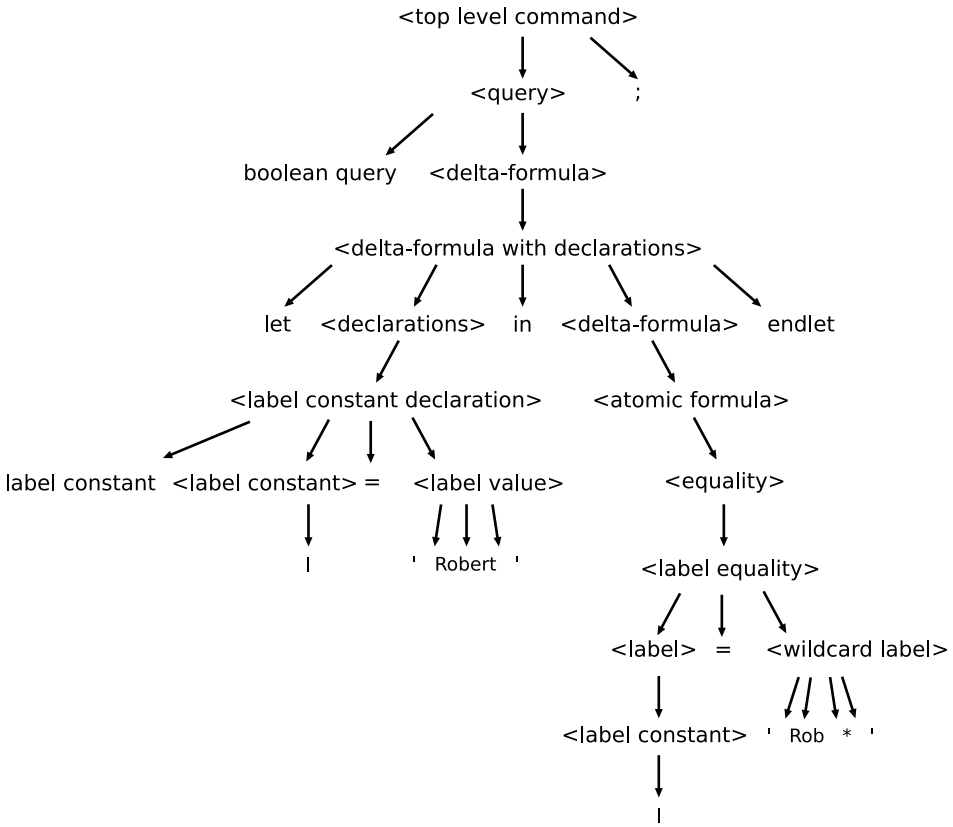


Figure 9.1: Example parse tree

Strictly speaking, some parts of this parse tree are omitted for brevity. Say, according to Section 9.1.2.1, between `<declarations>` and `<label constant declaration>` we should have a tree node `<declaration>`.

9.1.3.2 Aims of query parsing

Well-formedness of any query is determined according to the rules of the BNF grammar. However, when all possibilities for matching productions are unsuccessfully exhausted in any attempt to construct a parse tree then the query is considered as non-well-formed with appropriate error messages outputted.

Moreover, to further aid contextual analysis (see Section 9.2) the parser should output, in addition to the parse tree of the query, the list of all Identifier Nodes (see Section 9.1.2.2) in the parse tree labelled by:

```
<boolean query name>, <set query name>, <label variable>,
<label constant>, <set variable>, <set constant>.
```

9.1.4 Parsing ambiguities

The syntax of the implemented Δ-language (expressed as BNF) is intended for any user to understand the constructs of Δ, and how to write valid Δ-queries – well-formed and well-typed. However, the implemented parser alone cannot guarantee well-typedness of queries. Note that, well-typedness is checked by the contextual analysis algorithms described later in Section 9.2.

The problem is that the grammar of our implemented Δ-language is ambiguous concerning types as we briefly commented this in Note 4 above. Thus, the typing of identifiers, say as label constant or variable, or set constant or variable, etc., is actually decided from the context. For example, let us consider the equality query:

```
boolean query a=b;
```

Parsing of this query could realise two unique parse trees, where the statement `a=b` represents either `<label equality>` or `<set equality>`. Thus, the syntactical category of this statement depends wholly on the interpretation of the identifiers `a` and `b` as either, label constants or variables, or set constants or variables, respectively. The parse tree presented above in Figure 9.1 is also not unique one because the syntactic category `<label constant>` under `<label>` could be formally replaced according to syntax by `<label variable>`, however, intuitively contradicting the label constant declaration `let label constant l = ...`.

Furthermore, let us even strengthen the above example,

```
boolean query let
    label constant l='Robert',
    label constant m='John'
```

```
  in
    l=m endlet;
```

where the statement `l=m` intuitively represents the syntactic category `<label equality>` because according to the context the identifiers `l` and `m` are label constants. However, the BNF formally allows that `<label equality>` could be replaced with `<set equality>` and `l` and `m` are are taken as `<delta-term>`s, independently of the declarations that `l` and `m` are both label constants. Even the following query can be formally parsed, i.e. is well-formed,

```
boolean query let
    label constant l='Robert',
    set constant m={}
  in
    l=m endlet;
```

despite being evidently non-well-typed by equating label with set.

Therefore, the syntax (expressed as BNF) alone is insufficient and requires guessing which rule to apply to make the parse tree (and to guarantee that the parsed query is) well-typed. Therefore, such guesses by the parser should be subsequently checked, to ensure no contradictions with the actual typing of identifiers. Moreover, the syntactic categories of all nodes, not just IN, should be checked and possibly renamed (according to the grammar) without changing the structure of the parse tree. Such renaming is done by the *contextual analysis algorithm*, detailed in Section 9.2, whose role is to ensure query well-typedness and eliminate potential ambiguities, as above.

9.1.5 Grammar classification

Note that the syntax of Δ-query language, fully presented as BNF in Appendix A.1, can be classified as *context-free grammar* according to Chomsky's definitions of formal languages. Taking the definition from the textbook about parsing [75], all production rules of a context free grammar have the form:

$$A \longrightarrow \gamma$$

where A represents a unique non-terminal, and γ represents an ordered list of terminals and/or non-terminals (possibly empty). Context free grammars are those where each non-terminal A can be transformed by a production rule into corresponding γ without any additional criteria of context. Our grammar satisfies this property and therefore cannot grasp contexts which are necessary for correct typing of queries. Thus, an additional contextual analysis algorithm working jointly with the parser is required which we discuss in the following section.

9.2 Contextual analysis (well-typed queries)

9.2.1 Aim of contextual analysis

The aim of contextual analysis is to determine whether every *identifier occurrence* in a query q is declared[2], thereby having type, and whether the whole query is well-typed (all types are coherent). Each identifier occurrence should be appropriately typed as either: *set constant* or *variable*, *label constant* or *variable* or *query name* of some type[3]. Note that query names can have more complicated types than variables or constants,

$$(type_1, type_2, ..., type_n \longrightarrow type) \tag{9.1}$$

where each participating $type_i$ is either *set* or *label*, and $type$ after the arrow is either *set* or *boolean*[4]. Each $type_i$ is the expected type of i-th parameter of the query name q, and n is the required number of parameters – according to the declaration of this query name. From this type it should be already clear that the identifier q is a (set or boolean) query name, how many arguments it has, and the typing of each argument.

Furthermore, an identifier occurrence is considered declared if it is contained within the scope of an appropriate identifier declaration, and well-typed if both the identifier occurrence and identifier declaration have the same types. Moreover, for query to be well-typed, coherence of typing (for equalities, as in the examples above, membership statements and query calls) should be additionally required.

9.2.1.1 Strategies for computing contextual analysis

In principle there are two possible algorithms for performing contextual analysis of any query q, both algorithms are named after the way in which they walk the parse tree of q:

- **Top-down contextual analysis** – The parse tree is walked in breadth first manner starting at the root node, creating a list of the identifier declarations (called the context) which is used to check that all other identifier occurrences are closed and well-typed according to these declarations.

[2] An identifier occurrence in some expression e (not necessary a full-fledged query; e can be a fragment of a query q) which is non-declared inside e can also be called *free* in e, whereas those correctly declared inside e identifier occurrences are called *closed*. Therefore the terms "declared" and "closed", and "non-declared" and "free", have the same meaning. (This agreement on terminology is, however, non-traditional in the particular case of (set or label) constants for which it is more habitual to use the terms "declared" or "non-declared" instead of "closed" or "free".) We assume that each full-fledged query q must be closed in this sense (all its identifiers must be declared inside q).

[3] To simplify terminology, we consider *variable* or *constant* or *query name* as typing information of some identifier, alongside the proper types *set* or *label* or *boolean* or the complex type (9.1).

[4] Note that, we formally have no queries or query names in Δ of the type *label*. However, label values can be represented in the same way as atomic values, i.e. as singleton sets of the form $\{l : \emptyset\}$.

- **Bottom-up contextual analysis** – Walking of the parse tree starts from any identifier occurrence leaf i[5] ascending up the corresponding branch of the parse tree, searching for an identifier declaration which declares i[6]. The existence of a corresponding identifier declaration indicates that the identifier occurrence is declared. Moreover, the real types of all such i can be extracted from the corresponding declarations and compared with syntactical categories of these nodes i in the parse tree. In the case of coherence, the parse tree and hence the query is considered *well-typed*. Otherwise, syntactical categories of the parse tree nodes could be possibly corrected by (another bottom-up procedure of) renaming syntactical categories of some non-leaf nodes by the iterative algorithm described below in Section 9.2.3. If such a renaming is successful – giving rise to a correct parse tree according to both the BNF and the typing, then the resulting version of tree and the original query are also considered *well-typed*, otherwise *non-well-typed*.

9.2.2 Some useful definitions

Definition 1 (Identifier Node)**.** *Identifier Nodes* (IN) were introduced in Section 9.1.2.2, as those nodes in the parse tree labelled by one of the following syntactic categories:

```
<boolean query name>, <set query name>, <label variable>,
<label constant>, <set variable>, <set constant>.
```

Additionally, let us define *Identifier Node Name* (INN) as string of symbols labelling the unique child (in fact, a leaf called above as i) of the corresponding IN fork in the parse tree.

Definition 2 (Binder Node)**.** *Binder (or binding) Nodes* (BN) are those nodes in the parse tree labelled by one of the following syntactic categories:

```
<delta-term with declarations>, <delta-formula with declarations>,
<collect>, <separate>, <recursion>, <quantified formula>.
```

Binder nodes can have appropriate declarations like `"let..."`, `"forall..."`, `"exists..."`, etc., as described in Definition 3, and thereby can *bind* identifier occurrences (or IN).

Definition 3 (Identifier Declaration Node)**.** Following from Definition 2 those declarations belonging to BN are called *identifier declarations nodes (IDN) of a BN* and defined as follows.

- For BNs `<delta-formula with declarations>` with `"let"` declaration(s), and `<delta-term with declarations>` with `"let"` declaration(s) the IDNs are:

[5] For example, the second leaf labelled by the identifier l in Fig. 9.1 above

[6] In Fig. 9.1 above the corresponding node would be `<delta-formula with declarations>` having the declaration of the label constant l under it. Note that quantifiers and other quantifier-like constructs, called binders (see Section 9.2.2), are also considered as identifier declarations.

  - <set constant declaration> grandchild of <declarations>,
  - <label constant declaration> grandchild of <declarations>,
  - <set query declaration> grandchild of <declarations>, and
  - <boolean query declaration> grandchild of <declarations>.

- For BNs <separate> and <collect> the IDNs are:
  - <label variable> grandchild of <variable pair>, and
  - <set variable> grandchild of <variable pair>.

- For BN <recursion> the IDNs are:
  - <set variable> child of <recursion>,
  - <label variable> grandchild of <variable pair>, and
  - <set variable> grandchild of <variable pair>.

- For BN <quantified formula> the IDNs are:
  - <label variable> grandchild of <variable pair>, and
  - <set variable> grandchild of <variable pair>.

For example, Figure 9.2 depicts a fragment of a query parse tree, where the root node <separate> is a BN and the corresponding IDN nodes (described above) can be found by walking the paths from the <separate> node,

<variable pair> → <variable pair label> → <label variable>

<variable pair> → <variable pair set> → <set variable>

All other cases follow as the above. Note that there may be many IDNs of a given BN. Any IDN declares one or more identifiers (IN) each of which has its name as a string of symbols (the leaf under IN).

Definition 4 (Bounding Term or Formula or Label Value Node)**.**

(a) Following from Definition 2, the *bounding* term or formula or label value nodes (BTFLVN) of a BN

```
<collect>
<separate>
<recursion>
<quantified formula>
<delta-term with declarations>
<delta-formula with declarations>
```

is defined, respectively, as

- a unique <delta-term> child of:
  - <collect> or <separate> or <recursion> or
  - <forall> child of <quantified formula> or
  - <exists> child of <quantified formula> or
  - any <set constant declaration> grandchild of <delta-term with declarations> or <delta-formula with declarations> or
  - any <set query declaration> grandchild of <delta-term with declarations> or <delta-formula with declarations>, or
- a unique <label value> child of:
  - any <label constant declaration> grandchild of <delta-term with declarations> or <delta-formula with declarations> or
- a unique <delta-formula> child of:
  - any <boolean query declaration> child of <delta-term with declarations> or <delta-formula with declarations>.

(b) Each BTFLVN of a BN restricts the range of the value of some INs (variables, constants or query names) which BN binds[7] and which we also call *bounded or restricted IN(s) by the BTFLVN*[8]. These INs are defined as follows:

- In the case of BNs <collect>, <separate>, <recursion> and <quantified formula>, the bounded INs are respectively <label variable> and <set variable> grandchildren of <variable pair>.
- Additionally, in the case of BN <recursion> one more bounded IN is its immediate <set variable> child.
- In the case of BNs <delta-formula with declarations> or <delta-term with declarations>, the bounded IN is either the declared

[7] Which was briefly hinted in the Definition 2

[8] Moreover, the IN bounded by BTFLVN should not be free in the BTFLVN (i.e., if present in the BTFLVN, it should be declared inside this BTFLVN) as we will discuss later as one of the conditions to be checked by contextual analysis algorithm. This is the reason why we need Definition 4.

<set constant> or <label constant>, or <set query name>, or <boolean query name>.

For example, Figure 9.2 depicts the query parse tree for an expression e (fragment of a query q), where the root node <recursion> is a BN and the corresponding BTFLVN and the bounded INs can be found by walking the paths,

<recursion> → <delta-term> (BTFLVN)

<recursion> → <set variable> (IN)

<recursion> → <variable pair> →
<variable pair label> → <label variable> (IN)

<recursion> → <variable pair> →
<variable pair term> → <set variable> (IN)

whereas <label variable> (l) and <set variable> (x) are INs bounded by this <delta-term> (BTFLVN). Additional (recursion) <set variable> (r) is IN also bounded by <delta-term> (BTFLVN).

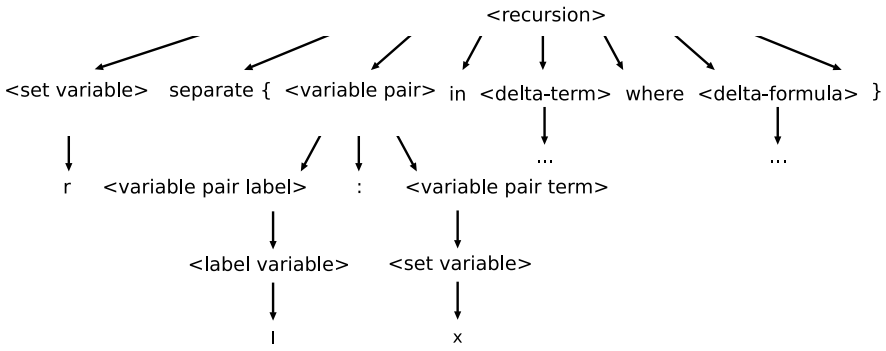


Figure 9.2: Fragment of a query parse tree

9.2.3 Bottom-up contextual analysis in detail

As stated in the brief description in Section 9.2.1, contextual analysis should check that the given well-formed query (according to the parser) is also well-typed. To this end, the bottom-up contextual analysis algorithm, first of all, iteratively searches for the nearest identifier declaration for each identifier occurrence, i.e. each IN in the parse tree. We assume that before starting contextual analysis the parser generates a list of all INs (not those INs of the declarations in IDNs) along with their currently chosen typing (immediately seen from syntactical categories of these INs, say, <set variable>, etc.) during the parsing process. The parser outputs this list if the query is well-formed.

9.2.3.1 Identifier declaration search (IDS) algorithm

Single iteration of the search for the nearest identifier declaration of an IN is determined by the *Identifier Declaration Search* (IDS) algorithm. The inputs to this algorithm is any qt (query parse tree) and some IN in qt. The output of the IDS algorithm is the ordered triple $< BN, IDN, IN >$ (if the required one exists at all) consisting of: BN (Binding Node), IDN (Identifier Declaration Node) and the given IN.

Note that, IDN contains typing information of the declared identifier (including the information whether it is a constant or variable, or a query name – also a kind of typing information). In fact, the IDN is recoverable from BN and IN in the parse tree, however, it is convenient to have IDN included in the triple obtained during this process.

Identifier Search Algorithm $IDS(qt, IN)$:

START with a given IN belonging to qt.

1. **Make this node (IN) the *current node*.**
2. **Ascend from the *current node* traversing up** qt to its unique parent node, making this node the *current node*.
3. **Is the *current node* a BN?**

   No – Move to step 4.

   Yes – Iterate from right to left through IDNs of the BN, searching for the first[9] suitable candidate identifier declaration whose declared identifier has the same name (INN) as the given IN. If a suitable candidate IDN exists then construct the ordered triple $< BN, IDN, IN >$ (end of algorithm), *otherwise* move to step 4.
4. **Is the *current node* the root node of** qt**?**

   Yes – No suitable candidate identifier declaration could be found, and therefore, the IN is non-declared. Output ordered triple $< NULL, NULL, IN >$ (end of algorithm).

   No – Continue searching for a suitable identifier declaration by moving to step 2.

   END with the ordered triple $< BN, IDN, IN >$ if a suitable identifier declaration exists, otherwise with $< NULL, NULL, IN >$.

[9] Formally, it is not forbidden that the same identifier name could be multiply declared even in the same binder, but only the right most one is that which binds the IN considered and which assigns a type to IN.

The IDS algorithm should iteratively generate the triples as above for all INs (actually, for those identifier occurrences not in a declaration) of the given parse tree qt. If all these are non-null triples then the query q is considered as *closed* (yet possibly not well-typed). Thus, any closed query q has all INs declared with preliminary typing according to the declarations (IDN) from the corresponding triples. For non-closed query an error message should be generated by the implementation saying that the query has non-declared identifiers. Moreover, any closed query q and its parse tree qt are considered also well-typed if all identifiers have coherent typing both in respect to their corresponding declarations and syntactical categories of the parse tree qt. More precisely, this means that:

1. Syntactical categories of IN (e.g. `<set variable>` or `<boolean query name>`, etc.) should be the same as declared in IDN (in corresponding triple), and

2. Types of participating parameters in query calls should agree with types discovered from IDNs declaring corresponding query names.

If these two clauses do hold then in other nodes the BNF itself supports correct typing and/or syntactical categories (such as `<set equality>` vs. `<label equality>`, etc.). Otherwise, an appropriate renaming of syntactical categories of the nodes in qt should be tried (as detailed in the next section), based on the initial partial correcting only the discrepancies in the clauses (1) and (2), with the aim to recover well-typed version of qt and conclude that the query q is *well-typed*. If such a renaming is impossible, then q is considered as *non-well-typed*.

9.2.3.2 Syntactic category renaming (SCR) algorithm

It is required that renaming should lead to a correct parse tree. This means that the *syntactic category renaming* (SCR) algorithm,

- takes a parse tree with some already correctly renamed nodes (such as INs, by removing the discrepancies mentioned above, and may be some other nodes as we will see below) and formally marked as "correct", and

- if necessary, attempts to rename other nodes ensuring that the parse tree remains faithful to the Δ-language BNF syntax (well-formed).

Thus, the *input* is a given parse tree qt with some (non-leaf) labels *already relabelled*[10] and additionally marked as "correct", with the output being either: (i) parse tree with all other

[10] Note that, INs are formally non-leaf nodes, although neighboring to leafs. As we will see below in Section 9.2.3.3, not only INs should be initially relabelled in the input parse tree. These may be also query call `<parameters>` which, unlike INs, may be far away from leaves in the parse tree.

nodes successfully relabelled (q is well-formed), or (ii) an error state (qt is inconsistent with the Δ-language syntax, even after further relabelling).

The procedure of relabelling starts from the leafs of the parse tree, and, while going bottom-up along the tree relabels according to the Δ-language BNF syntax (if necessary) those nodes which have not already been relabelled. Newly relabelled nodes are additionally marked as "correct", and visited nodes are marked also as "seen" as described formally below. At each stage of the computation some nodes are already marked by this procedure as "correct", and only a node N can be relabelled and then also marked as "correct" and "seen" which, (i) has not yet marked as "seen" (although probably marked as "correct" by the input marking), and (ii) all its children, $Children(N)$, have already marked as both "seen" and "correct".

Syntactical renaming algorithm $SCR(qt)$:

START with parse tree qt.

1. **Initially mark some nodes as "seen" and "correct".** Mark all leaf nodes, INs, IDNs and `<set name>` nodes both as "seen" and "correct"[11].

   Note: Syntactic categories of "correct" nodes will not be renamed by this algorithm. Furthermore, `<set name>` nodes should not be renamed (and thus, these are initial marked as "correct") as they evidently have unambiguous type *set* and definitely require no renaming.

2. **Find any node suitable for correcting.** Find node N, which is not marked as "seen", and whose all children are marked both as "correct" and "seen" (giving rise to a fork $N \longrightarrow Children(N)$ in qt). Does the required N exist in qt?

   No – Therefore, by induction, all nodes in the tree are already marked as "correct", (end of algorithm).

   Yes - Check and (if necessary, and possible) rename according to BNF the syntactical category of N:

   (a) **Find a suitable fork F in the BNF that matches the children of N.** Find a fork F from the BNF whose leaves match with $Children(N)$. As N is not an identifier node, it follows from Assertion 3 from Section 9.1.2 that there can exists only one such fork F, if any.

[11] In fact, as we discussed above, INs and query call `<parameters>` are already marked as correct in the input parse tree qt.

If the required fork F does not exist in the BNF, output error message "query is not well-typed" indicating the statement in the query q corresponding to the node N which "cannot be properly typed", and halt (end of algorithm).

Otherwise, if F exists, move to step 2b or 2c depending on whether N is already marked as "correct" or not.

Note: The term 'matching' means that the branching degree should be the same and the matching children nodes (in the natural order) have the same labels. The labels of N and the root of F are not required to coincide for matching to be successful.

(b) ***N* is not marked as "correct"** - relabel syntactical category of N exactly as the root of F, mark N as "correct" and "seen", and move to step 2.

(c) ***N* is marked as "correct"** - if the label of the root of F *coincides* with the label on N then mark N also as "seen" and move to step 2.

However, if the label of the root of F *differs* from the label on N, generate the error message "query is not well-typed; conflicts with the expected syntax" and indicate which syntactic category name (and corresponding place in the query) requires renaming. (End of algorithm.)

END with either correctly relabelled parse tree, or an appropriate error state.

The successful result of this algorithm would give us a full guarantee that the resulting relabelled tree is still the correct parse tree of the given query which is therefore well-formed. Most importantly[12], it will also guarantee that the query is well-typed: parse tree labelling is fully coherent, both with the typing and all other details in declarations of identifiers (such as to be a constant or variable or query name).

9.2.3.3 Contextual analysis algorithm

The complete algorithm for bottom-up contextual analysis consists of the following (macro) steps. The input is any query parse tree qt and the list of INs (both obtained from the parser). The output being either: (i) correctly relabelled query parse tree (q is well-typed), or (ii) an error message (q is non-well-typed).

Contextual analysis algorithm $CA(qt, \text{the list of INs})$**:**

START with the list of INs of the query parse tree qt**.**

1. **Find suitable candidate declaration (BN and IDN) for each identifier occurrence (each IN).** That is, iterate over the given list of INs calling IDS algorithm for each IN

[12] also, taking into account appropriate renaming of syntactical categories of query call `<parameters>` considered below

(see Section 9.2.3.1). The result of these identifier declaration searches is the list of declaration triples for all INs.

For those INs for which the algorithm IDS outputs $< NULL, NULL, IN >$ the corresponding error messages "identifier non-declared" should be outputted concerning all such identifier occurrences in the query q and additionally that the "query is not well typed".

If IDS outputted NULL triple for some IN then end of algorithm; otherwise move to step 2.

2. **Relabel syntactical categories of some parse tree nodes according to step 1.**

   (a) **Relabel syntactical categories of identifier occurrences.** Labels of nodes (i.e. syntactical categories) generated by the parser contain the preliminary information on the typing (assigned by the parser and possible contradicting the actual type). The real typing of any IN and, in fact, the real syntactical categories (the node labels) of the INs can be correctly determined using the IDN from the declaration triple of IN. The parse tree labelling for these INs should be updated accordingly (may be vacuously if the given IN, in fact, does not need updating according to the IDN) with marking these nodes as "correct". This can be done straightforwardly for all INs (in particular for query names to be discussed below). Thus after relabelling, all INs will be actually marked as "correct".

   (b) **Relabel syntactical categories of query call parameters**[13]**.** In the case of INs which are query names in query calls some additional renaming of some (possibly) non-IN nodes (query parameters) is required as described below.

   If we have a query call $q(t_1, ..., t_n)$ with the query name q of the type

   $$(type_1, type_2, ..., type_m \longrightarrow type)$$

   obtained from the appropriate IDN by the algorithm IDS (where all participating $type_i$ are *set* or *label*, and the type after arrow is *set* or *boolean*) then we should:

   i. Check whether $m = n$; if not, the query is not well-typed, and the algorithm should halt with an appropriate error message.

[13] In some cases similar to query parameters the parser already assumes some typing. For example, in the membership statement $l : a \in b$ the syntactical categories of l, a, b must be, respectively, <label>, <delta-term> and <delta-term>, according to the BNF. In the case of equality $a = b$, the expressions a and b must be of the same type according to BNF, although the choice of type is ambiguous as shown by those examples in Section 9.1.4. But, the case of query call parameters requires our special attention in the currently described algorithm.

ii. If $m = n$, rename (possibly vacuously) syntactical categories of parameter nodes t_i (<delta-term> or <label>) according to the types $type_i$ (*set* or *label*), and mark them as "correct".

3. **Relabel syntactical categories of all other parse tree nodes.** Apply SCR algorithm (Section 9.2.3.2) to the resulting partially relabelled parse tree. Thereby other nodes of the parse tree will also be potentially renamed.

   (a) **Were all other nodes successfully renamed?**

   Yes - If the SCR algorithm renamed and marked all nodes as "correct", then move to Step 4 to check for additional requirement (that query is properly "bounded").

   No - Parsing agreeing with typing is impossible, and appropriate error messages from SCR algorithm should be outputted. End of algorithm.

4. **Additional requirements on bounding terms or formulas (BTFLVNs)**

   (a) **Check that (the names of) bounded identifiers (INs) of: <separate>, <recursion>, <collect>, <delta-formula with declarations>, <delta-term with declarations>, and <quantified formula> have no non-declared occurrences inside the bounding term or formula (BTFLVN)**.

   For convenient implementation of this clause we assume additionally that the parser also generates for each bounding term or formula (BTFLVN) the sub-list of INs (from the list of all INs generated by the parser) lying *under* BTFLVN in qt[14]. In other words, these are some of the identifiers occurring in the query q. This can be represented as lists (for each BTFLVN) of the form:

   $$< BTFLVN, IN_1, \ldots, IN_k > .$$

   Using the list of these IN_i under the given BTFLVN and the declaration triples of the form $< BN, IDN, IN >$ generated by the IDS algorithm, it should be checked that each IN_i from the above list whose name coincides with the name of some bounded IN by the given BTFLVN (see Definition 4 (b)) is declared in this BTFLVN. The latter means that such an IN has its own binding node BN (from the appropriate unique triple), and this BN lies under or coincides with the given BTFLVN. This should hold for each BTFLVN in qt. Otherwise contextual analysis should be aborted with corresponding error message.

[14] If BTFLVN is LVN – a label value node – then this list is, of course, empty.

In particular, in the case or recursion, we should check that the recursion binding set variable, as well as variables from the binding variable pair, do not occur free in the bounding term. Also, each query name should not occur free in the defining term or formula, and set constant should not occur free (non-declared) in the defining term, etc. However, in the case of set constants and query names we need to add the following additional requirements.

(b) Check that for each <set constant declaration> the defining <delta-term> has all of its set or label *variables* declared within this term. That is, intuitively, <delta-term> defining a set constant should have a constant value. However, constants and query names inside this <delta-term> may be declared in the query outside this term.

To do this, use the list of INs of *variables* lying under the node <delta-term> of <set constant declaration> and the identifier declaration triples of the form $< BN, IDN, IN >$ generated by the above IDS algorithm, and check that each BN of such a variable IN lies in the <delta-term> node subtree. Otherwise, such a variable IN of the <delta-term> is considered as free, and the contextual analysis should be aborted with the corresponding error message.

(c) Check that for each <set query declaration> the defining <delta-term> has all its set or label *variables* declared (quantified, etc.) either inside this term or in the given <set query declaration> as <variables> parameters of the declared query. Constants, and query names inside this <delta-term> may be declared in the query outside this term. Quite similarly check for each <boolean query declaration> and corresponding <delta-formula>.

(d) The remaining check that <label constant declaration> uses closed <label value> is evidently vacuous, as actually there is nothing to check.

END with a correctly relabelled and well-typed and properly bounded parse tree ("query is well-formed and well-typed"), or a partially relabelled parse tree plus additional error messages ("query is well-formed but not well-typed", etc.).

9.2.4 Extension of contextual analysis to support libraries

That the library declarations are well-formed and well-typed can be checked by reducing these declarations to the ordinary queries, as it was shown in Section 3.4.2.2, and applying parsing and contextual analysis algorithm described above to the resulting query.

Chapter 10

XML Representation of Web-like Databases (XML-WDB Format)

10.1 Represention of WDB by graph or set equations

As we discussed in Chapter 2 the (hyper)set theoretic approach [40, 41, 43, 56, 57, 61] to WDB is based on the concept of hereditary finite sets or, more generally, hyperset theory [3, 5]. Such semi-structured data is represented as abstract sets (of sets of sets, etc.) with the possibility for membership relation to form cycles.

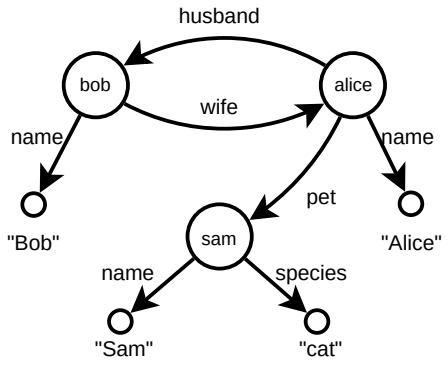


Figure 10.1: Example WDB representing a fictitious family

For visualisation purposes hyperset databases are represented as *graphs* (see Figure 10.1) where nodes correspond to set names and labelled edges to membership relation. When considering implementation (and also intuitively from the set theoretic view) it is far more appropriate to represent WDB as *system of set equations*. Each set equation consists of a *set name* equated to a *bracket expression*; *labelled elements* of such sets may be either atomic values, nested bracket expressions, or set names (described in some other equations). For example, system of *flat* set

equations corresponding to the WDB graph in Figure 10.1 looks as follows:

```
bob   = { name:"Bob", wife:alice }
alice = { name:"Alice", husband:bob, pet:sam }
sam   = { name:"Sam", species:"cat" }
```

or, equivalently, with the *nesting* allowed:

```
bob   = { name:"Bob", wife:alice }
alice = { name:"Alice", husband:bob,
          pet:{name:"Sam", species:"cat"} }
```

In particular, this demonstrates that the specific form of set names (e.g. `bob`, `alice`, `sam`) however helpful intuitively are formally not important. They can always be renamed (say by numbered "object identities" e.g. `&23`, etc.) or substituted as above. In general, the role of set names in any system of set equations depends on its position. Those set names occurrences on the left-hand side of set equation (simple set names) are also called *defined* set names, whereas, all other set name occurrences are called *referenced* set names. Each referenced set name should be defined somewhere in the system, and only once.

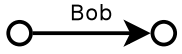
The implemented query system considers WDB as systems of flat set equations (without any nesting). As described below, WDB is represented practically as a system of XML files each containing a fragment of the whole system of set equations of the WDB, which proves convenient. From the perspective of any database designer, the informational content of WDB is carried by:

- Labels on WDB-graph edges e.g. `name`, `wife`, `husband`, etc.
- Atomic data (see Note 5) on leaves e.g. `"Bob"`, `"Alice"`, etc.
- Graph structure or, respectively, set-element nesting.

Note 5 (Atomic data)**.** Atomic data is, in fact, treated as singleton sets consisting of a labelled empty set or, equivalently, as labels on additional leaf edges in the WDB graph. For example, the atomic value `"Bob"` from the above example is formally represented as

```
{Bob:{}}
```

or, respectively, as the labelled edge with the target node being a leaf,



For example, taking into account the above description, the corresponding system of (almost) flat set equations (with atomic values simulated as labelled empty sets) representing the WDB graph depicted in Figure 10.1 should actually be:

```
bob        = { name:bob_name, wife:alice }
bob_name   = { Bob:{} }
alice      = { name:alice_name, husband:bob, pet:sam }
alice_name = { Alice:{} }
sam        = { name:sam_name, species:cat_name }
sam_name   = { Sam:{} }
cat_name   = { cat:{} }
```

To completely flatten this system we need to further replace all nested occurrences of `{}`, say, by the set name `empty` and add one more equation `empty = {}`. Of course, nesting is a reasonable notion, and atomic values are more user friendly from the external point of view. Thus, these concepts are included in the XML representation of WDB considered below, although the query system internally uses only completely flat set equations[1].

10.2 Practical representation of WDB as XML

Although set equations represent WDB in the most natural and intuitive way, directly suggesting that such data are hypersets, it makes sense to relate this approach to the popular XML representation of semi-structured data and use appropriate existing techniques. Thus, numerous and independently existing XML data can be treated by our approach, making its application considerably wider.

Extensible Markup Language (XML) is popular model for ordered (typically) tree-like semi-structured data. The portability, scaleability and tree (but extendable to graph) structure of XML has given rise to its wide spread useage. As such, systems of set equations, possibly allowing deep nesting, although very intuitively appealing could be represented practically as XML documents also based on the idea of representation of nesting data. However, the primary goal of our approach is not the implementation of XML querying, as much research and practical work has already been devoted to the latter: *CDuce* [7], *Lore* [33], *Quilt* [14] *XML-GL* [13], and *XML-QL* [23]; as well as the W3C standards *XSLT* [15], *XPath* [22], and *XQuery* [8] (based on Quilt).

[1] Note that WDB may (briefly) involve complicated equations, such as $res = q$ where q is an arbitrarily complicated term or formula, during the execution of queries q or after invoking the "splitting" rule during reduction. But, this extended system is, in fact, reduced to the flat form, and it is technically more convenient to work with other given WDB equations if they are presented in the flat form.

The main idea of the proposed XML-WDB format is to represent WDB systems of set equations as XML documents of a special form, and the most essential step consists in recursively replacing any labelled bracket expression

```
label : {...}
```

by the XML element:

```
<label>...</label>
```

Additionally, XML-WDB documents require: **(i)** the special root element `<set:eqns>` which denotes system of set equations, and **(ii)** the nested elements `<set:eqn>` denoting particular set equations. Defined set names participate as values of the `set:id` attribute of `<set:eqn>` tags, and referenced set names as values of the `set:ref` attribute (and also `set:href` attribute discussed later) of any other tags. Note that, as stated above, XML represents *ordered* tree-like semi-structured data, however, our set-theoretic approach to WDB ignores order. Thus, such XML documents are treated by our approach ignoring the order (and possible repetition) of elements.

Let us consider the system of set equations (with nesting allowed) in Section 10.1 (depicted visually in Figure 10.1) and its representation as an XML document in XML-WDB file 1. The names of the special elements (`set:eqns` and `set:eqn`) and special attributes (`set:id`, `set:ref` and `set:href`) should appeal to the readers' intuition that the XML-WDB document below corresponds to the above system of set equations.

XML-WDB file 1 Family database (cf. Figure 10.1).

```
<?xml version="1.0"?>
<set:eqns xmlns:set="http://www.csc.liv.ac.uk/~molyneux/XML-WDB">

  <set:eqn set:id="bob">
    <name>Bob</name>
    <wife set:ref="alice" />
  </set:eqn>

  <set:eqn set:id="alice">
    <name>Alice</name>
    <husband set:ref="bob" />
    <pet>
      <name>Sam</name><species>cat</species>
    </pet>
  </set:eqn>

</set:eqns>
```

Recall that atomic data such as `name:"Bob"` is interpreted as `name:{Bob:{}}`, and should therefore be translated into `<name><Bob></Bob></name>` or, equivalently, into `<name><Bob/></name>`. This might seem to contradict XML-WDB file 1 where rather `<name>Bob</name>` is used, but the inverse translation in Section 10.2.3 (Rule 2) shows that the empty element `<Bob></Bob>` or `<Bob/>` is treated equivalently as text data `Bob`. Here it appears as text data for the readers' convenience.

10.2.1 XML-WDB document format

In general, an arbitrary XML-WDB document is defined as follows.

Definition 5 (XML-WDB; see also Section 10.2.4 for the corresponding XML schema)**.** A well-formed and valid XML-WDB file is an XML document with the root element `<set:eqns>` containing possibly several `<set:eqn>` sub-elements. The `<set:eqns>` element should contain no attributes, whereas, the element `<set:eqn>` should contain the required `set:id` attribute only. The value of the attribute `set:id` should have a unique value (across the whole document) called the *defined set name* and can only be be a string of symbols which is any *simple set name* (according to the syntactical category `<simple set name>` in the BNF). The elements `<set:eqns>`, `<set:eqn>`, and the attribute `set:id` are not allowed to appear anywhere else in the document. The element `<set:eqn>` can contain possibly several arbitrary XML sub-elements. The attributes `set:ref` and `set:href` can appear (at any depth) in those arbitrary elements under `<set:eqn>`. The values of the attributes `set:ref` and `set:href` are called *referenced set names*, and must correspond to some existing `set:id` value in the same XML-WDB document in the case of `set:ref`, or `set:id` value in some other XML-WDB document in the case of `set:href`. To this end, the value of the attribute `set:href` should be *full set name* (as discussed in Section 10.2.2; cf. the syntactical category `<set name>` in the BNF) consisting of an (XML-WDB file) URL and simple set name defined in that file (delimited by #).

Everything else allowed by XML standard, what is not forbidden by the above restrictions, is permitted in the XML-WDB format.

Note 6. The important feature of this definition is that XML-WDB documents can contain quite arbitrary XML elements under `<set:eqn>`, thus allowing to include arbitrary XML data with any nesting, any text data and any attributes[2] (except `set:id`, and with restrictions on values of `set:ref` and `set:href`, as described above) into our hyperset approach to WDB. However, the order and repetitions of data will be irrelevant for our approach, and the usual XML attributes (except the attributes `set:ref` and `set:href` which have a special role, as described above) will be treated rather as tags which permit no further nesting.

[2] In general, arbitrary attributes are treated by the Rule 1 in Section 10.2.3 below.

10.2.2 Distributed WDB

Any WDB system of set equations may be divided into several subsystems (as XML-WDB files) with the possibility for the set names s participating in one subsystem (XML-WDB file) to be defined by set equations $s = \{\ldots\}$ either in the same or in some other subsystems (XML-WDB files). Thus, strictly speaking, we should always consider the corresponding full versions of set names defined in set equations of distributed WDB, even when a simple set name is used for simplicity. That is, each simple set name occurring as a value of `set:id` or `set:ref` attributes within an WDB-XML file should be understood as full set name obtained from the URL of this file by concatenating it with the simple name using # to delimite these parts. Moreover, this technique allows to avoid unintended simple set name clashes without cooperation or collaboration between the authors of distributed WDB-XML files. (Unfortunately, unintended clashes for using the same label for different intuitive meanings is still possible, however, this is not formal contradiction in our approach. Here the well-known idea of namespaces in XML could be used.)

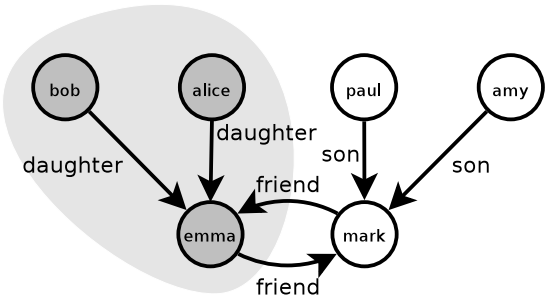


Figure 10.2: Example distributed WDB representing two fictitious families, divided into two fragments represented as white and grey nodes

Defined set names appearing in some XML-WDB file can participate as referenced set names in the same or other XML-WDB files. Those set names defined in the same XML-WDB file are referenced as simple set name values of the attribute `set:ref`, whereas, set names defined in some other XML-WDB file are referenced as full set name values of the attribute `set:href`. It is required that each full set name should refer to an existing XML-WDB file and the set equation within that file for the simple set name part (after the # symbol).

Let us now consider an example of distributed WDB, representing two families (visualised in Figure 10.2) and the corresponding XML-WDB files `family1.xml` and `family2.xml` (XML files 2 and 3) appearing below. Both simple and full set names participate as referenced set names in this example distributed WDB. For example, take the labelled element `daughter:emma` represented in XML-WDB file `family1.xml` as

```
<daughter set:ref="emma" />
```

where the attribute `set:ref` refers to simple set name `emma` defined within the same file. As an illustration of distribution, consider the labelled element `friend:mark` represented as

```
<friend set:href="...family2.xml#mark" />
```

where the attribute `set:href` refers to set name `mark` defined in the file `family2.xml`. Note that, the URL in this example has shorted for the sake of simplicity.

XML-WDB file 2 Family database fragment (cf. grey nodes Figure 10.2): family1.xml

```
<?xml version="1.0"?>
<set:eqns xmlns:set="http://www.csc.liv.ac.uk/~molyneux/XML-WDB">

  <set:eqn set:id="bob">
    <daughter set:ref="emma" />
  </set:eqn>

  <set:eqn set:id="alice">
    <daughter set:ref="emma" />
  </set:eqn>

  <set:eqn set:id="emma">
    <friend set:href="...family2.xml#mark" />
  </set:eqn>

</set:eqns>
```

XML-WDB file 3 Family database fragment (cf. white nodes Figure 10.2): family2.xml

```
<?xml version="1.0"?>
<set:eqns xmlns:set="http://www.csc.liv.ac.uk/~molyneux/XML-WDB">

  <set:eqn set:id="paul">
    <son set:ref="mark" />
  </set:eqn>

  <set:eqn set:id="amy">
    <son set:ref="mark" />
  </set:eqn>

  <set:eqn set:id="mark">
    <friend set:href="...family1.xml#emma" />
  </set:eqn>

</set:eqns>
```

The analogy of WDB with the WWW and, in particular possible distributed character of WDB does not imply it is necessarily so huge and unorganised as the WWW. It could be distributed between several sites, and supported by specialised WDB servers of some departments of an organisation owning this WDB and maintaining some specific structure of this WDB.

Thus, WDB might, in fact, be much more structured than the WWW, however, the general approach imposes no restrictions. Therefore, the concept of WDB *schema* or *typing* relation between hypersets or graphs (much more flexible than for the relational databases and based on the notion of bisimulation or "one-way" simulation) relativised to some typing relation on labels/atomic values can be considered for such databases [9, 41, 57, 69]. Here we will not go into details of this important topic as our main concern is the straightforward implementation of querying WDB which does not take into account any such WDB schemas.

10.2.3 Transformation rules from XML to systems of set equations

Let us show how any XML-WDB document, as described above, can be treated as a system of set equations by using the following simple transformations (applicable, in fact, to arbitrary XML documents, but giving the desired system of set equations only for the XML-WDB documents). There are however currently some restrictions on XML-WDB in these transformation rules which can easily be relaxed, for example attributes having many values `attr="value1 value2 ..."` are not taken into account.

10.2.3.1 Elimination of attributes and text data

The first two transformation rules, applied recursively, will eliminate attributes and atomic (text) data from arbitrary XML element by treating them as tags.

Rule 1 (Attribute elimination, except attributes `set:id`, `set:ref` and `set:href`).

XML tags which have attributes,

```
<tag attr="value" other-attributes>
  some-content
</tag>
```

transform to

```
<tag other-attributes>
  <attr>value</attr>
  some-content
</tag>
```

where `attr` is restricted to be any attribute name except the distinguished attributes `set:id`, `set:ref` and `set:href` belonging to the `set` namespace which will be considered later. Additionally, `some-content` means arbitrary XML content of an XML element.

In the case of empty element with attributes,

```
<tag attr="value" other-attributes />
```

transformation quite analogously gives the similar result,

```
<tag other-attributes>
  <attr>value</attr>
</tag>
```

This rule is applied until all attributes, except those attributes beglonging to the `set` namespace (`set:id`, `set:ref` and `set:href`), are eliminated. This way attributes are actually treated as tags.

Rule 2 (Atomic data elimination).

Text data with no white spaces

```
any-text-data
```

transforms to the empty XML element

```
<any-text-data/>
```

In the case of text data containing white characters (spaces, carriage-returns, tabs),

```
any text    data
```

all white characters are ignored, and the result is the corresponding sequence of the empty elements,

```
<any/><text/><data/>
```

As our set theoretic approach ignores order and repetitions (in contrast with the ordinary XML approach) this, in fact, means that a sentence (any text data) is considered rather as an unordered set of words. This way text data are actually treated as tags. (An another alternative would be to replace all white characters by the underscore symbol, thus giving rise to `<any_text_data/>`, like above.)

Iterated application of rules 1 and 2 eliminates all atomic (text) data and attributes except those attributes belonging to the `set` namespace (`set:id`, `set:ref` and `set:href`).

10.2.3.2 Elimination of tags

The remaining rules below allow transformation of XML elements with (simple) attributes and text data eliminated by the above rules into bracket expressions (possibly involving set names), and into set equations if there are tags `set:eqns` and `set:eqn` occurring as described in Definition 5. In the intermediate steps, the expression transformed will be in the mixed language.

Rule 3 (Tag elimination, except the tags `set:eqns` and `set:eqn`).

For arbitrary XML tags, except `set:eqns` and `set:eqn`, which have no attributes,

```
<tag>
  some-content
</tag>
```

transforms into

```
tag:{some-content}.
```

Those possibly remaining tags in sub-elements of `some-content` will be eliminated recursively by application of transformation rules 3 and 4. Quite analogously for the case of the empty element,

```
<tag/>
```

transforms to

```
tag:{}
```

Rule 4 (Elimination of tags with `set:ref` and `set:href` attributes).

```
<tag set:ref="set-name" />
```

transforms to the sequence

```
tag:set-name
```

Recall that other attributes were already eliminated by Rule 1. Furthermore, according to the definition of well-formed XML document an attribute name must only appear once in any tag, however, `set:ref` and `set:href` may participate together in any tag. The above elimination is considered as typical if only the attribute `set:ref` or `set:href` occurs.

Additionally, we must consider the following more general, however unlikely case when some content is present:

```
<tag set:ref="set-name1" set:href="set-name2">
  some-content
</tag>
```

transforms to

```
tag:set-name1,
tag:set-name2,
tag:{some-content}.
```

However, to be consistent with the first version of Rule 4, if `some-content` is empty, then (as an exception) the result should not contain the labelled element, `tag:{}`.

The above rules hold also for the case of the attribute `set:href`, or when both `set:ref` and `set:href` are present within a tag. Note that after applying Rule 4, the difference between these two attributes is not taken into account in generating the result. Recall that `set:ref` refers to a simple set name, whereas, `set:href` refers to a full set name which is actually an URL together with simple set name (see Section 10.2.2). Such syntax explicitly differentiating between simple and full set names is convenient for implementation. After applying this rule this feature will disappear, but the difference between the shapes of simple and full set names will remain, so that nothing essential will be lost.

Rule 5 (Elimination of tags `set:eqn` and `set:eqns`).

```
<set:eqn set:id="simple-set-name">some-content</set:eqn>
```

is replaced by the equation,

```
simple-set-name = {some-content}
```

and,

```
<?xml ... >
<set:eqns>some-content</set:eqns>
```

is replaced by

```
some-content
```

that is, by system of set equations (in the case of a well-formed XML-WDB document; cf. Definition 5 above).

Note that, all the above rules can be applied in arbitrary order, leading to a unique system of set equations.

10.2.4 XML schema for XML-WDB format

A well-formed and valid XML-WDB document must conform to Definition 5. As our general goal is implementation, let us also present the XML schema[3] (at the end of this section) which corresponds to this definition almost completely (as XML schemes are, in fact, insufficiently expressible).

First of all, the schema requires that all the declared elements `eqns` and `eqn`, and attributes `id`, `ref` and `href` are qualified under the namespace `http://www.csc.liv.ac.uk/~molyneux/XML-WDB`. In practice the author of any XML-WDB document can declare this namespace as the mnemonic `set`[4] and use `set:eqns` instead of just `eqns`, etc. to emphasise these special elements/attributes are subject to the rules of this schema.

The root element `eqns` of an XML-WDB document is declared in the schema as having the complex type `system_of_set_equations`, as follows,

```
<xsd:element name="eqns" type="system_of_set_equations"/>.
```

The complex type `system_of_set_equations` is defined as

```
<xsd:complexType name="system_of_set_equations">
   <xsd:sequence minOccurs="0" maxOccurs="unbounded">
     <xsd:element name="eqn" type="set_equation"/>
   </xsd:sequence>
</xsd:complexType>
```

where an arbitrary number (≥ 0) of set equations can participate in any XML represented system of set equations. Note that, by definition only, `eqn` subelements can participate under an `eqns` element. Here, `eqn` elements represent set equations by the given complex type `set_equation`, which is defined by two elements:

```
<xsd:sequence minOccurs="0" maxOccurs="unbounded">
  <xsd:any namespace="##any" processContents="lax"/>
</xsd:sequence>

<xsd:attribute form="qualified"
               name="id"
               type="xsd:ID"
               use="required"/>
```

Thus, any `eqn` element must contain the required attribute `id`, and may contain arbitrary XML sub-elements. Note that, by definition, only one attribute, `id`, must appear in `eqn`

[3] also available at `http://www.csc.liv.ac.uk/~molyneux/XML-WDB/schema/xml-wdb.xsd`

[4] In fact, the namespace `http://www.csc.liv.ac.uk/~molyneux/XML-WDB` could be declared by any chosen mnemonic, let us say `s`.

elements. The corresponding value of the `id` attribute must be unique over the entire XML-WDB document according the type `xsd:ID`. However, the schema only ensures the well-formedness with `lax` processing of arbitrary XML sub-elements, and therefore does not check that such elements are XML-WDB valid according to Definition 5. In particular this schema says nothing about `ref` and `href` attributes and how they can be used. Thus, our implementation additionally ensures the following:

- The elements `eqns` and `eqn` and attribute `id` qualified under the `http://www.csc.liv.ac.uk/~molyneux/XML-WDB/` namespace can not participate in arbitrary XML sub-elements.
- The attribute `ref` must have simple set name value, defined by the `id` attribute in the same XML-WDB file. Furthermore, the attribute `href` must have full set name value whose simple set name part is defined in some other well-formed and valid XML-WDB file.

Thus, any well-formed XML document is considered as valid XML-WDB document if it can be successfully validated against the above schema and conforms to these additional rules. However, our Δ language query implementation deals directly with systems of set equations, therefore it is necessary to rewrite from valid XML-WDB files into systems of set equations, by treating them with the rules from Section 10.2.3. The inverse transformation from systems of set equations to XML-WDB format is also implemented.

XML schema 1 XML-WDB file schema: xml-wdb.xsd

```
<?xml version="1.0" encoding="UTF-8"?>

<xsd:schema  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.csc.liv.ac.uk/~molyneux/XML-WDB"
  xmlns="http://www.csc.liv.ac.uk/~molyneux/XML-WDB"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified">

  <xsd:complexType name="system_of_set_equations">

     <xsd:sequence minOccurs="0" maxOccurs="unbounded">
       <xsd:element name="eqn" type="set_equation"/>
     </xsd:sequence>

  </xsd:complexType>

  <xsd:complexType name="set_equation">

     <xsd:sequence minOccurs="0" maxOccurs="unbounded">
       <xsd:any namespace="##any" processContents="lax"/>
     </xsd:sequence>

     <xsd:attribute form="qualified" name="id"
       type="xsd:ID" use="required"/>

  </xsd:complexType>

  <xsd:element name="eqns" type="system_of_set_equations"/>

</xsd:schema>
```

Part IV

Evaluation

Chapter 11

Comparative analysis

11.1 Preliminary comparison

There have been many proposed approaches for modelling and querying semi-structured data. Many of these approaches are based on the graph model, which has become the prevalent model for representation of semi-structured data. For example, the graphical Object Exchange Model (OEM) [51] was used in the integration of heterogeneous information sources in Tsimmis [31] and the semi-structured query language Lorel [2, 46]. Moreover, there has been some trend toward the XML document model, which is essentially the graph model restricted to ordered trees, but arbitrary graphs can be imitated by using the attributes `id` and `ref` to define links between tree branches. In fact, Lore (implementation of the Lorel language) was later migrated to XML [33].

The most natural and intuitive way of querying graphs employed in most approaches is path navigation by using path expressions. However, path expressions are evidently sufficiently complicated syntactical means to achieve expressive power in queries. This is practically very reasonable and means path expressions are a strong technical tool. But, on a logical level (in the wide sense of this word) such complicated things are always considered as definable in terms of some other more fundamental concepts. Thus, in foundation of mathematics such fundamental concepts are set, membership relation, logical quantifiers, etc. allowing to express all other concepts, constructions and proofs in mathematics and (theoretical) computer science. In a sense, the graph approach to semi-structured databases lacks natural logically fundamental concepts, and in these circumstances path expressions are included as the main tool for achieving expressive power. On the other hand, the set theoretic approach to semi-structured databases presented in this thesis does not require path expressions[1] to achieve high expressive

[1] besides the related classical operation of transitive closure of a set and a general recursion operator — classical inductive definitions

power which in fact captures exactly all "generic" polynomial time computable operations over hypersets [41, 43, 56, 57]. Therefore, the language can be considered theoretically as having in this sense no "gaps". But, from the point of view of practical usability and efficiency of implementation, path expressions should be eventually included in our implementation of the Δ-language although not increasing its expressive power (see [61]).

From the traditional theoretical point of view polynomial time computability of queries in Δ (which is usually theoretically considered as "feasible computability") allows to consider Δ as computationally viable. However, in a practical sense, we cannot insist on this usage of the term "feasible" because polynomials can be of high degree and with huge coefficients. Also, this makes less sense in the context of those most expensive computational steps assuming downloading numerous files from the World-Wide Web. Thus, we rather consider this characteristic not as a witness of efficiency of Δ but as a good witness of expressive power of the language. Anyway, when comparing this approach with others, it can be considered as top-down from theory to practice. In particular, this explains again our attitude to not include path expressions in the main conceptual version of the Δ-language, being a definable concept, and considering them only as technical "conservative" extension, although very important practically.

Recall that hypersets representing WDB can be visualised as graphs, and thus, in principle, our approach can treat graph structured data from other approaches, but assuming that the order and repetition of such data does not matter. As the latter is not always the case, the precise comparison with other approaches is not so straightforward. Similarly, our implementation can query arbitrary XML elements, rewriting from XML-WDB to systems of set equations and ignoring order. Although the aim of the project was not XML querying, this accomplishment extends possible applicability of our implementation.

Now, after these preliminary general comments, let us consider several known approaches to semi-structured databases and to set theoretic programming.

11.2 SETL

An important practical predecessor of our work is the set theoretic programming language SETL [62, 63, 64] which deals with hereditarily-finite well-founded sets (without cycles) and tuples. (Note that tuples or, more generally, records $[a_1 : x_1, \ldots, a_n : x_n]$ can be trivially treated in our approach as sets $\{a_1 : x_1, \ldots, a_n : x_n\}$ in which all labels a_i are different.) This general purpose programming language exploits the notion of set as fundamental data structure with its set theoretic style of constructs like collection in Δ. It is, however, an imperative language using such traditional operators as the assignment operator, loops, etc. For example, let us consider the SETL program:

```
A = {1,2,3,4,5};
B = { x: x in A | x >= 3 };
print(B);
```

where the statement on the second line reminds us of the Δ-term collect. In fact, the result of executing this SETL program is the output set of `B`, which is, in fact, defined as those numbers `x` belonging to the set `A` such that the number `x` is greater than or equal to three, as follows:

```
{3,4,5}.
```

Furthermore, in SETL, equality between sets is understood as "deep" set equality implemented as the following (recursive) procedure taken from [62]:

```
proc equal(S1,S2);
    if # S2 /= # S1
    then return false;
    else
        (forall x in S1)
            if x notin S2 then return false;
            end if;
        end forall loop;
        return true; -- S1 and S2 are equal
    end if;
end proc;
```

That is, the two sets `S1` and `S2` are equal if they have the same cardinality and each element `x` of the set `S1` participates as a member in the set `S2`. In fact, this equality procedure will be called recursively for each membership test `notin` (where, like in our case, $x \in y \iff \exists x' \in y\,.\,\texttt{Equal}(x, x')$). Hence, `S1` and `S2` are equal if their elements are equal and their elements are also equal, and so on. This is similar to bisimulation equivalence which is an important concept in our hyperset theoretic approach. The use of cardinality operator $\#$ either witnesses that hereditarily-finite sets are represented in SETL implementation in strongly extensional form and, anyway, assumes further recursive call of equality. In contrast to SETL, the implemented Δ language is actually a declarative query language to semi-structured or Web-like databases and, as such, is not intended to be a universal language. The degree of universality of Δ is characterised by its expressive power equivalent to polynomial time. Also, SETL does not have any construct similar to the decoration operator within the Δ-language which allows for restructuring, but its universal character should allow to define decoration for acyclic graphs. In contrast to SETL, the main characteristic feature of Δ is the extension of the ideas of descriptive complexity theory [37, 38, 55, 74] (usually considered in connection with the relational approach to databases) from finite relational structures to hereditarily-finite (hyper)sets and, thereby, to semistructured databases.

The most recent development on the SETL language was the implementation described in [4], which introduced Internet programming using sockets into the SETL language. In fact, these latest considerations further support that SETL is actually a general purpose programming language, and in this sense differs from Δ which is a query language.

11.3 UnQL

The UnQL query language [10, 11] is closest to our approach as it is based on bisimulation, with its operators also being bisimulation invariant as in our case. However, despite considering bisimulation, UnQL is based on the graph model, and the op. cit. do not even mention hyperset theory. UnQL can also be characterised as a bottom-up approach from graphs to something reminding us of hypersets. Moreover, there is no operator for testing equality between graph vertices (neither literal nor based on bisimulation) in the UnQL language. However, bisimulation should be used in defining the semantics of path expressions (patterns in their terminology) in the UnQL language, as shown in [61] and in our example in Section 3.6, ensuring that its operations really are bisimulation invariant. Much of the UnQL approach is devoted to the rather complicated way in which they deal with graphs, which appears more technical compared to the intuitive denotational and operational semantics of the hyperset approach. In a sense, UnQL has defined only operational semantics over graphs, which is bisimulation invariant. No abstract concept like hyperset and corresponding (hyper)set theoretical style of thought is explicitly described. Moreover, operational semantics of the structural recursion operator is rather complicated by working with multiple "input" and "output" vertices considered as essential part of graphs to be queried by UnQL. Therefore, semi-structured data represented in UnQL does not exactly correspond to hypersets, although it can be imitated by hypersets as shown in [61]. Also, the UnQL language and related language UnCal were shown in [61] to be embeddable within Δ, but, as reasonably conjectured, not vice versa. This embedding, although done in purely set theoretic terms, is based on the interpretation of arbitrary graphs as sets of ordered pairs. The bisimulation invariant operations on graphs of UnQL are defined set theoretically but as operations on graphs rather than as operations on abstract entities denoted by these graphs (with multiple "inputs" and "outputs") considered up to simulation. In particular, the main structural recursion construct of UnQL is definable in Δ by manipulating graphs using recursive separation and concluded by applying decoration operation to get a hyperset imitating the result (with multiple "inputs" and "outputs"). In fact, many of the operations in UnQL are based on various ways of appending such kind of graphs (via "inputs" and "outputs"), including structural recursion, all of which may be considered as a special versions of the decoration operator. However, the full version of the powerful decoration operator (which is much simpler and logically more fundamental

than its particular versions mentioned) is neither considered nor definable in UnQL (according to the conjecture in [61, page 813]).

11.4 Lore

Lore (Lightweight Object REpository) [46] is the implementation of the Lorel query language [2] based on the OEM graph model [51]. Lorel is an extention of the Object Query Language (OQL) [12] and, in fact, statements written in the Lorel are translated to OQL. Moreover, additional features of Lorel (such as path expressions, and type coercion) are syntactical sugaring of OQL. The OEM model is similar to the data model used in UnQL, but unlike UnQL and also our approach, does not consider graphs up to bisimulation. Therefore, bisimulation invariance is not pursued in this approach, hence, in this way it is crucially different from UnQL and Δ. In the OEM model equality is between graph nodes (OIDs) rather than value equality using bisimulation. Lorel also uses ordinary equality between sets of OIDs, which, however, is not the "deep" set equality assumed by bisimulation. Therefore, Lorel would treat some of our examples differently, and thus, only very informal and superficial comparison is possible, unlike the comparison with UnQL. However, the `select` operator of Lorel is very similar to our `collect` construct, as illustrated in the following example Lorel query:

```
SELECT pub
FROM pub in BibDB
WHERE pub.author = "Smith"
```

and the (strikingly similar) corresponding Δ-query,

```
set query collect {
    'null':pub
    where pub-type:pub in BibDB
    and author:"Smith" in pub
}
```

Note that only OIDs are `select`ed in Lorel, whereas in Δ (OIDs or) set names denote (hyper)sets which are, in fact (on the level of abstract semantics) `collect`ed. Note that, OIDs in Lorel denote just themselves and nothing more. Lorel can not express restructuring queries, unlike Δ which can perform restructuring queries with the decoration operation (at the final stage). Thus, informally (as formal comparison is impossible due to the above differences in data models – graphs vs. hypersets represented by graphs) Lorel (and also UnQL) can be said to be also strictly embeddable in Δ^2. Finally, there is also no recursion operator (except for Kleenes star in path expressions) and nothing similar to decoration operator (important for deep restructuring).

[2] ignoring so called path variables which may potentially lead to exponential complexity and, for simplicity, some less essential aspects like typing and coercion

11.5 Strudel

Strudel is a Web site management system [26] for creating Web pages from heterogeneous data sources via the StruQL query language [27] (see also [1]). In particular, the `link` clause in StruQL is able to do simple restructuring. In fact, Strudel allows to generate real Web sites in a declarative way from a site graph (a graphical "plan" of a site) that encodes the Web site's structure. The latter feature resembles the decoration construct although outside of hyperset approach. In Studel data is integrated from heterogeneous sources by mediators which rewrite from various data sources (such as XML files, bibtex files, etc.) to Strudel data graphs. StruQL queries over these data graphs, in fact, define the Web site structure creating Web pages and hyperlinks between Web pages.

11.6 G-Log

G-Log [19] is another query language for semi-structured data represented as arbitrary labelled graphs. However, unlike the other approaches consider so far (Lorel, UnQL, Δ) any query, as well as data, in G-log is represented graphically as a set of schematical red/green coloured "rule" graphs. Querying in G-log (in general, updating) is based on matching the query rule graph with the "concrete" black coloured data graph. This matching assumes one of three possible kinds of bisimulation (in particular, isomorphic embedding) of the red part of the rule with a subgraph of the black concrete data graph, and using the green part for updating the concrete data graph. This procedure is essentially non-deterministic and, in fact, can be executed in non-deterministic polynomial time (rather than polynomial time in the case of Δ). The expressive power of G-log in its present form, or its potential extensions, is unclear, as well as precise comparison with Δ. Granted, both are based on bisimulation but in a somewhat different way. The rule graphs of G-log can be described in some logical form, but it is unclear how to systematically relate this with the syntax of Δ to have a better comparison. In principle, extending Δ by quantification over the subset of a set, $\forall x \subseteq t, \exists x \subseteq t$, together with definability in Δ the necessary versions of bisimulation over graphs could make it possible to imitate matching of a rule graph with a subgraph of the data graph. But, it seems unclear whether there exists a natural unifying conceptual framework for both approaches. Furthermore, G-log is an open ended language with some ideas of its extension discussed in [19]. In any case, we can conclude that UnQL and even Lorel[3] are syntactically, as well as in terms of operational semantics, much closer to Δ than G-log. However, matching with a subgraph is somewhat similar to the idea of path expressions which appear in both UnQL and Lorel, the latter being imitated in Δ as illustrated in Section 3.6.

[3] ignoring that Lorel does not consider bisimulation

11.7 Tree (XML) model approaches

The XML data model is based on ordered trees, whereas the other approaches to querying semi-structured databases discussed so far deal with arbitrary graphs. (However, as we already mentioned, using attributes `id` and `ref` in XML allows imitate arbitrary graphs.) It might seem that querying XML data is formally outside of the (hyper)set theoretic view as the XML document model assumes a fixed order on the children of any node. Despite this our approach is able to query restricted XML documents (XML-WDB files which, however, can involve arbitrary nested XML elements) interpreted as systems of set equations.

The following comparisons focus on three contemporary XML data model approaches, XSLT, XQuery and XPath, all of which were developed by W3C working groups. In fact, these languages are the successors to many other XML model approaches, for example, XQuery is based on the Quilt query language [14]. However, for brevity no comparisons will be made with these predecessors.

XSLT

XSLT (eXtensible Stylesheet Language transformations) [15] is a rule based language for transforming the structure of an XML document, that is, XSLT rewrites an XML document to another XML document with different structure. Thus, XSLT does allow convenient manipulation of XML documents. XSLT rules are composed of template rules which *match* attributes/elements using XPath-like expressions (discussed below) and create new XML elements/attributes or apply other template rules. This style of language and its operational semantics is rather different from the Δ-query language. In particular XSLT is typically used to visualise XML documents by transforming them into HTML Web pages.

XQuery

XQuery [8] is declarative query language for XML documents, and was derived from Quilt [14], Lorel [2] (described above) and XML-QL [23]. XQuery is, in fact, Turing complete and thus can be considered as more than just a query language but also, in a sense, as a general purpose programming language.

Path expressions (XPath)

XQuery and XSLT include XPath path expressions in its syntax. XPath is a language especially created to express paths navigating over XML document trees, and, in fact, XPath itself can serve as a query language.

Currently path expressions are not included in the implemented Δ-query language, however, they were shown to be definable in the original language [61], and a simple example demonstrating how Δ could be extended syntactically to have path expressions and how it can define their meaning was shown in Section 3.6. Thus, our language is rich enough by fundamental operators over sets so that, at least theoretically, path expressions are unnecessary. Of course, practically they are very desirable and must be included in Δ to make it more practically convenient and user friendly. Moreover, path expressions, if implemented well, would make execution time of queries better than queries imitating path expressions in the current version of Δ.

In general, comparison of Δ with query languages for XML can be done only on a rather superficial level. In fact, they do not share a common data model and the levels of abstraction are so different that more detailed comparison in general terms is difficult. We can only repeat that the closest approach to ours is UnQL where comparisons can be done in quite precise mathematical formulations [61].

Chapter 12

Conclusion and future outlook

In this thesis we explored the experimental implementation of the hyperset approach to semi-structured or Web-like databases and the query language Δ originally known only on a pure theoretical level. The primary goal was to demonstrate working practically with the Δ-query language, and secondly, some considerations towards one crucial aspect of efficiency of such querying in the case of distributed WDB. The latter involves some theoretical considerations in Chapter 6 and empirical testing in Section 7.2.

This chapter begins by reviewing the hyperset approach to semi-structured databases in the context of this thesis. In Section 12.2 we summarise the main results of our work which, in brief, consist in (i) the implementation of the query language Δ and (ii) development the concept of local/global bisimulation and running experiments demonstrating its fruitfulness in making query execution more efficient when equality (bisimulation) is involved. Some further simple optimisations used in our implementation are also discussed. Then we recapitulate briefly in Section 12.3 comparisons of Δ with other most close query languages. Finally, we conclude in Section 12.4 with some closing discussion towards possible future extensions and optimisations.

12.1 Hyperset approach to semi-structured databases

First of all, the hyperset approach to semi-structured or Web-like databases and their querying was described in this thesis on the base of the earlier theoretical work done in [41, 57, 61]. This approach considers hypersets as the abstract data model for WDB where the concrete representation of hypersets is given by systems of set equations which can be saved either as plain text files or as XML-WDB files. Likewise in relational databases where the abstract data model is relations, our approach focuses on abstract hypersets and strongly distinguishes them from their concrete representations by set equations (or corresponding XML-WDB form).

Set theory is known to play an extraordinary foundational role in mathematics, and here we wanted to demonstrate in a practical context that very general set theoretic approach towards semi-structured or Web-like databases is also quite reasonable.

Systems of set equations can also be trivially represented as graphs where the latter, if considered literally, lead to the more traditional approach to semi-structured databases. To visualise our considerations we also use graphs, but they play only an auxiliary role. Abstractly, graph nodes as well as corresponding set names in set equations, denote hypersets. In fact, it is assumed that any user of our query system should mainly rely on pure set theoretic style of thought which is (mostly) simple and intuitive.[1] Otherwise it would not be so widely accepted both in the foundation of mathematics, and in everyday mathematical practice. As graphs or corresponding systems of set equations can involve cycles, their nodes or set names denote, in general, hypersets. They differ from the ordinary concept of sets in the fact that hypersets are not necessary well-founded. Based on well-developed and understood hyperset theory [3, 5], such sets pose no conceptual difficulty in our approach. This approach demonstrates on a practical level that hypersets are no more difficult than the usual concept of sets, and are quite useful by allowing arbitrary semi-structured data to be represented in a completely set theoretic manner.

An additional feature of our data model is its distributed character, that is any system of set equations representing a WDB is allowed to be distributed, with set names used in one (XML-WDB) file possibly described by set equations in the others files. This leads to distinctions between simple set names described in the same file, and full set names involving also the URL of the file where this set name is described. This does not change the hyperset approach but extends its possible applicability. On the other hand, this distributed character of a WDB poses an additional challenge on how to check practically whether two set names (possibly described in remote files) denote the same abstract hyperset, i.e. whether two given set names or graph nodes are bisimilar. However, the problem of computing bisimulation in the distributed case was shown here to be, in principle, resolvable practically, as remarked later in Section 12.2.2.

Respectively, the Δ-query language considered here is set theoretic with the denotation Δ bearing from logic and set theory and traditionally emphasising its bounded character. The latter guarantees that all queries in Δ are computable in finite, in fact, polynomial time with respect to the the size of the input WDB. Moreover, it is known to have expressive power exactly corresponding to polynomial time (see [43, 57] and particularly [41, 57] for precise formulations of the labelled case considered here).

[1] The most subtle concept in our approach is the decoration operation.

12.2 Novel contributions

The main results of this work are the implementation of the hyperset approach to semi-structured databases and the query language Δ, and, secondly, the local/global approach towards efficient computation of bisimulation in the case of distributed WDB.

12.2.1 Implementation of the hyperset approach to semi-structured databases

The implemented version of the language Δ is quite complex and even somewhat comparable with practical programming languages. In fact, there was not enough time to create the most optimal implementation. The general problem of efficiency is so difficult and involving so many various aspects (see e.g. [32]) that it is mostly outside the scope of this thesis (with one exception which is most essential to our hyperset approach; see Section 12.2.2). Taking this into account, the main criteria were correctness of the implementation and its user friendliness so that the language could be demonstrated to a more practically oriented, rather than just a mathematically inclined, audience. As far as we see, the implementation satisfies these criteria based on our testing and also writing and running the worked examples in Sections 3.5–3.7. This query system was also used by my supervisor, Vladimir Sazonov, as demonstration tool for undergraduate students. This initial practical goal of the project lead to the successful development of:

- **Implementation of the Δ-query language** as a declarative language, based on those theoretical constructs in the original Δ-language. Furthermore, for the convenience of writing queries some important features were included in the implemented language, such as *library declarations* and *query declarations* which, although very useful as the reader can see from the example queries, do not extend the theoretical expressive power of the language.

- **Algorithms for checking the validity of queries** to ensure both well-formedness and well-typedness. These algorithms add important low-level details for our implementation serving also as a sufficiently strong guarantee that the implementation was done correctly. The aim of the *parsing* algorithm is to ensure well-formedness, according to the BNF grammar in Appendix A.1; whereas the aim of the *contextual analysis* algorithm is to ensure well-typedness (which required considerable efforts to develop).

The above syntactical considerations were highly important for implementation, and much time was dedication to ensuring these algorithms were described and implemented correctly. In fact, the following developments strongly rely on these algorithms:

- **Implementation of operational semantics of Δ language** according to reduction rules in [61] with some additional low-level descriptions for the operators `recursion`, `decoration` and `TC` also given here to aid implementation.

- **XML representation of WDB** by developing the XML-WDB format for systems of set equations and implementing algorithms rewriting from XML-WDB documents into systems of set equations, and vice versa. Currently we accepted this XML-WDB format as the standard way of representing WDB. These files can be saved on various sites and hyperlinked via full set names as we discussed above, and thus, WDB can be distributed (and queried) over the Internet. In fact, the XML-WDB format allows our approach to treat arbitrary nested XML elements within a WDB. The aim of this practical representation of WDB as XML is the ability, in principle, to query any existing XML data in our hyperset approach (assuming order and repetition in these data play no essential role).

12.2.2 Local/global approach towards efficient implementation of bisimulation

Bisimulation between WDB graph nodes or set names (i.e. whether they denote the same hypersets) is a crucial concept for the whole hyperset approach to WDB. The equality symbol (=) in our language means, abstractly, the identity between hypersets. But, from the point of view of implementation which deals with set names, rather than with abstract hypersets, the equality operator (=) means bisimulation which assumes sufficiently complicated computation. Thus, if we want to remain faithful to this approach and really value this set theoretic style then we should not only implement bisimulation, as it is described in Chapter 4, but also work towards optimising this expensive operation. It can be particularly expensive in the case of distributed WDB when computing bisimulation would assume potentially downloading lots of (possibly) remote WDB files, and we pay special attention to this challenge.

The main idea of the local/global approach consists in computing the (global) bisimulation relation ($\approx$) on the whole distributed WDB from many couples of local approximation relations ($\approx^L_+$ and $\approx^L_-$) for each WDB site (or even for each WDB file), and that the latter relations are easily derivable locally. This way the global task is distributed between the main agent (Bisimulation Engine) and local agents (servers of WDB sites). Furthermore, empirical testing suggested that the exploitation of local approximations in the computation of global bisimulation relation $\approx$ can considerably improve performance. Also, the idea that the Bisimulation Engine is working in background time (similarly to Google) to compute the global bisimulation relation from local approximations was crucial in this performance improving strategy. Experiments described in Section 7.2 suggested that bisimulation, although a very challenging problem, especially in distributed case, is not so hopeless practically as it might

seem. In particular, taking such optimisations into account the hyperset approach to WDB seems also potentially feasible practically.

12.2.3 Further optimisation

The work done on local/global bisimulation was the main focus of our attempts to optimise our implementation of the hyperset approach in the case of distributed WDB. Also, some additional consideration was given on writing more efficient queries in the current implemented version of Δ, such as the removal of redundancies by using the so called canonisation query `Can(x)`. In fact, this query does not change its input (`Can(x)=x` as abstract hypersets) but transforms its representation into an equivalent strongly extensional (non-redundant) form. The effect of using `Can` in one particular example (in the query which linear orders any hyperset, Section 3.7) is quite impressive. Another general optimisation related with the recursion operator (and also crucially improving execution time of the linear ordering query mentioned above) is based on the possibility of replacing bisimulation to compare the iteration steps by simple comparison of participating set names only. Of course, further work on optimising the implementation of Δ (in comparison with writing optimal queries, for example exploiting `Can` above) remains to be done (see Section 12.4 below).

12.3 Comparisons with other approaches

After considering various approaches in Chapter 11 we have found that the UnQL and Lorel query languages are closest to our approach. However conceptually, i.e., in fact, from the point of view of the hyperset approach, UnQL is the most close to Δ. The implemented Δ-language does not include yet path expressions typical for other approaches. But, this language is already a very expressive, and, in a sense, subsumes both the UnQL and (the main features of) Lorel languages.

12.4 Further work

In short, the primary goal of implementation and attempts towards optimisation described in this thesis can be considered as successful. However, development of the implementation and the experiments was very time consuming, and there was insufficient time to implement all potential ideas. Many useful features have yet to be implemented, such as:

- **Extending the implemented Δ-query language to make it more user friendly** with quantification over multiple variables. Also, similarly for the case of collection, separation and recursion constructs.

- **Improving the library function**, in particular to allow multiple or user defined libraries.

- **Extending the implemented Δ-query language to include path expressions** which are typically included in other approaches towards semi-structured databases and, additionally, are very useful practically. In principle, path expressions could be implemented by rewriting them into Δ-queries according to definitions in [61]. But, straightforward implementation should be more efficient.

- **Extending the implemented Δ-query language by update queries.**

- **More user friendly interface** for inputting queries and WDB, as well as for outputting query results. In particular, the graphical visualisation of WDB and query results (developing a special WDB browser, as well as an editor for WDB files).

Additionally, suitable techniques should be developed for creating WDB, taking into account its hyperset theoretic character:

- **Using WDB schemas** in the context of hyperset approach to impose restriction on the structure of WDB, just like in the relational approach but not necessarily so rigid. In fact, enforcing structure makes queries easier to write, and, additionally, can serve to eliminate possible unintended redundancies in set equations which could arise otherwise due to poor WDB design.

Furthermore, although some suggestions towards efficiency were made here, there remains much work towards development of a practically efficient implementation:

- **Adapting known and developing new optimisation techniques** such as indexing, hashing and other data structures helping to implement efficient searching as described in [73] to the case of semi-structured data. Redundancies in set equations arising during computation should be regularly eliminated, thus allowing writing queries without explicit using the canonisation query. In this case equality between sets trivially becomes the identity relation rather than the bisimulation relation. Also, identical query calls should be executed only once.

- **Dealing with redundancies** in various circumstances by developing various techniques and methodology e.g. related with redundancies (bisimilarities) arising due to local updates in a WDB file (answering questions such as: are redundancies possibly arising in such local way easy to eliminate? under which conditions? etc.), or due to mirroring WDB sites, etc.

- **Further improvements on the bisimulation engine** transforming it from imitational to a more realistic version (Web service) assuming several levels (granularity) of locality (WDB-files, WDB-sites, the whole WDB) and extending the range of experiments with this engine.

- **Adopting known [24, 25] and developing new techniques for optimisation of bisimulation** which, for example, may take advantage of WDB scheme (see above).

There is great scope for further theoretical and practical work. In summary, this could mean developing a full-fledged WDB management system and also WDB design techniques, and other methodologies based on the hypeset approach. Of course, the hyperset approach could be further evolved, e.g. it can be extended to also involve standard datatypes like integers, reals, strings as atomic data or label values with arithmetical and other operations over them (completely lacking in the current version of Δ), etc. Also, multi-hypersets [44], records, lists, etc. could be allowed. Another version of the Δ language capturing LogSpace [40, 42] (currently for well-founded sets only) could be either implemented in its present form or, firstly, theoretically extended to the case of hypersets. Anyway, working on the theoretical level in various directions and simultaneously developing more practically oriented implementations, like in this thesis, seems a fruitful style of research.

Appendix A

Appendix

A.1 Implemented BNF grammar of Δ-query language

The grammar of the implemented Δ-language is represented by the metasyntax notation Extended Backus-Naur Form (EBNF) which allows for example to define the repetition of syntactical categories using $\star$ or + (unlike regular BNF which does not have these features). For example, the EBNF production rule of `<declarations>` in Section A.1 defines an infinite number of possible forks, with any number of leaves labelled by `<declaration>` each separated by the terminal leaf labelled by `","`.

The EBNF notation (used here to express the Δ-language grammar) defines production rules as sequence of terminals (symbols) or non-terminals,

`"xxx"` - Terminal
`<yyy>` - Non-terminal

where production rules are constructed (from those terminals or non-terminals) according to the following rules,

Parentheses, `()`	- Grouping
Vertical bar, `|`	- Alternation
Square brackets, `[]`	- Optional
Kleene star, $\star$	- Repeat 0 or more times
Kleene plus, +	- Repeat 1 or more times

Top level commands

```
<top level command> ::=
    ( "library" <library command> | <query> | "exit" ) ";"
```

```
<query> ::=
    "boolean query" <delta-formula> | "set query" <delta-term>
```

Library commands

```
<library command> ::=
    "add" <declarations> |
    "list" [ "verbose" ]
```

Declarations

```
<declarations> ::= <declaration> ( "," <declaration> )*

<declaration> ::=
    <set constant declaration> | <label constant declaration> |
    <set query declaration> | <boolean query declaration>

<set constant declaration> ::=
    "set constant" <set constant> ("be"|"=") <delta-term>

<label constant declaration> ::=
    "label constant" <label constant> ("be"|"=") <label value>

<set query declaration> ::=
    "set query" <set query name> "(" <variables> ")" ("be"|"=")
    <delta-term>

<boolean query declaration> ::=
    "boolean query" <boolean query name> "(" <variables> ")"
    ("be"|"=") <delta-formula>

<variables> ::= <variable> ( "," <variable> )*
<variable> ::=  ( "set" <set variable> | "label" <label variable> )

<parameters> ::= <parameter> ( "," <parameter> )*
<parameter> ::=  ( <delta-term> | <label> )

<boolean query name> ::= <identifier>
<set query name> ::=     <identifier>
```

Δ-terms

```
<delta-term> ::= <set variable> |
                 <set constant> |
                 <set name> |
                 <atomic value> |
                 <enumerate> |
                 <union> |
                 "(" <multiple union> ")" |
                 <collect> |
                 <separate> |
                 <transitive closure> |
                 <recursion> |
                 <decoration> |
                 <if-else term> |
                 <set query call> |
                 <delta-term with declarations>

<set name> ::= <URI> "#" <simple set name>

<atomic value> ::= """ <identifier> """

<enumerate> ::= "{" <labelled terms> "}"

<union> ::=     (  "U" | "union" ) <delta-term>

<multiple union> ::=
    <delta-term> ( ( "U" | "union" ) <delta-term> )*

<collect> ::=
    "collect" "{" <labelled term> ( "where" | "|" ) <variable pair>
    ("in"|"<-") <delta-term> [ "and" <delta-formula> ] "}"

<separate> ::=
    "separate" "{" <variable pair> ("in"|"<-") <delta-term>
    ( "where" | "|" ) <delta-formula> "}"

<transitive closure> ::=
    ( "tc" | "TC" | "transitiveclosure" ) <delta-term>

<recursion> ::=
    "recursion " <set variable> " {" <variable pair> (" in "| "<-")
    <delta-term> ( "where" | "|" ) <delta-formula> "}"
```

```
<decoration> ::=   "decorate" "(" <delta-term> ", " <delta-term> ")"

<if-else term> ::= "if" <delta-formula> "then" <delta-term>
                   "else" <delta-term> "fi"

<set query call> ::= "call" <set query name> "(" <parameters> ")"

<delta-term with declarations> ::=
    "let " <declarations> "in" <delta-term> " endlet"

<URI> ::=          ( <web prefix> | <local prefix> ) <file path>
<web prefix> ::=   "http://" <host> "/" [ "~" <identifier> "/" ]
<local prefix> ::= "file://" ( (A-Z) | (a-z) ) ":/"
<host> ::=         <identifier> [ "." <host> ]
<file path> ::=    <identifier> ( "/" <file path> | <extension> )
<extension> ::=    ".xml"
<simple set name> ::= <identifier>
```

Δ-formulas

```
<delta-formula> ::= <atomic formula> |
                    "(" <conjunction> ")" |
                    "(" <disjunction> ")" |
                    "(" <quasi-implication> ")" |
                    <quantified formula> |
                    <negated formula> |
                    <if-else formula> |
                    <delta-formula with declarations>

<atomic formula> ::=
    <equality> | <label relationship> | <membership> |
    <boolean query call> | "true" | "false"

<equality> ::=     <set equality> | <label equality>

<set equality> ::= <delta-term> "=" <delta-term>

<label equality> ::=
    <label> "=" <wildcard label> | <wildcard label>  "=" <label>
```

```
<wildcard label> ::=
    ["*"] ( <label variable> | <label constant> ) ["*"] |
    "'" ["*"] <identifier> ["*"] "'"

<label relationship> ::= <label> "<" <label>
                         <label> ">" <label>
                         <label> "<=" <label>
                         <label> ">=" <label>

<membership> ::=        <labelled term> ("in"|"<-") <delta-term>

<boolean query call> ::= "call" <boolean query name>
    "(" <parameters> ")"

<if-else formula> ::= "if" <delta-formula> "then" <delta-formula>
                      "else" <delta-formula> "fi"

<delta-formula with declarations> ::=
    "let" <declarations> "in" <delta-formula> "endlet"

<conjunction> ::=       <delta-formula> ( "and" <delta-formula> )*

<disjunction> ::=       <delta-formula> ( "or" <delta-formula> )*

<quasi-implication> ::= <delta-formula>
    ( <quasi-implication connective> <delta-formula> )*

<quasi-implication connective> ::=
    "<=" | "=>" | "implies" | "iff" | "<=>"

<quantified formula> ::= <forall> <delta-formula> |
                         <exists> <delta-formula> |

<forall> ::=
    "forall" <variable pair> ("in"|"<-") <delta-term> [ "." ]

<exists> ::=
    "exists" <variable pair> ("in"|"<-") <delta-term> [ "." ]

<negated formula> ::= "not" <delta-formula>
```

Variables, constants, literals etc.

```
<label> ::= <label variable> | <label value> | <label constant>
<label variable> ::= <identifier>
<label constant> ::= <identifier>
<label value> ::=    "'" <identifier> "'"

<set variable> ::=   <identifier>
<set constant> ::=   <identifier>

<labelled terms> ::= <labelled term> ( "," <labelled term> )*
<labelled term> ::=  <label> ":" <delta-term>

<variable pair> ::=  <variable pair label> ":" <variable pair term>
<variable pair label> ::= <label variable> | <label value>
<variable pair term> ::= <set variable>

<identifier> ::=     ( (A-Z) | (a-z) | (0-9) | "_" | "-" )+
```

A.2 Example XML-WDB files

XML-WDB file 4 XML-WDB file http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f1.xml (cf. Section 3.5).

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

<set:eqns
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation=
 "http://www.csc.liv.ac.uk/~molyneux/XML-WDB/schema/xml-wdb.xsd"
 xmlns:set="http://www.csc.liv.ac.uk/~molyneux/XML-WDB">

  <set:eqn set:id="BibDB">
    <paper set:href=
     "http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f2.xml#p1"/>
    <paper set:href=
     "http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f2.xml#p2"/>
    <paper set:href=
     "http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f2.xml#p3"/>
    <book set:ref="b1"/>
    <book set:ref="b2"/>
  </set:eqn>

  <set:eqn set:id="b1">
    <refers-to set:ref="b2"/>
    <refers-to set:href=
     "http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f2.xml#p1"/>
  </set:eqn>

  <set:eqn set:id="b2">
    <author>Jones</author>
    <title>Databases</title>
  </set:eqn>

</set:eqns>
```

XML-WDB file 5 XML-WDB file http://www.csc.liv.ac.uk/~molyneux/t/BibDB-f2.xml (cf. Section 3.5).

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

<set:eqns
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation=
 "http://www.csc.liv.ac.uk/~molyneux/XML-WDB/schema/xml-wdb.xsd"
 xmlns:set="http://www.csc.liv.ac.uk/~molyneux/XML-WDB">

  <set:eqn set:id="p1">
    <refers-to set:ref="p2"/>
  </set:eqn>

  <set:eqn set:id="p2">
    <author>Smith</author>
    <title>Databases</title>
    <refers-to set:ref="p3"/>
  </set:eqn>

  <set:eqn set:id="p3">
    <author>Jones</author>
    <title>Databases</title>
  </set:eqn>

</set:eqns>
```

A.3 Predefined library queries

```
set query Pair (set x,set y) be
    { 'fst':x, 'snd':y },

boolean query isPair (set p) be (
        exists l: x in p . (
            l='fst'
            and
            forall m:z in p . ( m='fst' => z=x )
        )
        and
        exists l:y in p . (
            l='snd'
            and
            forall m:z in p .( m='snd' => z=y )
        )
    ),

set query First (set p) be
    union separate { l:x in p where l='fst' },

set query Second (set p) be
    union separate { l:x in p where l='snd' },

set query CartProduct (set x,set y) be
    union collect {
        'null':collect {
            'null':call Pair ( xx, yy )
            where l:yy in y
        }
        where m : xx in x
    },

set query Square (set z) be
    call CartProduct ( z, z ),

set query LabelledPairs (set v) be
    collect { l:{ 'fst':v, 'snd':u } where l:u in v },

set query Nodes (set g) be
    union separate { m:p in g where call isPair ( p ) },
```

```
set query Children (set x,set g) be
    collect {
        l:call Second ( p )
        where l:p in g
        and (
            call isPair ( p )
            and
            call First ( p ) = x
        )
    },

set query Regroup (set g) be
    collect {
        'null':call Pair ( x, call Children ( x , g ) )
        where m : x in call Nodes ( g )
    },

set query CanGraph (set x) be
    union collect {
        'null':call LabelledPairs ( v )
         where m:v in TC ( x )
    },

set query Can (set x) be
    decorate ( call CanGraph ( x ), x ),

set query TCPure(set x) be
    collect{ 'null':v where l:v in TC ( x ) },

set query HorizontalTC (set g) be
    recursion p {
        'null':pair in call Square ( call Nodes ( g ) )
        where (
            call First ( pair ) = call Second ( pair )
            or
            exists m:z in call Nodes ( g ) . (
               'null':call Pair ( call First ( pair ), z ) in p
               and
               'null':call Pair ( z, call Second ( pair ) ) in g
            )
        )
    },
```

```
set query TC_along_label (label l,set z) be
    recursion p {
        k:x in TC ( z )
        where (
            ( x=z and k = 'null' )
            or
            ( k=l and exists m:y in p . l:x in y )
        )
    },

set query SuccessorPairs (set L) be
    separate {
        l:pair in L
        and not exists l:x in call Nodes(L) . (
            'null':call Pair ( call First ( pair ),x ) in L
            and
            'null':call Pair ( x, call Second ( pair ) ) in L
        )
    },

boolean query Precedes5(set R,label l,set x,label m,set y) be (
        l < m
        or (
            l=m
            and
            exists 'null':p in R . (
                'fst':x in p and 'snd':y in p
            )
        )
    ),
```

```
set query StrictLinOrder_on_TC (set z) be
    recursion R {
        'null':p_xy in call Square( call Can(call TCPure(z)) )
        where (
            (
              not 'null':p_xy in R
              and
              not exists 'fst':xx in p_xy .
                    exists 'snd':yy in p_xy .
                      exists 'null':inv_p in R . (
                        'fst':yy in inv_p
                        and
                        'snd':xx in inv_p
                      )
            )
            and
            exists 'snd':yyy in p_xy .
              exists lu:u in yyy . (
                exists 'fst':xxx in p_xy .
                  forall lv:v in xxx . (
                    call Precedes5(R,lu,u, lv,v)
                    or
                    call Precedes5(R,lv,v, lu,u)
                  )
                and
                forall fs:xy in p_xy .
                  forall lw:w in xy . (
                    call Precedes5(R,lu,u, lw,w) =>
                    exists 'fst':xxxx in p_xy .
                      exists lp:p in xxxx .
                        exists 'snd':yyyy in p_xy .
                          exists lq:q in yyyy . (
                            not call Precedes5(R,lp,p, lw,w) and
                            not call Precedes5(R,lw,w, lp,p) and
                            not call Precedes5(R,lq,q, lw,w) and
                            not call Precedes5(R,lw,w, lq,q)
                          )
                  )
              )
        )
    }
```

Bibliography

[1] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web - From Relations to Semi-structured Data and XML*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 2000.

[2] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet Lynn Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[3] Peter Aczel. *Non-Well-Founded Sets*. CSLI, Stanford, CA, USA, 1988.

[4] David Bacon. *SETL for Internet Data Processing*. PhD thesis, New York University, NY, USA, 2000.

[5] John Barwise and Lawrence Moss. *Vicious circles: on the mathematics of non-well-founded phenomena*. Center for the Study of Language and Information, 1996.

[6] Jon Barwise. *Admissible Sets and Structures*. Springer, Berlin, Germany, 1975.

[7] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce: an XML-centric general-purpose language. In Colin Runciman and Olin Shivers, editors, *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003*, pages 51–63. ACM, 2003.

[8] Scott Boag, Donald Dean Chamberlin, Mary Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML query language. W3C recommendation, W3C, January 2007. http://www.w3.org/TR/2007/REC-xquery-20070123/.

[9] Peter Buneman, Susan Davidson, Mary Fernández, and Dan Suciu. Adding structure to unstructured data. In *ICDT '97: Proceedings of the 6th International Conference on Database Theory*, pages 336–350, London, UK, 1997. Springer-Verlag.

[10] Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu. A query language and optimization techniques for unstructured data. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 505–516, Montreal, Quebec, Canada, 1996. ACM.

[11] Peter Buneman, Mary Fernández, and Dan Suciu. UnQL: a query language and algebra for semistructured data based on structural recursion. *The VLDB Journal*, 9(1):76–110, 2000.

[12] Roderick Geoffrey Galton Cattell and Tom Atwood, editors. *The Object Database Standard: ODMG-93*. Series in Data Management Systems. Morgan Kaufmann, 1993.

[13] Stefano Ceri, Sara Comai, Ernesto Damiani, Piero Fraternali, Stefano Paraboschi, and Letizia Tanca. XML-GL: a graphical language for querying and restructuring XML documents. *Computer Networks*, 31(11-16):1171–1187, 1999.

[14] Donald Dean Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: An XML query language for heterogeneous data sources. In Dan Suciu and Gottfried Vossen, editors, *The World Wide Web and Databases: Third International Workshop (WebDB 2000), Dallas, Texas, USA, May 18-19, 2000, Selected Papers*, volume 1997 of *Lecture Notes in Computer Science*, pages 1–25. Springer-Verlag, 2001.

[15] James Clark. XSL transformations (XSLT) version 1.0. W3C recommendation, W3C, November 1999. http://www.w3.org/TR/1999/REC-xslt-19991116.

[16] Edgar Frank Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 26(1):64–69, 1983.

[17] Thomas Connolly and Carolyn Begg. *Database Systems: A Practical Approach to Design, Implementation and Management*. Addison-Wesley Publishing Company, third edition, 2002.

[18] Mariano P. Consens and Alberto O. Mendelzon. GraphLog: a visual formalism for real life recursion. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, April 2-4, 1990, Nashville, Tennessee*, pages 404–416. ACM Press, 1990.

[19] Agostino Cortesi, Agostino Dovier, Elisa Quintarelli, and Letizia Tanca. Operational and abstract semantics of the query language G-Log. *Theoretical Computer Science*, 275(1-2):521–560, 2002.

[20] Elias Dahlhaus and Johann Andreas Makowsky. The choice of programming primitives for SETL-like programming languages. In *ESOP 86, European Symposium on Programming, Lecture Notes in Computer Science 213*, pages 160–172. Springer, March 1986.

[21] Elias Dahlhaus and Johann Andreas Makowsky. Query languages for hierarchic databases. *Information and Computation*, 101:1–32, 1992.

[22] Steven DeRose and James Clark. XML path language (XPath) version 1.0. W3C recommendation, W3C, November 1999. http://www.w3.org/TR/1999/REC-xpath-19991116.

[23] Alin Deutsch, Mary Fernández, Daniela Florescu, Alon Levy, and Dan Suciu. A query language for XML. *Computer Networks*, 31(11-16):1155–1169, 1999.

[24] Agostino Dovier, Carla Piazza, and Alberto Policriti. An efficient algorithm for computing bisimulation equivalence. *Theoretical Computer Science*, 311(1-3):221–256, 2004.

[25] Jean-Claude Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13(1):219–236, 1989.

[26] Mary Fernández, Daniela Florescu, Jaewoo Kang, Alon Levy, and Dan Suciu. Catching the boat with strudel: experiences with a web-site management system. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 414–425, NY, USA, 1998. ACM.

[27] Mary Fernández, Daniela Florescu, Alon Levy, and Dan Suciu. A query language for a web-site management system. *SIGMOD Record*, 26(3):4–11, 1997.

[28] Marcelo Pablo Fiore, Achim Jung, Eugenio Moggi, Peter O'Hearn, Jon Riecke, Giuseppe Rosolini, and Ian Stark. Domains and denotational semantics: History, accomplishments and open problems. *Bulletin of EATCS*, 59:227–256, 1996.

[29] Marco Forti and Furio Honsell. Set theory with free construction principles. *Annali Scuola Normale Superiore Pisa Classe di Scienza*, 10(4):493–522, 1983.

[30] Robin Oliver Gandy. Set theoretic functions for elementary syntax. *Proceedings of Symposia in Pure Mathematics*, 13(2):103–126, 1974.

[31] Hector Garcia-Molina, Yannis Papakonstantinou, Dallan Quass, Anand Rajaraman, Yehoshua Sagiv, Jeffrey Ullman, Vasilis Vassalos, and Jennifer Widom. The tsimmis

approach to mediation: data models and languages. *Journal of Intelligent Information Systems*, 8(2):117–132, 1997.

[32] Hector Garcia-Molina, Jeffrey David Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall Press, NJ, USA, 2008.

[33] Roy Goldman, Jason McHugh, and Jennifer Widom. From semistructured data to XML: Migrating the Lore data model and query language. In Sophie Cluet and Tova Milo, editors, *Proceedings of the 2nd International Workshop on the Web and Databases (WebDB '99)*, pages 25–30, Philadelphia, PA, USA, June 1999.

[34] Yuri Gurevich. Algebras of feasible functions. In *In Proceedings of the 24th Annual Symposium on Foundations of Computer Science*, pages 210–214. IEEE Computer Society Press, 1983.

[35] Vadim Guzev, Vladimir Sazonov, and Yuri Serdyuk. Distributed querying of web by using dynamically created mobile agents, `http://www.csc.liv.ac.uk/~sazonov/papers/distributed_querying_of_web.pdf`, 2002. Supported by an RDF grant from The University of Liverpool.

[36] Marc Gyssens, Jan Paredaens, Jan Van den Bussche, and Dirk Van Gucht. A graph-oriented object database model. *IEEE Transactions on Knowledge Data Engineering*, 6(4):572–586, August 1994.

[37] Neil Immerman. Relational queries computable in polynomial time. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, pages 147–152, San Francisco, CA, USA, May 1982. ACM.

[38] Neil Immerman. *Descriptive Complexity*. Texts in Computer Science. Springer-Verlag, 1999.

[39] Ronald Bjorn Jensen. The fine structure of the constructible hierarchy. *Annals of Mathematics and Logic*, 4:229–308, 1972.

[40] Alexander Leontjev and Vladimir Sazonov. Δ: Set-theoretic query language capturing logspace. *Annals of Mathematics and Artificial Intelligence*, 33(2-4):309–345, 2001.

[41] Alexei Lisitsa and Vladimir Sazonov. Bounded hyperset theory and web-like data bases. In *Proceedings of the Kurt Goedel Colloquium (KGC 1997)*, volume 1234, pages 178–188, 1997.

[42] Alexei Lisitsa and Vladimir Sazonov. Δ-languages for sets and LOGSPACE computable graph transformers. *Theoretical Computer Science*, 175(1):183–222, 1997.

[43] Alexei Lisitsa and Vladimir Sazonov. Linear ordering on graphs, anti-founded sets and polynomial time computability. *Theoretical Computer Science*, 224(1–2):173–213, 1999.

[44] Alexei Lisitsa and Vladmir Sazonov. Bounded multi-hyperset theory and polynomial computability. Unpublished manuscript, 2007.

[45] Kenneth C. Louden. *Compiler Construction: Principles and Practices*. PWS Publishing Company/International Thomson Publishing, Boston, MA, USA, 1997.

[46] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallan Quass, and Jennifer Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, 1997.

[47] Brett McLaughlin and Justin Edelson. *Java and XML*. O'Reilly Media, third edition, 2006.

[48] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[49] Richard Molyneux. Implementation of a hyperset Δ-language as query language to web-like databases. Undergraduate dissertation, The University of Liverpool, May 2004.

[50] Richard Molyneux and Vladimir Sazonov. Hyperset/web-like databases and the experimental implementation of the query language delta - current state of affairs. In *ICSOFT 2007 Proceedings of the Second International Conference on Software and Data Technologies*, volume 3, pages 29–37. INSTICC, 2007.

[51] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object exchange across heterogeneous information sources. In *In Proceedings of the Eleventh International Conference on Data Engineering*, pages 251–260, Taipei, Taiwan, 1995.

[52] Jan Paredaens, Paul De Bra, Marc Gyssens, and Dirk Van Gucht. *The structure of the relational database model*. Springer-Verlag New York, Inc., New York, NY, USA, 1989.

[53] David Park. Concurrency and automata on infinite sequences. In *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, pages 167–183, London, UK, 1981. Springer-Verlag.

[54] Mark A. Roth, Herry F. Korth, and Abraham Silberschatz. Extended algebra and calculus for nested relational databases. *ACM Transactions on Database Systems*, 13(4):389–417, 1988.

[55] Vladimir Sazonov. Polynomial computability and recursivity in finite domains. *Elektronische Informationsverarbeitung und Kybernetik*, 16(7):319–323, 1980.

[56] Vladimir Sazonov. Bounded set theory, polynomial computability and Δ-programming. In *Lecture Notes in Computer Science*, volume 278, pages 391–397, 1987.

[57] Vladimir Sazonov. Hereditarily-finite sets, data bases and polynomial-time computability. *Theoretical Computer Science*, 119(1):187–214, 1993.

[58] Vladimir Sazonov. A bounded set theory with anti-foundation axiom and inductive definability. In *CSL '94: Selected Papers from the 8th International Workshop on Computer Science Logic*, pages 527–541, London, UK, 1995. Springer-Verlag.

[59] Vladimir Sazonov. On bounded set theory. In *Invited talk on the 10th International, Congress on Logic, Methodology and Philosophy of Sciences, Florence, August 1995, in Volume I: Logic and Scientific Method*, pages 85–103. Kluwer Academic Publishers, 1997.

[60] Vladimir Sazonov. Using agents for concurrent querying of web-like databases via a hyperset-theoretic approach. In *PSI '02: 4th International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pages 378–394, London, UK, 2001. Springer-Verlag.

[61] Vladimir Sazonov. Querying hyperset / web-like databases. *Logic Journal of the IGPL*, 14(5):785–814, 2006.

[62] J. T. Schwartz, Robert B. K. Dewar, E. Schonberg, and E. Dubinsky. *Programming with sets: an introduction to SETL*. Texts and Monographs in Computer Science. Springer-Verlag New York, Inc., New York, NY, USA, 1986.

[63] Jacob T. Schwartz. Set theory as a language for program specification and programming. Technical report, Courant Institute of Mathematical Sciences, New York University, NY, USA, 1970.

[64] Jacob T. Schwartz. On programming, an interim report on the setl project. Technical report, Courant Institute of Mathematical Sciences, New York University, NY, USA, 1973.

[65] Dana Stewart Scott and Christopher Strachey. Toward a mathematical semantics for computer languages. In *Proceedings Symposium on Computers and Automata*, volume 21 of *Microwave Institute Symposia Series*. Polytechnic Institute of Brooklyn, 1971.

[66] Yuri Serdyuk. Delta-language implementation, `http://www.botik.ru/~logic/bst/delta_implementation.html`, 1996. Supported by RBRF (project 96-01-01717) and INTAS (project 93-0972).

[67] Simon St.Laurent and Michael Fitzgerald. *XML pocket reference*. O'Reilly Media, third edition, 2005.

[68] Christopher Strachey. Fundamental concepts in programming languages. Lecture Notes, International Summer School in Computer Programming, Copenhagen, August 1967. Reprinted in *Higher-Order and Symbolic Computation*, 13(1–2), pp. 1–49, 2000.

[69] Dan Suciu. Typechecking for semistructured data. In *Database Programming Languages, 8th International Workshop, DBPL 2001 Frascati, Italy, September 810, 2001 Revised Papers*, volume 2397, pages 1–20, London, UK, 2002. Springer-Verlag.

[70] Robert Walker Taylor and Randall L. Frank. Codasyl data-base management systems. *ACM Computing Survey*, 8(1):67–103, 1976.

[71] Stan J. Thomas and Patrick C. Fischer. Nested relational structures. *Advances in Computing Research*, 3:269–307, 1986.

[72] Dennis Tsichritzis and Frederick Horst Lochovsky. Hierarchical data-base management: A survey. *ACM Computing. Survey*, 8(1):105–123, 1976.

[73] Jeffrey David Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1 of *Principles of Computer Science Series, 14*. Computer Science Press, Rockville, MD, USA, 1988.

[74] Moshe Y. Vardi. The complexity of relational query languages. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, pages 137–146, San Francisco, CA, USA, 1982. ACM.

[75] Des Watson. *High-Level Languages and their Compilers*. Addison-Wesley Publishing Company, 1989.

[76] Reinhard Wilhelm and Dieter Maurer. *Compiler Design*. Addison-Wesley Publishing Company, 1995.