Implementation of Recursively Nested Relation of *JRelix*

Zhan Yu

School of Computer Science
McGill University, Montreal

January 2004

The project report submitted to the faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

T.H. Merrett, Supervisor

Abstract

This project report discusses the design and implementation of new features involved in *JRelix*, including semi-structured data loading, improved queries (recursive nesting, path expression and regular expression operators). Semi-structured data is self-describing thus more flexible. Acceptance of semi-structured data loading makes data loading for relation initialization more convenient in *JRelix.* To simply the edition of relation loading data, the data can be edited in a file and saved as a .txt file. The relation can then be declared and initialized by the data in the file. After allowing *JRelix* to accept recursive nesting, the regular expression operators (“*”, “+”, “.”, “?”) have been implemented to query the relations with a recursively nesting domain. In addition, path operator, which is likely to be frequently used in querying nested relations, has been implemented as a short-cut by using the / operator. The implementation of this project is part of the *Aldat* project at McGill University.

Acknowledgments

I would like to express my gratitude to all those who gave me the possibility to complete the project. My great appreciation goes to my supervisor **Dr T. H. Merrett** for his guidance, patience, advice and encouragement in all the time of this project. I would also express my thanks for his generous financial support and the time he spent with me on the topics of this project.

My appreciation goes to Zongyan Wang. I benefited from her valuable advices throughout the work. I would also like to thank the School of Computer Science for the graduate courses and the research environment.

Finally, I would like to thank my wife, Ying Wang, for her love, support during my study at McGill.

Contents

List of Figures

Chapter 1

Introduction

This project report describes the design and implementation of some new features involved in *JRelix*, including semi-structured data loading and improved queries (recursive nesting, path expression and regular expression operators). In section 1.1, a background material and motivation is given for the implementations. In section 1.2, a brief outline on the structure of this report can be found.

1.1 Backgrounds and Motivation

Relix System

Over the past 18 years, *Relix,* which is a relational database language, has been designed and developed at the Aldat lab of the School of Computer Science at McGill. The original version of *Relix* was developed in the C language and ran on the UNIX operating system. Since then, the system has been enhanced with further developments. In 1998, the *Relix* system was redesigned to be implemented it in Java [Yua98, Hao98]. The new system was named *JRelix* and covered the most important functions of the original *Relix* system, with a further extension to support a nested relational model.

The current *JRelix* system supports relational algebra, domain algebra and computations. Furthermore, the Internet capability has been integrated into *JRelix*, so it can also process remote data processing through the Internet [Wan02].

Semi-structured Data

The data resides in different forms ranging from unstructured data to highly structured data. At an extreme we find data coming form traditional relational or object-oriented databases, with a completely known structure. At another extreme, we have data that is

fully unstructured, such as sound or images. But most of the data falls somewhere in between those two extremes and the data have been given the term *semi-structured data* [Suc99].

Semi-structured data is often described as "schema-less" or "self-describing", and these terms mean that a pre-imposed schema or type system is needed for the interpretation of semi-structured data [Bun97]. Typically, when we store or program with data, we firstly describe the structure (type, scheme) of that data and then create instances of that type. In *semi-structured data* we directly describe the data using a simple syntax. For example, {*name: "Joe", ID: "23451", Tel: "514-398-0980"*} is a simple set of pairs such as *name*: "*Joe*" consisting of a label and a value. We are not constrained to make all the tuples the same type. One of the main strengths of semi-structured data is its ability to accommodate variations in structure. In addition, semi-structured data allows forgetting any type the data might have had, and serializes it by annotating each data item explicitly with its description (such as *name*, *ID*, etc). Such data is called *self-describing.* The term serialization signifies the conversion of the data into a byte stream that can be easily transmitted and reconstructed at receivers [ABS00].

Schemas for semistructured data differ from those for relational data. In a traditional database approach, types are always fixed prior to populating the database. Once the data is populated, its binary storage cannot be interpreted without having knowledge the schema. With semistructured data we may specify the type after the database is populated. The type may often describe the structure only for a part of the data and, even then, do that with less precision. An important consequence is that a data instance may have more than one type [ABS00].

The web provides numerous popular examples of semi-structured data. On the web, data consists of files in HTML format, with some structuring primitives such as tags and anchors. Secondly, the need for semi-structured data arises when integrating several sources. Finally, semi-structured data arises under a variety of forms for a wide range of application such as scientific databases and on-line documentations. [Abi95].

The most popular example of *semi-structured data* is XML (Extensible Markup Language). It is a language for representing data as a string of text that includes interspersed "markup" for describing properties of the data. Using markup allows the text to be interspersed with information related to its content or form [Gra02]. Element is the basic component of XML. XML can contain attributes with varied order and multiple elements with the same element type. This makes it easier to represent more complex data.

Semistructured Data DBMS

The exploration of application of semi-structured data in databases is due to the advent of semi-structured data. Firstly, database systems were developed to manage semi-structured data. The most famous is *Lore* (Lightweight Object Repository), which was implemented at Stanford University. The *Lore* system is designed specifically for the management of semi-structured data. In general, Lore attempts to take advantage of structure where it exists, but can also handle irregular cases gracefully. Implementing the Lore system requires rethinking all aspects of a DMBS, including storage, query and user interface [MAG+97]. Secondly, data is not stored as semi-structured, but it can be exported as semi-structured and presented to other applications or users. Semi-structured data can be entered into a database by compressing the data and storing the data in the structure of the database, such as relations [Gra02].

A semi-structured data interface to a relational DBMS provides access to robust database technology with the advantages of semi-structured data delivery. It is especially useful when a relational database already exists. Loading the semi-structured data into an existing relational database allows more flexible data input for the relational database system [GRA02].

Generally, there are two approaches in devising query language for semi-structured data. First, take SQL as a starting point and features are added to perform useful queries. The second one is to start from a language based on the notion of computation on

semi-structured data then to modify that language into acceptable syntax [Bun97]. Several query languages for semi-structure data have been proposed: LOREL [QRS+95,AQM+96,AQM+97,MAG+97], UnQL [BDS95, BDHS96], WebSQL [MMM96]. Their common feature is the ability to traverse arbitrary long paths in the data, usually specified in the form of a regular path expression: thus these query languages are recursive. [Suc99].

Motivation

Due to the advantages of semi-structured data, one part of this project is devoted to the implementation of embedding semi-structured data into *JRelix*. An advantage is that the data structure can be much more flexible. Alternatively, the self-describing data could be embedded in natural language text.

We can further explore the flexibility of semistructured data. For nested relations, path expressions are useful to describe the hierarchy. And the notion of path expression takes its full power when we start using it in conjunction with wild card (.) or regular expression operators, such as Kleene star (*), plus operator (+) and question mark operator (?). This project is also devoted to implementing path expression, recursive nesting and the regular operators.

1.2 Outline of the Report

This project report discusses the design and implementation of some new features in *JRelix*. Chapter 1 of this report introduces the project topic. In chapter 2, the overview of the current *JRelix* system is given. Chapter 3 describes how the new features of JRelix are used, including semi-structured data input, recursive nesting, path expression (“/”) and the regular expression operators (“*”, “+”, “.”, “?”). Chapter 4 explores the issues involved in implementating the new features. In chapter 5, a brief summary is provided and future work is presented.

Chapter 2

JRelix Overview

This chapter is to introduce *JRelix*, describing how to use *JRelix* to perform database operations. Section 2.1 explains how to start *JRelix*. Section 2.2 describes the declarations of domains and relations and the initialization of relations. The functional operations of relational algebra will be explored in section 2.4. The use of domain algebra operations will be presented in section 2.5. In the section 2.6, the usage of views and computations will be briefly introduced.

2.1 Getting started

JRelix runs on any platform that has the Java Runtime Environment (JRE) version 1.1 or up. To start *JRelix* interpreter, type the following

```
java JRelix
```

As the result, *JRelix* copyright information will be displayed in its run-time environment, as illustrated in Figure 2.1.

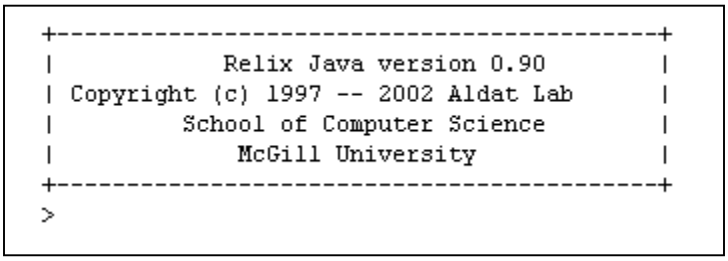


Figure 2.1 *JRelix* startup

After successful starting up, prompt > is shown and JRelix is ready to accept user inputs.

2.2 Commands

In this section, the most commonly used commands of *JRelix* are presented.

`pr Expression` Display the result of a relational expression.

`time` Time on/off interpreter.

`trace` Log on/off.

`sr (<IDENTIFIER>)?` Display the description of the relation identifier. All relations are shown if identifier is omitted.

`sd (<IDENTIFIER>)?` Display the description of the attribute identifier. All attributes are shown if identifier is omitted.

`dd IDList` Remove the attributes specified in IDList.

`dr IDList` Remove the relations, views or computations specified in IDList.

`quit` Exit the system. JRelix performs clean-up procedure and saves the information before it returns the original operating system.

2.3 Declarations

2.3.1 Domain Declaration

Use the following syntax to declare domains:

`domain` *`IDLIST Type;`*

where domain is a keyword and *IDLIST* specifies the list of domains being declared, Type donates the types of these domains. In the current JRelix system there are two kinds of domain types: atomic and complex. The atomic types are primary such as **integer, string, long, double,** etc. Nine atomic data types for domains declaration are shown in Figure 2.3.1.

Type	Short Form	Corresponding Java Type
integer	intg	signed int, 4 bytes
short	short	signed short int, 2 bytes
long	long	singed long, 8 bytes
double	double	singed double, 8 bytes
float	float	signed float, 4 bytes
string	strg	String
boolean	bool	true,false
attribute	attr	String
universal	univ	String
numeric	num	

Figure 2.3.1 Atomic domain types in *JRelix*

In addition, two complex domain types have been implemented in JRelix: i.e. nested relation and computation. In general, the syntax used to declare nested domains is as follows:

```
domain nest_domain_name(domName1,domName2…);
```

where the *nest_domain_name* specifies the name of the nested domain being declared and the domain list that the nested relational domain contains are present in the following bracket.

As well, the following syntax is to declare a computational domain:

```
domain comp_domain_name comp(domName1,domName2…);
```

Note that the current JRelix implementation required that the attributes on which a nested attribute is defined must be declared already, so the recursively defined nested attributes were now allowed. The current implementation allows them. The details about recursively defined nested attributes will be presented in next chapter.

Figure 2.3.2 gives some examples of declaring both atomic-type domains and complex-type domains.

```
>domain a intg;
>domain b float;
>domain name strg;
>domain N(name,b);      <<nested domain>>
>domain C comp(a,b);    <<computational domain>>
```

Figure 2.3.2. Examples of domain declaration

2.3.2 Relation Declaration and Initialization

The following syntax is used to declare and initialize relations

`relation` ***IDLIST "(" IDLIST ")" (Initialization)?***

where the first IDLIST specifies the name of a relation to be declared or initialized. The domain list that the relation being declared contains is specified in the second *IDLIST.* When initialization is absent, an empty relation is declared without any tuple data inside, otherwise, a relation is declared with actual data tuples, called *relation initialization.* The most often used initialization is the so called curly bracket syntax in which relations start and terminate with curly bracket { and }, while their tuples are surrounded with round bracket (and). Also, the use of the name of another relation can initialize a relation.

For a nested relation, surrogates are used to replace actual values of nest attributes. The actual data for a nest attribute are stored in a relation with additional attribute *.id* which function is to link surrogates of the attributes in its parent relation. The name of the relation is the name of the nested attributes prefixed with a dot(.).

The following is the examples of relational declaration and initialization.

In the example presented in Figure 2.3.3, the relation *EmployeeInfo* contains three domains, *name, department* and *ContactInfo.* The type of the domain *name* and *department* is *strg,* while *ContactInfo* is a nested domain, which is defined on *email* and *fax*.

```
EmployeeInfo(name      department       ContactInfo                          )
                                       (email                        fax     )
              Jane    Administration    jane8@chronogen-inc.com       288-1111
              Ying          IT          yingw@chronogen-inc.com       288-2222
                                        yingw@yahoo.com               288-3333
              Patrick       IT          patrick@chronogen-inc.com     288-9091
```

```
>domain name,email,fax strg;
>domain department strg;
>domain ContactInfo(email,fax);
>relation EmployeeInfo(name,department, ContactInfo)<-
           {("Jane","Administration",{("jane8@chronogen-inc.com","288-1111")}),
            ("Ying","IT",{("yingw@chronogen-inc.com","288-2222"),
                     ("yingw@yahoo.com","288-3333")}),
            ("Patrick","IT",{("patrick@chronogen-inc.com","288-9091")})};
```

Figure 2.3.3 Initialization of relation *EmployeeInfo*

To display the contents of the relation, the command *pr* can be used.

```
pr EmployeeInfo;
```

The relation *EmployeeInfo* is shown in Figure 2.3.4. Since *ContactInfo* is a nested domain, *.ContactInfo* is generated to store all the *ContactInfo* data. Note that in relation *EmployeeInfo,* surrogates *608*, *609*and *610* for nested attribute *Contact* link the values *608,609* and *610* of attribute *.id* in relation *.ContactInfo.*

```
>pr EmployeeInfo;
+----------------------+----------------------+----------------------+
| name                 | department           | ContactInfo          |
+----------------------+----------------------+----------------------+
| Jane                 | Administration       | 608                  |
| Patrick              | IT                   | 610                  |
| Ying                 | IT                   | 609                  |
+----------------------+----------------------+----------------------+
relation EmployeeInfo has 3 tuples

>pr .ContactInfo;
+----------------------+----------------------+----------------------+
| .id                  | email                | fax                  |
+----------------------+----------------------+----------------------+
| 608                  | jane8@chronogen-inc. | 288-1111             |
| 609                  | yingw@chronogen-inc. | 288-2222             |
| 609                  | yingw@yahoo.com      | 288-3333             |
| 610                  | patrick@chronogen-in | 288-9091             |
+----------------------+----------------------+----------------------+
relation .ContactInfo has 4 tuples
```

Figure 2.3.4. Contents of relation *EmployeeInfo*

2.4 Assignment

JRelix provides two assignment operators, which are replacement (<–) and incremental assignment (<+). The replacement operators completely replace the left-hand relation that may have been defined or not. The data in the right-hand relation is copied into the left-hand relation. The incremental assignment adds new tuples and the attributes of the right-hand relation must be compatible with those of the relation on the left. The renaming assignment allows attributes on the left to be matched with the attributes on the right. The syntax for assignment is shown below:

```
Identifier (“<–“|“<+” ) Expression
```

or

```
Identifier  “[“ IDList(“<–“|“<+”ExpressionList “)” Expression
```

2.5 Relational Algebra

In this section, the syntax and semantics of the relational algebra are presented. Firstly, we describe the unary operators and binary operators. Assignment and incremental assignment operations are then described.

2.5.1 Unary Operators

There are six unary operators, including projection, selection, T-selection, QT-selections implemented in JRelix system.

Projection

The syntax for projection is as follows:

```
“[“ (IDList)? “]” in (Projection | Selection)
```

Projection extracts a subset of attributes named in *IDList* from a source relation. Duplicate tuples in the result relation are removed. If *IDList* is absent, a relation containing only one tuple with a boolean domain “.bool” is projected. The value of the boolean domain is true if the relation resulting from the projection has at least one tuple, and otherwise the value is false. Figure 2.5.1 and Figure 2.5.2 show examples of projection.

Projection query1: *Retrieve the attribute name in relation EmployeeInfo*

```
>employeeName<-[name] in EmployeeInfo;
+----------------------+
| name                 |
+----------------------+
| Jane                 |
| Patrick              |
| Ying                 |
+----------------------+
relation employeeName has 3 tuples
```

Figure 2.5.1. Retrieve the attribute *name* from relation *EmployeeInfo*

Projection query 2: *Check if there is any tuple in relation EmployeeInfo*

```
>pr [] in EmployeeInfo;
+--------+
| .bool  |
+--------+
| true   |
+--------+
expression has 1 tuple
```

Figure 2.5.2. Check if there is any tuple in relation *EmployeeInfo*

Selection

Selection is used to return a subset of a source relation that satisfies certain conditions. The syntax for selection is as follows:

```
where  SelectClause in Projection
```

where *selectClause* specify the certain conditions that the result relations must satisfy. A example of selection is shown in Figure 2.4.3.

Selection query 1: *Retrieve the tuples of relations EmployeeInfo where department is "IT".*

```
>IT<- where department = "IT" in EmployeeInfo;
>pr IT;
+----------------------+----------------------+----------------------+
| name                 | department           | ContactInfo          |
+----------------------+----------------------+----------------------+
| Patrick              | IT                   | 610                  |
| Ying                 | IT                   | 609                  |
+----------------------+----------------------+----------------------+
relation IT has 2 tuples
```

Figure 2.5.3. Retrieve the tuples of relation *EmployeeInfo* where *department* is *"IT"*.

T-Selection

T-selection is a combination of projection and selection. The general syntax for the T-Selection is

```
"["(IDList)?"]" where SelectClause in Projection
```

Figure 2.4.4 gives a example of T-selection using the relation *EmployeeInfo*.

T-Selection query 1: *Find all employees of the department "IT"*

```
>ITEmployee<-[name] where department = "IT" in EmployeeInfo;
>pr ITEmployee;
+----------------------+
| name                 |
+----------------------+
| Patrick              |
| Ying                 |
+----------------------+
relation ITEmployee has 2 tuples
```

Figure 2.5.4. Find *employees* of the *department "IT"*

2.5.2 Binary Operators

There are two categories: μ-join and σ-join. μ-joins are set operations generalized to relations, and σ-joins generalize logical operations.

The syntax of join operators is as following:

```
Expression JoinOperator Expression
or
Expression "[" ExpressionList ":" JoinOperator (":")?
ExpressionList "]" Expression
```

In the first production, the common attributes of both sides are joined attributes. While in the second production, users can select the common attributes to be joined attributes.

μ-join

μ-join are used as set operations including union, intersection and difference. The μ-join can be defined in term of the left wing, the center wing and the right wing. The definitions of them are as following [Mer84]:

For relations $R(X,Y)$ and $S(Y,Z)$ sharing a common attribute set, Y

center $\equiv \{(x,y,z)|(x,y) \in R \wedge (y,z) \in S\}$

left $\equiv \{(x,y,DC)|(x,y) \in R \wedge \forall z,(y,z) \notin S\}$

right $\equiv \{(DC,y,z)|(y,z) \in S \wedge \forall x,(x,y) \notin R\}$

For relations $R(W,X)$ and $S(Y,Z)$ sharing no common attribute set

center $\equiv \{(w,x,y,z)|(w,x) \in R \wedge (y,z) \in S \wedge x = y\}$

left $\equiv \{(w,x,y,DC)|(w,x) \in R \wedge x = y \wedge \forall z,(y,z) \notin S\}$

right $\equiv \{(DC,x,y,z)|(y,z) \in S \wedge x = y \wedge \forall x,(x,y) \notin R\}$

The description of μ-join is summarized as following:

ijoin or **natjoin** ≡ center

ujion ≡ left wing ∪ center ∪ right wing

ljoin ≡ left wing ∪ center

rjoin ≡ center ∪ right wing

djoin or **dljoin** ≡ left wing

drjoin ≡ right wing

sjoin ≡ left wing ∪ right wing

For more details please refer to [Mer84]

σ-join

The σ-join extends truth-valued comparison operation on sets to relations by applying them to each set of values of join attribute for each of other values in the two relations[Mar84].

Given relations R(W, X) and S(Y, Z), R_w is the set of values of X associated by R with a given value, w, of W, and S_z is the set of values of Y associated by S with a given value, z, of Z are sets of attributes of S, the following definitions are general, and even allow for X and Y to be the same set of attributes. X and Y must by at least compatible attribute sets[Mer84].

$$R \textbf{ sup } S \equiv \{(w, z) \mid R_w \supseteq S_z\}$$

$$R \textbf{ sep } S \equiv \{(w, z) \mid R_w \not\supseteq S_z\}$$

$$R \textbf{ gtjoin } S \equiv \{(w, z) \mid R_w \supset S_z\}$$

$$R \textbf{ eqjoin } S \equiv \{(w, z) \mid R_w = S_z\}$$

$$R \textbf{ lejoin } S \equiv \{(w, z) \mid R_w \subseteq S_z\}$$

$$R \textbf{ ltjoin } S \equiv \{(w, z) \mid R_w \subset S_z\}$$

2.6 Domain Algebra

The algebra on attributes is called the domain algebra and contains two main components: scalar operations and aggregate operations. In the table view of relations, these can be thought of as “horizontal” and “vertical” operations [Mer84]. Horizontal domain operations work within the tuples, while vertical domain operations work among tuples.

2.6.1 Scalar Operations

Scalar operations work on a single tuple of a relation. In the current *JRelix* system, scalar operations include constant definition, renaming, arithmetic function, conditional statements etc. All these basic scalar operations are listed in Figure 2.6.1.

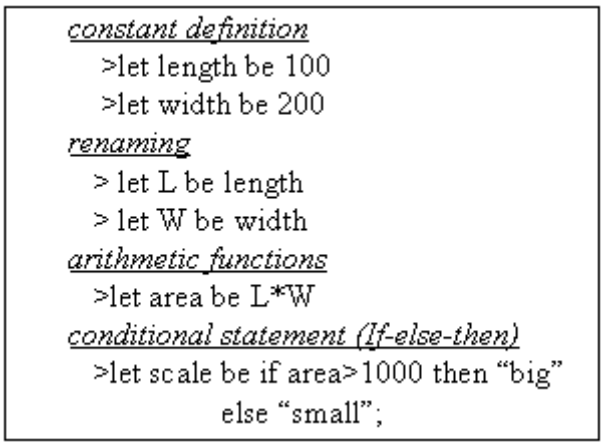


Figure 2.6.1. Scalar operations

2.6.2 Aggregate Operations

Aggregate operations, often referred as vertical operations, work on attribute values of all tuples in a relation. Basic vertical operations are listed as follows:

- Reduction
- Equivalence reduction
- Functional mapping
- Partial functional mapping

The examples used to illustrate these operations are shown in Figure 2.6.2.

```
domain product,Saledept strg;
domain saleAmount intg;
relation SaleInfo(product,Saledept,saleAmount)<-
        {("IBM", "Computer",20000),
         ("Dell", "Computer",10000),
         ("Canon", "Camera",2000),
         ("Kodak", "Camera",3000),
         ("JVC", "TV",5000)
        };

>let total be red + of saleAmount;

>let totalbyDept be equiv + of saleAmount by Saledept;

>saleData<-[Saledept,totalbyDept,total] in SaleInfo;

>pr saleData;
+----------------------+-------------+-------------+
| Saledept             | totalbyDept | total       |
+----------------------+-------------+-------------+
| Camera               | 5000        | 40000       |
| Computer             | 30000       | 40000       |
| TV                   | 5000        | 40000       |
+----------------------+-------------+-------------+
relation saleData has 3 tuples

>let rank be fun + of 1 order saleAmount;
>saleRank<-[Saledept,product,saleAmount,rank] in SaleInfo;
>pr saleRank;
+----------------------+----------------------+-------------+-------------+
| Saledept             | product              | saleAmount  | rank        |
+----------------------+----------------------+-------------+-------------+
| Camera               | Canon                | 2000        | 1           |
| Camera               | Kodak                | 3000        | 2           |
| Computer             | Dell                 | 10000       | 4           |
| Computer             | IBM                  | 20000       | 5           |
| TV                   | JVC                  | 5000        | 3           |
+----------------------+----------------------+-------------+-------------+
relation saleRank has 5 tuples

>let deptRank be par + of 1 order saleAmount by Saledept;

>DeptRank<-[Saledept,product,saleAmount,deptRank] in SaleInfo;
>pr DeptRank;
+----------------------+----------------------+-------------+-------------+
| Saledept             | product              | saleAmount  | deptRank    |
+----------------------+----------------------+-------------+-------------+
| Camera               | Canon                | 2000        | 1           |
| Camera               | Kodak                | 3000        | 2           |
| Computer             | Dell                 | 10000       | 1           |
| Computer             | IBM                  | 20000       | 2           |
| TV                   | JVC                  | 5000        | 1           |
+----------------------+----------------------+-------------+-------------+
relation DeptRank has 5 tuples
```

Figure 2.6.2. Examples of vertical operations

For more information about the domain algebra, please refer to [Mer84, Yua98].

2.7 Nesting

The relational algebra and domain algebra can be applied to relation-valued attributes in nested relations that are an expanded data structure, where a value of an attribute can be a

relation. Generally speaking, there is no new syntax for nested relations; we just subsume the relational algebra into domain algebra.

Unnesting operations and nesting operations are needed to raise and lower the levels of nesting. Since the nesting operation is still in a progress of implementation, here we are not going to discuss it.

An example is given to illustrate unnesting operations. To find all fax numbers of employees, we can use the query shown in Figure 2.7.1, but note that the result is itself a nested relation. To remove the nested structure of the result, called unnesting, two steps are followed. Firstly, do the reduction,

red ujoin of [*fax*] **in** *ContactInfo;*

Projecting the **red ujoin** still given a nested relation, but a singleton. The second step is to lift a level through anonymity (i.e., no giving the name of the attribute of result relation), by writing the reduction directly in a projection list.

AllFax'<-[**red ujoin of** [*fax*] **in** *ContactInfo*] **in** *EmployeeInfo*;

(The result shown in Figure 2.7.2)

Thus the system has no choice but to bring values of the attribute *fax* one level up, resulting in a single-level relation. The new syntax, syntactic sugar, which is likely to be frequently used in querying nested relations, has been implemented as a shorthand by using the / operator. So, the query can be

AllFax'<-*ContactInfo/ fax* **in** *EmployeeInfo*;

This is a path expression. Fully descripted in in chapter 3.

```
EmployeeInfo (name   department       ContactInfo                          )
                                      (email                        fax    )
              Jane   Administration   jane8@chronogen-inc.com       288-1111
              Ying       IT           yingw@chronogen-inc.com       288-2222
                                      yingw@yahoo.com               288-3333
              Patrick    IT           patrick@chronogen-inc.com     288-9091


let allFax be [fax] in ContactInfo;
AllFax<-[allFax] in EmployeeInfo;

>pr AllFax;
+----------------------+
| allFax               |
+----------------------+
| 611                  |
| 612                  |
| 613                  |
+----------------------+
relation AllFax has 3 tuples
>pr allFax;
+----------------------+----------------------+
| .id                  | fax                  |
+----------------------+----------------------+
| 611                  | 288-1111             |
| 612                  | 288-9091             |
| 613                  | 288-2222             |
| 613                  | 288-3333             |
+----------------------+----------------------+
relation .allFax has 4 tuples
```

Figure 2.7.1. All Fax Number. Version 1

```
AllFax'<-[red ujoin of [fax] in ContactInfo] in EmployeeInfo;

>pr AllFax';
+----------------------+
| fax                  |
+----------------------+
| 288-1111             |
| 288-2222             |
| 288-3333             |
| 288-9091             |
+----------------------+
relation AllFax' has 4 tuples
```

Figure 2.7.2. All Fax Number. Version 2

Although there are also important components such as views, update, computation, and so on in *JRelix*, we are not going to elaborate them here since they are not crucial for the implementations of semi-structured data input and recursive nesting.

Chapter 3

User's Manual

3.1 Semi-structured data input

Compared to conventional data, which is described by a scheme available to the database system separately from the data, semi-structured data is self-describing by embedding the scheme with the data by using markup language tags. The self-describing data structure is more flexible than in conventional relations. In the following example (Figure 3.1), the relation *EmployeeInfo*, will be used to demonstrate the rule of using semi-structured data input in *JRelix*.

3.1.1 An Example

EmployeeInfo(employeeName,	Id,	teleNum,	otherContactInfo) (FaxNum,	cell)
Ban	1001	235642	564215	dc
Patrick	dc	235643	dc	
Josee	dc	dc	dc	
dc	1002	987654	dc	564218

Figure 3.1.1 Example: Relation *EmployeeInfo*

The semi-structure input corresponding to the relation *EmployeeInfo* is shown in Figure 3.2.

```
relation EmployeeInfo <-<EmployeeInfo>
                            <employeeName type = strg>Ban</employeeName>
                            <Id type = strg>1001</Id>
                            <teleNum type = intg>235642</teleNum>
                            <otherContactInfo>
                              <FaxNum type = intg>564215</FaxNum>
                            </otherContactInfo>

                            <employeeName>Patrick</employeeName>
                            <teleNum>235643</teleNum>

                            <employeeName>Josee</employeeName>

                            <.tuple>
                            <Id>1002</Id>
                            <teleNum>987654</teleNum>
                            <otherContactInfo>
                               <cell type = intg>564218</cell>
                            </otherContactInfo>
                            </.tuple>
                          </EmployeeInfo>;

>pr EmployeeInfo;
+----------------------+----------------------+-------------+----------------------+
| employeeName         | Id                   | teleNum     | otherContactInfo     |
+----------------------+----------------------+-------------+----------------------+
|  dc                  | 1002                 | 987654      | 560                  |
| Ban                  | 1001                 | 235642      | 559                  |
| Josee                |  dc                  | dc          | dc                   |
| Patrick              |  dc                  | 235643      | dc                   |
+----------------------+----------------------+-------------+----------------------+
relation EmployeeInfo has 4 tuples

>pr .otherContactInfo;
+----------------------+-------------+-------------+
| .id                  | FaxNum      | cell        |
+----------------------+-------------+-------------+
| 559                  | 564215      | dc          |
| 560                  | dc          | 564218      |
+----------------------+-------------+-------------+
relation .otherContactInfo has 2 tuples
```

Figure 3.1.2 Example: Semi-structure input of Relation *EmployeeInfo*

3.1.2 Domain Declaration

The new syntax of domain declaration for the semi-structured data input is described below.

Since the semi-structure data is self-describing, it is not necessary to define the domains used in a relation before the semi-structured input of the relational initialization (schemaless). The domains are declared in their first occurrence in the semi-structured input of the relation initialization using the following syntax:

<domainName ***type*** *= data_type>... ...</domainName>*

Where

- *domainName* is the name of the new domain where type is being defined.
- The start tag is surrounded by angle brackets, while the end tag has the angle bracket and a slash "/". The *domainName* in the start tag and in the end tag should be the same.
- The text after " type = " represents the data type of the new domain.
- If the type of domain is not specified, the default type is *strg* (String).
- The text between the start tag and the end tag is the data value of the domain. Since the type of the domains is illustrated in its start tag, the values of the domains, whose type is *strg*, do not have to be surrounded by quotes.
- For subsequent occurrence of the same domains, the following syntax is used

  <domainName>... ...</domainName>
- For a nested domain, it is not necessary to specify the names of attributes that the nested domains contain. Similar to the initialization of a relation, the text between the start tag of a nest domain and its end tag is the initialization of the nested domain. In the example shown above, the nested domain *otherContactInfo* is initialized by the following text:

```
<otherContactInfo>
  <FaxNum type = intg>564215</FaxNum>
</otherContactInfo>
 ... ...
 <otherContactInfo>
    <cell type = intg>564218</cell>
 </otherContactInfo>
```

Figure 3.1.3 Initialization of nested domain *otherContactInfo*

3.1.3 Relation Declaration and Initialization

The relation declaration syntax for loading semi-structured data is:

relation *IDList* ***Initialization***

where

- The first *IDList* specifies the name of the relation being declared and the

production *Initialization* is the angle bracket syntax in which the relation starts and terminates with first angle bracket < and the last angle bracket >.

- Specification of the attributes on which relations are defined are omitted, since the semi-structured data input contains the information about the attributes of the relations to be initialized.
- The name of the relation that is to be initialized is not necessarily the same as the first tag and its corresponding end tag in its semi-structure data input.
  relation R<-<R1>…… </R1>;
  is acceptable.

In addition, to take further advantage of the semi-structured data, some entries for relations can be missed in their semi-structured data input for their initialization, since each entry has its own tags to describe it. As in the example relation *EmployeeInfo*, some entries are missed and in this situation the null values (dc) are added for the missing entries. For the nested relation, "dc" can be added automatically, if it is necessary, to terminate the initialization.

Furthermore, the tags *<.tuple> and </.tuple>*, used to separate tuples, are optional. However, they are compulsory when there is ambiguity. If *<.tuple> and </.tuple>* are missed, the reoccurrence of the same domains will be considered to be in a new tuple. And the separations of tuples are also dependent on the occurrence order of domains in the input data. To explain this more clearly, figure 3.1.4 illustrates a relation initialized by different cases of semi-structured data.

Refer back to the example in section 3.1.1. Both "*Patrick*" and "*Josee*" have the same attribute *employeeName,* so it would be regarded as being in two different tuples. But, since *Id* is just after *EmployeeInfo* in the domain list of the relation, if "*2002*" is not the *Id* of "*Josee*", *<.tuple>* and *</.tuple>* have to be used to avoid ambiguity, since without the tags "*Josee*" "*2002*" would be considered as being two domain entries in a same tuple.

Note that relations initialized with semi-structure data input are the same as with the

relations initialized with a curly bracket input. All relational algebra and domain algebra can be performed.

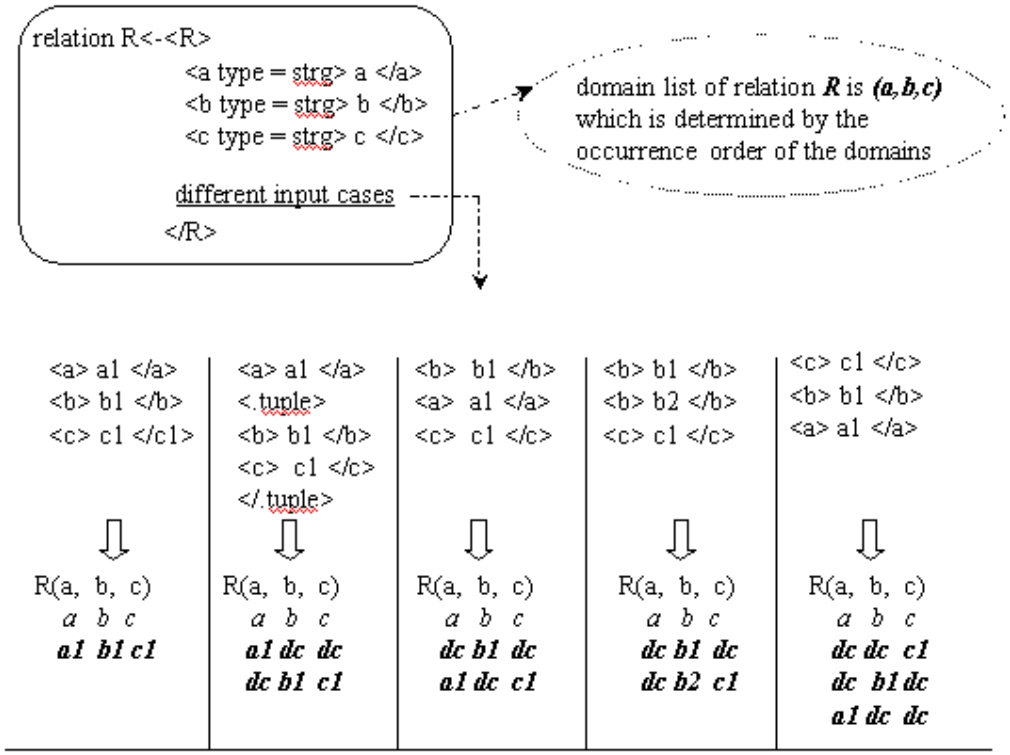


Figure 3.1.4 Example of different cases for relation R

3.2 Input Data From a File

To simplify the edition of input semi-structured data, a relation schema can be edited in a file and saved as a text file. Then a relation can be declared and initialized using the following syntax:

relation *rel_name* <- **"** file_path/ file_name **"**;

It is not necessary for the relation name to be same as the relation name in the file.

Take the relation *EmployeeInfo* as an example: firstly we edited the input of the relation *EmployeeInfo* in a file and saved it as "Emp_xml.txt" (Figure 3.2.1):

Emp_xml.txt:

```
<EmployeeInfo>
   <employeeName type = strg>Ban</employeeName>
   <Id type = strg>1001</Id>
   <teleNum type = intg>235642</teleNum>
   <otherContactInfo>
     <FaxNum type = intg>564215</FaxNum>
   </otherContactInfo>

   <employeeName>Patrick</employeeName>
   <teleNum>235643</teleNum>

   <employeeName>Josee</employeeName>

   <.tuple>
   <Id>1002</Id>
   <teleNum>987654</teleNum>
   <otherContactInfo>
      <cell type = intg>564218</cell>
   </otherContactInfo>
   </.tuple>
</EmployeeInfo>
```

Figure 3.2.1. Emp_xml.txt file for initialization of relation *EmployeeInfo*

Then we may declare the relation *EmployeeInfo* and initialize it with the following:
relation *EmployeeInfo* **<-"***Emp_xml.txt***";**

This initialization mechanism can also be used to declare and initialize relations where initialization requires curly bracket syntax. The only difference from the above case is that in this case the attributes on which the relation is defined should be specified. See the example in the following figure 3.2.2:

Emp_curly_bracket.txt:

```
{("Ban","1001",235642,{(564215,dc)}),
 ("Patrick",dc,235643,dc),
 ("Josee",dc,dc,dc),
 (dc,"1002",987654,{(dc,564218)})}
```

Figure 3.2.2. Emp_curly_bracket.txt file for initialization of relation *EmployeeInfo*

Then the relation can be initialized by the following:

relation *EmployeeInfo(employeeName,Id,teleNum, otherContactInfo)*

<- "Emp_curly_bracket.txt";

Note that in this case, it is necessary for domains to be declared before the relation initialization.

3.3 Recursive Nesting Declaration and Initialization

In recursive nesting, a relation name can be an attribute of itself.

Declaration of recursively nested attributes uses the syntax for nested domain declaration

domain *IDList "(" IDList ")"*

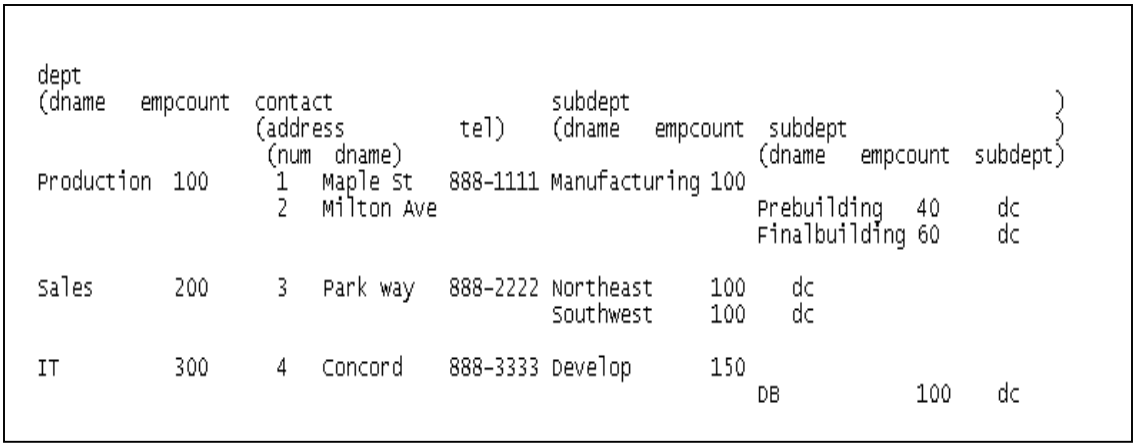
Here *IDList* specifies the name of a nested attribute being declared, *"(" IDList ")"* is used to specify the names of attributes on which the nested attribute is defined. The former Relix required that the attributes on which a new nested attribute is defined must be already defined. Therefore recursively defined nested attributes were not allowed. Now, recursively defined nested attributes are allowed, meaning the name of an attribute on which a nested attribute is defined can be the same as the name of the nested attribute. For example, the following is permitted in order to declare a nested attribute *N* and a relation containing the nested attribute.

domain *A data_type;*

domain *N(A,N);*

relation *R(N)*←……;

In the following sub-section, I will illustrate a recursive nesting example. The relation *"dept"* contains a recursive domain "*subdept* " to indicate the hierarchical structure involved in departments. (Figure 3.3.1)



```
dept
(dname   empcount  contact                  subdept                                  )
                   (address          tel)    (dname   empcount  subdept               )
                    (num  dname)                                (dname   empcount  subdept)
Production  100     1   Maple St   888-1111 Manufacturing 100
                    2   Milton Ave                             Prebuilding   40     dc
                                                               Finalbuilding 60     dc

Sales       200     3   Park way   888-2222 Northeast     100    dc
                                            Southwest     100    dc

IT          300     4   Concord    888-3333 Develop       150
                                                               DB            100    dc
```

Figure 3.3.1: Contents of the relation *Dept*

Similar to the non-recursive nested relation, the relation can be initialized as illustrated in Figure 3.3.2.

```
domain dname strg;
domain empcount intg;
domain num intg;
domain tel strg;
domain address(num,dname);
domain contact(address,tel);
domain subdept(dname,empcount,subdept);

relation dept(dname,empcount,contact,subdept)<-
        {("Production",100,
                        {({(1,"Maple St"),(2,"Milton Ave")},"888-1111")},
                        {("Manufacturing",100,
                                {("Prebuilding",40,dc),("Finalbuilding",60,dc)})})},
         ("Sales",200,
                        {( {(3,"Park way")},"888-2222")},
                        {("Northeast",100,dc),("Southwest",100,dc)}),
         ("IT", 300,
                       {( {(4,"Concord")},"888-3333")},
                       {("Develop",150,
                                {("DB",100,dc)})}
        )};

>pr dept;
+----------------------+-------------+----------------------+----------------------+
| dname                | empcount    | contact              | subdept              |
+----------------------+-------------+----------------------+----------------------+
| IT                   | 300         | 10                   | 12                   |
| Production           | 100         | 3                    | 5                    |
| Sales                | 200         | 7                    | 9                    |
+----------------------+-------------+----------------------+----------------------+
relation dept has 3 tuples


>pr .subdept;
+----------------------+----------------------+-------------+----------------------+
| .id                  | dname                | empcount    | subdept_0            |
+----------------------+----------------------+-------------+----------------------+
| 5                    | Manufacturing        | 100         | 6                    |
| 9                    | Northeast            | 100         | dc                   |
| 9                    | Southwest            | 100         | dc                   |
| 12                   | Develop              | 150         | 13                   |
+----------------------+----------------------+-------------+----------------------+
relation .subdept has 4 tuples


>pr .subdept_0;
+----------------------+----------------------+-------------+----------------------+
| .id                  | dname                | empcount    | subdept_1            |
+----------------------+----------------------+-------------+----------------------+
| 6                    | Finalbuilding        | 60          | dc                   |
| 6                    | Prebuilding          | 40          | dc                   |
| 13                   | DB                   | 100         | dc                   |
+----------------------+----------------------+-------------+----------------------+
relation .subdept_0 has 3 tuples

>pr .contact;
+----------------------+----------------------+----------------------+
| .id                  | address              | tel                  |
+----------------------+----------------------+----------------------+
| 3                    | 4                    | 888-1111             |
| 7                    | 8                    | 888-2222             |
| 10                   | 11                   | 888-3333             |
+----------------------+----------------------+----------------------+
relation .contact has 3 tuples


>pr .address;
+----------------------+-------------+----------------------+
| .id                  | num         | dname                |
+----------------------+-------------+----------------------+
| 4                    | 1           | Maple St             |
| 4                    | 2           | Milton Ave           |
| 8                    | 3           | Park way             |
| 11                   | 4           | Concord              |
+----------------------+-------------+----------------------+
relation .address has 4 tuples
```

Figure 3.3.2. Initialization of a recursive nesting relation and its attributes

Note that null value (*dc*) must be added to terminate the recursion. Finally, it is necessary to mention that the names *subdept*, *subdept_0* and *subdept_1* are automatically created by *JRelix* to represent the hierarchical structure of the entries. The reason to implement recursive nesting in this way is that in current JRelix implementation, relation names and relational domain's names are stored in the form of *hashTable* and the names are used as keys. However, it is not necessary for users to know the name *subdept_n*. As illustrated in Figure 3.3.3, *subdept* can represent any level of the recursive nested attribute *subdept* in queries.

```
query<-[red ujoin of [red ujoin of subdept] in subdept] in dept;
>pr query;
+----------------------+-------------+----------------------+
| dname                | empcount    | subdept_1            |
+----------------------+-------------+----------------------+
| DB                   | 100         | dc                   |
| Finalbuilding        | 60          | dc                   |
| Prebuilding          | 40          | dc                   |
+----------------------+-------------+----------------------+
relation query has 3 tuples
```

Figure 3.3.3. An example query for the recursive nested attribute

Recursively nested attributes can also be initialized in semi-structured data format. Figure 3.3.4 shows a small example where a relation R containing a recursively nested attribute is initialized with a semi-structured data input. Note that in the semi-structured input the null value (*dc*) will be added automatically to terminate the recursion.

```
relation familyTree<-<familyTree>
                        <Name type = strg>Bill</Name>
                        <Children>
                          <Name>Jack</Name>
                          <Children>
                             <Name>Smith</Name>
                          </Children>
                          <Name>Susan</Name>
                         </Children>
                       </familyTree>;

>pr familyTree;
+---------------------+---------------------+
| Name                | Children            |
+---------------------+---------------------+
| Bill                | 603                 |
+---------------------+---------------------+
relation familyTree has 1 tuple

>pr .Children;
+---------------------+---------------------+---------------------+
| .id                 | Name                | Children_0          |
+---------------------+---------------------+---------------------+
| 603                 | Jack                | 604                 |
| 603                 | Susan               | dc                  |
+---------------------+---------------------+---------------------+
relation .Children has 2 tuples

>pr .Children_0;
+---------------------+---------------------+---------------------+
| .id                 | Name                | Children_1          |
+---------------------+---------------------+---------------------+
| 604                 | Smith               | dc                  |
+---------------------+---------------------+---------------------+
relation .Children_0 has 1 tuple
```

Figure 3.3.4. Initialization for the recursive nested attribute

3.4 Path Expression Operator

Path expression will concatenate attribute names into a path by using the **"/"** operation. The syntax in using path expression is as follows:

[*rel_name/****] (****nested_rel_name "/")*[*** *domain_name****]***

We take the example relation *dept* shown in the last section to illustrate the syntactic sugar. Let's start with the following: suppose we would like to project the nested attribute *address,* instead of

addr<-[**red ujoin of** [**red ujoin of** *address*] **in** *contact*] **in** *dept;*

The **red ujoin of** raises the level of nesting as in section 2.7. Now, the simpler format

addr<-dept/contact/address

can be used. The result is listed in figure 3.4.1.

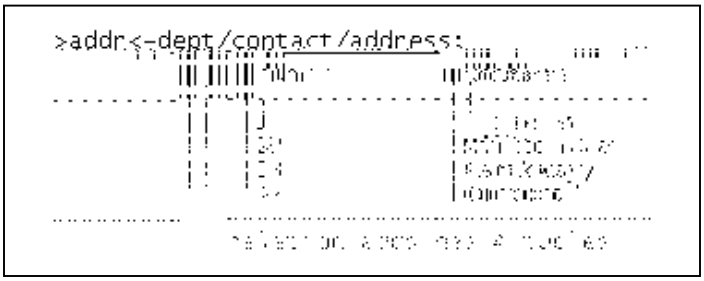


Figure 3.4.1 *Address* of *dept*

We can go further,

dname<-[**red ujoin of** [**red ujoin of** [*dname*] **in** *address*] **in** contact] **in** *dept;*

becomes

*dname<-dept/contact/address/dname**;*** (Figure 3.4.2)

```
>dname<-dept/contact/address/dname;
+----------------------+
| dname                |
+----------------------+
| Concord              |
| Maple St             |
| Milton Ave           |
| Park way             |
+----------------------+
relation dname has 4 tuples
```

Figure 3.4.2: *dname* of *address*

For the recursive nested attribute *subdept,*

*query<-dept/subdept/subdept**;***

can lift *subdept* on the second level to the top-level relation *dept* (Figure 3.4.3).

```
query<-dept/subdept/subdept;
>pr query;
+----------------------+-------------+----------------------+
| dname                | empcount    | subdept_1            |
+----------------------+-------------+----------------------+
| DB                   | 100         | dc                   |
| Finalbuilding        | 60          | dc                   |
| Prebuilding          | 40          | dc                   |
+----------------------+-------------+----------------------+
relation query has 3 tuples
```

Figure 3.4.3. Projection of the recursive nested attribute *subdept*

Furthermore, the path operator can be at any end of a regular T-selection. The two queries below (Figure 3.4.4) illustrate the path operator used in projection and relational expression.

```
Num<-[num] where dname="Maple St" in dept/contact/address;
+-------------+
| num         |
+-------------+
| 1           |
+-------------+
relation Num has 1 tuple


dName<-contact/address/dname in dept;
+----------------------+
| dname                |
+----------------------+
| Concord              |
| Maple St             |
| Milton Ave           |
| Park way             |
+----------------------+
relation dName has 4 tuples
```

Figure 3.4.4. Path operator in projection

To produce multiple attributes from a deeper level is also easy by using path operator. For instance,

addrs<-[num,dname] ***in*** *dept/contact/address;*

will project attributes *num* and *dname* from the deeper level *address.* The output is shown below.

```
addrs<-[num,dname] in dept/contact/address;
+-------------+----------------------+
| num         | dname                |
+-------------+----------------------+
| 1           | Maple St             |
| 2           | Milton Ave           |
| 3           | Park way             |
| 4           | Concord              |
+-------------+----------------------+
relation addrs has 4 tuples
```

Figure 3.4.5. Multiple attributes in projection

In addition, the path operator can be used in selection. We have the query: Find all *dname* in *dept* where *dname* in its *contact/address* is *"Park way"*, the query

dns<-[dname] ***where*** *contact/address/dname ="Park way"* ***in*** *dept;*

that can be expanded as:

dns<-[dname] ***where*** *([]* ***where*** *([]* ***where*** *dname="Park way"* ***in*** *address)*
in *contact)* ***in*** *R;*

will answer the query**.**

Suppose we want to find all *tel*s in *contact* where *dname* in *address* is *"Park way"*. The query

telp<-[tel] ***where*** *address/dname="Park way"* ***in*** *dept/contact;*

will complement the task. Outputs of these two examples are shown below.

```
dns<-[dname] where contact/address/dname ="Park way" in dept;
+----------------------+
| dname                |
+----------------------+
| Sales                |
+----------------------+
relation dns has 1 tuple

telp<-[tel] where address/dname="Park way" in dept/contact;
+----------------------+
| tel                  |
+----------------------+
| 888-2222             |
+----------------------+
relation telp has 1 tuple
```

Figure 3.4.6. Path operator in selection

Finally, the queries listed below will produce the same results:

*dept/contact/address/dname****;***

or

contact/address/dname ***in*** *dept;*

or

address/dname ***in*** *dept/contact;*

or

[dname] ***in*** *dept/contact/address;*

3.5 Regular Expression Operators

The regular expressions, similar to the regular expressions of XML, provide Kleene star(*), plus operator(+),question mark(?), and dot operator (.).

3.5.1 Kleene Star (*) and Plus Operator (+)

The Kleene star is involved for the recursive nesting in order to answer such queries as "find all *some attribute(s) of a recursive nesting domain* from *a relation or recursively nested relation*".

The syntax for Kleene star in projection or relational expression is the following:

*rel_name****(****/recursivelyNested_attribute_name****)*[****/attribuete_name****]*

or

*rel_name/****(****recursivelyNested_attribute_name/****)****attribuete_name*

The syntax for Kleene star in selection is the following:

where *[rel_name/]****(*** *recursivelyNested_attribute_name/****)****attribute_name* ***"="*** *value*

where *rel_name* is the names of relation or nested domains which contains a recursively nested attribute, while *recursivelyNested_attribute_name* is the names of recursively nested attributes and *attribute_name* is the names of domains to be projected from the recursively nested attribute.

Since Kleene star indicates zero or many occurrences of its operand [Mer03], the consequence of the expression should be the projection of the attribute whose name is *attribute_name* from all levels of the recursively nested relation where the name is *recursivelyNested_attribute_name* and from the top_level relation which is *rel_name***.**

Below I will display the use of the syntax with Kleene star by using the example relation *dept*. Firstly, if we would like to project *dname* from all levels of *subdept* and *dept,* the easiest way is:

dname<-dept(/subdept)/dname;*

It is obvious that it is actually the same as the following query:

dname<-dept/dname **ujoin** *dept/subdept/dname* **ujoin** *dept/subdept/subdept/dname;*

Apparently,

dname<-[dname] ***in*** *dept (/subdept)**;*

or

*dname<-(subdept/)*dname* ***in*** *dept;*

will produce the same result . The result is listed in Figure 3.5.1.

```
dnames<-dept(/subdept)*/dname;
dnames<-[dname] in dept(/subdept)*;
dnames<-(subdept/)*dname in dept;
+----------------------+
| dname                |
+----------------------+
| DB                   |
| Develop              |
| Finalbuilding        |
| IT                   |
| Manufacturing        |
| Northeast            |
| Prebuilding          |
| Production           |
| Sales                |
| Southwest            |
+----------------------+
relation dnames has 10 tuples
```

Figure 3.5.1. All *dname* in *dept* and *subdept*

The "+" operator represents one or many occurrences. Therefore, queries

dnameofSub<-dept(/subdept)+/dname;

or

dnameofSub<- [dname] ***in*** *dept/(subdept)+;*

or

dnameofSub<-(subdept)+/dname ***in*** *dept;*

will also project the *dname* from all levels of *subdept,* but do not project *dname* of *dept*.

Their output is listed in Figure 3.5.2.

```
dnameofSub<-dept(/subdept)+/dname;
dnameofSub<-[dname] in dept(/subdept)+;
dnameofSub<-(subdept/)+dname in dept;
+----------------------+
| dname                |
+----------------------+
| DB                   |
| Develop              |
| Finalbuilding        |
| Manufacturing        |
| Northeast            |
| Prebuilding          |
| Southwest            |
+----------------------+
relation dnameofSub has 7 tuples
```

Figure 3.5.2. All *dname* in *subdept*

In addition, to project all attributes from all levels of *subdept*,

allSubDept<- dept(/subdept)+;

is the simplest query to answer it.

While,

*allDept<-dept (/subdept)**

should produce all attributes from *dept* and from all levels of *subdept*.

In this example, relation *dept* and its nested attribute *subdept* have the common attribute *dname*. In this situation, all *dname* are projected from *dept* and from all levels of *subdept*. The result is shown in Figure 3.5.3.

```
allDept<-dept(/subdept)*;
+----------------------+-------------+----------------------+
| dname                | empcount    | contact              |
+----------------------+-------------+----------------------+
| DB                   | 100         | dc                   |
| Develop              | 150         | dc                   |
| Finalbuilding        | 60          | dc                   |
| IT                   | 300         | 8                    |
| Manufacturing        | 100         | dc                   |
| Northeast            | 100         | dc                   |
| Prebuilding          | 40          | dc                   |
| Production           | 100         | 1                    |
| Sales                | 200         | 5                    |
| Southwest            | 100         | dc                   |
+----------------------+-------------+----------------------+
relation allDept has 10 tuples

allSubDept<-dept(/subdept)+;
+----------------------+-------------+
| dname                | empcount    |
+----------------------+-------------+
| DB                   | 100         |
| Develop              | 150         |
| Finalbuilding        | 60          |
| Manufacturing        | 100         |
| Northeast            | 100         |
| Prebuilding          | 40          |
| Southwest            | 100         |
+----------------------+-------------+
relation allSubDept has 7 tuples
```

Figure 3.5.3. Projection of all attributes from the recursively nested relation

One point that should be mentioned is that these relations may be disjoint, that is, when top-level relations and recursively nested relations have no common attributes. In this case, N is a recursively nesting domain that contains A, B and R has domains X,Y, N. The query

q<-R(/N);*

that is,

q<-R ***ujoin*** *R/N* ***ujoin*** *R/N/N* ***ujoin…*** *R/N/N/../N;*

will be interpreted as a Cartesian product. A small example to illustrate this is shown in Figure 3.5.4.

```
domain X,Y intg;
domain A,B strg;
domain N(A,B,N);
relation R(X,Y,N)<-{(11,22,{("A","B",{("AA","BB",dc)})}),
                    (88,99,{("a","b",dc)})
                   }

q<-R(/N)*;
>pr q;
+-------------+-------------+----------------------+----------------------+
| X           | Y           | A                    | B                    |
+-------------+-------------+----------------------+----------------------+
| 11          | 22          | A                    | B                    |
| 11          | 22          | AA                   | BB                   |
| 11          | 22          | a                    | b                    |
| 88          | 99          | A                    | B                    |
| 88          | 99          | AA                   | BB                   |
| 88          | 99          | a                    | b                    |
+-------------+-------------+----------------------+----------------------+
relation q has 6 tuples
```

Figure 3.5.4. Projection of add attributes using Kleene Star

The benefits of exploring Kleene star in selections are obvious. A simple query,

deptName<-[dname] ***where (****subdept****)********empcount=100* ***in*** *dept****;***

will produce all *dname* from *dept* that has a *subdept* which *empcount* is *100* and *dname* from *dept* which *empcount* is *100* (Figure 3.5.5). Without using the Kleene Star, the query to answer the question will be too complicated to be performed. Firstly, users have to know how many levels the recursively nested attribute *subdept* contains. Then perform the long query listed below:

deptName<-[dname] ***where*** *empcount=100* ***in*** *dept* ***ujoin***

[dname] ***where*** *subdept/empcount=100* ***in*** *dept* ***ujoin***

[dname] ***where*** *subdept/subdept/empcount=100* ***in*** *dept;*

```
deptName<-[dname] where (subdept/)*empcount=100 in dept;
+----------------------+
| dname                |
+----------------------+
| IT                   |
| Production           |
| Sales                |
+----------------------+
relation deptName has 3 tuples
```

Figure 3.5.5. Kleene star in selection

Also keep in mind that

[dname] ***where (****subdept****)**** *empcount=100* ***in*** *dept;*

should include the *dname* from *dept,* where its *empcount* is 100, however, the expression

[dname] ***where (****subdept****)+*** *empcount=100* ***in*** *dept*;

that can be expanded to

[dname] ***where*** *subdept/empcount=100* ***in*** *dept* ***ujoin***

[dname] ***where*** *subdept/subdept/empcount=100* ***in*** *dept;*

only projects *dname* from *dept* that have a *subdept* where *empcount* is *100.*

Furthermore, the following Figure3.5.7 will show the use of "*" (or "+") both in selection and projection. The expression

Q1<-subdept100<-[dname] ***where (****subdept****)**** *empcount=100* ***in*** *dept****(****/subdept****)*;**

produces all *dname* from all levels of *subdept* which has a nested attribute *subdept* (no matter how many levels down) where *empcount* is *100* and all *dname* from *dept* where *empcount* is *100*. In comparison, the query Q2 will not produce all *dname* from *dept* where *empcount* is *100,* since in the example "+" is used instead of "*". In order to provide a clearer explanation, these queries have been expanded in the following figure 3.5.6.

Q1<-[dname] **where** (subdept/)*empcount=100 **in** dept(/subdept)*;

[dname] **where** empcount =100 **in** dept **ujoin**
[dname] **where** subdept/empcount =100 **in** dept **ujoin**
[dname] **where** subdept/subdept/empcount =100 **in** dept **ujoin**
[dname] **where** empcount =100 **in** dept/subdept **ujoin**
[dname] **where** subdept/empcount =100 **in** dept/subdept **ujoin**
[dname] **where** empcount =100 **in** dept/subdept/subdept;

Q2<-[dname] **where** (subdept/)+empcount=100 **in** dept(/subdept)+;

[dname] **where** subdept/empcount =100 **in** dept/subdept;

Q3<-[dname] **where** (subdept/)+empcount=100 **in** dept(/subdept)*;

[dname] **where** subdept/empcount =100 **in** dept **ujoin**
[dname] **where** subdept/subdept/empcount =100 **in** dept **ujoin**
[dname] **where** subdept/empcount =100 **in** dept/subdept;

Q4<-[dname] **where** (subdept/)*empcount=100 **in** dept(/subdept)+;

[dname] **where** empcount =100 **in** dept/subdept **ujoin**
[dname] **where** subdept/empcount =100 **in** dept/subdept **ujoin**
[dname] **where** empcount =100 **in** dept/subdept/subdept;

Figure 3.5.6. Queries with “*” and “+” in selection and projection

```
Q1<-[dname] where (subdept/)*empcount=100 in dept(/subdept)*;
+----------------------+
| dname                |
+----------------------+
| DB                   |
| Develop              |
| IT                   |
| Manufacturing        |
| Northeast            |
| Production           |
| Sales                |
| Southwest            |
+----------------------+
relation Q1 has 8 tuples

Q2<-[dname] where (subdept/)+empcount=100 in dept(/subdept)+;
+----------------------+
| dname                |
+----------------------+
| Develop              |
+----------------------+
relation Q2 has 1 tuple

Q3<-[dname] where (subdept/)+empcount=100 in dept(/subdept)*;
+----------------------+
| dname                |
+----------------------+
| Develop              |
| IT                   |
| Production           |
| Sales                |
+----------------------+
relation Q3 has 4 tuples

Q4<-[dname] where (subdept/)*empcount=100 in dept(/subdept)+;
+----------------------+
| dname                |
+----------------------+
| DB                   |
| Develop              |
| Manufacturing        |
| Northeast            |
| Southwest            |
+----------------------+
relation Q4 has 5 tuples
```

Figure 3.5.7. “*” and “+” in selection and projection

3.5.2 Dot Operator

In addition, we can avoid writing the names of intermediate nested attributes if we use a “wildcard”, namely “.”. Queries in figure 3.5.8 retrieve all *dname* from all relations that have the attribute *dname.*

```
   >allDepts<-dept/.*/dname;
or >allDepts<-dept/*/dname;

>pr allDepts;
+----------------------+
| dname                |
+----------------------+
| Concord              |
| DB                   |
| Develop              |
| Finalbuilding        |
| IT                   |
| Manufacturing        |
| Maple St             |
| Milton Ave           |
| Northeast            |
| Park way             |
| Prebuilding          |
| Production           |
| Sales                |
| Southwest            |
+----------------------+
relation allDepts has 14 tuples
```

Figure 3.5.8. Query: retrieve all *dname* from relation *dept* and its nested attributes

Here, "**.**" indicates any relation name. So, "**.***" stands for all relations and we don't care at which level they are. "**.*"** can be shortened to "*****" to resemble Unix conventions[Mer03].

In comparison, unlike the

addDepts<-dept/./dname;*

the queries

addDepts<-dept/.+/dname;

or

addDepts<-dept/+/dname; (Figure 3.5.9)

do not project the *dname* attribute of the relation *dept,* since "+" represents one or many.

```
allDepts<-dept/.+/dname;(or allDepts<-dept/+/dname)
>pr allDepts;
+----------------------+
| dname                |
+----------------------+
| Concord              |
| DB                   |
| Develop              |
| Finalbuilding        |
| Manufacturing        |
| Maple St             |
| Milton Ave           |
| Northeast            |
| Park way             |
| Prebuilding          |
| Southwest            |
+----------------------+
relation allDepts has 11 tuples
```

Figure 3.5.9. Query: retrieve all *dname* from the nested attributes of *dept*

We can explore the wildcard further. The example in figure 3.5.10 signifies that ***(././)*** means any two levels below the relation *dept*. The query is to find out all *dname* in the nested attributes that are two levels below the *dept*. Since both the relation *dept/contact/address* and the recursively nested attribute *subdept* of *dept* contain the domain *dname,* the query

q1<-dept/(././)dname;

will produce the result which is the same as the result of the query

q1<-dept/contact/address/dname **ujoin** *dept/subdept/subdept/dname;*

```
>q1<-dept/(././)dname

>pr q1;
+----------------------+
| dname                |
+----------------------+
| Concord              |
| DB                   |
| Finalbuilding        |
| Maple St             |
| Milton Ave           |
| Park way             |
| Prebuilding          |
+----------------------+
relation q1 has 7 tuples
```

Figure 3.5.10. Example of Wildcard

3.5.3 Question Mark Operator

Question mark operator (?) allows zero or one occurrence of its operands. Figure 3.5.11 and Figure 3.5.12 show the syntax for the question mark operator using the example relation *dept*.

```
>qm_f<-dept/(contact/address/)?dname;

pr qm_f;
+----------------------+
| dname                |
+----------------------+
| Concord              |
| IT                   |
| Maple St             |
| Milton Ave           |
| Park way             |
| Production           |
| Sales                |
+----------------------+
relation qm_f has 7 tuples
```

Figure 3.5.11. Example for the syntax of Question mark operator

In the figure 3.5.11 the query

*qm_f<-dept/(contact/addres/)**?** dname;*

means that if *contact/address* has the attribute *dname,* the *dname* of *contact/address* should be projected with *dname* in *dept*. Otherwise, only project *dname* in *dept.*

Sometimes we don't want to write down the names at intermediate levels. We can use (./) instead of writing down the specific names of relations in intermediate levels. In the figure 3.5.12, the query

*qm<-dept/(././)**?** dname;*

is used to find *dname* from the relation *dept* and from all relations that is two levels below *dept.* Since in the relation *dept* both *dept/contact/address* and its recursively nested attribute *subdept* have the domain *dname*, the query should produce the same result as the query listed below:

qm<-dept/dname ujoin dept/contact/address/dname ujoin dept/subdept/subdept/dname;

```
>qm<-dept/(././)?dname;

>pr qm;
+----------------------+
| dname                |
+----------------------+
| Concord              |
| DB                   |
| Finalbuilding        |
| IT                   |
| Maple St             |
| Milton Ave           |
| Park way             |
| Prebuilding          |
| Production           |
| Sales                |
+----------------------+
relation qm has 10 tuples
```

Figure 3.5.12. Example for question mark operator

Chapter 4

Implementation and Solution Strategy

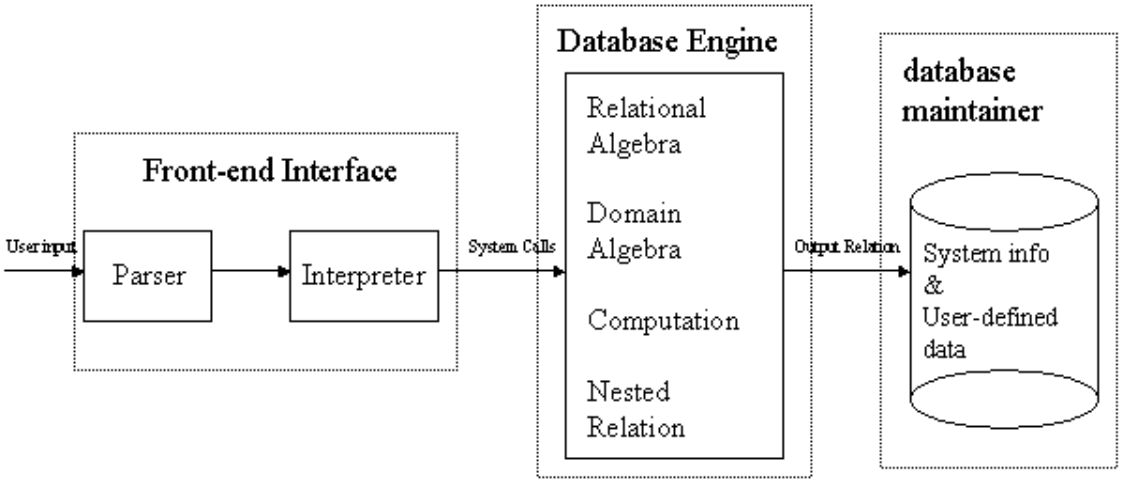
Implementation details for the new features in JRelix as shown in last chapter will be presented in this chapter. In section 4.1 and 4.2 an overview of the system architecture is given. In section 4.3, we introduce how to allow the current JRelix system to accept semi-structured data input. In section 4.4, we discuss the implementation of recursive nesting. In section 4.5, we describe the implementation of path expression operator. In the last section, the implementation of the expression operators, including "*", "+", ".", and "?", are presented.

4.1 Developing Environment

JRelix is written in Java. The old version Relix is implemented in the C programming language. It runs on UNIX, and Windows as well. The parser is generated by JavaCC and JJTree. JavaCC, is a java compiler compiler that acts as a parser generator. It reads high-level grammar specification and converts it to a Java program that matches the grammar. JJTree is a preprocessor for JavaCC that inserts parser tree building action at various places in the JavaCC source.

4.2 System Overview

The JRelix system contains three main conceptual modules: a front-end interface, a database engine and a system database maintainer. The picture is shown below:



The **Front-end Interface** consists of a parser and an interpreter. It is an interface between the user and the database engine. The parser accepts the user command input; then performs command syntax analysis. The user command can be translated into intermediate code that has a tree structure. Then the tree is passed to the interpreter. The interpreter performs error checking, traverses the tree and generates a set of method calls that can be accepted by the database engine.

The **Database engine** is the central part of the *JRelix* system. It implements relational algebra, domain algebra, computation and nested relation.

The **Database maintainer** maintains user-defined data and system information of the *JRelix* system. These system files are stored as ".rel", ".dom", ".rd", ".expr" and ".comp". Files ".rel" and ".dom" stores information about all relations and domains that are defined in the database. File ".rd" stores all information that links relations and the domains on which the relation are defined. File ".expr" stores the syntax trees for virtual domain and views and file ".comp" stores syntax trees for computations.

4.3 Implementation of Semi-Structured Data Input

All *JRelix* statements and input commands are parsed first and then they are transferred to the syntax trees in the *Parser* class. The syntax trees are then decomposed top-down into

some sub-trees in *Interpreter* class and are further processed. In order to make the current JRelix accept the semi-structure data input, quite a few additions have been brought to the *Parser* and *Interpreter* class.

Theoretically, the semi-structured data input should be parsed twice. In the first parsing, all domains should be declared. The domain list of the relation should be collected. In addition, the missing entries should be added to the original semi-structured data input before processing the second parsing to create a corresponding syntax tree. Subsequently the second-time parsing will be involved to generate a syntax tree, which can be processed correctly in the *Interpreter* class. However, the implementation avoids going back to the parser again after parsing the input in order to decrease the processing system time. After the *Interpreter* receives the immature tree translated by the parser from a semi-structured data input to initialize a relation, firstly the domain information is collected while interpreting the syntax tree and the domain list of the relation is then created. Secondly, the tree is modified by adding the nodes that are corresponding to the missing entries. Finally, the correct syntax tree resulting from the modification will be further processed to initialize the relation. The example below is used to explain this procedure.

Note that in the parsing time, only tags `<.tuple>` and `</.tuple>` can be used to separate tuples. When `<.tuple>` and `</.tuple>` are missing, entries are added to the same tuple. In order to clarify this explanation, the relation *EmployeeInfo* shown in chapter 3.1.1 was taken as an example. Firstly, the syntax tree corresponding to the semi-structured input of the relation *EmployeeInfo*, generated by the parser, is illustrated in Figure 4.3.1 below. It is clear from the figure that the resulting syntax tree of *EmployeeInfo* initialization has only two tuples since only one pair of `<.tuple>` and `</.tuple>` is includ

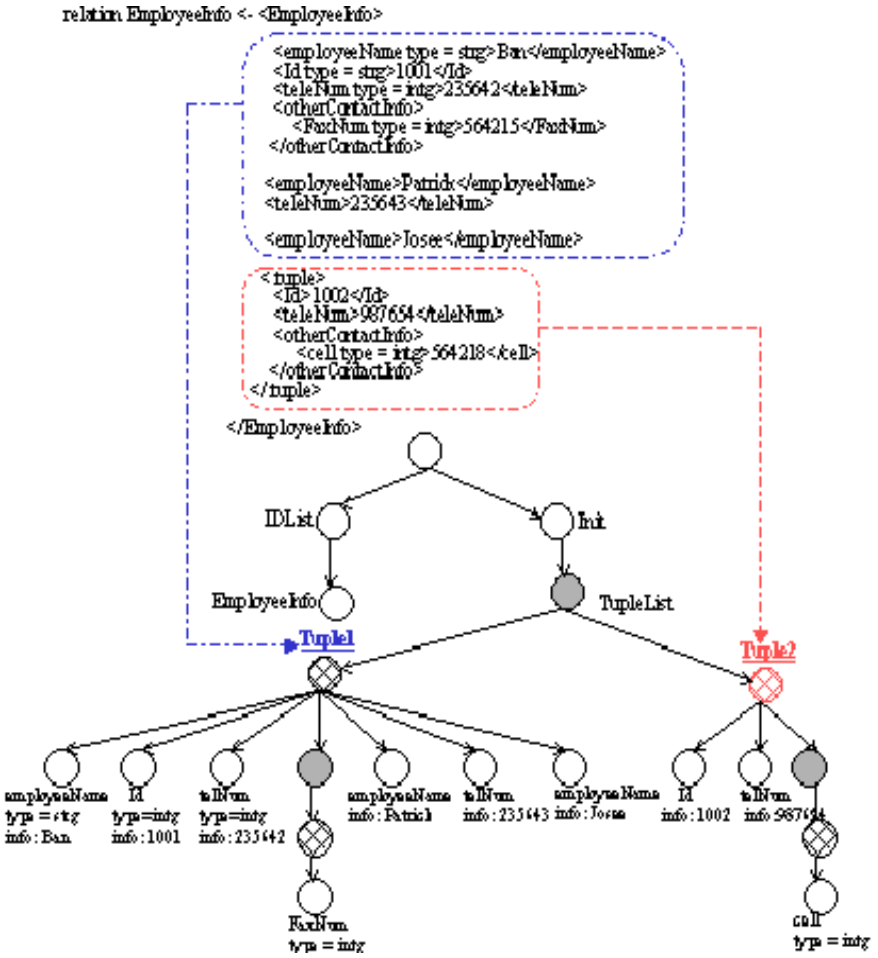


Figure 4.3.1. The tree created by *parser* for the initialization of *EmployeeInfo*

As declared before, since semi-structured data is self-describing, information of domains to be defined is in the input of a relation initialization, and entry missing is also allowed. Hence, during the parser time, all information, including the domain names and its defined types except for their values, have been saved in the syntax tree. In the syntax trees created by *parser* for semi-structured data inputs, the field *name* in nodes is used to store domain names. And if there is information about the defined types of domains, the defined types are also stored in the *name* field as a whole string with domain names instead of *null*. For example, parsing

<employeeName type = strg>Ban</employeeName>

results in a node with "*employeeName type = strg*".

After passing the whole syntax tree to *Interpret* class, extra steps are carried out to further

modify the syntax tree to be processed. The modifications include adding nodes that are corresponding to the entries that have been missed in the input (the value for the missing entries is set to be *dc*) and correctly group the entries to tuples. The *XMLInitialization* function is used to implement these modifications. For the relation *EmployeeInfo* initialization, the tree further modified by the function is shown below:

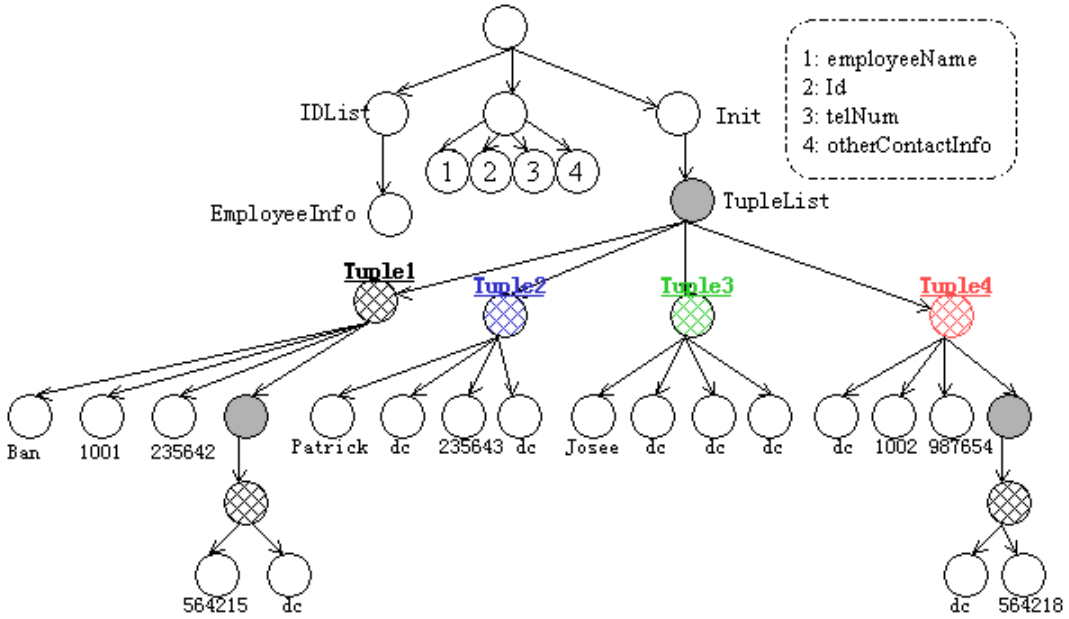


Figure 4.3.2. The tree modified for the initialization of *EmployeeInfo*

Before processing the tree correctly corresponding to the initialization data, new domains in the input must be declared. *XmlDomInfo* class is created to contain the names and the types of the domains being declared. *XmlRelInfo* class is introduced to save the domain list of a relation. In the case of the initialization of the relation *EmployeeInfo,* by traversing the tree passed from parser, the information of all domains and the domain lists of all relations can be obtained (Shown in Figure 4.3.3).

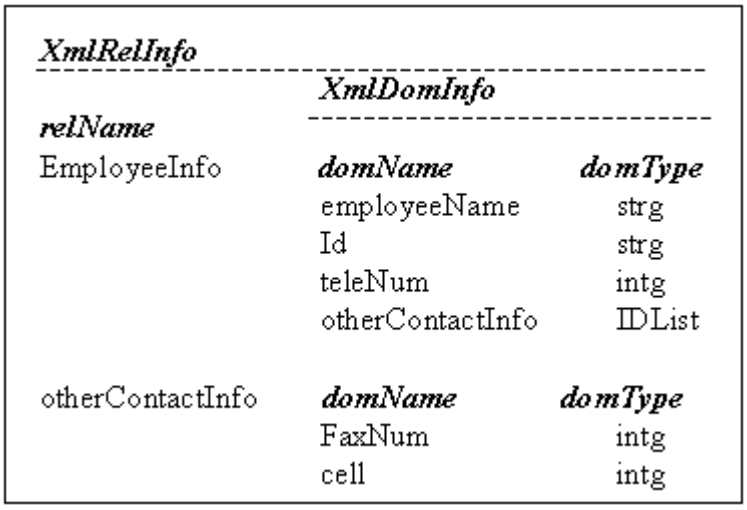


Figure 4.3.3. The domain information of *EmployeeInfo*

Instead of going back to the parser again, domain declaration trees corresponding to "domain *employeeName strg;*", "domain *Id strg;*", "domain *teleNum intg;*", "domain *FaxNum intg;*", "domain *cell intg*" and "domain *otherContactInfo(FaxNum,cell)*;" are created. The function *executeDaclaration* is then invoked to declare all these domains.

After declaring all domains, the tree which correctly corresponds to the initialization data will be processed to initialize the relation.

4.4 Implementation of Inputting Data From a File

To declare and initialize a relation using the data in a file, firstly the data is retrieved from the file. The data in the format of "String" is then combined with the relation name to produce a new input string that is acceptable to *JRelix*. The *Parser Class* is subsequently invoked to parse the new input string, and finally the syntax tree corresponding the new input is processed by *Interpreter* to initialize the relation. A small example is presented below to explain the process.

relation AldatLab<- "D:\Project\InputFile\AldatLab.tex";

Step1: Get the input data in the file

AldatLab.txt:

```
<AldatLab>
<name type = strg> Biao</name>
<name>Yi</name>
<name>Zhongyi</name>
<name>Shuhong</name>
<AldatLab>
```

Step2: Produce the new input string acceptable by *JRelix*

```
relation AldatLab(name)<-<AldatLab>
                         <name type = strg> Biao</name>
                         <name>Yi</name>
                         <name>Zhongyi</name>
                         <name>Shuhong</name>
                        <AldatLab>
```

Step3: Invoke the parser to parse the new string.

4.5 Implementation of Recursive Nesting

The current implementation of *JRelix* requires that the attributes on which a new nested attribute is defined must already be defined. Recursively defined nested attributes are not allowed. Hence, to implement recursive nesting, firstly of all, the recursive definition of domains [*e.g* domain *N(A, N)*] must be accepted. Modifications are made to the method *lookupDom( )* in the *Environment* class to allow the recursive definition of domains.

Secondly, careful modifications are made to the method *RelationalInitialization ( )* in the *Interpreter* class, which is used to initialize a relation. To keep the hierarchical structure of recursive nested relations, the number of levels of the recursive nested relations are recorded when the relations are initialized. As I mentioned in the last chapter, *hashtable* is used in the current system to save the data of relations. The key of the *reltable* is relation names, hence a duplicate name is not allowed in the table. So, the recursive relation names on each recursive level have been changed according the current level value. Furthermore, after a relation initialization the recursive domain names in the relations on all recursive levels have also been modified according to the recursive level values. The source code that implements this function is added to the

method *RelationalDeclaration()* in the *Interpreter* class.

Take the recursive nested relation *R(A,N)*, which contains the recursive domain *N*, as an example. Suppose *R* has 4 levels, then after the initialization of the relation *R*, four relations, as illustrated below, that have been created:

R(A,N)
.N(A,N_0)
.N_0(A, N_1)
.N_1(A,N_2)
.N_2(A,N_3)

Note that the type of domains *N, N_0, N_1,N_2* is IDLIST, while the type of domain *N_3* is a LONG and is used to stop the chain of DAG. However, compared to non-recursive domains, the types of the recursive domains should be changed when another relation, which contains the same recursive domain *N*, is to be initialized. For instance, another relation *Q(B,N),* where the recursive level of *N* is more than 4, is initialized after the initialization of the relation *R.* In the situation, after the initialization of relation *Q,* the domain type of *N_3* is changed to IDLIST from LONG.. The modification of recursive domain types is implemented by functions *putRecurDom* in the *Environment* class and *rmRecurDom* in the *domTable* class.

An advantage of this implementation is that operations on non-recursive nesting will still work on the recursive nesting, since the recursive nesting retains the exactly the same structure as the non-recursive nesting. However, it is not necessary for users to find values of recursive levels and use the different relation names on different recursive levels to query recursively nested relations. In queries that concern the recursive nesting relations users can simply use the recursive domain name at any level. For example, the queries:

*RN<-[**red ujoin of** [**red ujoin of** N] **in** N] **in** R;*

*RA<-[**red ujoin of** [**red ujoin of** [A] **in** N] **in** N] **in** R;*

etc. are acceptable. To allow the queries, modifications are made to the syntax trees, generated from the queries by parser. The modification is simple: change the *names* of

Identifier nodes that carry the *name* of the same recursive relation name to corresponding recursive relation names before implementating operations. Figure 4.4.1 presents a clearer explanation.

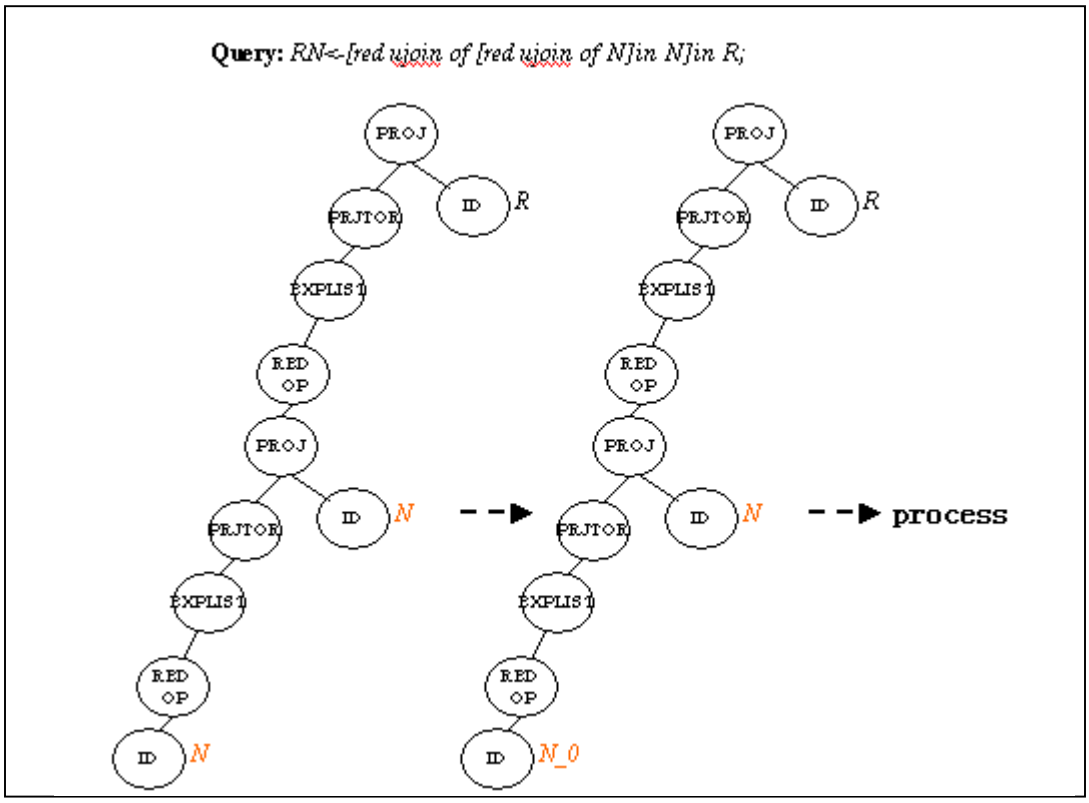


Figure 4.4.1. An example of the queries involved in the recursive nesting relation

It must be noted that in recursive nesting, the recursive loop will not terminate until the null value, *dc*, is found.

4.6 Implementation of Path Expression Operator

In the current *JRelix* system, the vertical operation ***red ujoin of*** is used for level-lifting in nested relations. For example, the query

*Address<-[**red ujoin of** otherContactInfo] **in** EmployeeInfo;*

accomplishes the raising of *otherContactInfo,* the nested attribute of the relation *EmployeeInfo,* to the relation *EmployeeInfo.* Using the operator "/", the path operator will turn this into

Address<- EmployeeInfo /otherContactInfo;

To implement the path operator, firstly the parser should recognize the command with the path operator "/' and build up the corresponding syntax trees for level lifting. Obviously, syntax trees for path operator are the same as the trees translated from the ***red ujoin of*** operation. However, "/" is used as the division operator in the current JRelix system. The path operator is overloaded. For example, for the query

let *p* ***be*** *R/A;*

where the "/" could be a division operator or a path operator. In order to check whether it is a path operator or a division operator, the function *lookupDom* in *Environment* class is called. If *R* is a relation and contains a domain *A*, "/" is a path operator; otherwise it is a division operator. Then the syntax trees corresponding to the path operator or division operator are built up.

To take fuller advantage of path operator, the last component of path operator query can either be a relation name or a domain name. Given the example relation *EmployeeInfo,* to project its domain *employeeName* from relation *EmployeeInfo,* we can simply use the expression

Ename<- EmployeeInfo/employeeName;

In this case, the syntax trees passed from parser to interpreter should be further modified. Firstly, the type of the last attribute is determined and the syntax tree passed from the *Parser* is then modified before being further processed if the last attribute is a domain, not a relation. The function *sugarInProject* in *Interpreter* class implements the operations. The following figure shows how the trees change if the last component of a path expression is a domain name. In the example, since *employeeName* is a domain, the syntax tree passed from the parser from the query *EmployeeInfo/employeeName* is modified by the method *sugarInProject* before being processed. If the last component in the path expression *EmployeeInfo/otherContactInfo* is a relation's name, the tree is processed directly without any modification.

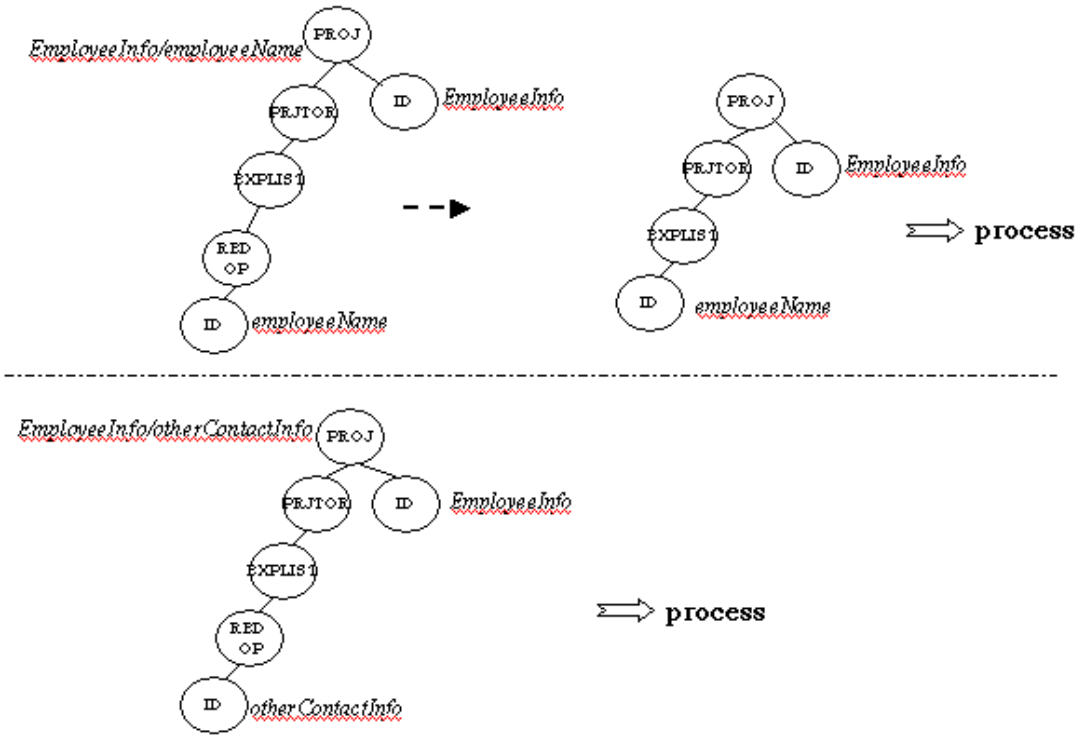


Figure 4.5.1 An example of trees modification for the path expression

As mentioned in previous chapter, path expression operator may be used in any part of a regular T-selector. In this case, the trees passed to the *Interpreter* are modified to their correct forms in order to enable further processing. For example, the query

street/cname in company/address;

(the structure of the relation *company* is shown in Figure 4.5.2) contains a path operator in both *projection* and *expression.* The syntax tree translated from the query will be modified, resulting in a tree corresponding to the query:

[cname] in company/address/street;

before being further processed in the *Interpret* class. Furthermore, as declared in the previous chapter, the path operator may also be used in selection, for example, the query

[cname] ***where*** *address/city ="Montreal"* ***in*** *company*;

contains a path operator in its selection. In this situation, the syntax tree, modified from the tree passed from parser and being further processed to accomplish this query, corresponds to the query below,

[cname] ***where*** *([] **where** city = "Montreal" **in** address) **in** company;*

The codes to implement these functions are used in the method *sugarInProject().*

```
company
(cname,    address                          )
           (street,      city,    codezip)
           (num, cname   )

chrono Inc. 1  Mess St  Montreal  H1W4A4
Future      10 Rechel   Quebec    R8P8C1
Microsoft   31 Peel     Toronto   T673R5
```

Figure 4.5.2. The relation *company*

4.7 Implementation of Expression Operators "*", "+", "." and "?"

In order to implement the operators "*", "+","." and "?", the subsets of queries that contain the expression operators are stored in the syntax trees built up by the parser. For example, the syntax tree for the query

R(/N)/A;*

built up by the parser contains a node with *name* (N)*. Then, in *Interpreter* class, significant modifications to the trees are required before further processing. The reason to implement this way is: to build up correct syntax trees, parser should get all information about the recursive nesting relations on that queries with which the operators are. However, it might be hard for the parser to get all the information.

First of all, in *Interpreter* the function *searchSTAR* is invoked before process syntax trees are passed from parser, to check whether there are expression operators in any node of the trees. If there are such operators in the trees, function *modifyNodeForKSTAR* is then celled to make necessary modifications to the trees.

For example, suppose *N* is a recursively nested domain that contains the attributes *A* and *B*. *R(/N)*/A* should be solved by lifting the attribute *A* in the nested attributes *N* at all

recursive levels to the top level relation *R* and **ujoin** them, as illustrated in Figure 4.6.1.

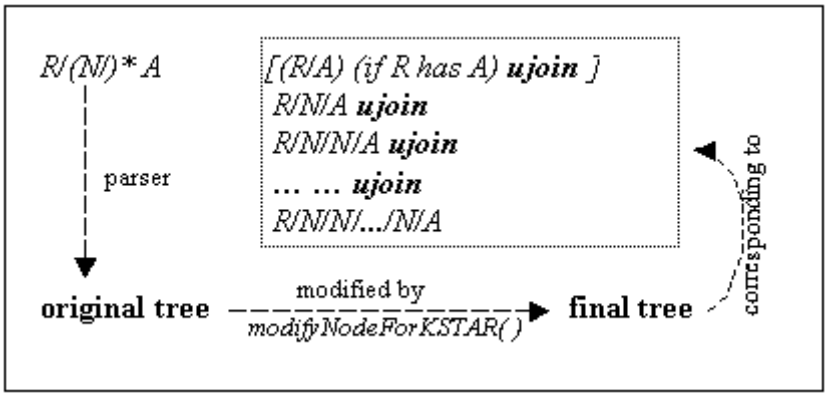


Figure 4.6.1. An example of Kleene Star operator

The figure 4.6.2 below shows the modification made to the original tree of R(/N)*/A before further processing.

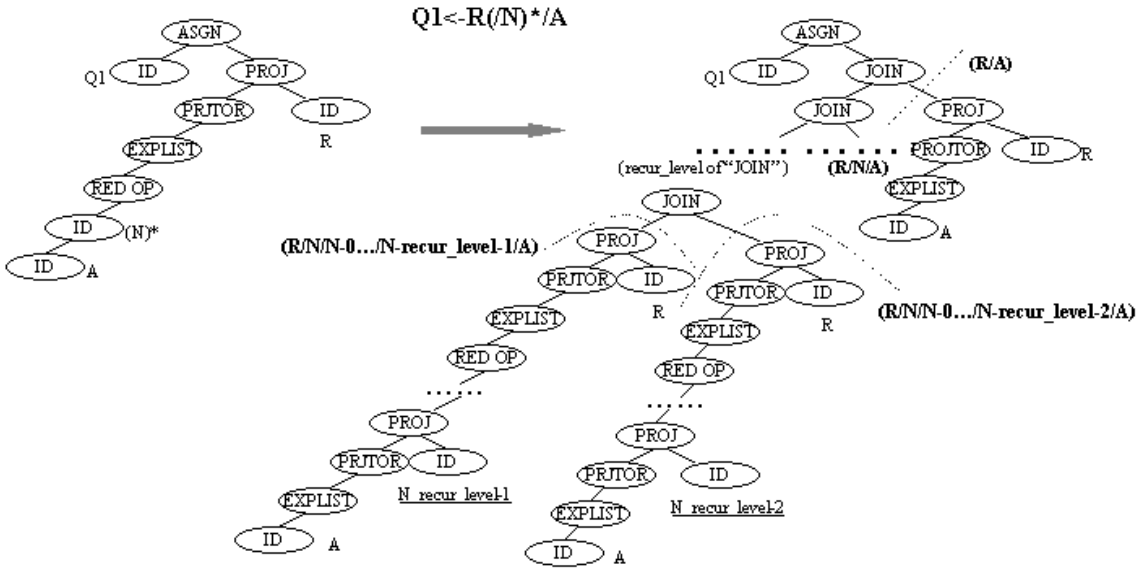


Figure 4.6.2. The original and final tree of R(/N)*/A

As mentioned in the previous chapter, the Kleene star "*" can be used in any part of a regular T-selector. An example for Kleene star in selection is shown in Figure 4.6.2.

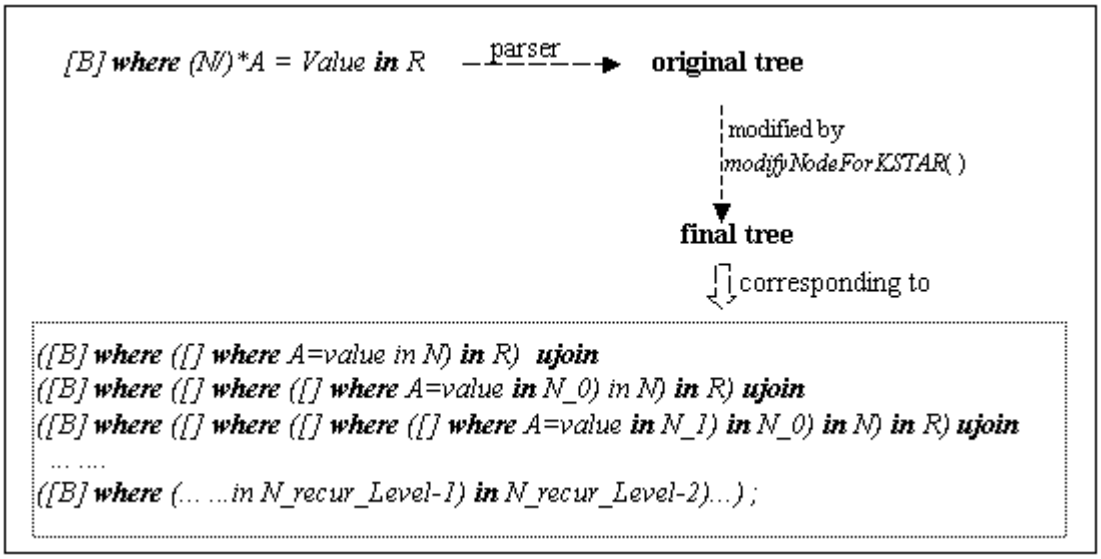


Figure 4.6.3. Kleene Star in Selection

The figure 4.6.4 below shows the modification made to the original tree of the query *[B] where (N/)*A=value in R* before further processing.

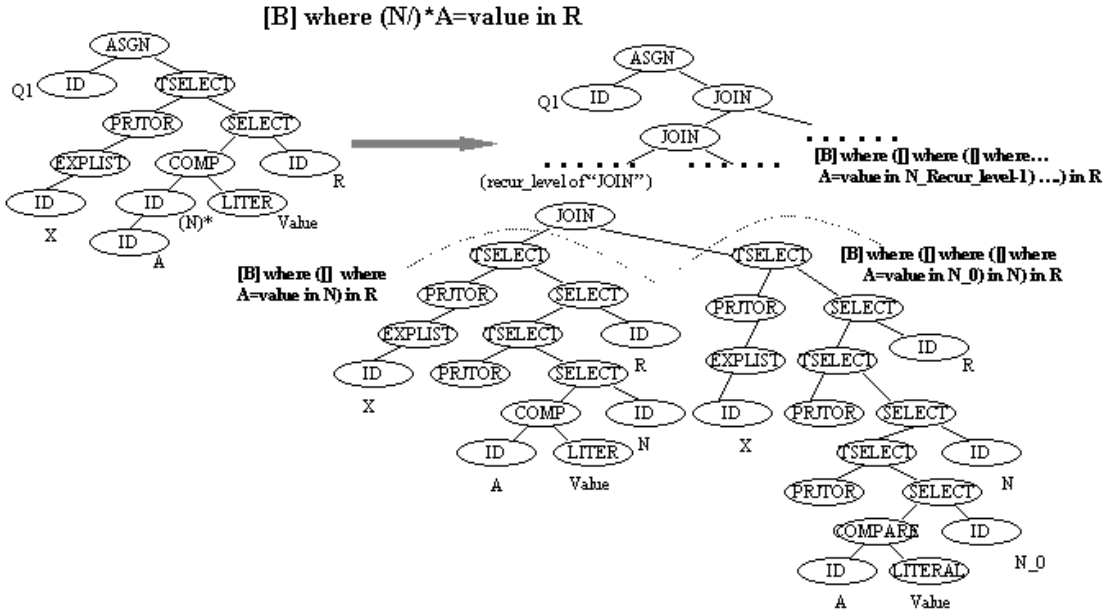


Figure 4.6.4. The original tree and final tree of query [B] where (N/)*A=value in R

It is more complicated if the Kleene star occurs in both projection and selection. Figure 4.6.3 illustrates an example of this case.

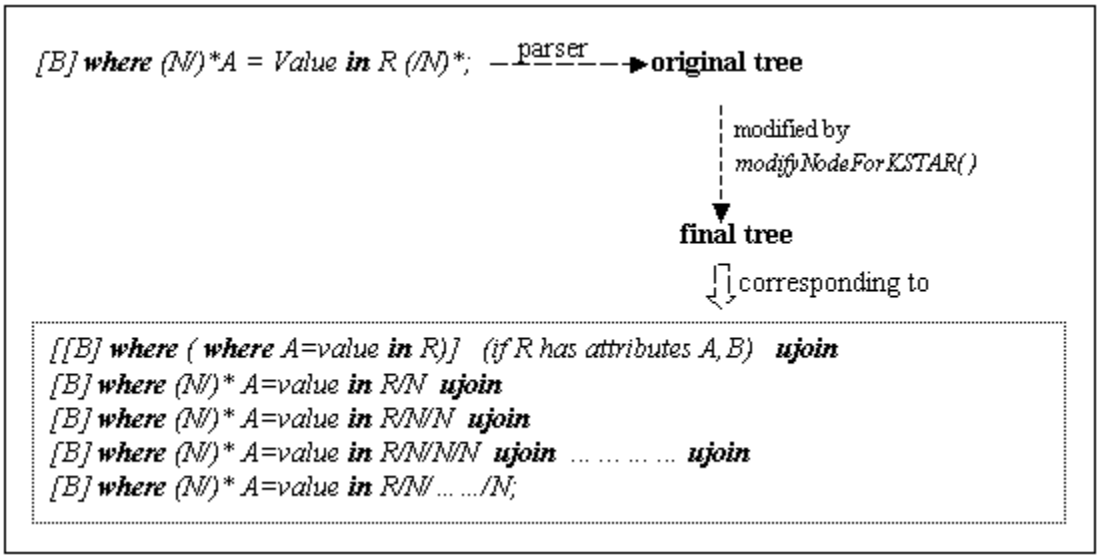


Figure 4.6.5. Kleene Star in projection and selection

For the case *R/.*/A* or *R/*/A*, extra steps to find all relations, regardless of level, containing the attribute *A* are required. The *nestDomsA* is created to handle this case. The example relation *dept* (Figure 4.6.4) used in the previous chapter is taken to illustrate this case in Figure 4.6.5.

dept

dname	empcount	contact (address (num	dname)	tel)	subdept (dname	empcount	subdept (dname	empcount	subdept)
Production	100	1	Maple St	888-1111	Manufacturing	100			
		2	Milton Ave				Prebuilding	40	dc
							Finalbuilding	60	dc
Sales	200	3	Park way	888-2222	Northeast	100	dc		
					Southwest	100	dc		
IT	300	4	Concord	888-3333	Develop	150			
							DB	100	dc

Figure 4.6.6. The relation *dept*

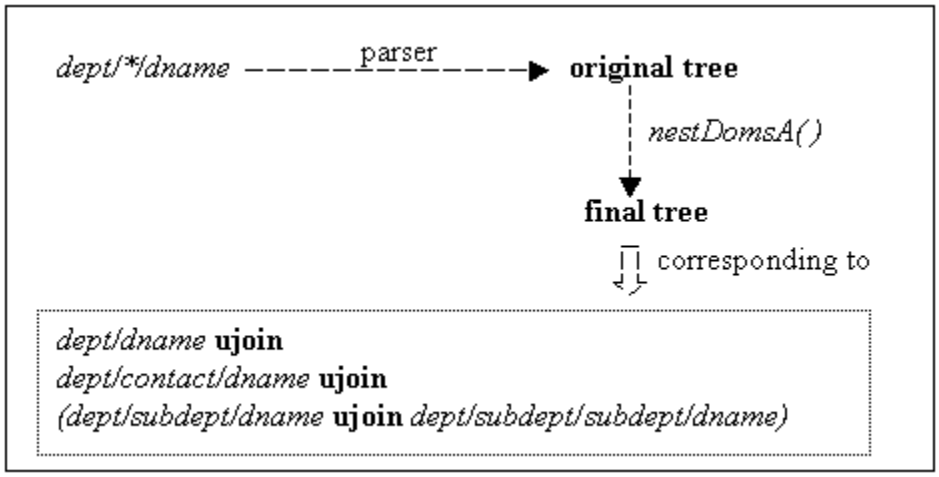


Figure 4.6.7. Projection of an attribute from all levels

The most popular queries using the operator "*" are presented above. Operator "+" is almost the same as the "*". The only difference between them is that for the queries with the operator "+", the top-level relation is ignored. The final trees modified from the original tree passed from the passer will be processed to accomplish the queries.

For the dot operator ".", search steps are also required to find the relations that satisfy specific conditions which are specified in queries. For example, the query

dept/(./.)dname;

will be accomplished in this way: first, all relations, which are two-levels below the relation *dept* and contain the attribute *dname,* should be found. Subsequently, the syntax trees, corresponding to ujoin and projection of the attribute *dname* from these relations, is generated by modifying the tree passed from the parser, which has a node with the *name* (./.). The new method *withoutRelName()* is created to accomplish these functions. Finally, the modified tree will be further processed in order to answer to the query.

The new method *modifyNodeForQuestionMarker()* is involved in the implementation of the question mark operator "?". The implementation policy for "?" is not complicated. Firstly, find whether the answer to the "?" is yes or no. If the answer is no, a warning will be given. If the answer is yes, then modifications to trees passed from the parser are required, since the original trees translated by parser contain the node which *name* is *(..)?*

and can not be processed. Take the query

*dept/(contact/addres/)**?** dname;*

as a example. In the function *modifyNodeForQuestionMarker,* firstly it is found that *dept/contact/address* contains a attribute *dname*, subsequently, the syntax tree corresponding to the query

dept/contact/addres/ dname;

is generated by modifying the syntax tree passed from parser. The result of the query will be given after processing the modified syntax tree.

Due to their complexity, not all original trees and final trees in the examples are illustrated in this section.

Chapter 5

Summary and Future Work

In this project, the design and implementation of some new features of *JRelix*, including semi-structured data loading, recursive nesting and improved query path expression and regular expression operators were described. In the implementation of these new features, the syntax of the former *JRelix* was used where it is possible. The new features are summarized below:

• Acceptance of semi-structured data loading makes data loading for relation initialization more convenient in *JRelix.* To simply the edition of relation loading data, the data can be edited in a file and saved as a .txt file. The relation can then be declared and initialized by the data in the file.

• The syntax for the semi-structured data input is similar to XML. The types of the domains in the relations to be initialized by a semi-structured data input are specified in the input data. While loading data some entries can be missed and in this situation the null values (*dc*) are added for the missing entries.

• To support recursive nesting, recursively defined nested attributes are now permitted After the relations containing a recursively defined domain are initialized, the modifications to the names of the nested relations created in the initialization are performed to indicate the hierarchical structure of the relations. The advantage of the implementation is that operations on non-recursive nesting can still work on the recursive nesting,

• Regular expression operators (“*”, “+”, “.”, “?”) have been implemented to query relations with a recursively nested domain.

• In addition, path expression operator, which is likely to be frequently used in

querying nested relations, has been implemented as a shorthand by using the / operator.

So far, only the major functions of these new features have been implemented. There is further work to be done to refine the implementation.

- To date, only semi-structured data loading is accepted. Query results can be output as semi-structured data by further implementations. In addition, further work on semi-structured data queries may be explored.
- Further implementations on the regular expression operators, which includes combination of these operators, additions of "*or*" operator (|) and etc, will improve the queries with the expression operators.
- Union type, which allows attributes alter their types, could be implemented with further work, permitting an attribute to have more than one type.

Biobiography

[Abi95] Serge Abiteboul. *Querying SemiStructured Data*. INRIA-Rocquencourt, 1995.

[ABS00] S.Abiteboul, P.Buneman, D.Suciu. *Data On the Web.* Morgan Kaufman Publishers, San Francisco, 2000.

[AQM+96] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet Wiener. *The Lorel query language for semistructured data*, 1996, Manuscript available from http://www-db.stanford.edu/lore/.

[AQM+97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. *The Lorel query language for semistructured data.* International Journal on Digital Libraries, 1(1):68-88, April, 1997.

[BDHS96] Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu. *A Query Language and Optimization techniques for unstructured data*. In SIGMOD, 1996.

[BDS95] Peter Buneman, Susan Davidson, and Dan Suciu. *Programming constructs for unstructured data*. In Pproceedings of DBPL'95, Gubbio, Italy, September 1995.

[Bun97] Peter Buneman. *Semistructure Data.* Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA.1997.

[GRA02] Mark Graves. *Designing XML Databases*. Page 4-24.

[Hao98] Biao Hao. *Implementation of the Nested Relational in Java.* Master's thesis, McGill University, Montreal, Canada, 2002

[MAG+97] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallan Quass, Jennifer Widom. *Lore: A Database Management System of Semistructure Data.* Stanford University, 1997.

[Mer77] T.H. Merrett. *Relations as programming language elements.* Information Processing Letters, 1977. Reston Publishing Co.

[Mer84] T.H. Merrett. *Relational Information Systems*. Reston Publishing Co. Reston, VA, 1984.

[Mer02] T.H. Merrett. *The class notes for 308-612 Fall 2002.* School of Computer Science, McGill University. www.cs.mcgill.ca/~612

[MMM96] A.Mendelzon, G.Mihaila, and T.Milo. *Querying the World Wide Web*. In Proceedings of the Fourth Conference on Parallel and Distributed Information Systems, Miami, Florida, December 1996.

[QRS+95] D.Quass, A.Rajaraman, Y.Sagiv, J. Ullman, and J. Widom. *Querying semistructure heterogeneous information* . In International Conference On Deductive and Object Oriented Databases, 1995.

[Suc99] Dan Suciu. *Management of Semistructure Data.* AT&T Labs, 1999

[Wan02] Zongyan Wang. *Implementation of Distributed Data Processing in a Database Programming Language.* Master's thesis, McGill University,

2002.

[Yua98] Zhongxia Yuan. *Java Implementation of the Nested Domain Algebra in a Database Programming Language*. Master's thesis, McGill University, 1998.

[Zhe02] Yi Zheng. *Abstract Data Types and Extended Domain Operations on Nested Relation Algebra*. Master's thesis, McGill University, Montreal, 2002