Java Implementation of the Domain Algebra for Nested Relations

Zhongxia Yuan
School of Computer Science
McGill University, Montreal

A Thesis 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, Advisor

Final draft printed on: August 29, 1998

0-612-50913-3

Contents

List of Figures

Abstract

This thesis discusses the design and implementation of a relational database programming language, focusing on the domain algebra for nested relations. While the *relational algebra* provides operations to manipulate relations as data primitives, the *domain algebra* allows the manipulation of the attributes of relations. With the nested relational model, the domain algebra subsumes the functionality of the relational algebra. The conventional relational operations (e.g. selection and projection) may be applied to the attributes of a relation.

The domain algebra for nested relations has many advantages. For example, the domain algebra makes the nesting and unnesting of relations very easy. This ensures that any hierarchical database schema can be validly translated into a conventional 1NF database schema. With the nested domain algebra, defining *abstract data types* for domains can be achieved as well.

The system consists of three modules; the relational algebra, the domain algebra and computations. This thesis deals with the domain algebra module.

The implementation is part of the *jRelix* project at McGill University. The most significant feature of jRelix is the support for the nested relations with an arbitrary but finite level of nesting. The Object-Oriented programming language Java was used exclusively during the implementation, which secures jRelix as a platform independent database programming language.

Résumé

Cette thèse documente la conception et l'implantation d'un langage de programmation de base de données relationnelle, se concentrant en particulier, sur l'algèbre des attributs pour les relations imbriquées. Alors que l'algèbre relationnelle offre des opérations qui manipulent des relations comme données primitives, l'algèbre des attributs permet la manipulation des attributs des relations. Avec le modèle des relations imbriquées, l'algèbre des attributs inclus le fonctionnement de l'algèbre relationnelle. Les opérations de l'algèbre relationnelle (e.g. selection et projection) peuvent être appliquées aux attributs des relations.

L'algèbre des attributs pour les relations imbriquées a plusieurs avantages. Par exemple, elle facilite l'imbrication et la désimbrication de relations. Ceci assure que n'importe quel schéma hiérarchique de base de données peut être traduit en un schéma conventionnel de base de données en première forme normale. En plus, l'algèbre des attributs pour les relations imbriquées permet la définition des ADT.

Le système comprend trois modules; l'algèbre relationnelle, l'algèbre des attributs et les computations. Cette thèse traite le module de l'algèbre des attributs.

L'implantation fait partie du projet jRelix de l'univerité McGill. Le trait le plus significatif de jRelix est le support pour les relations imbriquées à des niveaux arbitraires mais finis. Le langage de programmation orienté objet Java fut utilisé pour l'implantation. Ceci rend le langage de programmation de base de données jRelix indépendant de la plate-forme sur lequel il est utilisé.

Acknowledgments

Many people provided assistance to me, without which this thesis might never have been written. Among them, I especially wish to thank Prof. T. H. Merrett, my thesis supervisor, for his invaluable guidance and continuous encouragement throughout my research, for the financial support he provided me, and for the long hours he spent with me on the topics of the thesis.

I would like to thank Biao Hao and Patrick Baker, my colleagues in the jRelix project team at McGill, for their cooperation during the project. I benefited a lot from the discussions with them.

I am also indebted to my friends, Song Hu and Xiaoyan Zhao, for all their enthusiastic assistance on preparing this thesis.

Finally, I would like to thank my girlfriend, Jenny Hui Zhan, for her support, encouragement during my study and research work at McGill.

Chapter 1

Introduction

This thesis describes the design and implementation of a relational database language in the Java programming language in general, and the implementation of domain algebra in particular. The purpose of our work is to support the traditional relational model with an extension of nested relations.

1.1 Background and Motivation

Starting in 1986, a relational database language called *Relix* was designed and developed at the Aldat lab of School of Computer Science in McGill [Lal86]. The purpose of the original work was to provide an experimental interactive environment for exploring the concept of the so-called *Relational Database Model* proposed by Dr. E. F. Codd in his pioneering paper "A Relational Model of Data for Large Shared Data Banks" [Cod70]. The original system was developed in C language and ran on UNIX operating system. Relix was firstly designed to support both relational algebra [Cod70] and domain algebra [Mer84] for flat relations. Following the progress of development in the Aldat lab, the Relix system has been enhanced with further conceptual functionalities such as procedures, event handling, computations, and concurrent control etc. In 1996, an improvement to the system made Relix capable of supporting the basic operations of the so-called *Nested Relational Model* [Mak77] introduced in 1977, although the operations were limited to one-level of nesting [He97].

In the summer of 1997, a new project team was formed with the intention to redesign the whole system in an Object-Oriented manner in general, and to implement it in a new programming language i.e. Java [GJS96] [AG96] in particular. The new system was named *jRelix* and was supposed to cover the most important functionalities of the original Relix system, with a further extension to support the concept of a nested relational model with an arbitrary but limited level of nesting.

The project is still in progress at the time of this writing, although most most expected functionalities have been implemented. In general, the system was classified into three major modules, i.e. *Relational Algebra*, *Domain Algebra* and *Computation*. The *relational algebra* module was designed and implemented by Biao Hao, a graduate student in Computer Science at McGill. I have designed and implemented the *domain algebra* module. Patrick Guillaume Baker, another graduate student at McGill, was responsible for the *computation* module. This thesis will discuss the system design and implementation of the jRelix, with a focus on *Domain Algebra*, i.e. what I have been working on.

Domain Algebra for Nested Relations

As mentioned before, the purpose of our work is to design and implement a database language/model that supports the *Relational Database Model* with the extension of *Nested Relations*. To achieve this, the domain algebra needs to subsume the capability of dealing with nested relations.

Why do we need a domain algebra for nested relations? With domain algebra capable of dealing with nested relations, the *nesting* and *unnesting* of relations can be easily performed, which ensures that any hierarchical database schema can be validly translated into a 1NF relational database schema. On the other hand, the potential of defining *abstract data types for domains* can be achieved by nested domain algebra as well.

The idea is to integrate the notion of "nested relations" into both the relational algebra and the domain algebra. In other words, the key point is to make relational expressions part of domain expression so that relational operations can be applied to the nested attributes/domains. This way, any operations that are performed on the top level relations can be performed at the lower level attributes which are also relations.

Apparently, in order to fulfill this achievement, the domain algebra must be given more power, i.e. apart from its original ability to perform domain operations, it must be competent in relational operations as well. In fact, the way jRelix domain algebra was designed and implemented shows that domain algebra in the nested relational model is a super set of both traditional domain algebra and relational algebra, i.e. it subsumes almost all the functionalities of relational algebra, and becomes the most important concept in the *Nested Relational Model.*

Java Programming Language

The old version of Relix was developed in C programming language, and has being running in UNIX environment. Rather than continue development on the C version of the Relix system, we decided to choose Java as our developing language and start the design from scratch. There are multiple reasons behind this decision.

First of all, Java is an Object-Oriented programming language. The *Objectory* methodology i.e. to develop a system in an Object-Oriented manner gives much more benefits than the traditional methodology of structured system design and implementation, e.g. the easiness of system development and maintenance supported by the *Objectory* concepts of data encapsulation, inheritance and polymorphism.

Second, development using the Java programming language results in an object system that is *platform-independent*, as a declared feature of Java's *"neutral architecture"*. The benefit is significant: it realizes many software designers' dream of "compile once, and run everywhere". Compared with the C version of the Relix system which runs in UNIX environment (and needs a new compilation for each UNIX system), jRelix runs on almost all operating systems without additional compilation, as long as the operating system has a "Java Virtual Machine" running. And as we know, the fact is that almost all non-trivial operating systems have become "*Java-capable*".

Third, Java has a strong connection with Internet. The Java development environment provides abundant libraries of networking facilities. It is very easy for a Java application to migrate into Internet applications. The jRelix system, with no exception, can be converted into an Internet application that will become accessible by a remote web browser e.g. Netscape Navigator, which gives the strong potential of system scalability. A further migration of the jRelix system into a so-called Client/Server model, which theoretically takes less effort than that of a non-Java application, can enhance the system performance dramatically.

The discussion of the benefits that Java provides is beyond the scope of this thesis. However, the advantage of choosing Java as the developing language instead of staying with C/C++ for jRelix system development is obvious and desirable.

1.2 Thesis Outline

Chapter 1 of this thesis introduces the thesis topic. In Chapter 2, a background review of the literature and related work done at McGill is given. Chapter 3 describes in detail how the relational database operations are performed using the system developed as our implementation. Much discussion is focused on the nested relational model in this chapter. Chapter 4 explores the implementation issue involved in the design of a relational database language that supports nested relational model in general, and so-called domain algebra in particular. In Chapter 5, a conclusion of our present work as well as the future work for this topic is discussed.

Chapter 2

Background and Related Work

The well-known and widely used relational database systems all conform to the basic relational model first proposed by Dr. E. F. Codd in his pioneering paper "A Relational Model of Data for large Shared Data Banks" [Cod70]. The relational model, unlike other data models, has a rigorous mathematical definition that is beyond the scope this thesis, but it has since then been recognized for its simplicity, uniformity, data independence, integrity and evolvability [Ger75]. The basic technology shared by all relational databases can be summarized simply as follows:

- The database system maintains a clear distinction between the logical views of the data presented to the user and the physical structure of the data as it is stored. The user need not understand the physical structure of the data in order to access and manage data in the database.
- There is a simple logical data structure that is easily understood by users who are not database specialists.
- There are high-level languages provided for accessing the data in the database, and for performing various operations on databases.

Codd later went considerably beyond just providing this model, however [Cod72b]. It also included:

- Relational calculus, a mathematically rigorous definition of the "set operations" that a relational database should support for manipulation of tables.

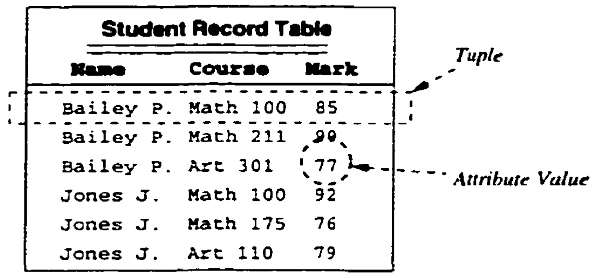
- Rules defining how a relational database should operate. The rules cover matters ranging from the database access that must be provided for users, to issues of data security.

2.1 Relational Model

In his relational model, Codd showed that a collection of tables that he termed *relations* could be used to model aspects of the real world and store data about objects in the real world. The form of a relation is deliberately chosen to be simple, yet it is capable of capturing many of the relationships represented by the more complex data structures.

The relational model for representing data specifies that information is represented in a table format with the following characteristics:

- *all rows are distinct*
- *the ordering of the rows is immaterial*
- *each column has a unique label, and, hence, the order of the columns in a row is insignificant*
- *the value of a given column in a row is of a simple type such as an integer or a floating point or a character string, as opposed to complex type such as a table*



Student Record Table

Name	Course	Mark
Bailey P.	Math 100	85
Bailey P.	Math 211	90
Bailey P.	Art 301	77
Jones J.	Math 100	92
Jones J.	Math 175	76
Jones J.	Art 110	79

Figure 2.1: Relational Model

As showed in Figure 2.1, the terminology associated with a relational model consists of:

- **tuple**: *a row in the relation*
- **attribute**: *a column in the relation*
- **domain**: *the set of legal values that an attribute can have*

2.1.1 Operations on Relations

All data within a relational database is viewed as being held in tables or relations. Each relation is a model of real-world data relationships. At the same time, a relation is a simple enough structure that users can readily understand. A system that supports the relational model can perform well-defined operations on these relations to retrieve information.

On the other hand, relational algebra (which is based on function application and the evaluation of algebraic expressions) is a procedural query language which is used to process relational data. The basic operations of relational algebra were first suggested by [Cod70]. He also established that queries formulated using his calculus *DSL-ALPHA* could be formulated in algebra and vice versa (1972); in consequence he called both languages *relationally complete* [Cod71]. In the relational algebra, there is no concept of tuples. The relational operators take relations as operands and return a relation as a result which can be further manipulated. The property that any relational algebra operation evaluates to a relation is also called the "*closure principle*" of relational algebra. The *closure principle* allows complex relational expressions by building up a series of simple operations.

The relational algebra operations are usually classified as unary or binary, depending on the number of their operands. Unary operators act on a single relation, binary operators act on two relations, and both produce a single relation as their result.

- Unary operations
  - *Projection: makes a copy of a relation with a specific subset of the attributes*
  - *Selection: selects tuples that satisfy a specific condition*
- Binary operations
  - *Mu-join: join operators that generalize set-valued set operations*
  - *Sigma-join: join operators that generalize logic-valued set operations*

2.1.2 Operations on Domains

The need for arithmetic and similar processing of the values of attributes in individual tuples is apparent. The domain algebra [Mer84] was proposed entirely to avoid tuple-at-a-time operations for processing attributes in individual tuples. It allows the user to create new domains from existing ones. It allows the generation of a new value from many values within a tuple or from values along an attribute. The domain algebra operations are defined as follows:

- horizontal operations
  - *Constant*
  - *Rename*
  - *Function*
  - *If-then-else*
- vertical operations
  - *Reduction*
  - *Equivalence Reduction*
  - *Functional Mapping*
  - *Partial Functional Mapping operations*

Various combinations and permutations of the above-mentioned operations e.g. selections, projections and joins etc. are used in practice to retrieve information from a collection of relations in a relational database. Many of these have been implemented on commercial DBMS in the form of SQL (Structured Query Language) and other specialized devices. The actual data retrieval process thus becomes transparent to the user making the query. The user only sees the output as a relation [TPB87].

2.2 Normalization of Relational Databases

Normalization is a prominent aspect of relational database theory. It addresses how data ought to be organized within a database in order to make the database as compact and as easy to manage as possible and to ensure that is produces consistent results. Normalization rules provide guidelines for defining the schema (design) of a relational database. Simply put, the rules specify how a database should be divided into tables and how the tables should be linked together. There are two major objectives of normalization:

1. Minimize the duplication of data.

Student			
Name	Advisor	Courses	
		Course#	Mark
Bailey P.	Smith A.	Math 100	85
		Math 211	99
		Art 301	77
Jones J.	Thomas P.	Math 100	92
		Math 175	76
		Art 110	79
		Music 210	88
Martin R.	Smith A.	Math 100	85
		Math 191	100

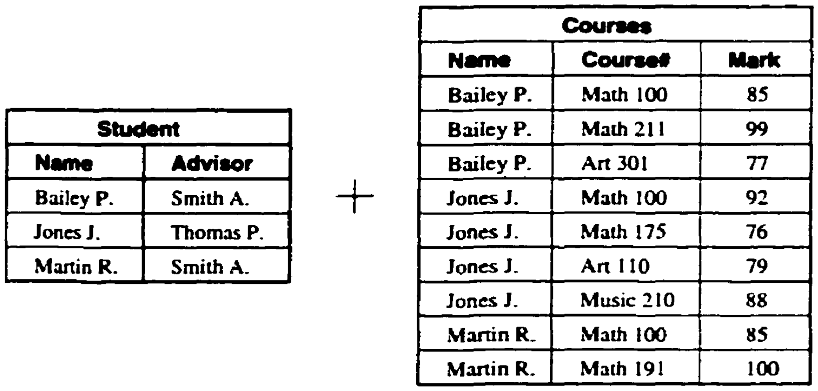
Figure 2.2: Nested Student Record Table

2. Minimize the number of attributes that must be updated when changes are made to the database, thereby making maintenance of the data easier and reducing the possibility of error.

There are several ways in which data in a database can be normalized, three initially defined by [Cod72a], [Cod72b], and some others defined by others since then. They are called *normal forms*. In order for a database to conform to the *first normal form (1NF)*, attributes must be atomic; that is, an attribute must not be an n-tuple and therefore can't be a set, list or, most importantly, a table or a complex object. This means that tables can not be nested in a 1NF database. Figure 2.2 shows a nested table that does not conform with 1NF. Figure 2.3 shows how the nesting of *Courses* is eliminated by creating a separate *Student* table and *Courses* table, and creating a relationship between these two tables, i.e. the student and his/her course records.

Adherence to the first normal form is a matter of the design of the database or relations. If the database does not support non-atomic attributes, then the user has no choice and conformity with the first normal form is guaranteed.

The second through fifth normal forms (hereafter the higher normal forms) define certain conditions for each of the normal forms that must be met. For example, the second normal form declares that if a table has a multi-valued key and contains an attribute that depends on only part of a multi-valued key, then that attribute should be moved to a separate table. The conversion of the table in Figure 2.4 which is necessary to achieve 2NF conformance is shown in Figure 2.5. The example illustrates how conforming to 2NF can reduce the amount of data

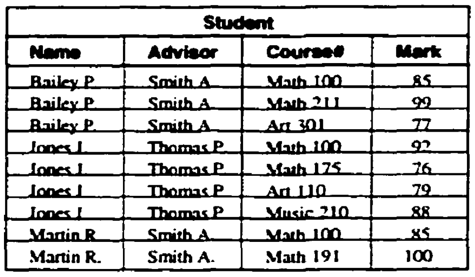


Student	
Name	**Advisor**
Bailey P.	Smith A.
Jones J.	Thomas P.
Martin R.	Smith A.

+

Courses		
Name	**Course#**	**Mark**
Bailey P.	Math 100	85
Bailey P.	Math 211	99
Bailey P.	Art 301	77
Jones J.	Math 100	92
Jones J.	Math 175	76
Jones J.	Art 110	79
Jones J.	Music 210	88
Martin R.	Math 100	85
Martin R.	Math 191	100

Figure 2.3: 1NF-conformant Student Record Table



Student			
Name	**Advisor**	**Course#**	**Mark**
Bailey P.	Smith A.	Math 100	85
Bailey P.	Smith A.	Math 211	99
Bailey P.	Smith A.	Art 301	77
Jones J.	Thomas P.	Math 100	92
Jones J.	Thomas P.	Math 175	76
Jones J.	Thomas P.	Art 110	79
Jones J.	Thomas P.	Music 210	88
Martin R.	Smith A.	Math 100	85
Martin R.	Smith A.	Math 191	100

Figure 2.4: 1NF(but Non-2NF) conformant Student Record Table

stored in the database and the number of values that must be modified when a change is made. In Figure 2.4, the *Advisor* name is stored once for every occurrence of a student record. In Figure 2.5, the *Advisor* name is stored only for each student. If it were necessary to change a student's advisor, there would be many fewer fields in Figure 2.5 that would require updating than in Figure 2.4.

Student	
Name	**Advisor**
Bailey P.	Smith A.
Jones J.	Thomas P.
Martin R.	Smith A.

+

Courses		
Name	**Course#**	**Mark**
Bailey P.	Math 100	85
Bailey P.	Math 211	99
Bailey P.	Art 301	77
Jones J.	Math 100	92
Jones J.	Math 175	76
Jones J.	Art 110	79
Jones J.	Music 210	88
Martin R.	Math 100	85
Martin R.	Math 191	100

Figure 2.5: 2NF-conformant Student Record Table

3NF, 4NF and 5NF similarly define increasingly stringent requirements, and adherence to each likewise can reduce storage space, the number of updates required, or both.

The normalization technique has been discussed by Ullman [Ull82] and by Date [Dat81], while several others have presented informal outlines of it [Gra85], [Ken83], [KS86], [Sal86]. Yao [Yao85] and Ceri et al. [CG86] have summarized the various normalization algorithms that are available, including some of their own modifications. Yang [Yan86] has discussed a graph-theoretic approach to normalization.

2.3 Limitations of 1NF Relational Databases

As with any relational database system, conformance to the higher normal forms is completely up to the database designer - the software imposes no constraints that prevent attaining an optimal schema, whether fully conformant or not. But databases that provide for the storage of atomic values give the designer no choice but to conform with 1NF. Conforming with the higher normal forms generally produces an optimal schema, albeit at the expense of greater complexity. But database conformance with 1NF often increases the amount of storage used, makes maintenance more difficult, and most importantly greatly increases the processing required to produce results, while still making the schema more complex. When comparing Figure 2.2 and Figure 2.3, the follow observations can be achieved:

- The number of tables increases from one to two.
- Normalization of the tables requires the student *Name* attribute to be stored twice for each student.
- Producing a report to show the student data requires that the two tables be joined. Joins are highly compute-intensive operations.

For some potential users of relational databases, the joins that would be required to resolve relations in 1NF databases would affect performance enough to preclude the use of relational databases. For example, 1NF relational databases are generally acknowledged to be unacceptable for CAD/CAM systems, which are used to design mechanical parts for manufacturing [MRS88]. One reason is that CAD/CAM data are inherently hierarchical in nature and the database structure used to store the part information must be traversed very quickly in order to display the part on the user's screen within an acceptable response time. Hundreds or thousands of join operations are required to display a complex part. These joins simply cannot be performed fast enough to provide acceptable display times. That is one reason 1NF databases are not used for CAD/CAM data. Such systems instead use proprietary hierarchical databases that provide high performance but are expensive to develop and maintain.

Apart from performance considerations, 1NF relational databases also have practical limitations for many applications. While any hierarchical database schema can validly be translated to a 1NF relational database schema, the practical considerations in doing so are daunting. Take for example a mechanical part. A hierarchical structure naturally and compactly stores the data that describe the part. The translation (mapping) of that hierarchical structure to a 1NF schema, however, is far from intuitive and leaves a confusing, awkward, complicated set of interrelated tables, including many tables for storing relationship relations. As a practical matter, such schemas are not possible to implement. These same considerations apply to many other types of data.

2.4 Nested Relation Model

Consideration of the limitations imposed by the 1NF constraint lead naturally to the question, "Can the 1NF constraint be removed from relational database without invalidating the underlying relational model?"

As mentioned earlier, the relational model has a mathematically rigorous definition to guarantee predictable, correct results on any database systems that faithfully implements the model. Detailed examination [AB84] [FT83] [KK89] [Mak77] [SP82] [SS86] has been made of the relational model with the 1NF constraint removed. The analysis has proven that the resulting model is equally robust. In other words, removing the 1NF constraint will not cause a relational database to produce invalid or inconsistent results as long as the database conforms to higher normal forms.

The removal of the 1NF restriction has led to investigations which retain much of the advantages of the relational model. The need to introduce complex objects into relations in order to make them more qualified to handle non-business data processing applications such as image and map processing, CAD/CAM, office automation, expert systems and certain scientific applications was realized in the late 1970's and lead to the introduction of nested relations [Mak77] and non-first-normal-form (NF^2)) [JS82].

Due to extensive research, significant progress has been made in the field of nested relations since the nested relational model was first proposed in 1977 [Mak77]. Fisher and Van Gucht [FG85] discussed the one-level nested relations and developed a polynomial time algorithm to test if a structure is an one-level nested relation. Jaeschke and Schek [JS82] introduced a generalization of the ordinary relational model by allowing relations with set-valued attributes and adding two restructuring operators, the **nest** and **unnest** operators, to manipulate such (one-level) nested relations. Thomas and Fischer[TF86] generalized Jaeschke and Schek's model and allowed nested relations of arbitrary (but fixed) depth. The definition of recursively nested relations was also discussed [LS88].

On the other hand, various query languages have been introduced for the nested relational model, and extensions have been proposed to practical query languages such as SQL to accommodate nested relations [PA86] [KR89] [PT86]. Graphics-oriented query languages [HP87] and datalog-like languages [BK86] [BNR+87] have been introduced for this model or slight generalizations of it. Also, various groups [BRS82] [DKA+86] [DPS86] [SPS87] have started with the implementation of the nested relational database model, some on top of an existing database management system [DG88] [SAB+89], others from scratch.

2.4.1 Nested Relations

Most information can be represented in a hierarchcal structure. The hierarchical database structure is base on a tree structure. Every data item except the roots of trees has a parent in the structure and may be the parent for other data items. To illustrate this idea, let us consider an example of a database for a university with a record for each department. Each department has students and professors. Each student has an advisor and a list of courses etc. These relationships can be represented diagrammatically as a tree, as shown in Figure 2.6.

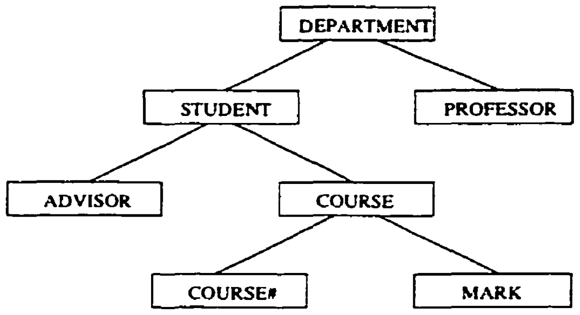


Figure 2.6: Schematic of Hierarchical Structure Example

As well, the contents of above information structure can be illustrated more or less like that shown in Figure 2.7.

Alternatively, the information can be described in a table of format of nested relations as illustrated in Firgure 2.8.

The relation *Student* in Figure 2.8 gives an example of nesting. Relation *Student* consists of three tuples each having three attributes:

- Name: The name of the student. Its data type is string (atomic).

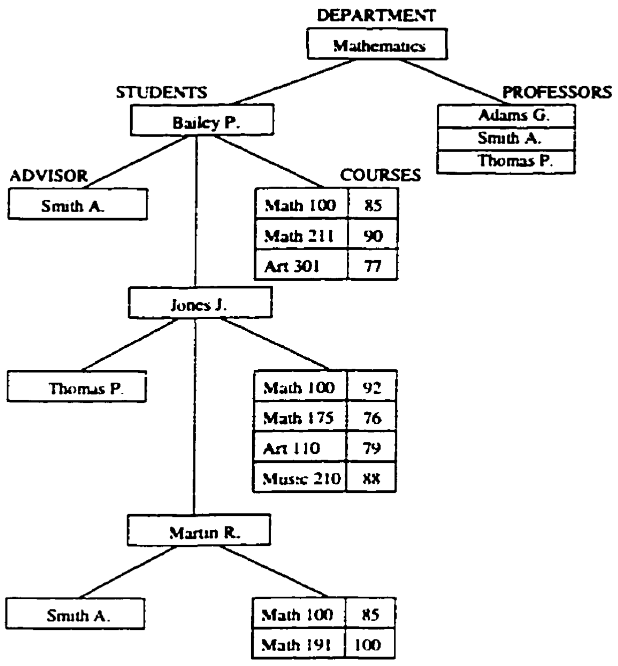
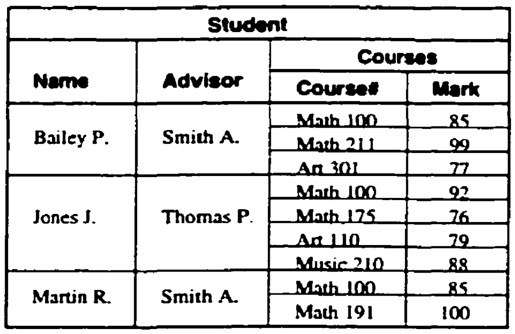


Figure 2.7: An Example of a Hierarchical Record



Student			
Name	Advisor	Courses	
		Course#	Mark
Bailey P.	Smith A.	Math 100	85
		Math 211	99
		Art 301	77
Jones J.	Thomas P.	Math 100	92
		Math 175	76
		Art 110	79
		Music 210	88
Martin R.	Smith A.	Math 100	85
		Math 191	100

Figure 2.8: An Example of a Nested Relation Representation

- Advisor: The name of this student's advisor. Its data type is string (atomic).
- Courses: A nested relation containing the course information the student is registered in. Each tuple in relation *Courses* contains a whole relation as an attribute. The first tuple contains a relation with 3 tuples. The second tuple contains a relation with 4 tuples.

The NF^2 relations have some advantages over 1NF relations, such as:

- Nested relations minimize redundancy of data. Related information can be stored in one relation only without redundancy. For example, if relation ***Student*** in Figure 2.8 were to be represented by 1NF, either it would have had to have redundant values for attribute Name and Advisor, or it would have had to be split into two different relations i.e. ***Student*** and *Courses*, with a foreign key *Course#*;
- Nested relations allow efficient query processing since some of the joins are realized within the nested relations themselves. In our example in Figure 2.8, if information about the student's marks needed to be retrieved in the 1NF representation, a join would have had to be performed between ***Student*** and *Courses*, while no joins are needed in the NF^2 representation.
- Low level implementation techniques such as clustering and repeating fields can be represented using the formalism defined by the nested relation model.

2.4.2 Abstract Data Types for Domains

A traditional database application involves storing large numbers of similar records of a few varieties, with insertions, updates, deletions, and simple queries being performed on these data. Recently, many application areas with more complex and varied data are being explored, with quantities of data being large and important enough that archiving these data in a database is desirable to help organize and keep track of the data as well as to gain security and consistency. Such applications might have variable-length character strings which are very long, such as abstracts or full text of articles or books, geographic maps, information describing a single television image, the pixels for a raster scan image, programs and their version, VLSI chip designs, and so on. For such applications, some attributes in each relation will be of the standard, built-in types, whereas other attributes will be defined as being of "new data types" such as "program" or "picture" [OH86].

The nested relational model allows abstract data types to be defined for domains, and allows operations to be defined on them. In the nested relational model, the domain algebra subsumes the functionalities of relational algebra, which means domain algebra is also capable of relational operations such as selections, projections and joins. A so-called *virtual domain* that is defined by domain algebra could have a complex data type of *relation* instead of atomic types of integer or float etc. As well, the virtual domains are capable of implying different kinds of relational operations performed upon other non-virtual domains which are lower-level relations. This gives a equivalent scheme of *abstract data type* in the relational data model.

2.4.3 Nesting and Unnesting

In the literature, defining nested relational models was done by extending relational operators to nested relations, and adding two restructuring operators, nest and unnest. The nest operator creates partitions which are based on the formation of equivalence classes. Tuples are equivalent if the values of the same attributes which are not nested are the same in the different tuples. All equivalent tuples are replaced with a single tuple in the resulting relation; the attributes of this tuple consists of all the attributes that are not nested, having the common value in the original tuples, as well as a nested relation whose tuples are the values of the attribute to be nested.

$$\boxed{\mathbf{UNNEST}_{\mathrm{attribute}}\ (\mathbf{NEST}_{\mathrm{attribute}}\ (\ \mathit{Relation}\ ))\ =\ \mathit{Relation}}$$

Figure 2.9: Nest and Unnest (which holds)

The Unnest operator undoes the result of the nest operator. It creates a new relation whose tuples are the concatenation of all the tuples in the relation being unnested to the other attributes in the relation. Thus the equation in Figure 2.9 always holds. On the other hand, however, the reverse does not necessarily hold, i.e. the equation in Figure 2.10 is not always true.

$$\boxed{\textbf{NEST}_{\text{attribute}}\ (\textbf{UNNEST}_{\text{attribute}}\ (\textit{Relation}\,))\ \overset{?}{=}\ \textit{Relation}}$$

Figure 2.10: Nest and Unnest (which does not always hold)

As the price of the advantages over 1NF relations, nested relations pose a non-trivial problem of data representation. That is, generally there are alternative representations of data in a nested relation, while the data is uniquely represented by a 1NF relation.

There are two different assumptions [Tak89] with respect to whether these alternative representations are the same. The first one [Mak77] takes the structure of the representation into account, since it catches certain semantics of the application. It follows that each nested relation should be recognized as a unique representation of data. This assumption, however, has a drawback that the information is lost when it is normalized into a set of simple tuples. This assumption poses a semantic gap between 1NF and nested form relations although it enables us to represent complex objects in a natural way using nested relations. The second one, on the contrary, assumes that each set of values is just a union of single values rather than a specific object. This assumption allows us to identify the different nested representations with their unique 1NF relation. In fact, this is an implicit assumption of many research papers such as transformation between 1NF relations and nested relations using NEST and UNNEST operators [JS82] [FT83], designing nested relations [OY87], and data manipulation [AMM83].

2.4.4 Domain Algebra for Nested Relations

In order to implement the nested relation model, not only is the relational algebra required to handle different joins of nested relations, more importantly, the domain algebra must be capable of dealing with lower-level nested relations. With a domain algebra capable of nested relational operations, the *nesting* and *unnesting* of relations can be easily performed, which ensures that any hierarchical database schema can be validly translated into a 1NF relational database schema. On the other hand, the potential of defining ***abstract data types*** for domains can be achieved by nested domain algebra as well.

In the jRelix system concerning the implementation of the domain algebra for nested relations, the basic strategy is to integrate the notion of "nested relations" into the domain algebra and make relational expressions part of domain expression so that relational operations can be applied to the nested attributes/domains. This way, any operations that are performed on the top level relations can be performed at the lower level attributes which are also relations.

Clearly, in order to fulfill this achievement, the domain algebra must be given more power, i.e. apart from its original ability to perform domain operations, it must be competent in relational operations as well. In fact, the way jRelix domain algebra was designed and implemented shows that domain algebra is a super set of both traditional domain algebra and relational algebra, i.e. it subsumes almost all the functionalities of relational algebra.

Since I was mainly responsible for implementation of domain algebra in jRelix, I will give much more details about this in the rest of this thesis.

2.5 Introduction to Relix

Relix, a **Rel**ational database programming language in U**nix**, was developed at the Aldat lab of School of Computer Science, McGill, starting in 1986 [Lal86]. Relix is based on an algebraic data manipulation language proposed by Merrett [Mer77]. It is basically an experimental interactive environment built to explore the concept of the relational database model described in [Mer84]. This section discusses the conceptual framework of the existing Relix system. Since current implementation of jRelix is heavily based on the existing Relix system, a background knowledge of Relix helps the reader to better understand the rest of this thesis.

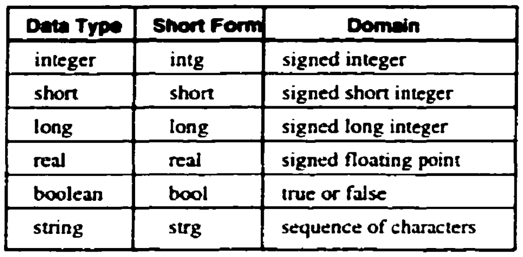
Generally speaking, Relix is an interpreted language written in the C programming language. It can accept and execute commands or statements interactively from the command line; while it also can run a batch file of Relix commands and statements.

2.5.1 Domains and Relations

Relix mainly deals with two kinds of data, i.e. domains and relations. A relation is defined on one or more attributes, and the data for a given attribute is from a particular domain of values.

The domain of a given attribute determines its data type.

There are totally six atomic data types defined in the original Relix system, which are illustrated in Figure 2.11. Some complex data types such as *nested relation* were implemented subsequently following the development of the system [He97].



Data Type	Short Form	Domain
integer	intg	signed integer
short	short	signed short integer
long	long	signed long integer
real	real	signed floating point
boolean	bool	true or false
string	strg	sequence of characters

Figure 2.11: Atomic Data Types Defined in Relix

Given the relation illustrated in Figure 2.1, Figure 2.12 shows the declaration and initialization of a Relix domain and relation.

```
>domain Name string;
>domain Course string;
>domain Mark integer;
>relation StudentRecord(Name, Course, Mark) <-
  {("Bailey P.", "Math 100", 85), ("Bailey P", "Math 211", 99),
  {("Bailey P.", "Art 301", 77), ("Jones J.", "Math 100", 92),
  {("Jones J.", "Math 175", 76), ("Jones J.", "Art 110", 79)};
```

Figure 2.12: Declaration and Initialization in Relix

2.5.2 Relational Algebra

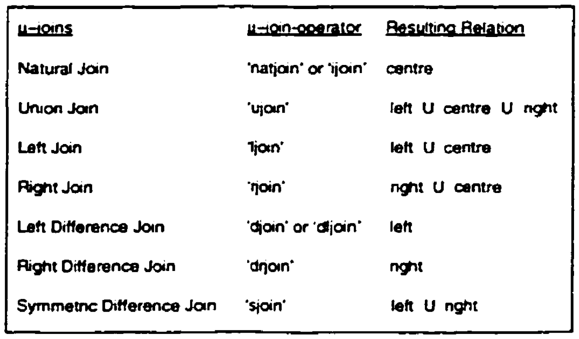
Relix supports relational algebra operations including selection, projection, μ-joins and σ-joins. *Selection* is the operation that creates a new relation by extracting specific tuples that satisfy certain conditions from the source relation; while *projection* is the operation that creates a new relation by extracting named domains from the source relation.

μ-joins are derived from the set operations such as intersection, union and difference etc. The μ-joins on two relations are based on three parts:

1. **center**, the combined tuples of the two relations that have equal values on the join attributes.

2. **left**, the tuples of the left operand relation, such that the value of its join attributes is the difference between the value of join attributes of the left operand relations and the right operand relation.

3. **right**, the tuples of the right operand relation, such that the value of its join attributes is the difference between the value of join attributes of the right operand relations and the left operand relation.



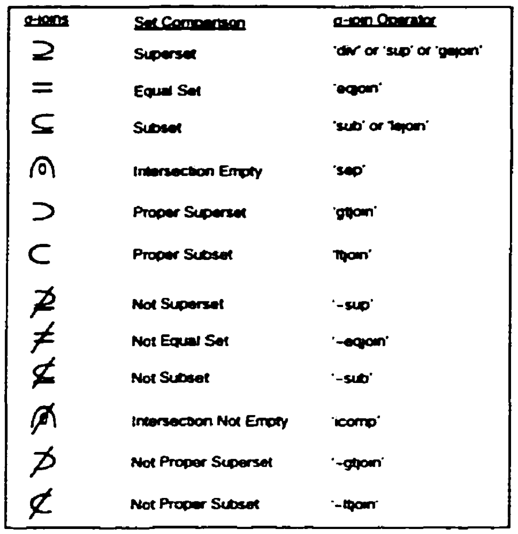
μ-joins	μ-join-operator	Resulting Relation
Natural Join	'natjoin' or 'ijoin'	centre
Union Join	'ujoin'	left U centre U right
Left Join	'ljoin'	left U centre
Right Join	'rjoin'	right U centre
Left Difference Join	'djoin' or 'dljoin'	left
Right Difference Join	'drjoin'	right
Symmetric Difference Join	'sjoin'	left U right

Figure 2.13: μ-joins in Relix

In Relix system, μ-joins include natural join (i.e. intersection join), union join, symmetric difference join, left and right joins, as well as left and right difference joins etc. They are illustrated in Figure 2.13.

On the other hand, σ-joins are based on set comparison operators and they include division (super set $\supseteq$), proper super set ($\supset$), equal set ($=$), proper subset ($\subset$), subset ($\subseteq$), intersection ($\cap$), and the their corresponding negative operations. Figure 2.14 illustrates the sigma joins defined in the Relix system.

Readers may refer to [Mer84] for a formal definition and detailed explanation of both μ-join and σ-join implemented in Relix.



σ-joins	Set Comparison	σ-join Operator
$\supseteq$	Superset	'div' or 'sup' or 'gejoin'
$=$	Equal Set	'eqjoin'
$\subseteq$	Subset	'sub' or 'lejoin'
$\overset{0}{\cap}$	Intersection Empty	'sep'
$\supset$	Proper Superset	'gtjoin'
$\subset$	Proper Subset	'ltjoin'
$\not\supseteq$	Not Superset	'~sup'
$\neq$	Not Equal Set	'~eqjoin'
$\not\subseteq$	Not Subset	'~sub'
$\not\overset{0}{\cap}$	Intersection Not Empty	'icomp'
$\not\supset$	Not Proper Superset	'~gtjoin'
$\not\subset$	Not Proper Subset	'~ltjoin'

Figure 2.14: σ-joins in Relix

2.5.3 Domain Algebra

Relational algebra considers relations to be the data primitives [Mer84] and therefore does not provide the power to manipulate attributes. As a result, domain algebra is proposed to overcome this problem [Mer77]. Even though attribute and domain hold different meanings, they are sometimes used interchangeably in Relix literature.

Apart from creating a domain by declaring its type as illustrated in Figure 2.12, a new domain can be created by expressing the domain as operations on existing domains. Domains defined in this way are called "*virtual domains*" in the sense that there are no actual values associated with them. The value of virtual domains is *actualized* in a Relix statement, notably, projection or selection etc.

The domain algebra is usually classified into two categories, i.e. horizontal and vertical operations. The horizontal operations work on a single tuple of a relation. The horizontal domain expressions are formed by applying renaming mechanisms, mathematical operators, predefined functions, and if-then-else clauses on constants or attribute names. On the other hand, vertical operations are those domain algebra operations that combine values from more than one tuple in a relation. They include simple reduction, equivalence reduction, functional mapping, and partial functional mapping. Given the relation illustrated in Figure 2.1, Figure 2.15 shows some examples of declaring virtual domains using both horizontal and vertical operations of domain algebra. Readers may refer to [Mer84] for a formal definition and detailed explanation on domain algebra operations implemented in Relix.

```
>let StudentName be Name;          // Renaming
>let NewMark be Mark-5;            // Arithmatic operation

// If-then-else clause
>let Result be if Mark>=60 then "Pass" else "Fail";

// Simple reduction
>let Average be (red+ of Mark)/(red+ of 1);

// Equivalent reduction
>let SubTotal be equiv+ of Mark by Name;
```

Figure 2.15: Examples of Domain Algebra in Relix

2.6 Scope of the Present Work

This thesis discusses the design and implementation of Relix as a database language which is built with a new Object-Oriented programming language, i.e. Java. The purpose of our work is to extend Relix with a set of full-fledged functionalities for the nested relational model. The system developed in accordance with this thesis is called *jRelix* (i.e. Java implementation of Relix). A detailed description of how to use jRelix to perform relational database operations and programming is firstly given. The concept and idea for jRelix design and implementation in general, and the domain algebra for nested relations (which was my major responsibility) in particular, will be explored thereafter.

Chapter 3

User's Manual on jRelix

This chapter serves as a jRelix tutorial. It describes how to use jRelix to perform relational database operations and programming. Section 3.1 describes how to start and exit the jRelix system. Section 3.2 explains how to declare domains and relations, and how to initialize a relation (both flat and nested). In this section, jRelix data types will also be introduced briefly. Section 3.3 tells reader how to remove a declared domain or relation, and what kind of restrictions may be encountered when trying to remove a domain or relation. The fundamental operations of relational algebra e.g. projection, selection and joins etc. will be explored in section 3.4. Subsequently in section 3.5, the use of domain algebra operations will be explained in detail. In section 3.6 and section 3.7, the usage of views and computations will be briefly introduced. Details can be found in [Hao98] and [Bak98]. Finally in section 3.8, some of the more advanced system commands in jRelix will be presented.

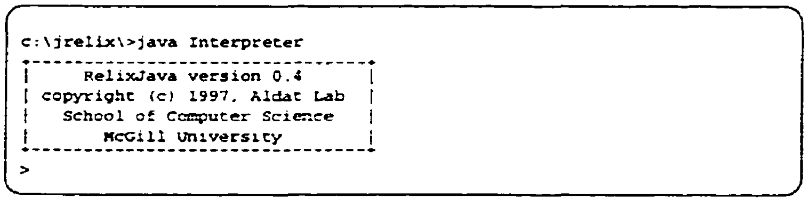
In this manual, the jRelix commands are basically introduced in a practical way which is easy to understand and yet sufficient for basic operations. On the other hand, readers who are interested in details can find a complete description on jRelix command syntax in Appendix A.

3.1 Starting and Exiting jRelix

Suppose both the Java run-time system and jRelix software are successfully installed on the user system. To start jRelix, the following command is typed on the command line of the operating system:

> java Interpreter

As the result, jRelix copyright information is displayed in its run-time environment, as illustrated in Figure 3.1. After certain internal initializations, jRelix shows its prompt sign "> " and waits for user input.



```
c:\jrelix\>java Interpreter
+--------------------------------+
|      RelixJava version 0.4     |
| copyright (c) 1997, Aldat Lab  |
|   School of Computer Science   |
|        McGill University       |
+--------------------------------+
>
```

Figure 3.1: Initial Screen upon Starting jRelix

To exit the system, the user types "quit;" after the system prompt sign. Upon receiving this command, jRelix performs its clean-up procedure and then returns to the original operating system.

In the jRelix environment, it is required that commands and statements end with a semi-colon (";"). Multiple lines of commands can be entered by the user but jRelix only starts to interpret the command when it catches a semi-colon, which serves as an *end-of-command* signal. This provides an efficient way of inputting multi-line commands and statements in jRelix.

3.2 Declarations

When entering a new jRelix environment, the first thing a user may want to do is to declare some attributes (i.e. domains) and relations which are based on the attributes already declared. This section describes both domain and relation declarations.

As we know, a relation is defined on one or more attributes, and the data for a given attribute is from a particular domain of values. The domain of a given attribute determines its data type. jRelix provides several data types, which will also be introduced in this section.

The terms "*domain*" and "*attribute*" usually have different meanings. However, throughout this thesis, these two terms are used interchangably. In general, they both refer to the same concept as "*attribute*". In the case a conflict of concepts exists, readers will be notified explicitly.

3.2.1 Declare Domains

There are two kinds of actual domain declarations in jRelix, i.e. atomic-typed domains and complex-typed domains. The term *"atomic data type"* means the primitive types such as integer, string etc., as opposed to *"complex data type"* such as text, statement, computation and nested relation etc.

Figure 3.2 gives some examples of declaring atomic-typed domains.

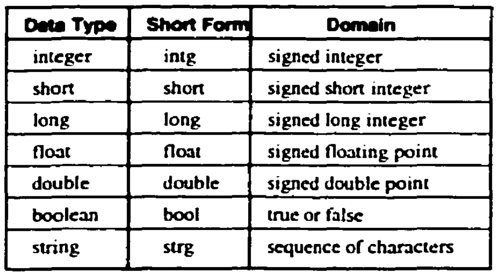
```
>domain A intg;
>domain B float;
>domain C long;
>domain D bool;
>domain E string;
>
```

Figure 3.2: Declaring Atomic-Typed Domains

In general, the syntax used to declare a domain/attribute of atomic data type is as follows:

> **domain** *dom_name1, dom_name2 data_type*;

Note that jRelix provides seven atomic data types for domain declaration, as showed in Figure 3.3.

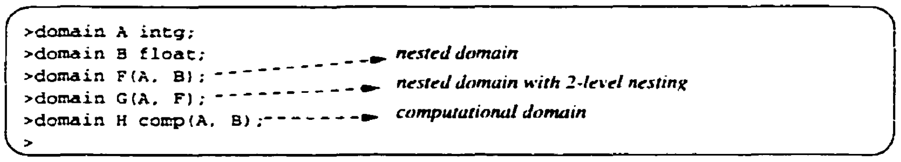


Data Type	Short Form	Domain
integer	intg	signed integer
short	short	signed short integer
long	long	signed long integer
float	float	signed floating point
double	double	signed double point
boolean	bool	true or false
string	strg	sequence of characters

Figure 3.3: Atomic Data Types in jRelix

On the other hand, two complex data types have been implemented in current jRelix. i.e. nested relation and computation. Nested relational domain is used when the attribute in a relation is a further relation, i.e. the attribute in a table is a table as well. This mechanism constructs a nested table, and as a matter of fact, multi-level (though not recursive) nesting is allowed in jRelix. A computational domain is a virtual computation which will be actualized based on the actual tuple data in the source relation [Bak98].

Figure 3.4 gives some examples of declaring both nested relational domains and computational domains.



```
>domain A intg;
>domain B float;
>domain F(A, B);
>domain G(A, F);
>domain H comp(A, B);
>
```

Figure 3.4: Declaring Complex-Typed Domains

Note that in Figure 3.4, domain *F* is a nested domain which is defined on atomic-typed domains *A* (integer) and *B* (float). Domain *G* is a 2-level nested domain which is defined on an atomic-typed domain *A* (integer) and a complex-typed domain *F* (i.e. nested domain). Something that needs to be mentioned here is that when a new nested domain is declared, an invisible relation (whose name starts with a ".") is created automatically in the system. This relation is supposed to hold the data that belong to the nested domain in question. The invisible relation can be seen by using a jRelix command introduced in section 3.8.2, while its contents can be printed by a command described in section 3.2.5, although readers do not need to bother with that at the present stage.

In general, the syntax used to declare a nested relational domain is as follows:

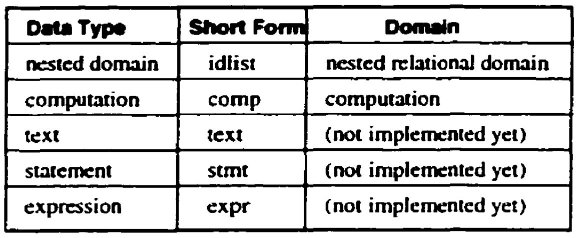
> \> **domain** *nest_dom_name(dom_name1, dom_name2, ...)*;

As well, the syntax for declaring a computational domain is as follows.

> \> **domain** *comp_dom_name* **comp** *(dom_name1, dom_name2, ...)*;

It is required by current jRelix implementation that the domains on which a new nested relational domain or computational domain is defined must be declared already, i.e. the domains of *(dom_name1, dom_name2, ...)* in above syntax must be declared and be existing in the system; Otherwise, a *"domain not found"* warning message is generated and the user is notified.

Finally as a complement, Figure 3.5 lists five complex data types that are included in the jRelix system (though not completely implemented yet).



Data Type	Short Form	Domain
nested domain	idlist	nested relational domain
computation	comp	computation
text	text	(not implemented yet)
statement	stmt	(not implemented yet)
expression	expr	(not implemented yet)

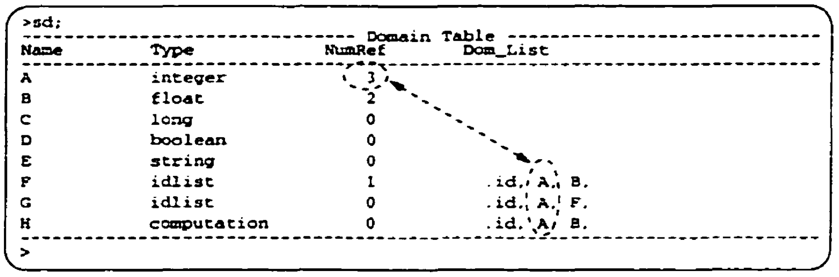
Figure 3.5: Complex Data Types in jRelix

3.2.2 Show Declared Domains

This section describes how to display the domain items that have been declared in the system. This is particularly useful when user wants to check if the domains are declared correctly or to see which domains are available for further relational declaration. On the other hand, readers who are more interested in relation declaration and relational operations may skip this section and jump to section 3.2.3 for information on relation declaration; and refer back to this section later when a need occurs.

The command to list all domains that have been declared in the system is "**sd;**". Given the domains declared in the previous sections, a sample output of this command is shown in Figure 3.6.

It is clear that domain information is displayed in a table format with four fields, i.e. *Name, Type, NumRef and DomList.* A type of *idlist* indicates a nested relational domain, with corresponding *DomList* field indicating the attributes/domains on which the current nested domain is defined. The *NumRef* field contains an integer value called *"reference counter"* that indicates how many times current domain is used by other domains or relations. For example, domain *A* in Figure 3.6 is used by domains *F, G* and *H*, hence its *NumRef* value is *3*. Needless to say that

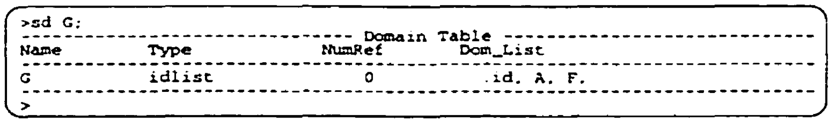


```
>sd;
------------------------------ Domain Table ------------------------------
Name          Type               NumRef        Dom_List
--------------------------------------------------------------------------
A             integer              3
B             float                2
C             long                 0
D             boolean              0
E             string               0
F             idlist               1          .id. A. B.
G             idlist               0          .id. A. F.
H             computation          0          .id. A. B.
--------------------------------------------------------------------------
>
```

Figure 3.6: Sample Output of "**sd;**" Command

when a new nested domain or a new relation is declared, the *NumRef* value of the referenced domains will be incremented by 1. Later in section 3.3, we will see that a domain is not allowed to be removed when it is used by any other domains or relations, i.e. when its *NumRef* value is not equal to *0*.

When "**sd**" is followed by a domain name, this particular domain's information will be displayed as illustrated in Figure 3.7. If the relevant domain is not found in domtable, a *"domain not found"* warning message will be generated.



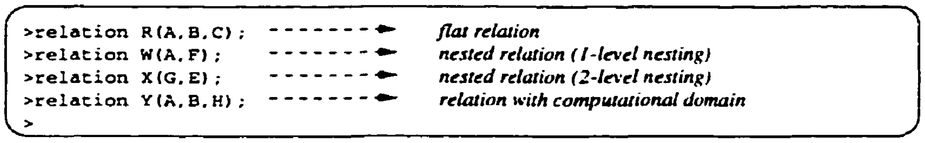
```
>sd G;
------------------------------ Domain Table ------------------------------
Name          Type               NumRef        Dom_List
--------------------------------------------------------------------------
G             idlist               0          .id. A. F.
--------------------------------------------------------------------------
>
```

Figure 3.7: Sample Output of Displaying a Particular Domain Information

Finally, combined with a command described in section 3.8.2, the "**sd**" command can also display some invisible domains which are so-called "*system domains*". Details will be presented later.

3.2.3 Declare and Initialize Relations

As mentioned before, relations are defined on one or more attributes (or domains) which must have been declared before the relation is declared or initialized. Otherwise, a "*domain not found*" error message will be generated and the declaration fails. Figure 3.8 gives some examples of relation declaration.



```
>relation R(A,B,C);    -------→   flat relation
>relation W(A,F);      -------→   nested relation (1-level nesting)
>relation X(G,E);      -------→   nested relation (2-level nesting)
>relation Y(A,B,H);    -------→   relation with computational domain
>
```

Figure 3.8: Declare Relations

In this figure, relation *R* is a flat relation since it is solely defined on atomic data types. Relation *W* is a nested relation with 1-level nesting since one of its attributes *F* is a nested domain; and relation *X* is a 2-level nested relation because of domain *G* (refer to Figure 3.4 for information on domains *F* and *G*). Finally, *Y* is a relation with computational domain (i.e. *H*) involved.

Meanwhile, it is not hard to see from above examples that the general syntax for declaring a relation is as follows:

> **relation** *rel_name*(*dom_name1*, *dom_nam2*, ...);

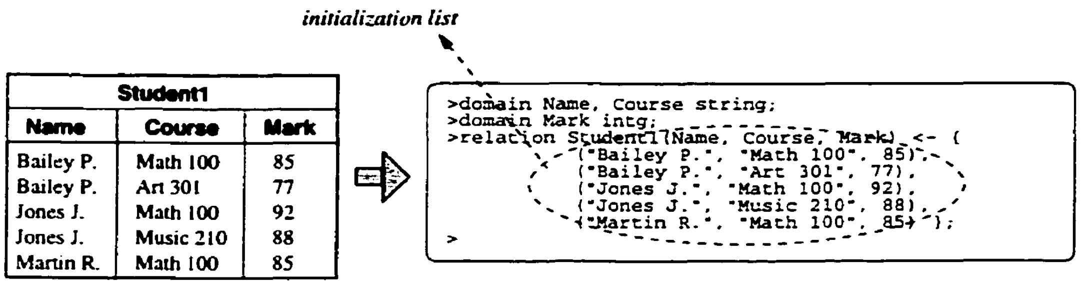
Note that the doamin *dom_name*'s can be any valid domain declared already in the system, e.g. atomic-data-typed domains, nested relational domains and computational domains etc. They can also be virtual domains which will be introduced later in section 3.5.1.

On the other hand, however, the syntax given above declares only a relation structure in the system, which means it is an empty relation without any tuple data inside. A relation can also be declared with actual data tuples. This is called *relation initialization*, and the tuple data is contained in a so-called *initialization list*. Some examples will be given below to illustrate different types of initialization.

Declare and Initialize Flat Relations

A flat relation is a relation whose domains are of atomic type. Usually a 1NF-relation is regarded as a flat relation, such as relation *Student1* in Figure 3.9.

As showed in this figure, the initialization list in a relation declaration is surrounded by a pair of curly brackets. Inside the curly brackets, each tuple is represented by a pair of round



Student1		
Name	**Course**	**Mark**
Bailey P.	Math 100	85
Bailey P.	Art 301	77
Jones J.	Math 100	92
Jones J.	Music 210	88
Martin R.	Math 100	85

```
>domain Name, Course string;
>domain Mark intg;
>relation Student1(Name, Course, Mark) <- (
        ("Bailey P.", "Math 100", 85),
        ("Bailey P.", "Art 301", 77),
        ("Jones J.", "Math 100", 92),
        ("Jones J.", "Music 210", 88),
        ("Martin R.", "Math 100", 85) );
>
```

Figure 3.9: Declare a Flat Relation

brackets separated by comma signs. Different domain values in each tuple are separated by commas as well.

In general, the syntax for relation initialization is defined as follows:

> **relation** *rel_name*(*dom_name1*, *dom_nam2*, ...) < − *initialization_list*;

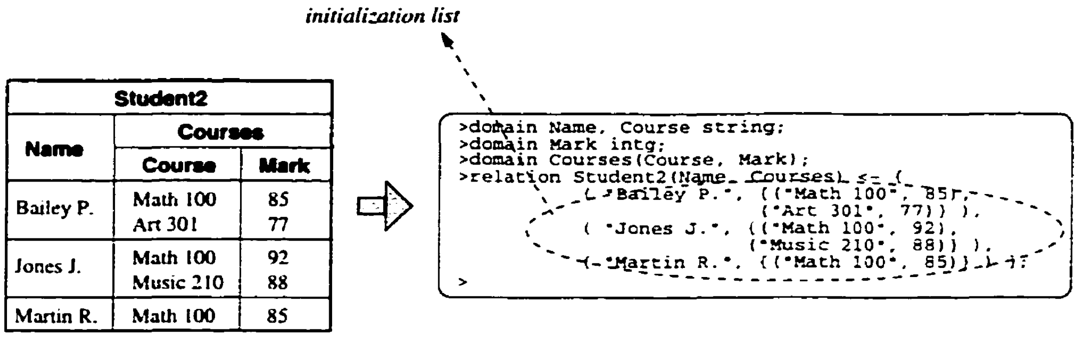
Different data types can be figured out in the initialization list by a type-tag associated with the actual values. For example, a long integer *101245* is represented as *101245l* with the trailing "*l*" implying a "*long*"-type; and strings are surrounded by quotation-marks as illustrated by the *Name* and *Course* field value in Figure 3.9. Some examples about the type-tag usage are give in Figure 3.10. Note that same rule is used when declaring constant domains e.g. "*let x be 23.8657d;*" etc.

Data Type	Short Form	Examples
integer	int	12, 150
short	short	12s, 78s
long	long	120l, 456l
float	float	23.5f, 125.45f, 2.1e8f
double	double	56.86d, 102.137d, 5.3e9d
boolean	bool	true, false
string	strg	"Mark P.", "12345"

Figure 3.10: Type-tags in Initialization List

Declare and Initialize Nested Relations

One of the most important features of jRelix is that nested relations (i.e. NF^2 relations) are supported; which means, for example, tables such as *Student2* in Figure 3.11 are allowed to further contain table fields (e.g. the *Courses* field is another table).



Student2		
Name	**Courses**	
	Course	**Mark**
Bailey P.	Math 100 Art 301	85 77
Jones J.	Math 100 Music 210	92 88
Martin R.	Math 100	85

```
>domain Name, Course string;
>domain Mark intg;
>domain Courses(Course, Mark);
>relation Student2(Name, Courses) <- {
          ( "Bailey P.", {("Math 100", 85),
                          ("Art 301", 77)} ),
          ( "Jones J.", {("Math 100", 92),
                         ("Music 210", 88)} ),
          ( "Martin R.", {("Math 100", 85)} ) };
>
```

Figure 3.11: Declare a Nested Relation

It is clear from this figure that same rule is used in the initialization list for nested relation declaration, i.e.

- A relation/table is always surrounded by a pair of curly brackets.
- Inside a relation, each tuple is surrounded by a pair of round brackets.
- Different tuples are separated by comma signs.

Obviously, this rule also applies to the nested domain fields, e.g. *Courses* in Figure 3.11. In other words, the *Courses* field values are themselves relations surrounded in curly brackets, as it is showed in the initialization list. Theoretically, jRelix supports multi-level nesting (though not recursive), but this will cause the initialization list to be much more complicated than what is showed in this example.

Something that needs to be clarified here is that, although it seems only one relation (e.g. *Student2* in this case) is initialized during a nested relation declaration, multiple relation initializations might potentially be involved. As mentioned in section 3.2.1, when a nested domain is declared, an invisible relation whose name is prefixed with a "." is created in the system automatically, and this relation is supposed to hold the data that belongs to the nested domain. Therefore, during a nested relation initialization, tuples for nested domains are saved in their corresponding relations which are not visible. In the example given in Figure 3.11, all the *Courses* data including *Names* and *Marks* are stored in the relation *.Courses* which was generated when the nested domain *Courses* is declared.

The linkage between the top-level relation (e.g. *Student2* in this case) and the relations associated with each nested domain (e.g. *.Courses*) is achieved through a so-called "*surrogate*", which is represented as a long integer in jRelix implementation. Figure 3.12 illustrates this mechanism.

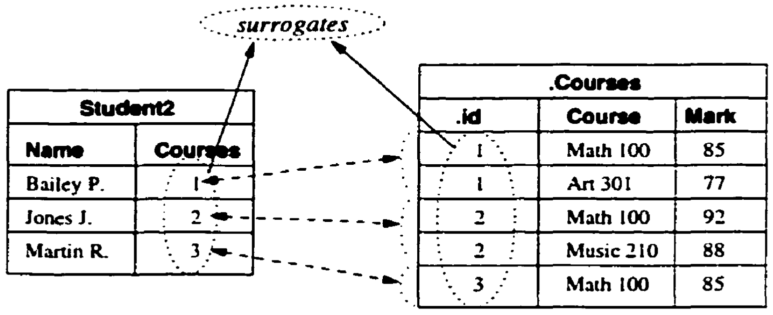


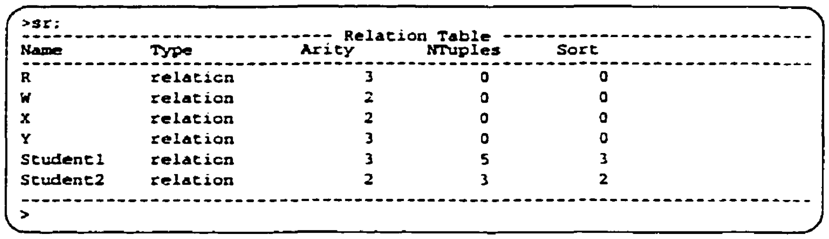
Figure 3.12: Link Two Relations Through Surrogates

Finally as a reminder, the invisible relations that are associated with nested domains can be listed by using a command introduced in section 3.8.2, and section 3.2.5 describes how to print the contents of a relation, regardless of whether it is visible or not.

3.2.4 Show Declared Relations

This section describes how to display the relation items that have been declared in the system. This is particularly useful when the user wants to check if the relations are declared correctly or to see which relations are available for further operation.

The command to list all relation entries that have been declared in the system is "**sr;**". Given the relations declared in previous sections, a sample output of this command is shown in Figure 3.13.

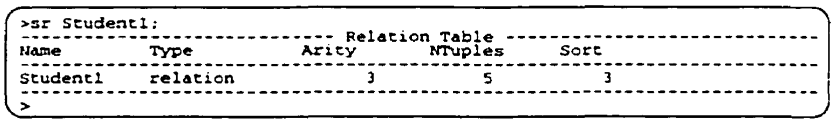


```
>sr;
------------------------------ Relation Table ------------------------------
Name         Type           Arity       NTuples      Sort
----------------------------------------------------------------------------
R            relation         3            0           0
W            relation         2            0           0
X            relation         2            0           0
Y            relation         3            0           0
Student1     relation         3            5           3
Student2     relation         2            3           2
----------------------------------------------------------------------------
>
```

Figure 3.13: Sample Output of "**sr;**" Command

Obviously, relation entries are displayed in a table format with five fields, i.e. *Name, Type, Arity, NTuples* and *Sort*. The *type* field indicates the type of the current entry, which can be "relation", "view" and "computation" as all of the three types are co-existent in the system. The *Arity* field contains an integer which tells how many attributes/domains this relation is defined on (however, another command need to be used in order to display exactly which domains are used in the current relation. Details in this connection will be described in section 3.2.6). In the case that a relation contains tuple data, field *NTuples* indicates how many tuples there are in that relation. For example in Figure 3.13, it is easy to know that relation *Student1* contains 5 tuples. Finally, the *Sort* field tells how many attributes the current relation is sorted on.

When "**sr**" is followed by a relation name, the information of this particular relation will be displayed as illustrated in Figure 3.14. If the relevant relation entry is not found in the system (e.g. relation is not declared), an error message will be displayed.



```
>sr Student1;
------------------------------ Relation Table ------------------------------
Name         Type           Arity       NTuples      Sort
----------------------------------------------------------------------------
Student1     relation         3            5           3
----------------------------------------------------------------------------
>
```

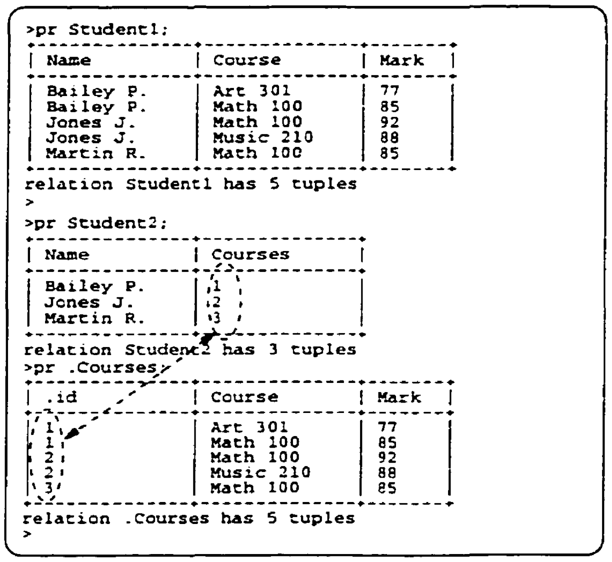
Figure 3.14: Sample Output of Displaying a Particular Relation Information

Note that the "**sr**" command also displays the information on all views and computations that have been declared in the system. Details will be explained in section 3.6 and section 3.7 respectively.

Finally, combined with a command described in section 3.8.2, the "**sr**" command can display those invisible relations mentioned in section 3.2.1 and section 3.2.3, as well as so-called "*system relations*" mentioned in section 3.2.6. Details will be presented later.

3.2.5 Print Contents of Relations

To print the contents of a relation, the command "**pr**" is used and followed by the relation name as illustrated in Figure 3.15.



```
>pr Student1;
+-----------------+-----------------+--------+
| Name            | Course          | Mark   |
+-----------------+-----------------+--------+
| Bailey P.       | Art 301         | 77     |
| Bailey P.       | Math 100        | 85     |
| Jones J.        | Math 100        | 92     |
| Jones J.        | Music 210       | 88     |
| Martin R.       | Math 100        | 85     |
+-----------------+-----------------+--------+
relation Student1 has 5 tuples
>
>pr Student2;
+-----------------+-----------------+
| Name            | Courses         |
+-----------------+-----------------+
| Bailey P.       | 1               |
| Jones J.        | 2               |
| Martin R.       | 3               |
+-----------------+-----------------+
relation Student2 has 3 tuples
>pr .Courses;
+-----------------+-----------------+--------+
| .id             | Course          | Mark   |
+-----------------+-----------------+--------+
| 1               | Art 301         | 77     |
| 1               | Math 100        | 85     |
| 2               | Math 100        | 92     |
| 2               | Music 210       | 88     |
| 3               | Math 100        | 85     |
+-----------------+-----------------+--------+
relation .Courses has 5 tuples
>
```

Figure 3.15: Sample Output of "**pr;**" Command

The general syntax is as follows, where *rel_name* can also be a relation name prefixed by a ".". In other words, the "**pr**" command prints a relation no matter the relation is visible or not. Needless to say, this helps to print the tuple data of nested domains.

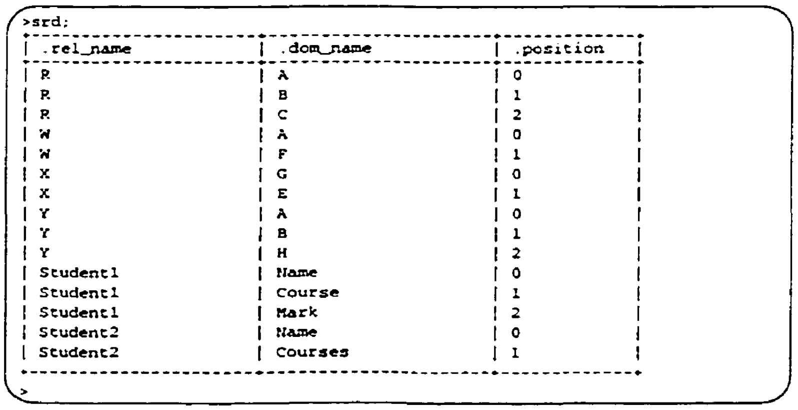
> **pr** *rel_name*

Ideally, the "**pr**" command should print the tuple data of a nested relation along with the data of all the nested domains. However, this is not implemented in the current jRelix system yet. As described in subsection "Declare and Initialize Nested Relations" of section 3.2.3, in the case of nested relations, the top-level relation is connected with the nested domain relations by long-integer-typed surrogates, as illustrated in Figure 3.12. Therefore, when the command "**pr**" is used to print the contents of a nested relation, the surrogate value instead of the nested domain tuples are printed, as illustrated by the output of *Student2* in Figure 3.15. In order to see the actual *Courses* data (e.g. course name and marks) instead of surrogates, another **pr** command has to be issued with the name of the nested domain *Courses* (prefixed by a ".") as the parameter, as illustrated in the same figure. It is clear that two relations (i.e. *Student2* and *.Courses*) are linked together by the surrogates.

Note that the "**pr**" command can also be used to print a view information when it is followed by a view name. Upon receiving this command, jRelix will evaluate and actualize the view based on its definition and corresponding tuples' data will be generated and printed on the fly. For details please refer to section 3.6.

3.2.6 Show Relation-Domain Information

As mentioned in section 3.2.4, the *Arity* field displayed by "show relation" command "**sr;**" only indicates how many attributes/domains the relation is defined on. In order to know exactly which domains are used in a relation (i.e. on which domains a relation is defined), a "show relation-domain (RD)" command "**srd;**" is provided by jRelix. This command basically displays the relationships between relations and domains, i.e. which relation is defined on which domains. Figure 3.16 is a sample output of this command, given the relations declared in previous sections.



```
>srd;
+------------------------------+------------------------------+---------------+
| .rel_name                    | .dom_name                    | .position     |
+------------------------------+------------------------------+---------------+
| R                            | A                            | 0             |
| R                            | B                            | 1             |
| R                            | C                            | 2             |
| W                            | A                            | 0             |
| W                            | F                            | 1             |
| X                            | G                            | 0             |
| X                            | E                            | 1             |
| Y                            | A                            | 0             |
| Y                            | B                            | 1             |
| Y                            | H                            | 2             |
| Student1                     | Name                         | 0             |
| Student1                     | Course                       | 1             |
| Student1                     | Mark                         | 2             |
| Student2                     | Name                         | 0             |
| Student2                     | Courses                      | 1             |
+------------------------------+------------------------------+---------------+
>
```

Figure 3.16: Sample Output of "**srd;**" Command

Obviously, "RD" entries are displayed in a table format with three fields, i.e. *rel_name,* *dom_name* and *position*. The *position* field indicates the index of a domain in the relation. For example, relation *Student2* is defined on two domains *Name* and *Courses*, while *Name* is the first attribute in the relation and *Courses* is the second one.

Note that the "**srd;**" command also produces information on system relations which are not illustrated in the sample figure. Although readers may not be interested in this information at the current stage, it is necessary to mention that there are three system relations maintained by jRelix system, i.e. *.rel, .dom* and *.rd*, which can also be used in normal relational operations such as joins etc.

3.3 Removing Domains & Relations

Removing an existing domain or relation is quite easy in jRelix. The syntax for removing a domain is as follows.

> **dd** *dom_name1, dom_name2, ...*;

And the syntax for removing a relation is as follows.

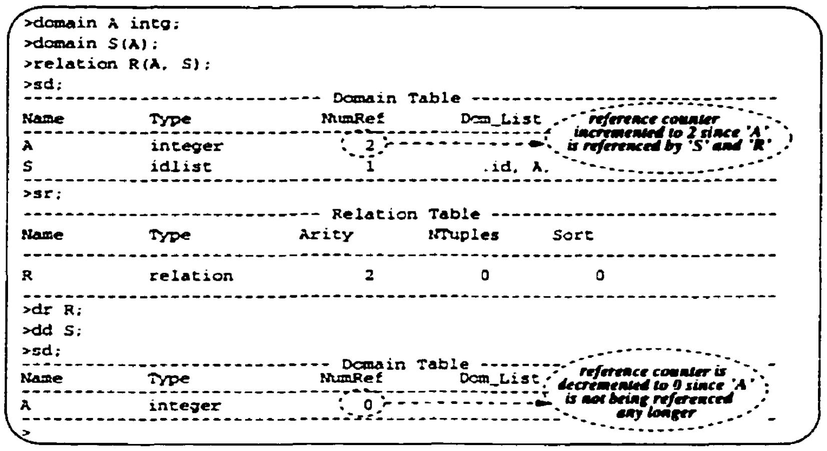
> **dr** *rel_name1, rel_name2, ...*;

Error messages will be displayed if the user tries to remove domains or relations which are not existing in the system, or if the user tries to remove a domain which is being used by other domains or relations (otherwise those domains or relations will reference to something which is not existing). On the other hand, after a domain or relation is removed, the *NumRef* field value (i.e. reference counter) of those domains that were used by the removed domain or relation will be decremented correspondingly. Figure 3.17 illustrates this case.

Finally as a remainder, when a nested relational domain is removed, the invisible relation that is associated with the nested domain will also be removed automatically.

3.4 Relational Algebra

Relations provide a simple but static structure that can be used to represent both entities and relationships. The relational algebra provides the operations needed to manipulate information stored logically in this form [RS95].



```
>domain A intg;
>domain S(A);
>relation R(A, S);
>sd;
---------------------------- Domain Table ------------------------------
Name         Type               NumRef        Dom_List
------------------------------------------------------------------------
A            integer              2
S            idlist               1            .id, A,
------------------------------------------------------------------------
>sr;
---------------------------- Relation Table ----------------------------
Name         Type             Arity       NTuples      Sort
------------------------------------------------------------------------
R            relation           2           0            0
------------------------------------------------------------------------
>dr R;
>dd S;
>sd;
---------------------------- Domain Table ------------------------------
Name         Type               NumRef        Dom_List
------------------------------------------------------------------------
A            integer              0
------------------------------------------------------------------------
>
```

reference counter incremented to 2 since 'A' is referenced by 'S' and 'R'
reference counter is decremented to 0 since 'A' is not being referenced any longer

Figure 3.17: Removing Domains and Relations

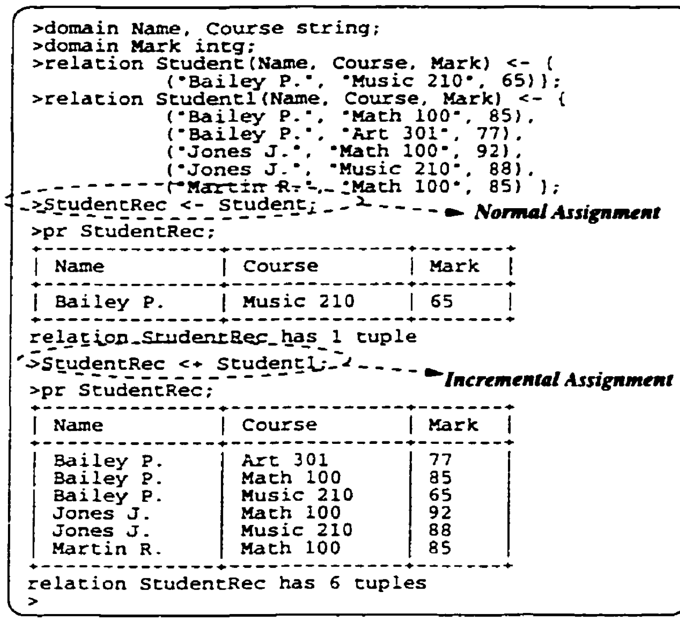
The relational operators are classified as unary or binary, depending on the number of their operands. Unary operators act on a single relation, binary operators act on two relation, and both produce a single relation as result.

This section firstly introduces a basic relational operation *assignment* which may be involved in other relational operations introduced later. After that, two relational operations associated with unary operators, i.e. *selection* and *projection* are described. A more flexible operation *tselection* which combines the two unary operations together is introduced thereafter. Binary operations i.e. various *joins* will be explored subsequently. In addition, a special relational operation *update* is described.

3.4.1 Assignment

The assignment operation assigns a "*relation value*" to a relation. In other words, it establishes an instance of a relation. There are two types of assignment in jRelix, i.e. *normal assignment* and *incremental assignment.* The former creates a new instance of the source relation, while the latter adds the tuple data of the source relation to the assigned relation. Figure 3.18 gives some examples of assignment operation.

In this figure, the first assignment creates a new relation instance *StudentRec* which has exactly the same tuple data as the source relation *Student*; whereas the second incremental



```
>domain Name, Course string;
>domain Mark intg;
>relation Student(Name, Course, Mark) <- {
           ("Bailey P.", "Music 210", 65)};
>relation Student1(Name, Course, Mark) <- {
           ("Bailey P.", "Math 100", 85),
           ("Bailey P.", "Art 301", 77),
           ("Jones J.", "Math 100", 92),
           ("Jones J.", "Music 210", 88),
           ("Martin R.", "Math 100", 85) };
>StudentRec <- Student;
>pr StudentRec;
+--------------+----------------+-------+
| Name         | Course         | Mark  |
+--------------+----------------+-------+
| Bailey P.    | Music 210      | 65    |
+--------------+----------------+-------+
relation StudentRec has 1 tuple
>StudentRec <+ Student1;
>pr StudentRec;
+--------------+----------------+-------+
| Name         | Course         | Mark  |
+--------------+----------------+-------+
| Bailey P.    | Art 301        | 77    |
| Bailey P.    | Math 100       | 85    |
| Bailey P.    | Music 210      | 65    |
| Jones J.     | Math 100       | 92    |
| Jones J.     | Music 210      | 88    |
| Martin R.    | Math 100       | 85    |
+--------------+----------------+-------+
relation StudentRec has 6 tuples
>
```

Figure 3.18: Assignment Operations

assignment adds the tuple data of relation *Student1* to relation *StudentRec*.

The general syntax for assignment operations is as follows:

> new_relname < − source_relation; *(normal assignment)*

> new_relname < + source_relation; *(incremental assignment)*

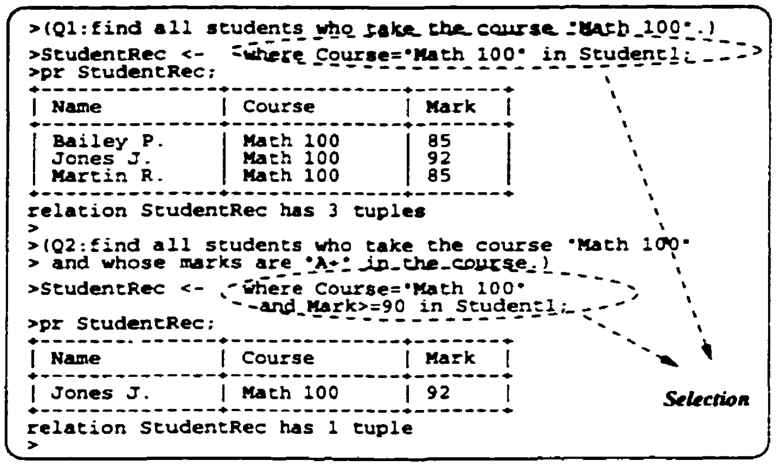
It is important to mention here that the *source_relation* in the above syntax for assignment operation is not necessarily an actual relation entry. It might however be a arbitrary combination of multi-step relational algebra operations such as selections, projections and joins etc. which will be introduced soon in the next sections. The multi-step arbitrary combination of relational algebra (and also domain algebra) operations are usually called "*relational expression*".

On the other hand, assignment operation with nested relations involved is exactly the same as the case of flat relations. When assigning a nested relation to a new relation, the surrogates' data is copied instead of the actual data values. For the definition of "*surrogate*", the reader can refer to the subsection *Declare and Initialize Nested Relations* of section 3.2.3.

3.4.2 Selection

The selection operation creates a new relation by extracting specific tuples from the source relation. The result relation contains the subset of tuples in the source relation that match a "*selection condition*". The selection condition may be any logical expression that can be evaluated to **true** or **false** on any one tuple of the source relation.

Given the sample relation *Student1* in previous section (refer to Figure 3.18), Figure 3.19 illustrates some examples of selection.



```
>(Q1:find all students who take the course 'Math 100'.)
>StudentRec <-  where Course='Math 100' in Student1;
>pr StudentRec;
+----------------+------------------+--------+
| Name           | Course           | Mark   |
+----------------+------------------+--------+
| Bailey P.      | Math 100         | 85     |
| Jones J.       | Math 100         | 92     |
| Martin R.      | Math 100         | 85     |
+----------------+------------------+--------+
relation StudentRec has 3 tuples
>
>(Q2:find all students who take the course 'Math 100'
> and whose marks are 'A+' in the course.)
>StudentRec <-  where Course='Math 100'
                 and Mark>=90 in Student1;
>pr StudentRec;
+----------------+------------------+--------+
| Name           | Course           | Mark   |
+----------------+------------------+--------+
| Jones J.       | Math 100         | 92     |
+----------------+------------------+--------+
relation StudentRec has 1 tuple
>
```

Figure 3.19: Examples of Selection Operation

In this figure, *Query 1* retrieves information of all students that take course "*Math 100*", while *Query 2* finds all students that secured "A" in the same course. Both queries create a new relation *StudentRec* as their search result.

The general syntax for selection is as follows:

where *selection_condition* **in** source_relation

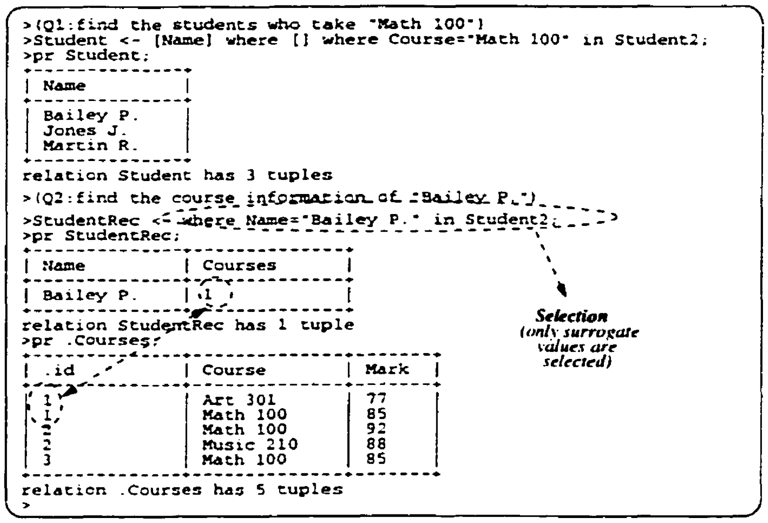
A more general syntax for selection with two advanced keywords is as follows:

where|when *selection_condition* **from|in** source_relation

Here the keyword **when** is of advanced usage, i.e. it provides a synchronization primitive for a multi-process environment [Dou91], which reader may put aside at the present stage.

The keyword **from** combines two operations together, i.e. selection and update (which will be introduced in section 3.4.6): the matched tuples are selected and removed from the source relation. Note that although these two advanced keywords are acceptable by jRelix, they are yet to be implemented.

On the other hand, selection on nested relations is a little different from that of flat relations. In Figure 3.20, *Query 1.* asks to find the students who take the course "Math 100" from the nested relation *Student2* given in section 3.2.3 (refer to Figure 3.11). To perform this query, the empty projection list, *[]*, is used to indicate that *"there is something in the ..."*.



```
>(Q1:find the students who take "Math 100")
>Student <- [Name] where [] where Course="Math 100" in Student2;
>pr Student;
+--------------+
| Name         |
+--------------+
| Bailey P.    |
| Jones J.     |
| Martin R.    |
+--------------+
relation Student has 3 tuples
>(Q2:find the course information of "Bailey P.")
>StudentRec <- where Name="Bailey P." in Student2;
>pr StudentRec;
+--------------+--------------+
| Name         | Courses      |
+--------------+--------------+
| Bailey P.    | 1            |
+--------------+--------------+
relation StudentRec has 1 tuple
>pr .Courses;
+--------------+--------------+--------+
| .id          | Course       | Mark   |
+--------------+--------------+--------+
| 1            | Art 301      | 77     |
| 1            | Math 100     | 85     |
| 2            | Math 100     | 92     |
| 2            | Music 210    | 88     |
| 3            | Math 100     | 85     |
+--------------+--------------+--------+
relation .Courses has 5 tuples
>
```

Figure 3.20: Selection on Nested Relation

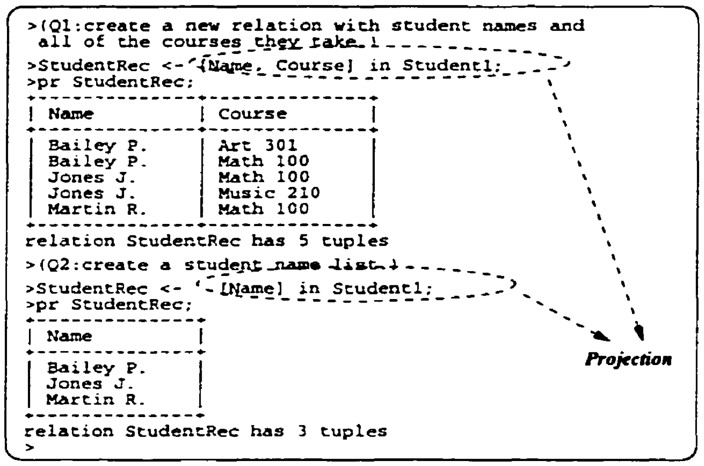
Another way to perform selection on nested relations is quite similar to the selection on flat relations. Instead of actual nested domain data, the corresponding surrogates' data are investigated and selected. *Query 2.* in Figure 3.20 illustrates this case.

It is clear from the example that the result relation *StudentRec* in *Query 2.* only contains the surrogate value of nested domain *Courses.* As explained in section 3.2.3 and section 3.2.5, this surrogate is the linkage between relation *StudentRec* and *.Courses* (as illustrated in the figure). The final realization of this relationship can be achieved by join operations introduced in section 3.4.5.

3.4.3 Projection

The projection operation creates a new relation by extracting named domains from the source relation. The result relation contains only the domains specified in the "*projection list*". It is therefore a subset of the domains of the source relation.

Given the sample relation *Student1* in the previous section (refer to Figure 3.18), Figure 3.21 illustrates some examples of projection.



```
>(Q1:create a new relation with student names and
 all of the courses they take.)
>StudentRec <- [Name, Course] in Student1;
>pr StudentRec;
+---------------+---------------+
| Name          | Course        |
+---------------+---------------+
| Bailey P.     | Art 301       |
| Bailey P.     | Math 100      |
| Jones J.      | Math 100      |
| Jones J.      | Music 210     |
| Martin R.     | Math 100      |
+---------------+---------------+
relation StudentRec has 5 tuples
>(Q2:create a student name list.)
>StudentRec <- [Name] in Student1;
>pr StudentRec;
+---------------+
| Name          |
+---------------+
| Bailey P.     |
| Jones J.      |
| Martin R.     |
+---------------+
relation StudentRec has 3 tuples
>
```

Figure 3.21: Examples of Projection Operation

In this figure, *Query 1* extracts student names and names of courses that the students registered for. The marks field is however cast out. Note the if we project just the *Name* attribute from *Student1*, the result has only three tuples as illustrated by *Query 2* in the same figure. The name "*Bailey P.*" and "*Jones J.*" occur twice in the source relation, but only once in the result. This is because each tuple in a relation must be distinct from all others. Project extracts the required domains and also removes any duplicate tuples from the result.

The general syntax for projection (including assignment after projection) is as follows:

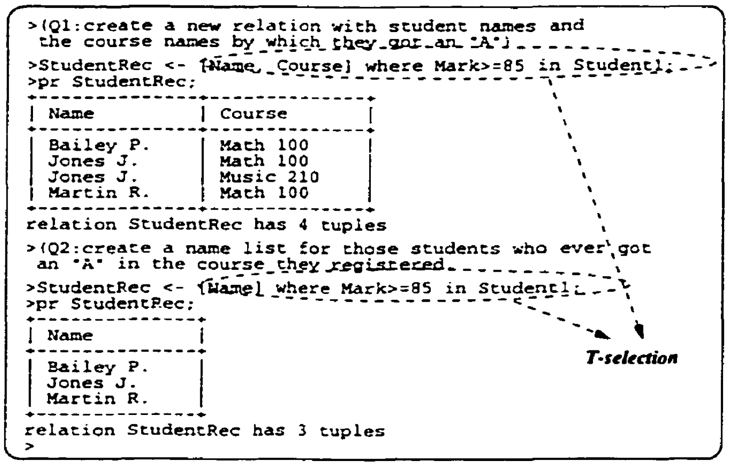
[*dom_name1*, *dom_name2*, ...] **in** source_relation

Note that the sequence of *dom_name*'s in the brackets in above syntax definition is called "*projection list*".

On the other hand, projection on nested relations is similar to the case of flat relations. Instead of actual nested domain data,the corresponding surrogates' data are projected. As this is similar to the selection operation introduced in section 3.4.2, examples are omitted.

3.4.4 T-selection

T-selection is a combination of selection and projection. It provides more flexible operations on relations. Figure 3.22 gives some example of *T-selection* using the sample relation *Student1* from the previous section.



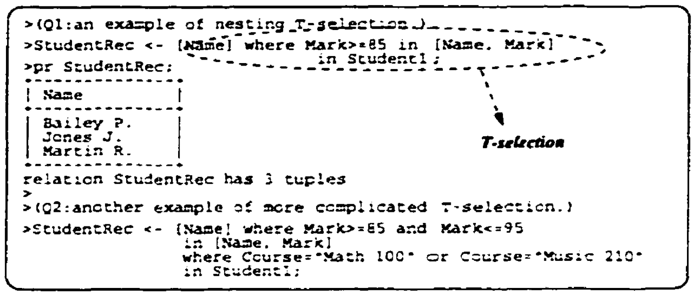
```
>(Q1:create a new relation with student names and
 the course names by which they got an "A")
>StudentRec <- [Name, Course] where Mark>=85 in Student1;
>pr StudentRec;
+--------------+--------------+
| Name         | Course       |
+--------------+--------------+
| Bailey P.    | Math 100     |
| Jones J.     | Math 100     |
| Jones J.     | Music 210    |
| Martin R.    | Math 100     |
+--------------+--------------+
relation StudentRec has 4 tuples
>(Q2:create a name list for those students who ever got
 an "A" in the course they registered.
>StudentRec <- [Name] where Mark>=85 in Student1;
>pr StudentRec;
+--------------+
| Name         |
+--------------+
| Bailey P.    |
| Jones J.     |
| Martin R.    |
+--------------+
relation StudentRec has 3 tuples
>
```

Figure 3.22: Examples of T-selection Operation

The queries in the examples are straight-forward, hence explanations are omitted. The general syntax for T-selection is as follows, although variations exist:

[*dom1, dom2, ...*] **where** *select-condition* **in** source_rel

According to the closure principle of relational model, any relational expression evaluates to a relation. This allows the arbitrary combinations of primary constructs to form complex expressions in T-selection, which is a type of "*composition*". *Query 1.* in Figure 3.23 gives an example of T-selection with composition. Much more complicated T-selections can be formed with composition, e.g. the *Query 2.* in Figure 3.23.



```
>(Q1:an example of nesting T-selection.)
>StudentRec <- [Name] where Mark>=85 in [Name, Mark]
                                         in Student1;
>pr StudentRec;
+---------------+
| Name          |
+---------------+
| Bailey P.     |
| Jones J.      |
| Martin R.     |
+---------------+
relation StudentRec has 3 tuples
>
>(Q2:another example of more complicated T-selection.)
>StudentRec <- [Name] where Mark>=85 and Mark<=95
                in [Name, Mark]
                where Course="Math 100" or Course="Music 210"
                in Student1;
```

Figure 3.23: Examples of T-selection with Composition

3.4.5 Joins

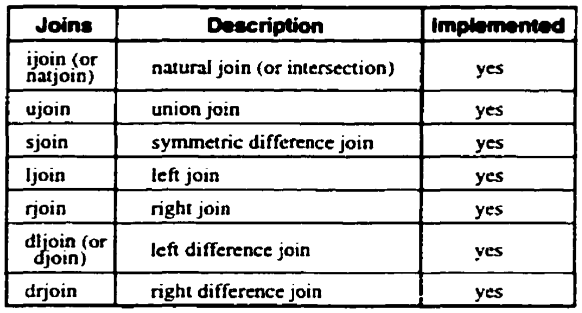
The relational model divides all objects, no matter how complex, into simple normalized relations and represents relationships between them by common values in shared attributes. Information retrieval thus depends on the ability to realize relationships by combining relations according to these shared attributes. This is achieved using join operators. For this reason, join is the characteristic relational operation [RS95].

Joins are made according to a join or linkage condition over a pair of (possibly compound) attributes, one in each relation, which are draw from the same domain. There are two classes of join operations defined in jRelix, i.e. μ-joins, the family of set-valued set operations; and σ-joins, the family of logical-valued set operations [Mer84].

Usually joins can be implemented by two approaches, i.e. pointer-based join algorithms and non-pointer-based algorithms [SC90]. The former approach takes advantage of pointer dereferencing, and provides significant performance gains in the situation where few tuples are involved in joins; while the latter, also called "***bulk algorithm***", usually deals with relations with large amounts of tuple data. Bulk algorithms include nested-loops [SM94], sort-merge [ICRR81] [LT95], hash-join [KO90] [ZJM94], hybrid-hash join [SC90] and partitioned band join [DNS91] algorithms etc. In jRelix implementation, the traditional *sort-merge* algorithm is applied for joins.

μ-joins

μ-joins are derived from set operators such as intersection, union, difference, etc. in mathematical set theory. Figure 3.24 lists the μ-joins that are defined and implemented in current jRelix, while detailed descriptions can be found in [Mer84].



Joins	Description	Implemented
ijoin (or natjoin)	natural join (or intersection)	yes
ujoin	union join	yes
sjoin	symmetric difference join	yes
ljoin	left join	yes
rjoin	right join	yes
dljoin (or djoin)	left difference join	yes
drjoin	right difference join	yes

Figure 3.24: μ-join Operations

The most frequently used join is natural join (i.e. **ijoin** or **natjoin**), which secures equality between the join attributes, and combines tuples from two relations together. Therefore, it can be regarded as the intersection of the two join relations. Figure 3.25 gives some examples of the natural join operation.

In Figure 3.25, a new relation *Course1* is introduced, which describes the course information e.g. the course name, credit of the course and the text book for the course. A natural join between relation *Student1* used in previous examples, and relation *Course1*, gives all information on the students and the courses they registered for, as illustrated in *Query 1*. *Query 2* performs a projection on certain attributes after the join operation. Some explanations will be made below.

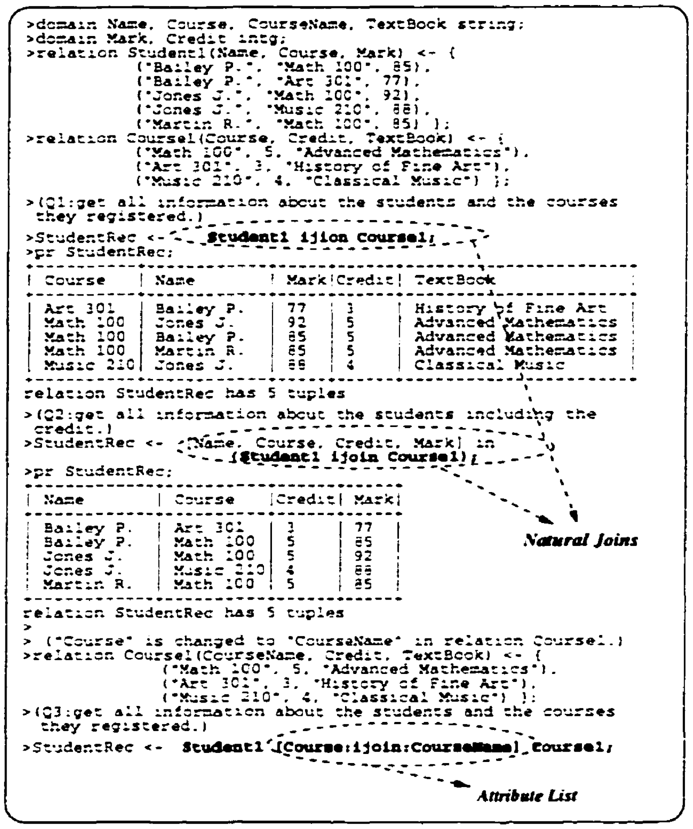
In general, the syntax for natural join is as follows:

rel_name1 **ijoin** *rel_name2*

or:

rel_name1 [*dom_name1,..* **:ijoin:***dom_name2, ..*]*rel_name2*

In the first syntax, two relations *rel_name1* and *rel_name2* are joined on their common attributes. In the case that there are no common attributes among the join relations, the cartesian



```
>domain Name, Course, CourseName, TextBook string;
>domain Mark, Credit intg;
>relation Student1(Name, Course, Mark) <- {
           ("Bailey P.", "Math 100", 85),
           ("Bailey P.", "Art 301", 77),
           ("Jones J.", "Math 100", 92),
           ("Jones J.", "Music 210", 88),
           ("Martin R.", "Math 100", 85) };
>relation Course1(Course, Credit, TextBook) <- {
           ("Math 100", 5, "Advanced Mathematics"),
           ("Art 301", 3, "History of Fine Art"),
           ("Music 210", 4, "Classical Music") };
>(Q1:get all information about the students and the courses
 they registered.)
>StudentRec <- Student1 ijion Course1;
>pr StudentRec;
+-----------+----------------+----+------+----------------------+
| Course    | Name           | Mark|Credit| TextBook             |
+-----------+----------------+----+------+----------------------+
| Art 301   | Bailey P.      | 77 | 3    | History of Fine Art  |
| Math 100  | Jones J.       | 92 | 5    | Advanced Mathematics |
| Math 100  | Bailey P.      | 85 | 5    | Advanced Mathematics |
| Math 100  | Martin R.      | 85 | 5    | Advanced Mathematics |
| Music 210 | Jones J.       | 88 | 4    | Classical Music      |
+-----------+----------------+----+------+----------------------+
relation StudentRec has 5 tuples
>(Q2:get all information about the students including the
 credit.)
>StudentRec <- [Name, Course, Credit, Mark] in
                (Student1 ijoin Course1);
>pr StudentRec;
+---------------+-----------+------+-----+
| Name          | Course    |Credit| Mark|
+---------------+-----------+------+-----+
| Bailey P.     | Art 301   | 3    | 77  |
| Bailey P.     | Math 100  | 5    | 85  |
| Jones J.      | Math 100  | 5    | 92  |
| Jones J.      | Music 210 | 4    | 88  |
| Martin R.     | Math 100  | 5    | 85  |
+---------------+-----------+------+-----+
relation StudentRec has 5 tuples
>
> ("Course" is changed to "CourseName" in relation Course1.)
>relation Course1(CourseName, Credit, TextBook) <- {
               ("Math 100", 5, "Advanced Mathematics"),
               ("Art 301", 3, "History of Fine Art"),
               ("Music 210", 4, "Classical Music") };
>(Q3:get all information about the students and the courses
  they registered.)
>StudentRec <-  Student1 [Course:ijoin:CourseName] Course1;
```

Figure 3.25: Examples of Natural Join i.e. **ijoin**

product of the two join relations will be generated. The second syntax tells how to join two relations which do not have common attributes. In this case, two relations join on the attributes listed in a pair of brackets ("[" and "]"). In particular, these attributes are called "*join attributes*"; and the set of attributes surrounded by the brackets are called "*join attributes list*".

Taking the example in Figure 3.25, consider the case that the name of domain *Course* in relation *Course1* is changed from "*Course*" to "*CourseName*". The two relations *Student1* and *Course1* can not be joined as expected in *Query 1*, since they do not have common attributes. In that case, the *join attributes list* is used to achieve the correct result for *Query 3*.

In addition, there are some rules that need to be mentioned here.

- In both cases, the common attributes will be listed as the starting attributes in the result relation. This is illustrated by the result of *Query 1* in Figure 3.25. To generated a relation with a desired attribute sequence, a projection after join operation is usually necessary, as illustrated by *Query 2* in the same figure.

- Domain names in *common attributes list* must exist in their respective join relations (e.g. in above syntax, *dom_name1* must be existing in *rel_name1* while *dom_name1'* must be existing in *rel_name2*). Otherwise, an error message is generated by the system and the join fails.

- In the case that common attributes exist in the join relations, they must appear in the *common attributes list* when the second natural join syntax is used. Otherwise, a warning message is generated by the system and the join fails.

- It is clear that the result of a join operation is a relation. Therefore, ijoin operations can be combined with other operations e.g. selection, projection etc. as illustrated in Figure 3.26.

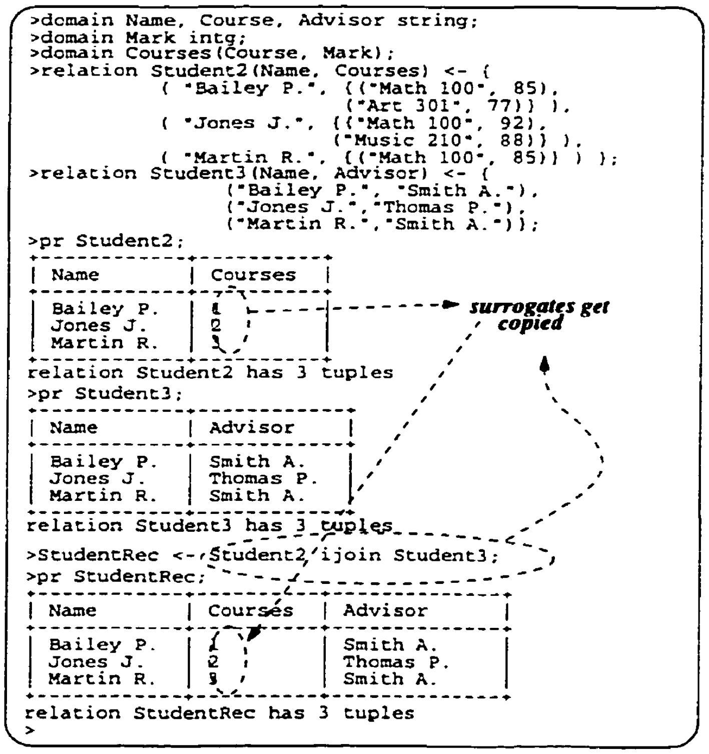
```
>StudentRec <- Student1 ijoin [Course, Credit] in Course1;
>StudentRec <- [Name, Course, Credit] in (Student1 ijoin Course1);
>StudentRec <- Student1 ijoin where Course="Math 100" in Course1;
>StudentRec <- [Name, Course, Credit] where Name="Jones J." in
        (Student1 ijoin where Course="Math 100" in Course1);
>
```

Figure 3.26: Combine ijion with Other Operations

- The syntax for the natural join operation listed above is basically applicable to all μ-joins, except that the keyword **ijoin** is changed to corresponding μ-join keywords. As well, the common rules for natural join are also applicable to other μ-join operations.

On the other hand, natural joins involving nested relations behave a little differently:

1. When the join attributes are not nested relational domains, natural join is the same as the ijions with flat relations as described above, except that the surrogates of nested domains are copied to the result relation. Figure 3.27 illustrates this case. In this example, a new relation *Student3* (which is defined on student *Name* and his/her *Advisor*) is created. The ijoin between *Student2* and *Student3* happens on their common attributes *Name*, and the surrogates of nested domain *Courses* are copied to the result relation *StudentRec* as illustrated in the figure.

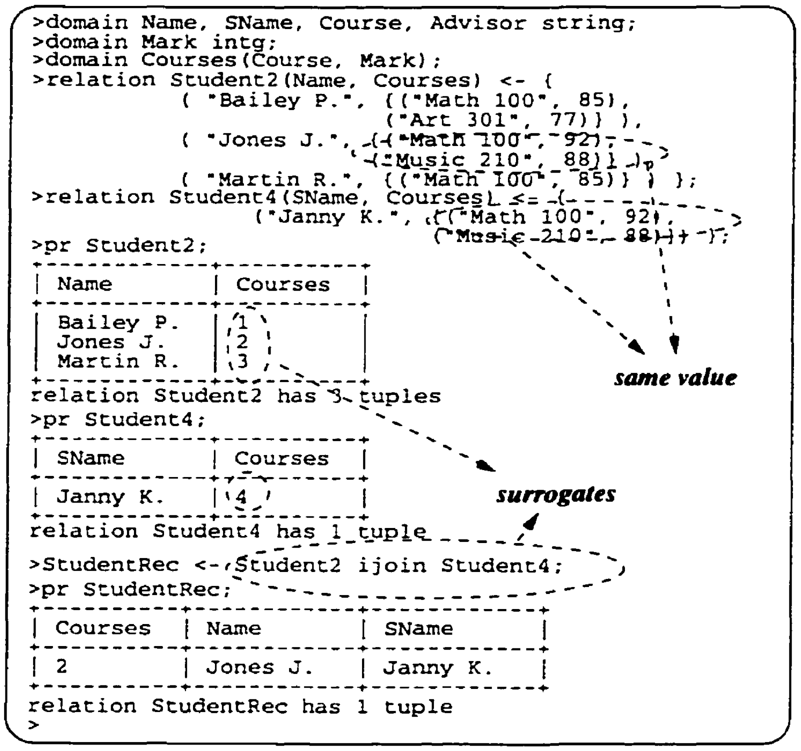


```
>domain Name, Course, Advisor string;
>domain Mark intg;
>domain Courses(Course, Mark);
>relation Student2(Name, Courses) <- {
           ( "Bailey P.", {("Math 100", 85),
                           ("Art 301", 77)} ),
           ( "Jones J.", {("Math 100", 92),
                          ("Music 210", 88)} ),
           ( "Martin R.", {("Math 100", 85)} ) };
>relation Student3(Name, Advisor) <- {
                ("Bailey P.", "Smith A."),
                ("Jones J.","Thomas P."),
                ("Martin R.","Smith A.")};
>pr Student2;
+------------+----------+
| Name       | Courses  |
+------------+----------+
| Bailey P.  | 1        |
| Jones J.   | 2        |
| Martin R.  | 3        |
+------------+----------+
relation Student2 has 3 tuples
>pr Student3;
+------------+--------------+
| Name       | Advisor      |
+------------+--------------+
| Bailey P.  | Smith A.     |
| Jones J.   | Thomas P.    |
| Martin R.  | Smith A.     |
+------------+--------------+
relation Student3 has 3 tuples
>StudentRec <- Student2 ijoin Student3;
>pr StudentRec;
+------------+----------+--------------+
| Name       | Courses  | Advisor      |
+------------+----------+--------------+
| Bailey P.  | 1        | Smith A.     |
| Jones J.   | 2        | Thomas P.    |
| Martin R.  | 3        | Smith A.     |
+------------+----------+--------------+
relation StudentRec has 3 tuples
>
```

Figure 3.27: Natural Join of Nested Relations (on Atomic Domains)

2. When the join attributes are nested relational domains, it is the tuple data of the nested data that are compared, but not the surrogate value, as illustrated in Figure 3.28. In this example, a new relation *Student4* (which is defined on domain *SName* and nested domain *Courses*) is introduced. There is only one student (i.e. "*Jenny k.*") declared in this relation. Her registered courses and the course results are exactly the same as that of student "*Jones J.*" in relation *Student2*. The natural join of *Studnet2* and *Student4* finds out those students who take exactly the same courses and whose course marks are exactly the same, as illustrated in the figure. Clearly, the nested domain data are compared when the join is performed.

   An exception occurs when one of the join attributes is "*.id*", which will be explained next.

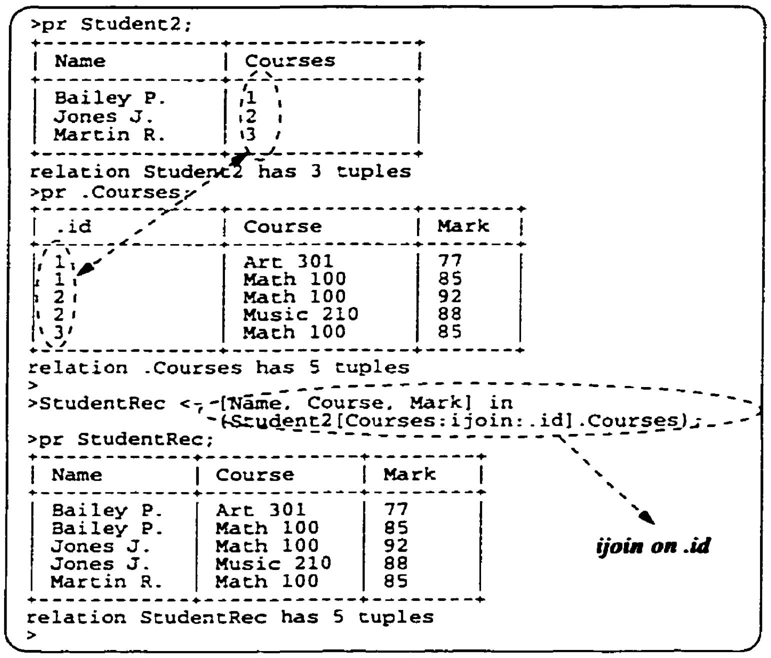


```
>domain Name, SName, Course, Advisor string;
>domain Mark intg;
>domain Courses(Course, Mark);
>relation Student2(Name, Courses) <- {
           ( "Bailey P.", {("Math 100", 85),
                           ("Art 301", 77)} ),
           ( "Jones J.", {("Math 100", 92),
                          ("Music 210", 88)} ),
           ( "Martin R.", {("Math 100", 85)} ) };
>relation Student4(SName, Courses) <- {
                ("Janny K.", {("Math 100", 92),
                              ("Music 210", 88)} )};
>pr Student2;
+------------+-----------+
| Name       | Courses   |
+------------+-----------+
| Bailey P.  | 1         |
| Jones J.   | 2         |
| Martin R.  | 3         |
+------------+-----------+
relation Student2 has 3 tuples
>pr Student4;
+------------+-----------+
| SName      | Courses   |
+------------+-----------+
| Janny K.   | 4         |
+------------+-----------+
relation Student4 has 1 tuple
>StudentRec <- Student2 ijoin Student4;
>pr StudentRec;
+----------+--------------+--------------+
| Courses  | Name         | SName        |
+----------+--------------+--------------+
| 2        | Jones J.     | Janny K.     |
+----------+--------------+--------------+
relation StudentRec has 1 tuple
>
```

Figure 3.28: Natural Join of Nested Relations (on Nested Domains)

3. When "*.id*" is one (or two) of the join attributes, the surrogate value of the nested domain is used for comparison during the join operation. Figure 3.29 illustrates this case. In this

example, the nested relation *Student2* declared in the previous section (refer to Figure 3.15) is used. The result is a flat relation containing student information including the course he/she registered for and their marks. This join is obviously a way of converting a nested relation into a flat relation.



```
>pr Student2;
+----------------+----------------+
| Name           | Courses        |
+----------------+----------------+
| Bailey P.      | 1              |
| Jones J.       | 2              |
| Martin R.      | 3              |
+----------------+----------------+
relation Student2 has 3 tuples
>pr .Courses;
+----------------+----------------+--------+
| .id            | Course         | Mark   |
+----------------+----------------+--------+
| 1              | Art 301        | 77     |
| 1              | Math 100       | 85     |
| 2              | Math 100       | 92     |
| 2              | Music 210      | 88     |
| 3              | Math 100       | 85     |
+----------------+----------------+--------+
relation .Courses has 5 tuples
>
>StudentRec <- [Name, Course, Mark] in
               (Student2[Courses:ijoin:.id].Courses);
>pr StudentRec;
+-------------+--------------+--------+
| Name        | Course       | Mark   |
+-------------+--------------+--------+
| Bailey P.   | Art 301      | 77     |
| Bailey P.   | Math 100     | 85     |
| Jones J.    | Math 100     | 92     |
| Jones J.    | Music 210    | 88     |
| Martin R.   | Math 100     | 85     |
+-------------+--------------+--------+
relation StudentRec has 5 tuples
>
```

Figure 3.29: Natural Join of Nested Relations (on ".*id*")

Union join (**ujoin**) is another frequently used μ-join. It is an operation that results in a union of the set of tuples from the natural join, together with the tuples from the relations of both sides that are not equal to each other in their join attributes, with the missing attributes filled up with so-called "*null value*" i.e. *DC* which denotes *don't care* and which describes irrelevant information. Figure 3.30 gives an example of union join.

In this example, an union join is performed between relations *Student* and *Course.* Since "Bailey P." takes a course "Art 301" that is not in the *Course* relation, and since nobody takes the course "Chemistry 108", the corresponding (missing) attributes are filled with *dc.* Apart from *DC*, there is another null value *DK* which denotes *don't know* and implies missing data. Readers may consult [Mer84] for detailed descriptions of null values.

```
>domain Name, Course, TextBook string;
>domain Credit intg;
relation Student(Name, Course) <- {
            ("Bailey P.", "Math 100"),
            ("Bailey P.", "Art 301"),
            ("Jones J.", "Math 100"),
            ("Jones J.", "Music 210"),
            ("Martin R.", "Physics 202") };
relation Course(Course, Credit, TextBook) <- {
            ("Math 100", 5, "Advanced Mathematics"),
            ("Physics 202", 3, "Principle of Physics"),
            ("Chemistry 108", 3, "Elementary Chemistry"),
            ("Music 210", 4, "Classical Music") };
>StudentRec <- [Name, Course, Credit] in   (Student ujoin Course);
>pr StudentRec;
+------------+-----------------+--------+
| Name       | Course          | Credit |
+------------+-----------------+--------+
| dc         | Chemistry 108   |  3     |
| Bailey P.  | Art 301         |  dc    |
| Bailey P.  | Math 100        |  5     |
| Jones J.   | Math 100        |  5     |
| Jones J.   | Music 210       |  4     |
| Martin R.  | Physics 202     |  3     |
+------------+-----------------+--------+
relation StudentRec has 6 tuples
>
```

Figure 3.30: Union Join

The operations of other μ-joins e.g. symmetric difference join (**sjoin**) etc. are similar to the natural join (**ijoin**) introduced above, except that different keywords (as illustrated in Figure 3.24) are used in the place of **ijion**. Therefore, detailed descriptions are omitted here.

σ-joins

The family of σ-joins are based on set comparison operators. In operations, the tuples in each of the operand relations are grouped such that for each group, all the non-join attributes are identical. Then, the set comparison operator is applied to the cartesian product of the groups. The values of the non-join attributes of the comparing groups are accepted if the specified set comparison on the join attributes is satisfied.

Figure 3.24 lists the σ-joins that are defined in jRelix, while detailed descriptions can be found in [Mer84].

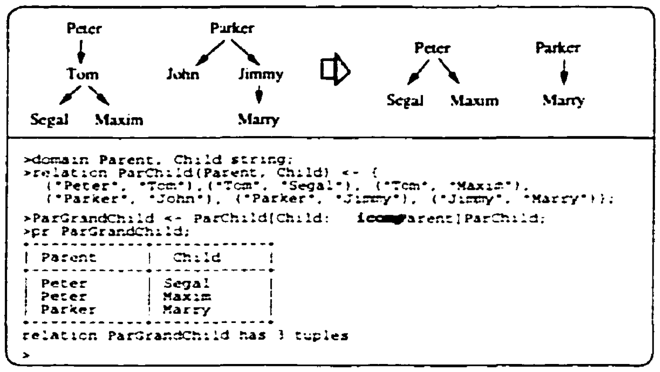
One of the frequently used σ joins is natural composition, i.e. **icomp**. The operation and result of natural composition are quite similar to that of natural join (i.e. **ijoin**), except that the join attributes are removed from the result relation. Figure 3.32 gives an example of natural composition.

In this example, natural composition is used to find the relation of a parent and their grand-

Joins	Description	Implemented
icomp (or natcomp)	natural composition	yes
sup (or div, gejoin)	super-set join or division	no
gtjoin	proper super-set (no inclsion)	no
eqjoin	equal set join (=)	no
lejoin (or sub)	sub-set join	no
ltjoin	proper sub-set join	no

Figure 3.31: σ-join Operations

children. Note that the pair of join attributes *Child* and *Parent* are not part of the result relation.



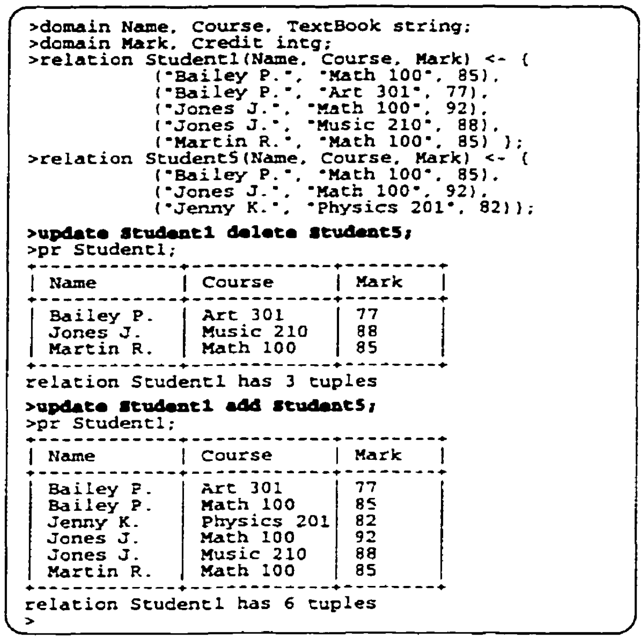
```
>domain Parent, Child string;
>relation ParChild(Parent, Child) <- {
  ("Peter", "Tom"),("Tom", "Segal"), ("Tom", "Maxim"),
  ("Parker", "John"), ("Parker", "Jimmy"), ("Jimmy", "Marry")};
>ParGrandChild <- ParChild[Child:   icomp Parent]ParChild;
>pr ParGrandChild;
+--------------------+--------------+
| Parent             |  Child       |
+--------------------+--------------+
| Peter              | Segal        |
| Peter              | Maxim        |
| Parker             | Marry        |
+--------------------+--------------+
relation ParGrandChild has 3 tuples
>
```

Figure 3.32: Example of Natural Composition (i.e. **icomp**)

σ-joins are not implemented in current jRelix yet. Therefore, introductions to the usage of σ-join operations are postponed.

3.4.6 Update

Adding and deleting tuples of a relation is relatively straightforward using the relational algebra described so far, but changing values of tuples in a relation is a little more complex. This section introduces a special relational operation **update** that provides a mechanism to change a relation locally.



```
>domain Name, Course, TextBook string;
>domain Mark, Credit intg;
>relation Student1(Name, Course, Mark) <- {
          ("Bailey P.", "Math 100", 85),
          ("Bailey P.", "Art 301", 77),
          ("Jones J.", "Math 100", 92),
          ("Jones J.", "Music 210", 88),
          ("Martin R.", "Math 100", 85) };
>relation Student5(Name, Course, Mark) <- {
          ("Bailey P.", "Math 100", 85),
          ("Jones J.", "Math 100", 92),
          ("Jenny K.", "Physics 201", 82)};
>update Student1 delete Student5;
>pr Student1;
+------------+------------+--------+
| Name       | Course     | Mark   |
+------------+------------+--------+
| Bailey P.  | Art 301    | 77     |
| Jones J.   | Music 210  | 88     |
| Martin R.  | Math 100   | 85     |
+------------+------------+--------+
relation Student1 has 3 tuples
>update Student1 add Student5;
>pr Student1;
+------------+------------+--------+
| Name       | Course     | Mark   |
+------------+------------+--------+
| Bailey P.  | Art 301    | 77     |
| Bailey P.  | Math 100   | 85     |
| Jenny K.   | Physics 201| 82     |
| Jones J.   | Math 100   | 92     |
| Jones J.   | Music 210  | 88     |
| Martin R.  | Math 100   | 85     |
+------------+------------+--------+
relation Student1 has 6 tuples
>
```

Figure 3.33: Update Operations: Add and Delete

Three types of **update** are provided in jRelix, i.e. *add, delete* and *change.* Figure 3.33 gives some examples of the update operation for addition and deletion.

In the **delete** example, those tuples in relation *Student1* that appear in relation *Student5* are taken off or removed; while in the **add** operation, all the tuples of relation *Student5* are added to relation *Student1*. Note that duplicate tuples are removed in the result relation for **add** operation.

Figure 3.34 shows the update operation that changes the attribute data in a Relation. As illustrated in the example, *marks* are decremented by 5 for the course "*Math 100*".

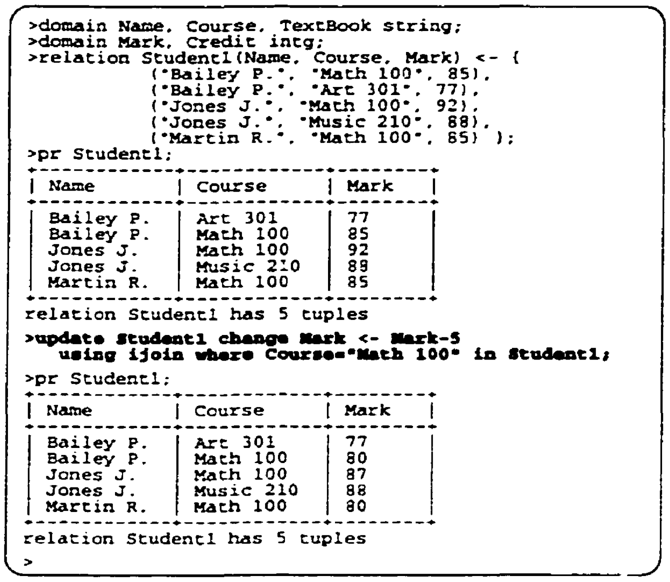
The general syntax for update is as follows:

> *new_rel* < − **update** *src_rel* **add** *add_rel*;

> *new_rel* < − **update** *src_rel* **delete** *del_rel*;

> *new_rel* < − **update** *src_rel* **change** *change_stmt* **using** *rel_expr*;

As illustrated in Figure 3.34, the *change_stmt* (change statement) tells what kind of change should be performed for certain attributes; while the *rel_expr* (relational expression) constrains the tuples in the source relation that should be changed. Note that *rel_expr* can be any valid



```
>domain Name, Course, TextBook string;
>domain Mark, Credit intg;
>relation Student1(Name, Course, Mark) <- {
          ("Bailey P.", "Math 100", 85),
          ("Bailey P.", "Art 301", 77),
          ("Jones J.", "Math 100", 92),
          ("Jones J.", "Music 210", 88),
          ("Martin R.", "Math 100", 85) };
>pr Student1;
+------------+-------------+---------+
| Name       | Course      | Mark    |
+------------+-------------+---------+
| Bailey P.  | Art 301     | 77      |
| Bailey P.  | Math 100    | 85      |
| Jones J.   | Math 100    | 92      |
| Jones J.   | Music 210   | 88      |
| Martin R.  | Math 100    | 85      |
+------------+-------------+---------+
relation Student1 has 5 tuples
>update Student1 change Mark <- Mark-5
   using ijoin where Course="Math 100" in Student1;
>pr Student1;
+------------+-------------+---------+
| Name       | Course      | Mark    |
+------------+-------------+---------+
| Bailey P.  | Art 301     | 77      |
| Bailey P.  | Math 100    | 80      |
| Jones J.   | Math 100    | 87      |
| Jones J.   | Music 210   | 88      |
| Martin R.  | Math 100    | 80      |
+------------+-------------+---------+
relation Student1 has 5 tuples
>
```

Figure 3.34: Update Operations: Change

combination of relational operations introduced so far, e.g. projection, selection and joins etc. Readers may consult [Hao98] for more information on the update operation.

3.5 Domain Algebra

Relational algebra considers relations to be the data primitives [Mer84] and therefore does not give the user the power to manipulate attributes. On the other hand, domain algebra [Mer77] [Mer84] consists of a set of operations to manipulate attributes such as mathematical operations, attribute group and ordering etc. Domain algebra is used through the declaration of virtual attributes and the actualization of them on relations.

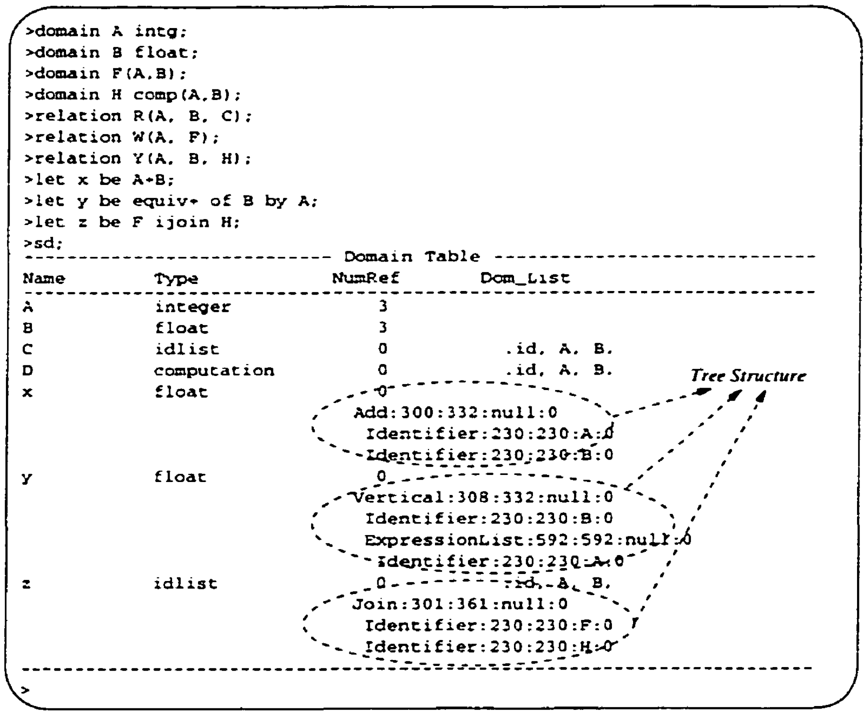
During the development of the jRelix system, my major responsibility was to design and implement the domain algebra. This section firstly discusses the virtual domain declarations, followed by a general description of actualization including the various error checking performed by jRelix. After that, horizontal operations e.g. renaming, function and if-then-else operation etc. as well as vertical operations e.g. reduction (both simple and equivalent) are explored respectively.

3.5.1 Virtual Domain Declaration

Virtual domains are domains that do not originally exist in a relation. They are declared on a set of actual domains or virtual domains which are subsequently based on actual domains. Virtual domains usually appear in projection introduced in section 3.4.1 and are *actualized* based on the actual domains' data in the source relation. However, virtual domains can theoretically appear wherever actual domains exist.

Virtual domains must be either directly or indirectly defined on the actual domains of the relation in question in order to be *actualizable*. Declaring a virtual domain is quite similar to defining a small procedure call in some programming languages such as C, with the procedure body represented in the form of an expression.

Figure 3.35 gives some example of declaring virtual domains, as well as displaying the declared domains.



```
>domain A intg;
>domain B float;
>domain F(A,B);
>domain H comp(A,B);
>relation R(A, B, C);
>relation W(A, F);
>relation Y(A, B, H);
>let x be A+B;
>let y be equiv+ of B by A;
>let z be F ijoin H;
>sd;
--------------------------- Domain Table ---------------------------
Name         Type             NumRef       Dom_List
--------------------------------------------------------------------
A            integer            3
B            float              3
C            idlist             0            .id, A, B.
D            computation        0            .id, A, B.
x            float              0
                                 Add:300:332:null:0
                                  Identifier:230:230:A:0
                                  Identifier:230:230:B:0
y            float              0
                                 Vertical:308:332:null:0
                                  Identifier:230:230:B:0
                                  ExpressionList:592:592:null:0
                                   Identifier:230:230:A:0
z            idlist             0            .id, A, B.
                                 Join:301:361:null:0
                                  Identifier:230:230:F:0
                                  Identifier:230:230:H:0
--------------------------------------------------------------------
>
```

Figure 3.35: Declaring Virtual Domains

In general, the syntax to declare a virtual domain is as follows:

> **let** *vir_dom_name* **be** *expression*;

There are however somethings that need to be mentioned here:

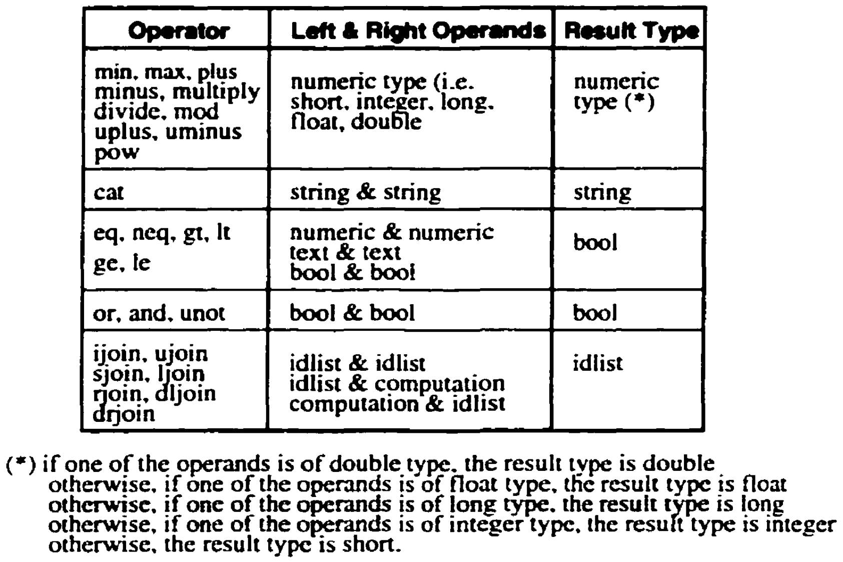
1. Virtual domain declaration does not affect the reference counter of the referenced domains (For the meaning of "*reference counter*", please refer back to section 3.2.2). For example, the virtual domain x in Figure 3.35 is defined on domain A and B, but the reference counter of domain A and B are not incremented because of this fact.

   An exception is for virtual domain z, which is defined on actual domain C and D. Obviously, the reference counters of domain C and D are not affected by this fact. However, since the resulting type of virtual domain z is *idlist* 3.2.2, this means domain$z$ is a nested relational domain. It is not hard to figure out that this nested domain (z) has an attribute list of *(.id, A, B)*. As we know, a nested domain is always associated with an invisible relation (refer to section 3.2.1) which is supposed to hold the tuples data of this domain. Hence, an invisible relation $.z$ is automatically generated in the system when virtual domain z is declared, and this relation is defined on domain A and B. As the result, the reference counters of domain A and B are incremented by 1.

2. The resulting type of a virtual domain is decided according to certain rules illustrated in Figure 3.36. For example, virtual domain x is defined on domain A which is of *integer* type, and domain B which is of *float* type. The resulting type of x is however *float*. Similarly, domain z is defined on the domains of type *idlist* and *computation*, and the resulting type is *idlist*. On the other hand, if a virtual domain is declared on domains with incompatible types, an error message "*mismatched types*" will be generated by the system and the declaration fails.

3. The *expression* part of virtual domain declaration is interpreted by jRelix system as a *tree structure* which can be seen by displaying the definition of virtual domains using "**sd;**" command. For example, in Figure 3.35, virtual domain "x" is defined as domain A plus domain B. When displaying the definition of x, a tree structure is printed apart from the basic information such as *Type, NumRef* and *DomList* etc.

   The interpretation of a virtual domain's expression tree is a little cryptic, but readers are



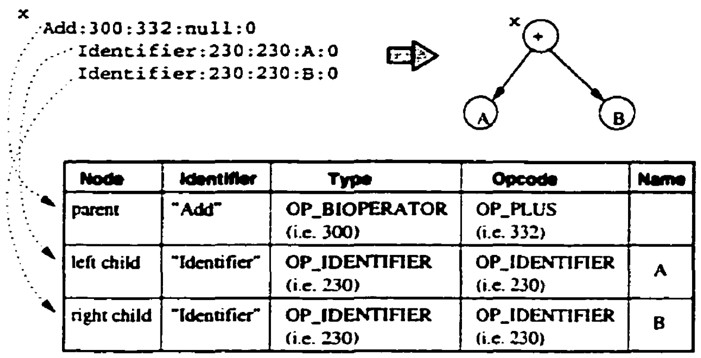
Operator	Left & Right Operands	Result Type
min, max, plus minus, multiply divide, mod uplus, uminus pow	numeric type (i.e. short, integer, long, float, double	numeric type (*)
cat	string & string	string
eq, neq, gt, lt ge, le	numeric & numeric text & text bool & bool	bool
or, and, unot	bool & bool	bool
ijoin, ujoin sjoin, ljoin rjoin, dljoin drjoin	idlist & idlist idlist & computation computation & idlist	idlist

(*) if one of the operands is of double type, the result type is double
otherwise, if one of the operands is of float type, the result type is float
otherwise, if one of the operands is of long type, the result type is long
otherwise, if one of the operands is of integer type, the result type is integer
otherwise, the result type is short.

Figure 3.36: Rule of Type Operations

not required to understand it completely in order to perform domain algebra operations. Basically, an expression tree is made of a set of nodes each of which has the attributes *identifier*, *type*, *opcode* and *name*, where *identifier* tells the node's name, *type* describes the general type of the node, e.g. "*bi-operation*" in this example, *opcode* indicates the more specific type, e.g. "*plus*"; and *name* tells the actual identifier's name that the current node represents. The display of a virtual domain's expression tree is a list of these nodes with indentations implying *parent-children* relationships. For each node, a list of its attributes is printed. Figure 3.37 gives an example of the expression tree of virtual domain x in Figure 3.35.

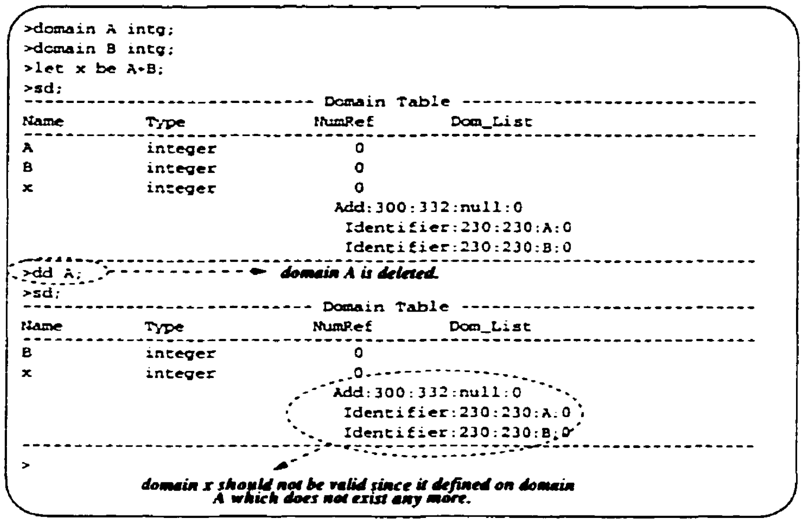
4. During a virtual domain declaration, all identifiers (i.e. domains) in the expression tree must be already declared in the system; otherwise, an error message will be generated and the declaration will fail. This is a weak check to make sure that the virtual domain is actualizable. Usually we call this a "*declaration check*". It is however possible that a valid virtual domain definition is changed to become invalid later by the user, as illustrated in Figure 3.38. Therefore, a stronger check has to be performed whenever the virtual domain is actualized. This is called a "*run-time check*". Details about *run-time check* will be



Node	Identifier	Type	Opcode	Name
parent	"Add"	OP_BIOPERATOR (i.e. 300)	OP_PLUS (i.e. 332)	
left child	"Identifier"	OP_IDENTIFIER (i.e. 230)	OP_IDENTIFIER (i.e. 230)	A
right child	"Identifier"	OP_IDENTIFIER (i.e. 230)	OP_IDENTIFIER (i.e. 230)	B

Figure 3.37: Example of an Expression Tree

described in section 3.5.2.



```
>domain A intg;
>domain B intg;
>let x be A+B;
>sd;
--------------------------- Domain Table ----------------------------
Name           Type              NumRef        Dom_List
---------------------------------------------------------------------
A              integer              0
B              integer              0
x              integer              0
                                  Add:300:332:null:0
                                   Identifier:230:230:A:0
                                   Identifier:230:230:B:0
---------------------------------------------------------------------
>dd A; ------------ domain A is deleted.
>sd;
--------------------------- Domain Table ----------------------------
Name           Type              NumRef        Dom_List
---------------------------------------------------------------------
B              integer              0
x              integer              0
                                  Add:300:332:null:0
                                   Identifier:230:230:A:0
                                   Identifier:230:230:B:0
---------------------------------------------------------------------
>
```

domain x should not be valid since it defined on domain A which does not exist any more.

Figure 3.38: Example of a Valid Virtual Domain Declaration Becomes Invalid

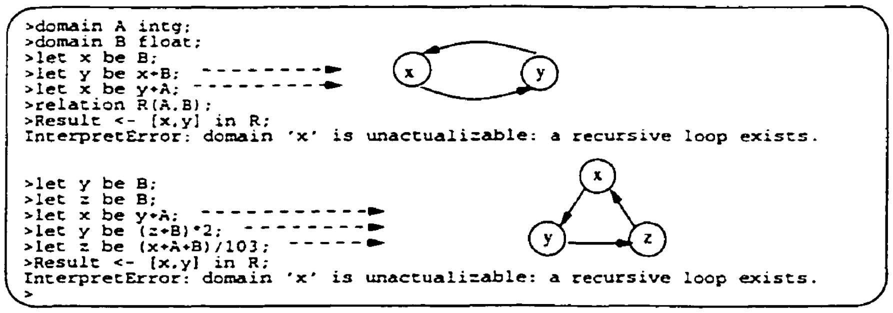
3.5.2 Virtual Domain Actualization

Generally speaking, once declared, virtual domains can appear wherever an actual domain appears, e.g. they can appear in the projection list of project operations, in the selection condition expression of select operations etc. The virtual domains are *actualized* when the relational algebra operations are performed. Given the domains and relations declared in Figure 3.35, Figure 3.39 gives some examples of virtual domain actualization.

```
>Result <- [A,B,x] in R;
>Result <- [x,y] in (R ijoin Y);
>Result <- where x=100 in R;
>Result <- [A,B] where y=x in Y;
>Result <- [z] in [F,H] where x=100 in (W ijoin Y);
>
```

Figure 3.39: Actualize Virtual Domains

As mentioned in last section, a "*run-time check*" which is much stronger than the "*declaration check*" will be performed during actualization. If a virtual domain is found to be unactualizable, an error message is generated and the actualization fails. Apart from the condition described in last section under which a virtual domain is not actualizable, jRelix also considers a virtual domain as unactualizable if this virtual domain is recursively defined on itself, i.e. there is a recursive loop in the definition of virtual domains in question. Figure 3.40 illustrates this condition.



```
>domain A intg;
>domain B float;
>let x be B;
>let y be x+B;  ---------►
>let x be y+A;  ---------►
>relation R(A,B);
>Result <- [x,y] in R;
InterpretError: domain 'x' is unactualizable: a recursive loop exists.

>let y be B;
>let z be B;
>let x be y+A;  ------------►
>let y be (z+B)*2;  ---------►
>let z be (x+A+B)/103;  ------►
>Result <- [x,y] in R;
InterpretError: domain 'x' is unactualizable: a recursive loop exists.
>
```

Figure 3.40: Recursive Loop in Virtual Domain Declaration

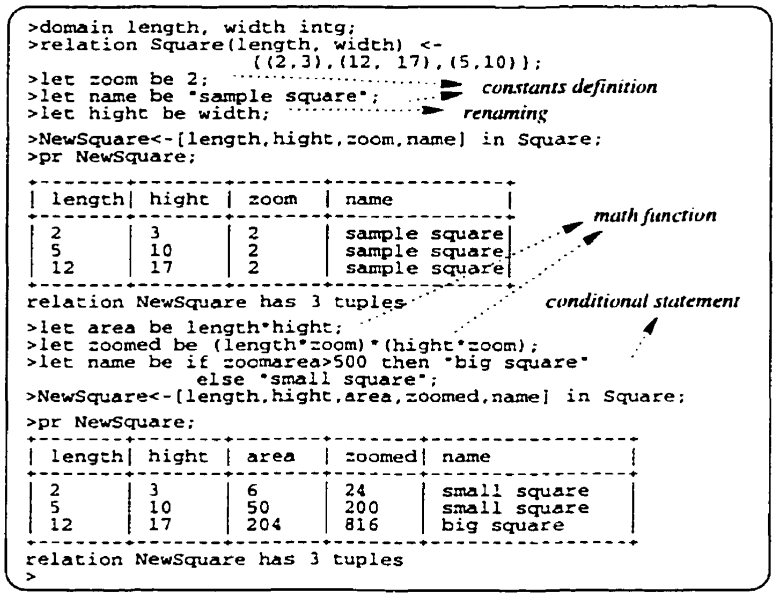
In this figure, two examples of recursive definition are given. In the first case, virtual domain x is defined on virtual domain y, while y is further defined on x. The result is that virtual domain x is defined on itself. In the second example, domain x is defined on itself at a three-hop, i.e. through virtual domain y and z as illustrated in the figure. All these definitions are not allowed by jRelix.

3.5.3 Horizontal Operations

Horizontal operations of domain algebra work on a single tuple of a relation. They generate the value in a tuple for the virtual attribute in terms only of the values in the same tuple of the operand attributes.

In the jRelix system, horizontal operations include constant definition, renaming, arithmetic functions, conditional statements (if-then-else) etc. which are called "basic horizontal operations" here; and most notably, all relational operations on the tuple level etc.

Figure 3.41 gives some examples for basic horizontal operation. Since the examples are quite self-explainable and easy to understand, the detailed explanation is omitted.



```
>domain length, width intg;
>relation Square(length, width) <-
                  {(2,3),(12, 17),(5,10)};
>let zoom be 2;
>let name be "sample square";
>let hight be width;
>NewSquare<-[length,hight,zoom,name] in Square;
>pr NewSquare;
+-------+-------+-------+---------------+
| length| hight | zoom  | name          |
+-------+-------+-------+---------------+
| 2     | 3     | 2     | sample square|
| 5     | 10    | 2     | sample square|
| 12    | 17    | 2     | sample square|
+-------+-------+-------+---------------+
relation NewSquare has 3 tuples
>let area be length*hight;
>let zoomed be (length*zoom)*(hight*zoom);
>let name be if zoomarea>500 then "big square"
            else "small square";
>NewSquare<-[length,hight,area,zoomed,name] in Square;
>pr NewSquare;
+-------+-------+-------+-------+---------------+
| length| hight | area  | zoomed| name          |
+-------+-------+-------+-------+---------------+
| 2     | 3     | 6     | 24    | small square  |
| 5     | 10    | 50    | 200   | small square  |
| 12    | 17    | 204   | 816   | big square    |
+-------+-------+-------+-------+---------------+
relation NewSquare has 3 tuples
>
```

Figure 3.41: Basic Horizontal Operations

Figure 3.42, 3.43 and 3.44 together give an example how relational algebra is involved in horizontal operation of domain algebra.

Figure 3.42 lists a table of students and the courses they registered for the fall and winter terms. Some courses can be taken by a student in both terms continuously; while some were taken in one of the terms. The queries to be performed are to find those courses that a student registered for both terms and to find a summarization of courses that a student registered for

Name	Fall		Winter	
	Course	Mark	Course	Mark
Bailey P.	Math 100 Art 301	88 75	Math 100 Physics 200	77 90
Jones J.	Math 100 Music 210 Physics 200	82 89 90	Music 210 Physics 200	83 100
Martin R.	Math 100 Art 301 Music 210	73 79 93	Math 100 Art 301 Music 210	91 78 90

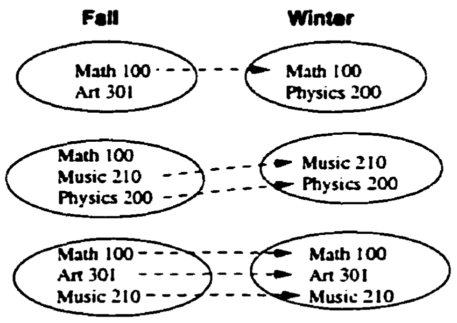
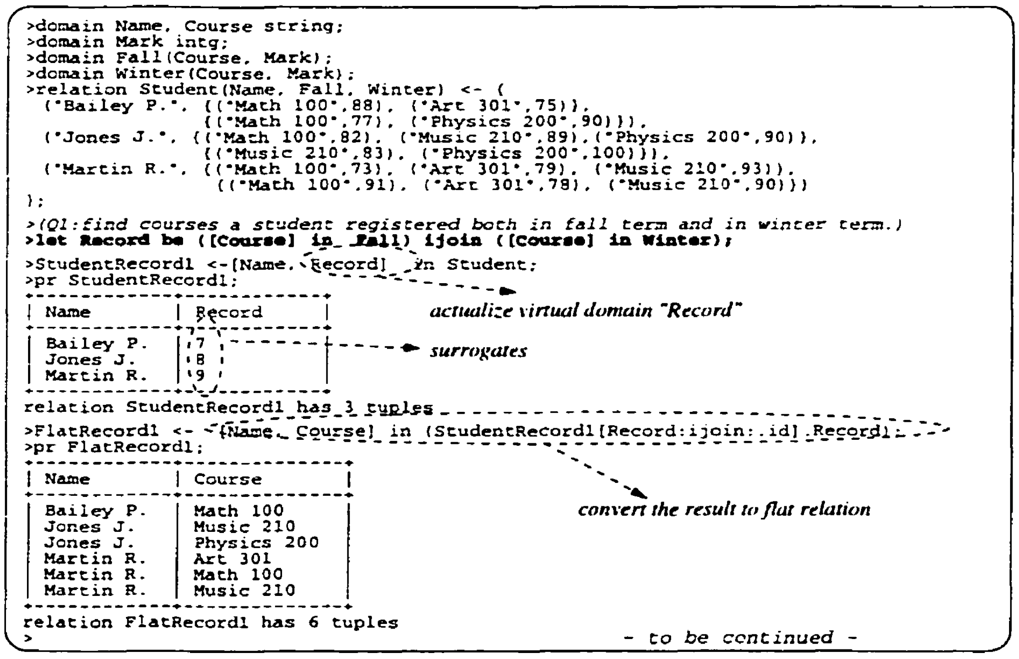


Figure 3.42: Example of Student Course Registration


actualize virtual domain "Record"
surrogates
convert the result to flat relation

```
>domain Name, Course string;
>domain Mark intg;
>domain Fall(Course, Mark);
>domain Winter(Course, Mark);
>relation Student(Name, Fall, Winter) <- (
  ("Bailey P.", (("Math 100",88), ("Art 301",75)),
                (("Math 100",77), ("Physics 200",90))),
  ("Jones J.", (("Math 100",82), ("Music 210",89),("Physics 200",90)),
                (("Music 210",83), ("Physics 200",100))),
  ("Martin R.", (("Math 100",73), ("Art 301",79), ("Music 210",93)),
                 (("Math 100",91), ("Art 301",78), ("Music 210",90)))
);
>(Q1:find courses a student registered both in fall term and in winter term.)
>let Record be ([Course] in Fall) ijoin ([Course] in Winter);
>StudentRecord1 <-[Name, Record] in Student;
>pr StudentRecord1;
+-------------+-------------+
| Name        | Record      |
+-------------+-------------+
| Bailey P.   | 7           |
| Jones J.    | 8           |
| Martin R.   | 9           |
+-------------+-------------+
relation StudentRecord1 has 3 tuples
>FlatRecord1 <- [Name, Course] in (StudentRecord1[Record:ijoin:.id].Record);
>pr FlatRecord1;
+-------------+---------------+
| Name        | Course        |
+-------------+---------------+
| Bailey P.   | Math 100      |
| Jones J.    | Music 210     |
| Jones J.    | Physics 200   |
| Martin R.   | Art 301       |
| Martin R.   | Math 100      |
| Martin R.   | Music 210     |
+-------------+---------------+
relation FlatRecord1 has 6 tuples
>                                                    - to be continued -
```

Figure 3.43: Relational Algebra in Horizontal Operation of Domain Algebra

during the two terms.

As illustrated in Figure 3.43, a nested relation *Student* is created with attributes *Name, Fall* and *Winter* where *Fall* and *Winter* are nested domains defined on *Course* and *Mark.* The nested domains hold the information of courses a student may register for during a term. To answer the first query, a virtual domain *Record* is declared to be the natural join of the nested domains *Fall* and *Winter* projected on their *Course* attributes. This will give the intersection of courses taken in both the *Fall* and *Winter* terms, which is supposed to be the result of query 1. As we will see, the virtual domain *Record* itself is a nested domain which will hold the result courses. A projection is performed on relation *Student* with *Record* as one of the domains in the projection list, which causes *Record* to be actualized. The result *StudentRecord1* is a nested relation that is defined on *Name* and *Record* where the *Record* field contains only surrogates. A further natural join is perform between relations *StudentRecord1* and *.Record* which converts the nested relation *StudentRecord1* to a flat relation *FlatRecord1.* It is clear that *FlatRecord1* lists the courses the students registered for in both terms (readers may consult Figure 3.42).

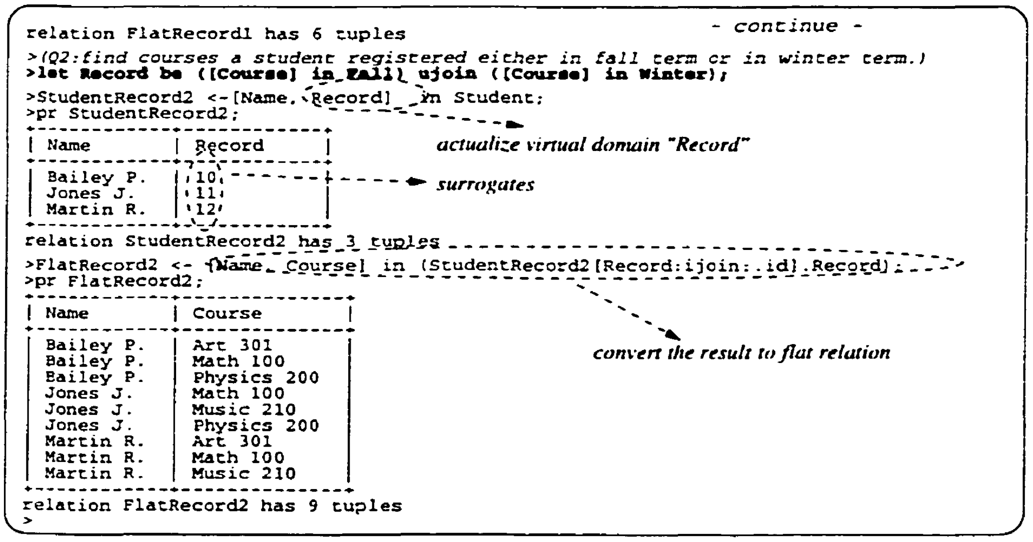
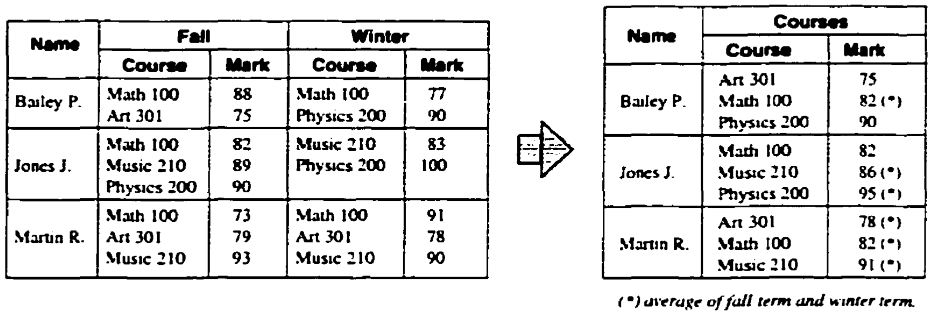

```
relation FlatRecord1 has 6 tuples                                   - continue -
>(Q2:find courses a student registered either in fall term or in winter term.)
>let Record be ([Course] in Fall) ujoin ([Course] in Winter);
>StudentRecord2 <-[Name, Record] in Student;
>pr StudentRecord2;
+-------------+-------------+
| Name        | Record      |
+-------------+-------------+
| Bailey P.   | 10          |
| Jones J.    | 11          |
| Martin R.   | 12          |
+-------------+-------------+
relation StudentRecord2 has 3 tuples
>FlatRecord2 <- [Name, Course] in (StudentRecord2[Record:ijoin:.id].Record);
>pr FlatRecord2;
+-------------+----------------+
| Name        | Course         |
+-------------+----------------+
| Bailey P.   | Art 301        |
| Bailey P.   | Math 100       |
| Bailey P.   | Physics 200    |
| Jones J.    | Math 100       |
| Jones J.    | Music 210      |
| Jones J.    | Physics 200    |
| Martin R.   | Art 301        |
| Martin R.   | Math 100       |
| Martin R.   | Music 210      |
+-------------+----------------+
relation FlatRecord2 has 9 tuples
>
```
actualize virtual domain "Record"
surrogates
convert the result to flat relation

Figure 3.44: Relational Algebra in Horizontal Operation of Domain Algebra

Similarly as illustrated in Figure 3.44, query 2 is performed by declaring a virtual domain which is a union join between *Fall* and *Winter*, and by actualizing this virtual domain on relation

Student. The result is listed in relation *FlatRecord2.*

A point that needs to be mentioned here is that theoretically any operation that can be performed in relational algebra (e.g. selection, projection, t-selection and all kinds of join operations) can also be performed on nested domains by using the horizontal operations of domain algebra. Figure 3.45 and Figure 3.46 together give a more complex example of horizontal operations on nested domains.



Name	Fall		Winter	
	Course	Mark	Course	Mark
Bailey P.	Math 100 Art 301	88 75	Math 100 Physics 200	77 90
Jones J.	Math 100 Music 210 Physics 200	82 89 90	Music 210 Physics 200	83 100
Martin R.	Math 100 Art 301 Music 210	73 79 93	Math 100 Art 301 Music 210	91 78 90

Name	Courses	
	Course	Mark
Bailey P.	Art 301 Math 100 Physics 200	75 82 (*) 90
Jones J.	Math 100 Music 210 Physics 200	82 86 (*) 95 (*)
Martin R.	Art 301 Math 100 Music 210	78 (*) 82 (*) 91 (*)

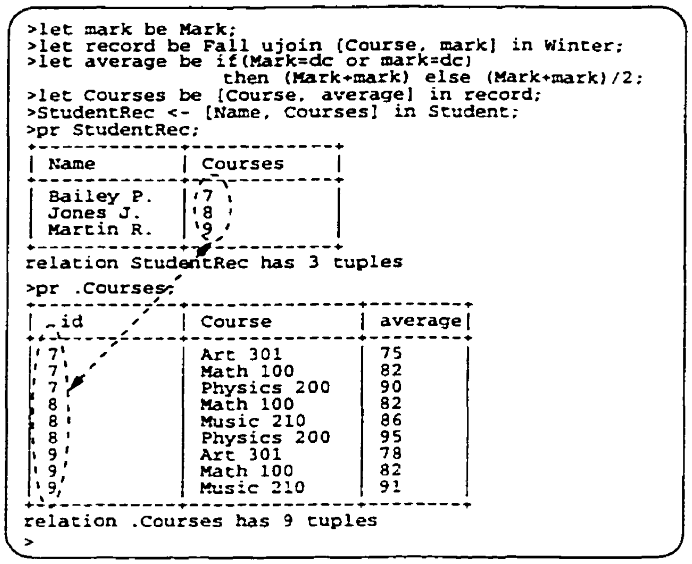
(*) average of fall term and winter term.

Figure 3.45: Calculate Average Marks of Fall and Winter Terms

The same student record information in Figure 3.42 is used in this example. A new record list is to be created which contains the student's name, the courses registered for in both terms and their marks. In the case that same course was taken during both terms, the average mark needs to be calculated. The result is illustrated on the right-side of the table in Figure 3.45.

As illustrated in Figure 3.46, various horizontal operations (e.g. renaming, union join, projection, math and if-then-else etc.) are involved in order to finish the query. Firstly, domain *Mark* has to be renamed in order to perform ujoin with *Fall*, since they are supposed to join on attribute *Course* only. Second, null value *dc* is used in the if-then-else construct, which calculates an average mark if the courses were taken both in fall and winter. The result *StudentRec* is a nested relation which is defined on the student's *Name* and the *Courses* information which is a further relation with attributes *Course* and *average.*

Horizontal operations with relational algebra can also be applied to deeper levels of nested domains. They behave in the same way previously introduced. However, due to their complexity, further explanations are omitted here.



```
>let mark be Mark;
>let record be Fall ujoin [Course, mark] in Winter;
>let average be if(Mark=dc or mark=dc)
                   then (Mark+mark) else (Mark+mark)/2;
>let Courses be [Course, average] in record;
>StudentRec <- [Name, Courses] in Student;
>pr StudentRec;
+------------+------------+
| Name       | Courses    |
+------------+------------+
| Bailey P.  | 7          |
| Jones J.   | 8          |
| Martin R.  | 9          |
+------------+------------+
relation StudentRec has 3 tuples
>pr .Courses;
+------------+--------------+--------+
| _id        | Course       | average|
+------------+--------------+--------+
| 7          | Art 301      | 75     |
| 7          | Math 100     | 82     |
| 7          | Physics 200  | 90     |
| 8          | Math 100     | 82     |
| 8          | Music 210    | 86     |
| 8          | Physics 200  | 95     |
| 9          | Art 301      | 78     |
| 9          | Math 100     | 82     |
| 9          | Music 210    | 91     |
+------------+--------------+--------+
relation .Courses has 9 tuples
>
```

Figure 3.46: More Relational Algebra in Horizontal Operation of Domain Algebra

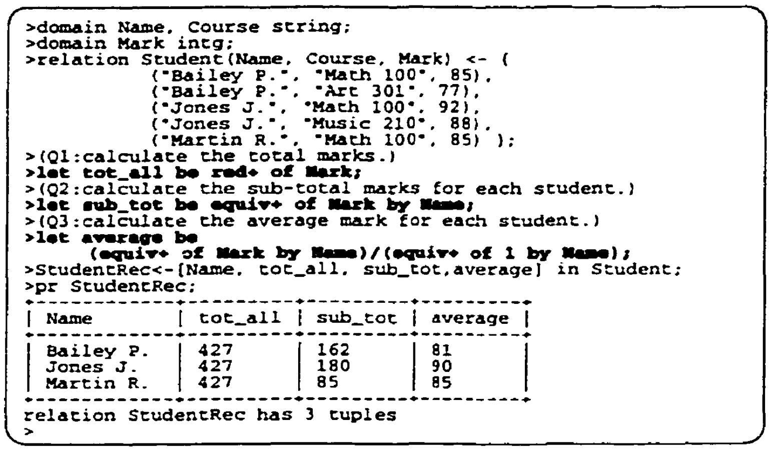
3.5.4 Vertical Operations

Vertical operations [Mer84] of domain algebra work on attribute values of all tuples in a relation. Basically, four types of vertical operations are defined in jRelix, although only the first two are implemented in current version:

- Simple reduction
- Equivalence reduction
- Functional mapping
- Partial functional mapping

Simple reduction produces a single result from the values from all tuples of a single attribute in the relation, while *equivalence reduction* provides a grouping mechanism not present in simple reduction [Mer84]. Figure 3.47 gives some examples of the reduction operation.

In this example, *tot_all* calculates the total mark regardless of the student and course; *sub_tot* calculates the total mark for each student; and *average* computes the average mark for each student.



```
>domain Name, Course string;
>domain Mark intg;
>relation Student(Name, Course, Mark) <- (
          ("Bailey P.", "Math 100", 85),
          ("Bailey P.", "Art 301", 77),
          ("Jones J.", "Math 100", 92),
          ("Jones J.", "Music 210", 88),
          ("Martin R.", "Math 100", 85) );
>(Q1:calculate the total marks.)
>let tot_all be red+ of Mark;
>(Q2:calculate the sub-total marks for each student.)
>let sub_tot be equiv+ of Mark by Name;
>(Q3:calculate the average mark for each student.)
>let average be
     (equiv+ of Mark by Name)/(equiv+ of 1 by Name);
>StudentRec<-[Name, tot_all, sub_tot,average] in Student;
>pr StudentRec;
+------------+---------+---------+---------+
| Name       | tot_all | sub_tot | average |
+------------+---------+---------+---------+
| Bailey P.  | 427     | 162     | 81      |
| Jones J.   | 427     | 180     | 90      |
| Martin R.  | 427     | 85      | 85      |
+------------+---------+---------+---------+
relation StudentRec has 3 tuples
>
```

Figure 3.47: Example of Reduction Operations

The general syntax for declaring a virtual domain that performs a reduction operation is as follows:

> **let** *virname* **be red** operator **of** *expr*; (simple reduction)

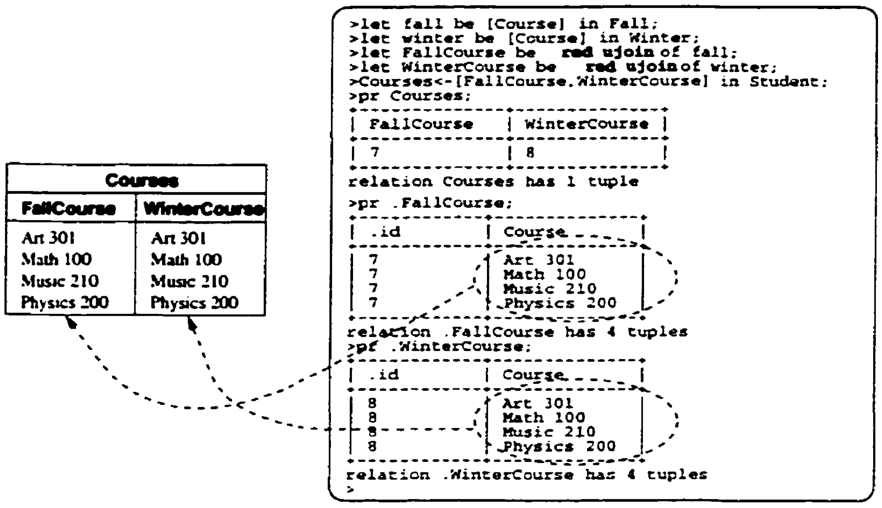
> **let** *virname* **be equiv** operator **of** *expr* **by** *expr_list*; (equivalence reduction)

In the syntax for equivalence reduction, the *expr_list* after the keyword **by** describes the sort attributes according to which the reduction is performed. The list is also called "*by-list*" of equivalence reduction.

The *operator* in the above syntax must be both **commutative** and **associative**. The operators satisfying this condition are *addition* (+), *multiplication* (*), *max* and *min* for numeric operations, *and* and *or* for boolean operations, and **ijoin, ujoin** and **sjoin** for relational operations etc.

As mentioned already, relational operations can be involved in vertical operations as well. Figure 3.48 gives some examples for this case.

This example uses relation *Student* introduced in Figure 3.42 and Figure 3.43, and produces a nested relation *Courses* which contains all the courses given in the fall and winter terms respectively (which happen to be same). As it is illustrated in the example, "*reduction of ujoin*" is used to generate such a nested relation.



Courses	
FallCourse	**WinterCourse**
Art 301 Math 100 Music 210 Physics 200	Art 301 Math 100 Music 210 Physics 200

```
>let fall be [Course] in Fall;
>let winter be [Course] in Winter;
>let FallCourse be  red ujoin of fall;
>let WinterCourse be  red ujoin of winter;
>Courses<-[FallCourse.WinterCourse] in Student;
>pr Courses;
+------------------+-----------------+
| FallCourse       | WinterCourse    |
+------------------+-----------------+
| 7                | 8               |
+------------------+-----------------+
relation Courses has 1 tuple
>pr .FallCourse;
+-------------+--------------+
| .id         | Course       |
+-------------+--------------+
| 7           | Art 301      |
| 7           | Math 100     |
| 7           | Music 210    |
| 7           | Physics 200  |
+-------------+--------------+
relation .FallCourse has 4 tuples
>pr .WinterCourse;
+-------------+--------------+
| .id         | Course       |
+-------------+--------------+
| 8           | Art 301      |
| 8           | Math 100     |
| 8           | Music 210    |
| 8           | Physics 200  |
+-------------+--------------+
relation .WinterCourse has 4 tuples
>
```

Figure 3.48: Example of Reduction with Relational Operation

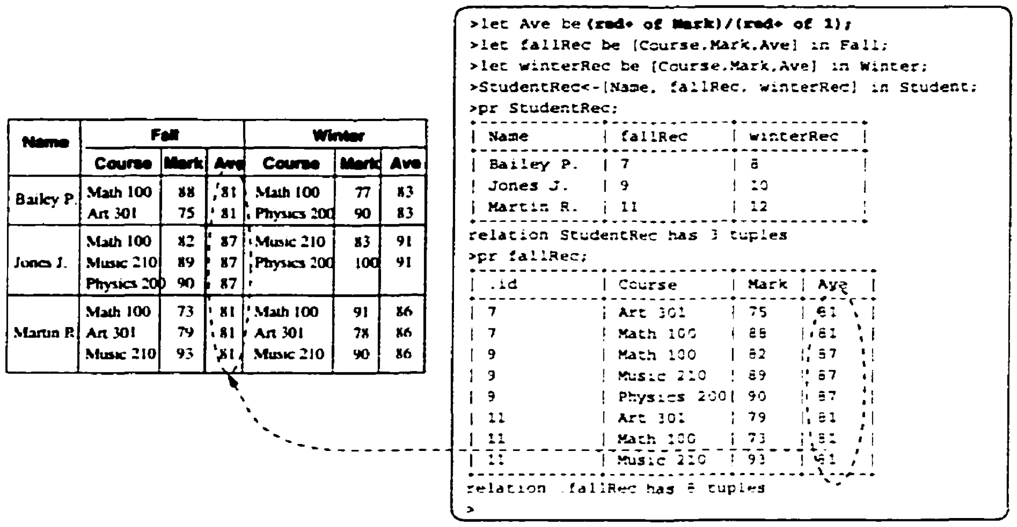
Vertical operations can also be applied to lower-level (sub-)relations in a nested relation. This is illustrated in Figure 3.49.

In this example, the average marks are calculated for each student in each term. This requires vertical operation to work on nested domains *Fall* and *Winter*, as illustrated by the virtual domain *Ave* combined with nested virtual domains *fallRec* and *winterRec*.

Finally, horizontal and vertical operations of domain algebra my be combined together to produce quite sophisticated queries. However, detailed explanations are omitted here.

3.6 Views

Views are computed relations defined on relations (including computations and views themselves). Unlike a relation, view does not hold actual data upon being declared (and initialized). They are usually regarded as a functional definition which is similar to a procedure call in other programming languages such as C and Java etc. Tuple data are generated *on the fly* for a view when it is invoked by certain mechanism, which is similar to the *actualization* of a virtual domain. Readers are encouraged to refer to [Hao98] for detailed information on views in the jRelix system.



Name	Fall			Winter		
	Course	Mark	Ave	Course	Mark	Ave
Bailey P.	Math 100	88	81	Math 100	77	83
	Art 301	75	81	Physics 200	90	83
Jones J.	Math 100	82	87	Music 210	83	91
	Music 210	89	87	Physics 200	100	91
	Physics 200	90	87			
Martin R.	Math 100	73	81	Math 100	91	86
	Art 301	79	81	Art 301	78	86
	Music 210	93	81	Music 210	90	86

```
>let Ave be (red+ of Mark)/(red+ of 1);
>let fallRec be [Course,Mark,Ave] in Fall;
>let winterRec be [Course,Mark,Ave] in Winter;
>StudentRec<-[Name, fallRec, winterRec] in Student;
>pr StudentRec;
+----------------+-----------------+---------------+
| Name           | fallRec         | winterRec     |
+----------------+-----------------+---------------+
| Bailey P.      | 7               | 8             |
| Jones J.       | 9               | 10            |
| Martin R.      | 11              | 12            |
+----------------+-----------------+---------------+
relation StudentRec has 3 tuples
>pr fallRec;
+----------------+-----------------+-------+-------+
| .id            | Course          | Mark  | Ave   |
+----------------+-----------------+-------+-------+
| 7              | Art 301         | 75    | 81    |
| 7              | Math 100        | 88    | 81    |
| 9              | Math 100        | 82    | 87    |
| 9              | Music 210       | 89    | 87    |
| 9              | Physics 200     | 90    | 87    |
| 11             | Art 301         | 79    | 81    |
| 11             | Math 100        | 73    | 81    |
| 11             | Music 210       | 93    | 81    |
+----------------+-----------------+-------+-------+
relation .fallRec has 8 tuples
>
```

Figure 3.49: Example of Reduction on Lower-level Nested Relations

3.7 Computations

Basically, computations are similar to the procedure calls in some programming languages such as C and Java etc. They accept parameters which are usually relations and output a relation as the result of computation. Readers are encouraged to refer to [Bak98] for detailed information on computations in the jRelix system.

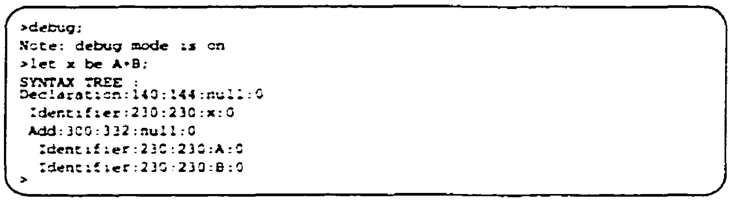
3.8 Advanced System Commands

System commands can be used to set the jRelix environment and display system information. This section introduces some of the more advanced jRelix commands. By using these commands, the user can know more about his/her environment upon starting the jRelix run-time system.

3.8.1 Setting Up Environment

In the current jRelix implementation, two types of environment modes (i.e. *debugging* mode and *timing* mode) are provided like switches. The commands to toggle these switches are described as follows:

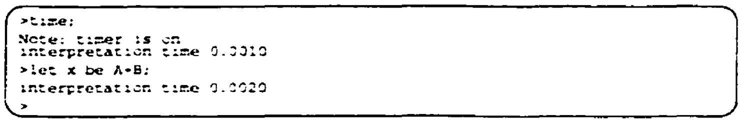
- **debug;** turn the *debugging* mode on/off. When the *debugging* mode is on, the system prints a syntax tree for the user command or statement whenever the user makes an action. This is particularly useful when doing debugging and when the syntax tree needs to be investigated. Figure 3.50 gives an example of this mode.



```
>debug;
Note: debug mode is on
>let x be A*B;
SYNTAX TREE :
Declaration:140:144:null:0
 Identifier:230:230:x:0
 Add:300:332:null:0
  Identifier:230:230:A:0
  Identifier:230:230:B:0
>
```

Figure 3.50: Turning Debugging Mode On

- **time;** turn the timer on/off. When timer is turned on, the interpretation time of user command is displayed in seconds whenever the user makes an action. Figure 3.51 gives an example of this mode.



```
>time;
Note: timer is on
interpretation time 0.0010
>let x be A*B;
interpretation time 0.0020
>
```

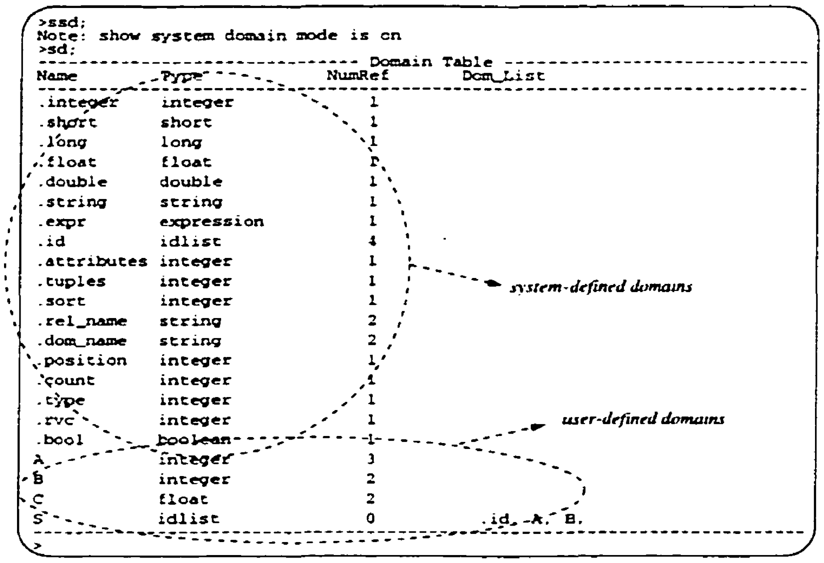
Figure 3.51: Turning Timer On

3.8.2 Displaying System Table Information

As mentioned in section 3.2.6, there are three system relations maintained by jRelix, i.e. *.dom*, *.rel* and .rd. They have corresponding memory versions i.e. *domtable* and *reltable* etc. which are usually called "*system tables*". System tables are maintained by the jRelix system. System table information is consulted and modified when declaring actual or virtual domains, relations, and views etc.

The commands to show system tables are introduced in sections 3.2.2 and 3.2.4. However, these commands usually display normal relation/domain information. There is, on the other hand, certain information that is not displayed by the commands introduced so far, and that may be of interest to the jRelix user. This section introduces two jRelix commands that deal with this problem.

- **ssd;** toggles the mode of the "**sd;**" command. When this switch is on, the "**sd;**" command displays both user-defined and system-defined domain information. A sample output of this case is shown in Figure 3.52.

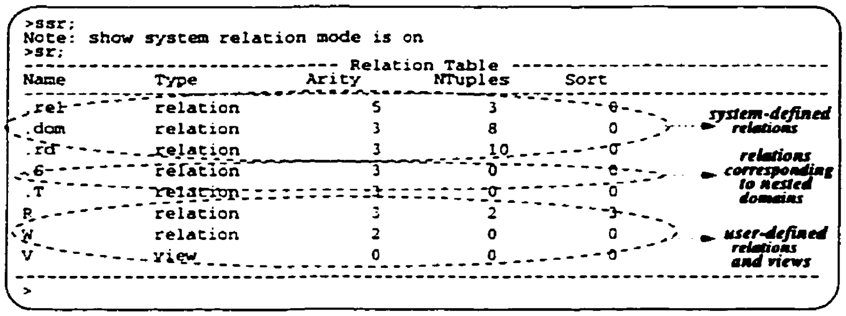


```
>ssd;
Note: show system domain mode is on
>sd;
------------------------------ Domain Table ------------------------------
Name          Type              NumRef        Dom_List
--------------------------------------------------------------------------
.integer      integer            1
.short        short              1
.long         long               1
.float        float              1
.double       double             1
.string       string             1
.expr         expression         1
.id           idlist             4
.attributes   integer            1
.tuples       integer            1
.sort         integer            1
.rel_name     string             2
.dom_name     string             2
.position     integer            1
.count        integer            1
.type         integer            1
.rvc          integer            1
.bool         boolean            1
A             integer            3
B             integer            2
C             float              2
S             idlist             0             .id, A, B,
--------------------------------------------------------------------------
>
```

Figure 3.52: Sample Output of "**ssd;**" + "**sd;**"

It's clear to see from the sample output that the system-defined domain name starts with a ".".

- **ssr;** toggles the mode of the "**sr;**" command. When this switch is on, the "**sr;**" command displays both user-defined and system-defined relation entries in reltable. Since a nested relational domain is always connected with a (sub)relation entry in reltable, this entry will also be displayed. A sample output of this case is shown in Figure 3.53.



```
>ssr;
Note: show system relation mode is on
>sr;
---------------------------- Relation Table ----------------------------
Name          Type           Arity       NTuples      Sort
-------------------------------------------------------------------------
.rel          relation           5             3         0
.dom          relation           3             8         0
.rd           relation           3            10         0
.6            relation           3             0         0
.T            relation           1             0         0
R             relation           3             2         3
W             relation           2             0         0
V             view               0             0         0
-------------------------------------------------------------------------
>
```

system-defined relations
relations corresponding to nested domains
user-defined relations and views

Figure 3.53: Sample Output of "**ssr;**" + "**sr;**"

3.8.3 Batch Processing

In jRelix, large databases (relations) are usually created as text files by hand by using a text editor, and then loaded into the jRelix run-time system by using the **input** command. In fact, "*input*" is a useful command to perform batch processing, which means, any jRelix commands and statements can be stored as batch files on disk and be loaded into the system like a sequence of jRelix commands. For example, suppose that the disk file *combat* is edited to hold a batch of jRelix commands and statements such as domain and relation declarations and initializations, and certain operations of relational as well as domain algebra. The following command is used to load and perform all the operations in the jRelix run-time system.

> **input** "combat";

The contents of file *combat* might look like something listed in Figure 3.54.

```
domain Name, Course string;
domain Mark intg;
relation Student(Name, Course, Mark) <- {
               ("Bailey P.", "Math 100", 85),
               ("Bailey P.", "Art 301", 77),
               ("Jones J.", "Math 100", 92),
               ("Martin R.", "Math 100", 85) };
StudentList <- [Name] in Student;
CourseList <- [Course] in Student;
let average be (equiv+ of Mark by Name) / (equiv+ of 1 by Name);
StudentRecord <- [Name, average] in Student;
pr StudentList;
pr CourseList;
pr StudentRecord;
```

Figure 3.54: Example of Batch File *combat*

Chapter 4

Implementation and Solution Strategy

As introduced in chapter 3, the jRelix system consists of such conceptual modules: relational algebra, domain algebra and computations. In the jRelix implementation, each conceptual module is designed to correspond to several low-level function modules (or components) in an object-oriented manner. The goal is to break the problem down into a number of smaller problems that are easier to understand and implement. Ideally, the function modules (or components) can be implemented directly as objects in the Java language.

In this chapter, we will explore some of the implementation details in jRelix. In section 4.1, the jRelix developing environment and tools are briefly discussed. The advantage of the Java programming language over other programming languages is shown to readers. Section 4.2 gives a general overview of the jRelix system. Different function modules and their relationship are described.

In section 4.3, something regarding the jRelix parser and interpreter is generally discussed. The jRelix parser and interpreter together serve as the front-end processor for the entire system. They are the interface between the end user and the central jRelix database engine.

Section 4.3 also roughly talks about the top-level expression evaluator. In the jRelix system, all user inputs are captured and translated into a syntax tree by parser/interpreter; while the evaluator makes the syntax tree understandable by the rest of the jRelix system. In addition, the top-level expression evaluator is also involved in actualization of lower-level nested virtual domains.

Section 4.4 deals with the system table mechanism in jRelix. Apart from user-defined relational or domain information, jRelix maintains so-called "*system information*" which describes the current system execution state and controls the system behavior either during a single jRelix session or across multiple sessions. As an example, the three system relations *.rel*, *.dom* and *.rd* mentioned in section 3.2.6 contain highly important information about user-defined relations in the system. Any error with these system relations may result in the malfunctioning of the entire system or a system crash in the worst case. System information is maintained in several so-called "*system tables*". System tables exist both in memory and on hard disk with different formats, which will also be described in section 4.4.

Section 4.5 explores the virtual domain actualizer, which deals with domain algebra in jRelix and is therefore one of the most important modules in jRelix. Apart from the implementation of horizontal and vertical operations in domain algebra, the actualizer is also in charge of the virtual domain's validation check, operands' type compatibility testing and mutually recursive definition detection etc., which will also be discussed in this section.

A virtual domain is usually actualized on a tuple-by-tuple level, which means the relation on which the virtual domain is to be actualized is scanned from the first tuple to the last one, and for each tuple data, the virtual domain's value is calculated. This is particularly true with the horizontal operation of domain algebra. For vertical operations, the relation is still scanned and relevant tuple data is stored somewhere for the vertical (e.g. reduction) calculation. This approach is called as "*tuple-by-tuple approach*". On the other hand, the tuple-by-tuple approach has efficiency problems since a loop within the entire relation is involved. This poses an even more serious problem when actualizing a virtual domain with relational operations on a nested relation, since, for example, joins on a tuple level are supposed to slow down the whole actualization procedure, as highly time-consuming sorting and disk I/O are involved with joins. Therefore, an alternative way named "*top-level approach*" is also available in the jRelix system. The top-level approach deals with top-level relation operations during virtual domain actualization, and yet fulfills the same result as tuple-level data calculation. Both approaches are discussed in section 4.5.

Note that the so-called "*relational processor*" and "*computation processor*" are not discussed in this thesis due to the space and time constraints. The relational processor is in charge of implementing relational algebra. Some of the most important relational operations e.g. projection, selection and various join operations are implemented within this module. The computation processor deals with computations in the jRelix system. As both of these modules are of the same importance and weight as the virtual domain actualizer, detailed descriptions are documented in [Hao98] and [Bak98] respectively.

4.1 Developing Environment and Tools

When choosing a developing environment, the following questions might firstly come to the decision-maker's mind.

- *What's the target operating system?*
- *Which programming language should be chosen?*
- *Are there any handy developing tools/utilities to speed up the development procedure?*
- *How about the experimental/testing environment, e.g. profiler?*

These questions are frequently asked in the initial stage of almost all software development. They will be well discussed in this section. Certain comparisons between jRelix and its counterpart (C version Relix) will also be discussed briefly.

4.1.1 Java Programming Language

The old version of Relix was written in C programming language, and is portable across different platforms running the UNIX operating system. Although C language provides applications with high performance in speed, flexibility in programming, and portability across different UNIX environments, it is fairly hard to program and debug C code due to the complexity of memory manipulation etc. This is especially true when building a medium/large-sized application such as a database engine like Relix. On the other hand, there are no built-in network facilities with standard C. To gain the power of network, C language will need additional network layers/libraries, which are mostly platform dependent however.

The Java programming language [AG96] [GJS96], developed at Sun Microsystems under the guidance of James Gosling and Bill Yoy, was designed to be a machine independent programming language that is both powerful enough to replace native executable code and safe enough to traverse networks. The authors of Java have written an influential "White Paper" that explains their design goals and accomplishments. Their paper is organized along the characteristics as showed in Figure 4.1 [GJS96] [Gos96]:

- Object Oriented
- Architecture Neutral & Portability
- Simple
- Network Facility and Distribution
- Robust & Secure
- Multithreaded

Figure 4.1: Java Buzzwords

It is almost impossible to discuss all the details about the above-mentioned features in this thesis. From Relix' point of view, Java is a good choice because of the following reasons.

1. It is an Object-Oriented language hence it is easy to program.
2. It is platform independent.
3. It has both robust and safe built-in network facilities.
4. It supports multi-threading.

The biggest program with Java is its speed. Java is an interpreted language. Generally speaking, Java compiler generates bytecodes which are interpreted by the Java Virtual Machine (JVM). Needless to say, this procedure slows down the execution speed especially for an application as database engine. However, there are ways to circumvent this drawback, e.g. Java bytecodes can be translated by a native code compiler (such as JIT) into machine code for the particular CPU the application is running on. This makes the target executing code just as fast as code written in other languages such as C.

4.1.2 JavaCC and JJTree

To build a non-trivial application that can intercept the user commands and respond interactively, a grammatical parser and a syntactic interpreter must be built as the front-end modules. In the Java world, two utility applications are freely available as tools to create such front-ends with sophisticated functionalities and with less design and coding efforts.

the Java Compiler Compiler (JavaCC) [SDV96] is a parser generator for use with Java applications. A parser generator is a tool that reads a high-level grammar specification and converts it to a Java program that can recognize matches in the grammar. In addition to the parser generator itself, JavaCC provides other standard capabilities related to parser generation such as tree building, actions, debugging etc. JavaCC uses the top-down parsing technique [ASU86].

JJTree is a preprocessor for JavaCC that inserts parse tree building actions at various places in the JavaCC source. The output of JJTree is run through JavaCC to create the parser. The relationship between JavaCC and JJTree is illustrated in Figure 4.2.

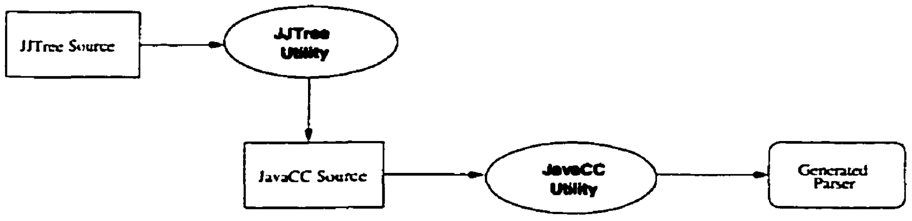


Figure 4.2: JavaCC and JJTree

JavaCC and JJTree are used to build the front-end parser for the jRelix system. Details will be introduced later in this chapter.

4.1.3 Debugger and Profiler

Two utility programs called *"jdb"* and *"javap"* are shipped with the *Java Development Kit (JDK)*. *jdb* is the official debugger provided by Sun Microsystems© for developing Java applications, while *javap* is a profiler provided by Sun Microsystems© for the Java developers to statistically measure the performance of different code pieces.

However, since both of these utilities are hard to use, they have not been adopted by the jRelix project team.

4.2 System Overview

Theoretically there are three conceptual aspects in the jRelix system, i.e. relational algebra, domain algebra and computation. They correspond to three basic function modules in the implementation, which work together (and also support each other) to fulfill the tasks of a database engine. Apart from the three modules, there are other supporting modules such as parser and interpreter which function as the front-end processor, and act as an interface between the end-user and the central database engine. Figure 4.3 is an overview of the system.

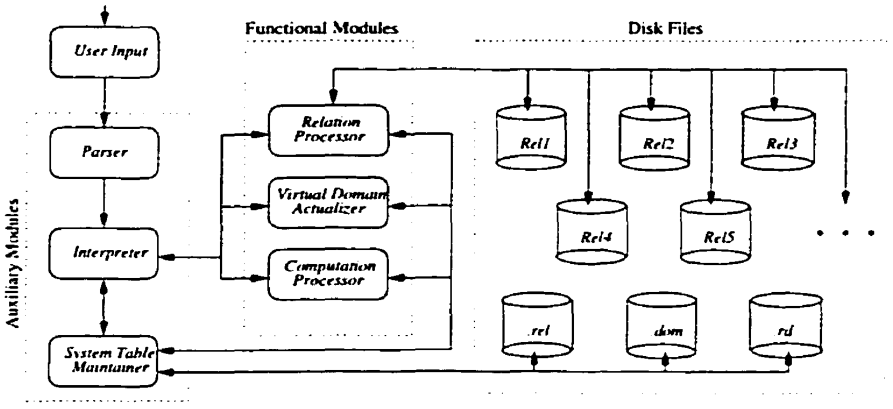


Figure 4.3: jRelix System Overview

A jRelix command entered by the end-user is first accepted by the parser. The parser is a Java class named Parser which is generated by JavaCC (refer to 4.1.2). It reads the command-line inputs and performs syntax analysis and finally translates jRelix commands into intermediate codes which have a tree structure. In JavaCC terminology, they are called syntax trees. More about the parser will be discussed in section 4.3.

The interpreter receives syntax trees passed by the parser and does certain evaluations such as error checking etc. It then calls different function modules to perform the operations. In the jRelix implementation, an Interpreter class is built to represent the interpreter. The evaluator is embedded in the interpreter. See section 4.3 for details on the interpreter.

The central database engine is represented by three modules i.e. the Relation Processor [Hao98], the Virtual Domain Actualizer (see section 4.5) and the Computation Processor [Bak98]. Three Java classes (i.e. Relation, Actualizer and Computation) are built correspondingly.

It is clear to see from Figure 4.3 that only the relation processor is responsible for disk I/O in a jRelix relation, whereas other modules access secondary storage via the relation processor. On the other hand, the system table maintainer is in charge of disk I/O for the system tables. Details are given in section 4.4.

4.3 Front End Processor

This section briefly discusses the front-end processor which consists of the parser, interpreter and top-level evaluator. The front-end processor is the interface between the end-user and the central jRelix database engine. The relationship of the three components in the front-end processor is illustrated in Figure 4.4.

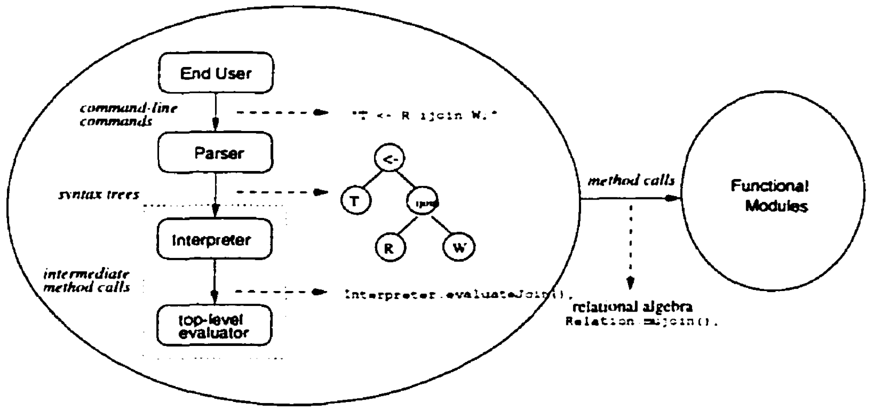
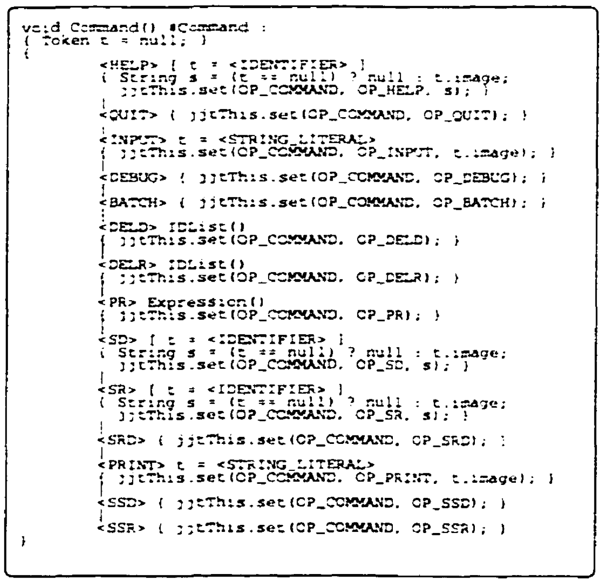


Figure 4.4: jRelix Front End Processor

As mentioned before, a jRelix command entered by the end-user is first accepted by the jRelix parser. The parser reads the command-line input, analyzes the command syntax and finally translates the command into an intermediate code which has a tree structure and is therefore called the *syntax tree.* The jRelix parser is created by using JavaCC (refer to section 4.1.2). The Backus-Naur form of all the jRelix command grammar is summarized in appendix A. In the jRelix implementation, a Parser class is created to correspond with this module.

The former version of Relix was developed in the C programming language and used Lex as its lexical analyzer and Yacc as its parser generator. The combination of these two utilities created a parser for the Relix interpreter which runs in the UNIX environment. In the jRelix implementation, the JavaCC utility introduced in section 4.1.2 is used to generate the front-end parser. In fact, JavaCC works as both lexical analyzer and parser generator which improves the processing efficiency.

On the other hand, the JJTree utility (refer to section 4.1.2) is used as an auxiliary program for JavaCC to specify the actions to be performed when a syntax match is found by JavaCC. Specifically, a source conforming to the JJTree syntax, which defines the jRelix grammar and actions for a match of the grammar, is introduced to the JJTree utility, which generates the intermediate code for JavaCC. JavaCC, upon receiving the output of JJTree, continues the process and generates source code in Java which is supposed to work together as the front-end parser of jRelix. Figure 4.2 depicts the procedure of parser generation in jRelix.



```
void Command() #Command :
{ Token t = null; }
{
        <HELP> [ t = <IDENTIFIER> ]
        { String s = (t == null) ? null : t.image;
          jjtThis.set(OP_COMMAND, OP_HELP, s); }
        |
        <QUIT> { jjtThis.set(OP_COMMAND, OP_QUIT); }
        |
        <INPUT> t = <STRING_LITERAL>
        { jjtThis.set(OP_COMMAND, OP_INPUT, t.image); }
        |
        <DEBUG> { jjtThis.set(OP_COMMAND, OP_DEBUG); }
        |
        <BATCH> { jjtThis.set(OP_COMMAND, OP_BATCH); }
        |
        <DELD> IDList()
        { jjtThis.set(OP_COMMAND, OP_DELD); }
        |
        <DELR> IDList()
        { jjtThis.set(OP_COMMAND, OP_DELR); }
        |
        <PR> Expression()
        { jjtThis.set(OP_COMMAND, OP_PR); }
        |
        <SD> [ t = <IDENTIFIER> ]
        { String s = (t == null) ? null : t.image;
          jjtThis.set(OP_COMMAND, OP_SD, s); }
        |
        <SR> [ t = <IDENTIFIER> ]
        { String s = (t == null) ? null : t.image;
          jjtThis.set(OP_COMMAND, OP_SR, s); }
        |
        <SRD> { jjtThis.set(OP_COMMAND, OP_SRD); }
        |
        <PRINT> t = <STRING_LITERAL>
        { jjtThis.set(OP_COMMAND, OP_PRINT, t.image); }
        |
        <SSD> { jjtThis.set(OP_COMMAND, OP_SSD); }
        |
        <SSR> { jjtThis.set(OP_COMMAND, OP_SSR); }
}
```

Figure 4.5: JJTree Source Code

Figure 4.5 gives a simplified example of the JJTree source code for the jRelix *"Command"* syntax, where the syntax for all jRelix commands e.g. *"deld"*, *"delr"* etc. are specified. As well, the actions to be performed when various jRelix commands are entered by the end user are also declared here. The description for how to write JJTree source code is beyond the scope of this thesis. Readers may, however, refer to corresponding documentation on JJTree and JavaCC for detailed explanation.

After the parser is generated by JJTree and JavaCC, the generated Java code is ready to be compiled in order to produce the object code for the jRelix parser. When executing the parser, the input is the user command, while the output is the so-called *"syntax tree"*. A syntax tree is an internal representation of the user command. It comprises nodes and relationships. Figure 4.6 illustrates the syntax tree of the jRelix command *"let SumAB be A + B;"*. In the figure, the nodes are represented by labeled circles (e.g. *"Declaration"*), and the relationships are represented by arrows. Details of the jRelix syntax tree will be explored later in this chapter.

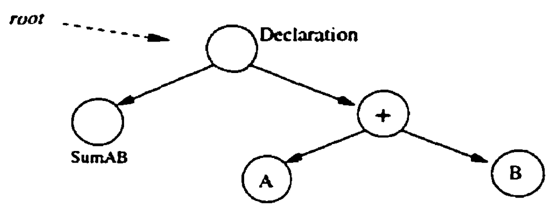


Figure 4.6: An Example of Syntax Tree Produced by the Parser

The jRelix interpreter receives syntax trees passed from the parser and does certain evaluations such as error checking etc. It then calls different function modules to perform operations. In the jRelix implementation, an Interpreter class is built to represent the interpreter. The evaluator is embedded in the interpreter.

In the jRelix system, all user inputs are captured and translated into a syntax tree by the parser/interpreter; the evaluator makes the syntax tree understandable to the rest of the jRelix system. In addition, a top-level expression evaluator is also involved in actualization of lower-level nested virtual domains.

4.4 System Table Maintainer

As mentioned before, upon declaration and initialization, a relation is stored in a file whose name corresponds to the name of the relation. User-defined relations including domains etc. are maintained in a jRelix database. Every jRelix database maintains a set of "*system tables*" which represent the data dictionary of the database and are stored permanently as system files. Three basic system tables are used to store information about domains, relations and relevant components. Sections 4.4.1 to 4.4.3 discuss these system tables respectively. The term "*system table*" has two different meanings regarding their storage formats, i.e. the storage format as permanent files on hard disk, and the memory format stored in RAM. Both formats will be discussed.

Figure 4.7 describes the maintenance of the system tables. Details about the maintenance mechanism will be explored in section 4.4.5.

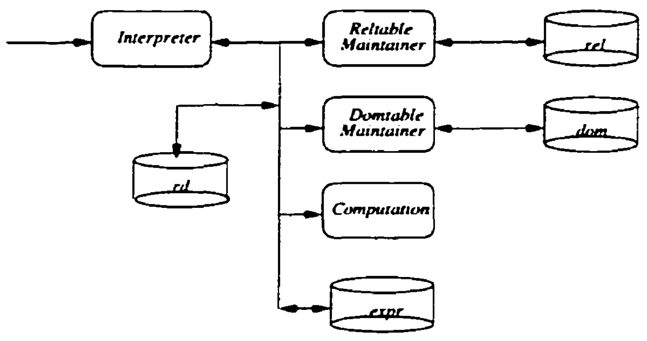
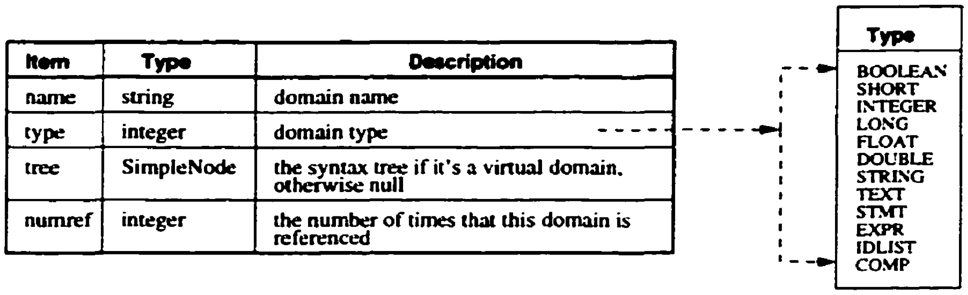


Figure 4.7: System Table Maintainer

4.4.1 Domain Table

In the jRelix implementation, the memory version of the information on all domains in the database is maintained in a hash-table, with domain names as hash keys. Each item in the hash-table has the structure depicted in figure 4.8. The hash-table is maintained by a DomTable class, which performs various operations on domain items (e.g. adds a new domain to the hash-table etc.); the domain item structure is represented by a DomEntry class.



Item	Type	Description
name	string	domain name
type	integer	domain type
tree	SimpleNode	the syntax tree if it's a virtual domain. otherwise null
numref	integer	the number of times that this domain is referenced

Type
BOOLEAN
SHORT
INTEGER
LONG
FLOAT
DOUBLE
STRING
TEXT
STMT
EXPR
IDLIST
COMP

Figure 4.8: Domain Table Format (In-RAM Version)

Usually we call the memory-version of the domain table "*domtable*"; On the other hand, the domain table information that is stored permanently in a disk file is named "*.dom*". The storage format of a *.dom* file is depicted in Figure 4.9. Obviously, the format of file *.dom* is quite similar to that of the "*In-RAM*" version domtable. The only difference is the storage of the syntax tree for virtual domains. Section 4.4.4 describes how jRelix handles the syntax tree information, and section 4.4.5 explains the details of domain table maintenance.

Item	Type	Description
name	string	domain name
type	integer	domain type
numref	integer	the number of times that this domain is referenced

Figure 4.9: Storage Format of File *.dom*

4.4.2 Relation Table

Information on all relations in the database is also maintained by a hash-table in memory, with relation names as hash keys. Each item in the hash-table has the structure depicted in figure 4.10. In the jRelix implementation, the hash-table is maintained by the RelTable class, which performs various operations on the system relation table (e.g. add a new relation to the hash-table); and the Relation class describes the relation entry structure, as well as perform the relational operations (e.g. joins, projections and selections etc.).

Item	Type	Description
name	string	relation name
rvc	integer	type (RELATION. VIEW or COMPUTATION)
numtuples	integer	the number of tuples in this relation
numattrs	integer	the number of attributes in this relation
numsortattrs	integer	the number of sorted attributes
tree	SimpleNode	syntax tree root if it is a view
domains	Domain[]	array of domain objects
data	Object[]	pointer to relation data
capacity	int	capacity of data

Figure 4.10: Relation Table Format (In-RAM Version)

The memory-version of a relation table is usually called "*reltable*". On the other hand, the relation table information that is stored permanently in a disk file is named "*.rel*". The storage format of a *.rel* file is illustrated in Figure 4.11. The items in file *.rel* are part of the items of the *In-RAM* version of reltable. The information about domain items a relation is defined on is not stored in file *.rel*, instead it is maintained in another file called *.rd*, which will be discussed in section 4.4.3. Also, syntax tree information for views is not stored in file *.rel*. Section 4.4.4 describes how jRelix handles the syntax tree information, and section 4.4.5 explains the details of reltable maintenance.

Item	Type	Description
name	string	relation name
rvc	integer	type (RELATION. VIEW or COMPUTATION)
numtuples	integer	the number of tuples in this relation
numattrs	integer	the number of attributes in this relation
numsortattrs	integer	the number of sorted attributes

Figure 4.11: Storage Format of File *.rel*

4.4.3 RD Table

Information that links the relations with the domains on which they are defined is maintained by the so-called RelDom (or RD) table. This kind of information is stored permanently in a disk file named "*.rd*". Figure 4.12 describes the file format for this *.rd* table. However, different from *domtable* and *reltable*, there is no memory-version of *RD* table. As explained in section 4.4.5, RD table information is loaded from the *.rd* disk file by Interpreter and inserted into *reltable* and *domtable* on the fly. The same is true when jRelix stores RD information to the disk file *.rd*.

Item	**Type**	**Description**
relName	string	relation name
domName	string	domain name
position	integer	the position of this domain in current relation

Figure 4.12: File Structure of .rd

4.4.4 Expression in System Tables

As we know, the definitions of virtual domain and views are represented by expression syntax trees which are a set of SimpleNode's connected in a tree structure. Figure 4.13 depicts the expression tree of virtual domain x, where x is declared by "*let x be S ijoin T ujoin U;*".

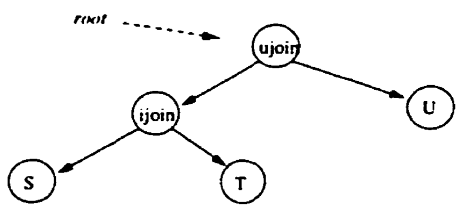


Figure 4.13: Expression Tree of "*let x be S ijoin T ujoin U;*"

A temporary hash-table is used by the ExprTable class to store expression trees when loading/saving the trees from/to a system disk file *.expr*. It is *temporary* because only when the system is starting/exiting is this hash-table used to hold the expression trees. During most of the execution time, the expression trees are maintained within domtable (for virtual domains) and reltable (for views).

Since expression trees are composed of SimpleNode objects, they are stored in the *.expr* file by means of Java's serialization I/O. ExprTable is in charge of serializing the tree streams to/from the disk. Details are explained in section 4.4.5.

4.4.5 System Table Initialization and Saving

As illustrated in Figure 4.7, jRelix maintains three basic system tables on disk, i.e. domtable (*.dom*), reltable (*.rel*) and "*rd*" table (*.rd*). The Java serialized expression table (*.expr*) is a complementary table that maintains the syntax tree for virtual domains and views. During a system session, jRelix basically maintains the domtable and reltable in memory. All information contained in the "*rd*" table and expression table is loaded into or extracted from the domtable and reltable during the system initialization and/or the system exiting time.

System Table Initialization

Figure 4.14 illustrates the system table initialization procedure.

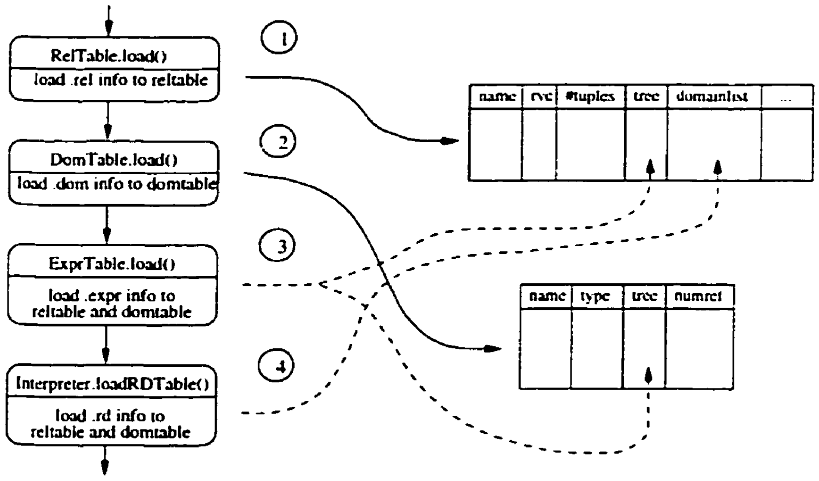

RelTable.load()
load .rel info to reltable
DomTable.load()
load .dom info to domtable
ExprTable.load()
load .expr info to reltable and domtable
Interpreter.loadRDTable()
load .rd info to reltable and domtable
1
2
3
4
name
rvc
#tuples
tree
domainlist
...
name
type
tree
numrel

Figure 4.14: System Table Initialization Procedure

1. Upon starting the system, the interpreter firstly initializes the reltable which involves constructing a RelTable object. The RelTable constructor calls its *load()* method which loads the *.rel* from disk. The loading is achieved by using a BlockInputStream. At this

stage however, the expression trees for view and the relations domain list are not yet loaded.

If no *.rel* file is found in the current database directory, jRelix initializes the reltable with the items illustrated in Figure 4.15, which are basic system relation items. This deals with the case where the jRelix database is newly created. Obviously, the system relation's name starts with a ".".

Name	#Tuples	#Attributes	Type
.rel	3	5	RELATION
.dom	8	3	RELATION
.rd	10	3	RELATION

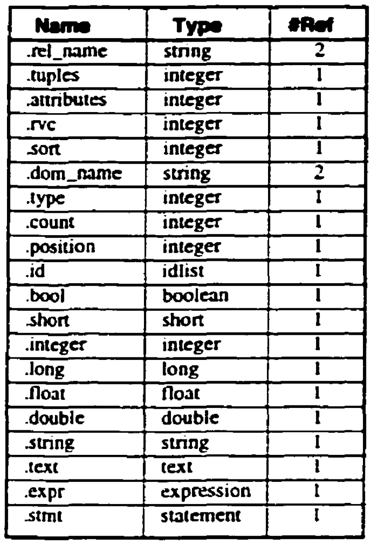
Figure 4.15: Initial Entries in Reltable

2. The interpreter then initializes the domtable by constructing a DomTable object. The DomTable constructor calls its *load()* method which loads the *.dom* from disk. The loading is achieved by using a BlockInputStream. At this stage, the expression trees for virtual domains are not yet loaded into domtable.

   If no *.dom* file found in the current database directory, jRelix initializes the domtable with the items illustrated in Figure 4.16, which are basic system domain items. This deals with the case where the jRelix database is newly created. Obviously, the system domain's name starts with a ".".

3. Next, an ExprTable object is constructed by the interpreter, and thus the expression trees are loaded from *.expr* file. As mentioned in section 4.4.4, expression trees are stored on disk by Java's object serialization mechanism. Hence, an ObjectInputStream is used to load the expression information.

   Expression trees are loaded into a temporary hash-table which is maintained by the ExprTable class. ExprTable then calls the *insertRoot()* methods of domtable and reltable to insert loaded tree objects into these two tables respectively.



Name	Type	#Ref
.rel_name	string	2
.tuples	integer	1
.attributes	integer	1
.rvc	integer	1
.sort	integer	1
.dom_name	string	2
.type	integer	1
.count	integer	1
.position	integer	1
.id	idlist	1
.bool	boolean	1
.short	short	1
.integer	integer	1
.long	long	1
.float	float	1
.double	double	1
.string	string	1
.text	text	1
.expr	expression	1
.stmt	statement	1

Figure 4.16: Initial Entries in Domtable

As we'll see in next section (System Table Saving), expressions for a virtual domain are prefixed by a "." before saving, so that the loader of ExprTable can correctly decide which expression goes to domtable and which goes to reltable.

4. Finally, the interpreter calls its *loadRDTable()* method to load the RD information from the *.rd* file using a BlockInputStream. As mentioned in Section 4.4.3, RD information describes which relation is defined on which attributes. Attributes that belong to a relation are inserted in the reltable by its insertIDList() method call.

   If no *.rd* file is found in the current database directory, jRelix initializes the RD information with the items illustrated in Figure 4.17, which are basic system relations corresponding to their attributes (i.e. domains). This deals with the situation where the jRelix database is newly created.

   It's easy to see that the initial system relations (i.e. *.rel, .dom and .rd* in Figure 4.17) are predefined in Figure 4.15, which includes the initial relation items. Their corresponding domains are predefined in Figure 4.16.

.rel	.dom	.rd
.rel_name .tuples .attributes .rvc .sort	.dom_name .type .count	.rel_name .dom_name .position

Figure 4.17: Initial RD Entries

System Table Saving

System table saving is kind of a reverse procedure of the system table initialization explained previously. Figure 4.18 illustrates the system table saving procedure which occurs at the end of a jRelix session.

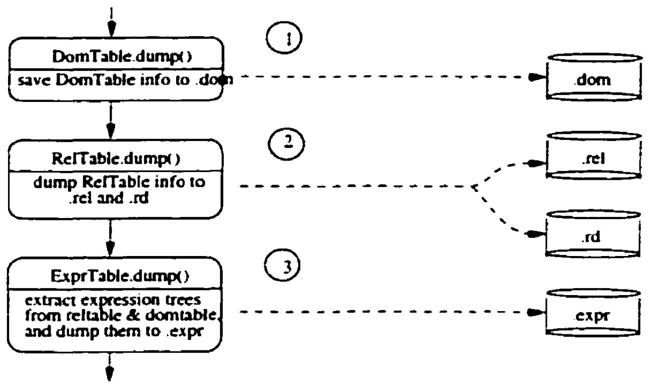

DomTable.dump()
save DomTable info to .dom
1
.dom
RelTable.dump()
dump RelTable info to .rel and .rd
2
.rel
.rd
ExprTable.dump()
extract expression trees from reltable & domtable, and dump them to .expr
3
expr

Figure 4.18: System Table Saving Procedure

1. Right before exiting, the jRelix interpreter calls DomTable's *dump()* method, which saves the domtable information to the *.dom* file by using a BlockOutputStream. Note however, that this procedure only saves domain's *name, type* and *#reference* fields to disk. It does not touch the expression tree of virtual domains, because the expression tree information will be handled by ExprTable object, as explained in step 3.

2. The interpreter then calls RelTable's *dump()* method which saves the reltable information to *.rel* by using a BlockOutputStream. It also saves the relation and its corresponding

domain information to *.rd*. But it does not do anything for a view's expression tree due to same reason as that given in step 1.

3. Finally, ExprTable's *dump()* method is called by the interpreter. This method extracts the expression trees from domtable and reltable by calling their *fillExprTable()* methods, and serializes the tree objects to the disk file *.expr*. Before that, DomTable's *fillExprTable()* method prefixes a "." to the virtual domain's name. This tells the loader of the expression trees which expression tree belongs to virtual domains, and which belongs to views.

4.5 Virtual Domain Actualizer

The virtual domain actualizer is responsible for the functionalities of domain algebra in jRelix. Hence, it is needless to say that it is one of the key components in the system. The actualizer implements both horizontal and vertical operations on a relation. In particular, it supports nested domain operations such that domain algebra and relational algebra are well integrated. In other words, domain algebra becomes a super-set of relational algebra in the jRelix implementation. When the user runs a nested domain operation, relational operations are invoked and run against a set of *sub-relations* which are attributes of the upper-level relations.

Computation is also integrated into the actualizer. It can be regarded as a virtual procedure call which accepts parameters from its environment, and outputs the result as a relation. From the actualizer point of view, the computation is applied onto a tuple-by-tuple level, which is quite similar to a virtual domain. But compared with the domain actualizer, the computation processor provides a much stronger handling capability of complex operations.

Figure 4.19 illustrates the basic control flow of a virtual domain actualizer.

Apart from actualization, a virtual domain actualizer is also in charge of virtual domain validation checks,, operand type compatibility testing and mutually recursive definition detection etc. This sort of run-time check is much stronger than the checking during declaration time. Since the virtual domain declaration has a close relationship with the actualizer, we move its description from section 4.4 to the next section. Section 4.5.2 describes the procedure used to construct a virtual domain actualizer. The details of virtual tree building and various validation

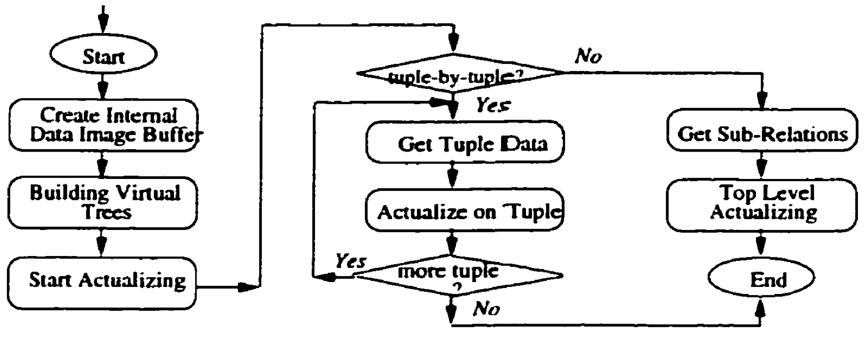


Figure 4.19: Basic Control Flow of a Virtual Domain Actualizer

checks are explored in section 4.5.3. Section 4.5.4 and 4.5.5 describe the detailed actualization procedures for the tuple-by-tuple approach and the top-level approach respectively, which are central parts of an actualizer.

4.5.1 Virtual Domain Declaration

Declaring a virtual domain is quite similar to defining a procedure call in other programming languages such as C and Java, with the procedure body represented in the form of an expression tree. Figure 4.20 illustrates this idea.

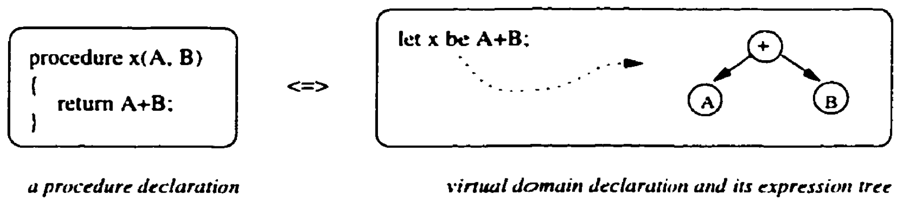


Figure 4.20: Virtual Domain Declaration

On the other hand, the virtual domain declaration command itself is in the form of a syntax tree as depicted in Figure 4.21. Obviously, this syntax tree includes the expression tree for the virtual domain definition (the dashed rectangle in Figure 4.21). Therefore, the syntax tree of a virtual domain can be simply "cut off" from the syntax tree of the declaration command.

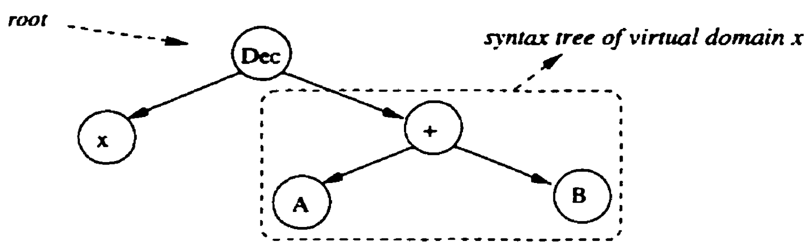


Figure 4.21: Syntax Tree for the Command *"let x be A+B;"*

However, before the virtual domain expression tree is cut off from the syntax tree of the declaration command and is inserted into the domtable, the following procedure is performed.

1. Traverse the expression tree to make sure that all identifiers in the tree have been already declared in domtable or reltable (for top-level relations). This means that a virtual domain must be defined on something that already exists, otherwise, an error message is displayed. This procedure is executed by *DomTable.traverseTree()*.

2. If we are redeclaring an existing domain, we just insert/replace the expression tree in that domain's 'tree' field. Note, the reference counters of those referenced domains are not incremented. However, before we do some actual work, we must make sure that old/new domains' types are identical even if we want to overwrite a virtual domain. This procedure is partially done by *DomTable.traverseType()*.

3. If we are declaring a new virtual domain, simply put a new Domain entry with the expression tree in the hash-table maintained by DomTable. Note, the reference counters of those referenced domains are not incremented.

4. If the type of the new virtual domain is *idlist* (as returned by *DomTable.travseType()*), this must be a nested relational domain (refer back to section 3.2.2 for introduction of the *idlist* type). The attributes list of this virtual domain is figured out by calling the *DomTable.getIDList()* method and a new relation entry is added into reltable by using *Relation.addRel()* method. This relation has the same name as the virtual domain except that it is prefixed with a "." (hence is an invisible relation. Refer to section 3.2.1). It is

supposed to hold tuples for the nested relational domain. Finally, as described in section 3.2.2, the reference counters of those domains that are used by this invisible relation are incremented by one.

DomTable.traverseType() traverses the expression tree and checks the type compatibilities. Figure 4.22 lists possible type combinations for various operators. Any operation that does not agree with the rules illustrated in this figure cause a type-mismatch error message.

Operator	Left & Right Operands	Result Type
min, max, plus minus, multiply divide, mod uplus, uminus pow	numeric type (i.e. short, integer, long, float, double	numeric type (*)
cat	string & string	string
eq, neq, gt, lt ge, le	numeric & numeric text & text bool & bool	bool
or, and, unot	bool & bool	bool
ijoin, ujoin sjoin, ljoin rjoin, dljoin drjoin	idlist & idlist idlist & computation computation & idlist	idlist

(*) if one of the operands is of double type, the result type is double
otherwise, if one of the operands is of float type, the result type is float
otherwise, if one of the operands is of long type, the result type is long
otherwise, if one of the operands is of integer type, the result type is integer
otherwise, the result type is short.

Figure 4.22: Possible Type Combinations

Even though a virtual domain's expression tree is inserted in the memory version of domtable, it is not stored in the *.dom* file on disk, which is the system file for domtable information. It is actually stored in a separate *.expr* file as explained in section 4.4.4 and 4.4.5.

One thing that should be noted here is that domtable does not hold the attribute list for an idlist-type nested relational domain. As already explained, when declaring an idlist-type domain, a new relation is inserted into domtable which will hold the data for this domain. The attribute list of the new nested relational domain is retrievable from the reltable but not the domtable. Hence, when displaying domain information using the "*sd*" show-domain command, jRelix has to go get the corresponding attribute list from *reltable* for this nested domain and then display other information using *domtable*. Figure 4.23 illustrates this scenario.

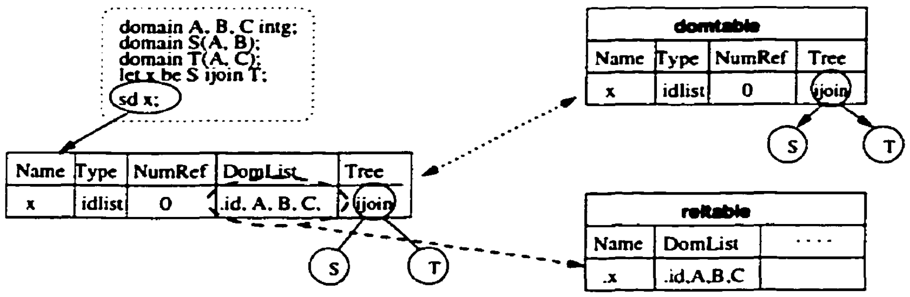


Figure 4.23: Displaying an IDLIST-typed Domain

4.5.2 Using an Actualizer

This section discusses how to use the actualizer from the application programmer's point of view. The *Actualizer* in jRelix is implemented as a *source code component* which can be reused by other modules e.g. relational algebra or computation at the source code level. This section talks about the usage of this source code component, i.e. the construction of an actualizer, the central actualization procedure, and the final clean-ups when finished using an actualizer. Readers who are interested in implementation issues regarding the use of the existent actualizer mechanism to build other functional modules may want to read this section, whereas the details about how an actualizer works to fulfill the actualization task are described in the rest of the chapter.

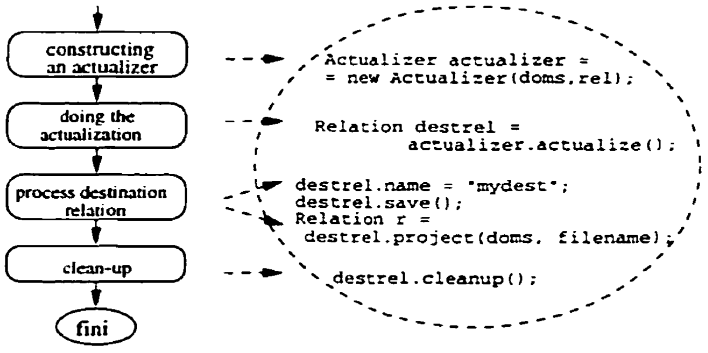
In the jRelix implementation, an Actualizer class is designed to take the responsibility of the virtual domain's actualization. Whenever actualization is necessary (e.g. when virtual domains are involved in projection, selection and joins etc.), an object of the Actualizer class must be constructed.

From an application developer's point of view, constructing an actualizer is quite simple. When an actualizer is involved, there must be a set of virtual domains which need to be actualized, and a source relation on which these virtual domains will be actualized. Hence, the constructor of the actualizer accepts these two elements as parameters, i.e. it has the following prototype:

Actualizer(Domain[] domains, Relation srcrelation);

Usage of an actualizer is also very easy. When the actualizer is constructed, the virtual domains' actualization can be performed at any time by calling Actualizer's *actualize()* method, which returns a destination relation containing the actualized virtual domain's fields. The application programmer can subsequently do certain operations e.g. projection and selection etc. on the destination relation.

When the operations are finished, an actualizer user should not forget to call Actualizer's *cleanup()* method. As we will see in subsection *Virtual Tree Truncation* of section 4.5.4, temporary intermediate domains may be created (and inserted into the system tables) during the virtual domain's actualization, and data for those intermediate domains will be filled into their corresponding data columns. The *cleanup()* method is in charge of removing all intermediate domains from the domtable in order to make it consistent with the system status before running an actualizer. This is important since system tables will be permanently stored on disk. If certain intermediate domains (or relations) are not removed cleanly, they will exist on the system forever.



```
Actualizer actualizer =
= new Actualizer(doms,rel);

Relation destrel =
        actualizer.actualize();

destrel.name = "mydest";
destrel.save();
Relation r =
 destrel.project(doms, filename);

destrel.cleanup();
```

Figure 4.24: An Example of Using an Actualizer

Figure 4.24 gives an example of using an actualizer.

4.5.3 Actualizer Initialization

As mentioned in section 4.5.2, to initialize an actualizer for a (set of) virtual domain(s), two parameters (i.e. the source relation and an array of the domains which are to be actualized) need to be passed to the constructor of the Actualizer class. Upon receiving these parameters, the Actualizer's constructor does the following initialization procedure:

1. Initializes all the internal buffers and data members which are used during the actualization. This includes creating internal vectors and hash tables that are supposed to hold various intermediate data objects such as syntax trees for all the virtual domains, and virtual domains that perform vertical operations etc.
2. Goes through the domain list and for each domain, does the following:
   - If it is an actual domain which already exists in the source relation, adds this domain to the actualizer's *actdoms* vector. This domain's data can be found in the source relation directly. No further actualization is necessary.
   - If it is an actual domain which does not exist in the source relation, gives an error message since it cannot be actualized.
   - If none of the above cases occurs, this domain must be a virtual domain. The constructor expands this domain's expression tree.
3. If it is a nested relational domain and a top-level approach is being used, the expanded expression tree must be passed to a *processIDListDom()* method to do further processing.

The last step is very important in order to make sure the virtual domain is actualizable. The basic idea is that after tree expansion, all nodes in the tree must be actual domain (identifier) nodes which can be actualized on the source relation. An exception is reduction nodes. As we will see later, virtual domains including reductions must be actualized in multiple passes. Figure 4.25 gives an example of this case.

In this example, virtual domain x is defined by *"let x be A+(red+ of (equiv* of B by A))"*. There are two reduction nodes in domain x's expression tree. Since it is impossible for the actualizer to actualize this tree in only one pass, temporary domains e.g. domain *0* and *1* must be created to hold the intermediate data. In the tuple-by-tuple approach, this work is basically done by the tree expansion procedure (i.e. *buildTree()* method); whereas in the top-level approach, it is done by the *processIDListDom()* method. Note that *processIDListDom()* method also does

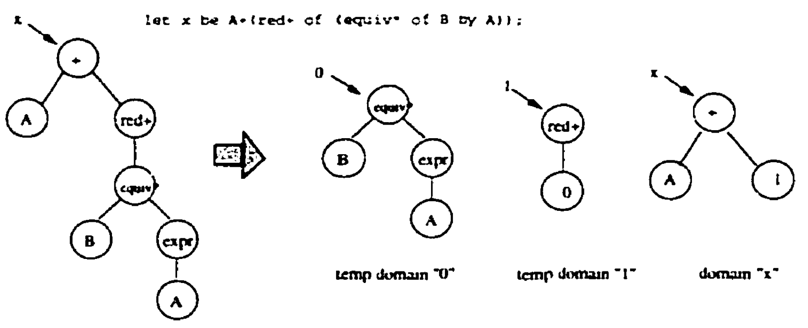


Figure 4.25: A Virtual Domain's Expression Tree with Reduction Nodes

similar work for virtual domains with multi-level joins involved, as illustrated by Figure 4.26. For a detailed description of *processIDListDomain()*, readers may refer to section 4.5.6.

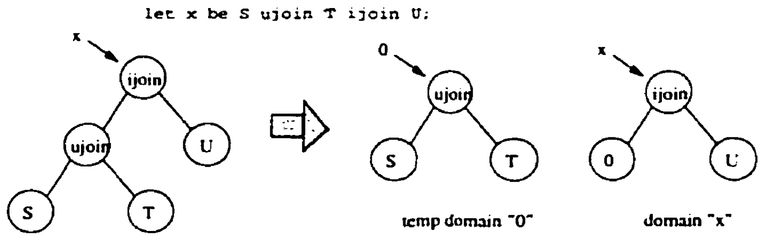


Figure 4.26: A Virtual Domain's Expression Tree with Multiple Join Nodes

The reason for this is that although virtual domains with multiple joins involved can be actualized directly in the tuple-by-tuple approach, it cannot be handled directly by the top-level approach, in which only one join can be dealt with at a time. Details will be explained in section 4.5.5 and section 4.5.6.

4.5.4 Building Virtual Trees

Virtual tree building is quite an important procedure during actualization. The purpose of building virtual trees is to make sure that the virtual domain (the tree is associated with) is safely actualizable on the source relation in the future. There are many possibilities that hinder the virtual tree from *"blossoming"* i.e. being actualized, for example:

- the virtual domain is defined on some actual domains which can never be actualized on the source relation, or which do not exist in the system at all.
- the virtual domain is defined on some other virtual domains which can never be actualized on the source relation.
- the virtual domains that *this* virtual domain is defined on are mutually defined on each other, i.e. there is a transitive loop in the virtual tree.
- there are semantic errors in the virtual tree, e.g. the type-mismatch error in *"let x be string_name + relation_name;"*.

Apart from these potential problems, a virtual tree must be reorganized, and intermediate virtual domains must be generated in order to handle the situations mentioned in last section.

In the jRelix implementation, a *buildTree()* method is created for the Actualizer class to handle the task of virtual tree building. Its major functions will now be described.

Validity Check

The *buildTree()* method is basically a recursive routine which frequently calls itself by passing a SimpleNode parameter during tree building. The node passed to *buildTree()* is carefully analyzed and its children are taken out and passed to *buildTree()* again as a lower-level invocation. *buildTree()* analyzes the node according its *type* field. When an identifier node is encountered, a validity check has to be performed. The idea of a validity check is quite simple, i.e. the domtable is first consulted. If no entry is found in domtable but the current node's parent is an IDLIST-related node (e.g. of OP_JOINOPERATOR type), the reltable is consulted. If no entry is found in both cases, the validity check fails and further processing is stopped by the actualizer.

When a relevant entry is found in the system tables (either domtable or reltable), the actualizer looks into its source relation and verifies that this entry belongs to the source relation's attribute (or domain) list. In that case, the identifier node representing the current entry is actualizable; otherwise, the actualizer continues to check if the current node represents a virtual domain. In this case, the tree has to be expanded as described in the next sub-section; otherwise, it is clear that the current entry represents an actual domain which is however not actualizable on the source relation. Figure 4.27 illustrates this procedure.

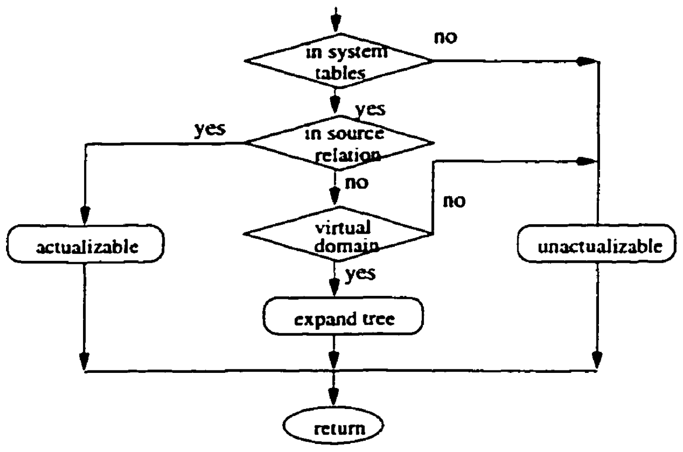


Figure 4.27: Validity Check

Virtual Tree Expansion

As we know, building the virtual tree is an important procedure during virtual domain actualization. When the final actualization happens, the virtual domain's syntax tree is passed to the actualizer engine, which subsequently consults the tree definition and generates tuple data based on different approaches i.e. the tuple-by-tuple approach (refer to section 4.5.5) or the top-level approach (refer to section 4.5.6).

The purpose of tree expansion is to make sure that there is no virtual domain node in the resulting tree. In other words, all identifier nodes in the final syntax tree must be actual domains which are actualizable on the source relation. Figure 4.28 illustrates the idea.

In this example, virtual domain z is to be actualized. Since it is defined on two other virtual domains x and y, its syntax tree must be expanded in order to replace x and y with actual domains they are defined on. Furthermore, virtual domain y is defined on virtual domain x, hence another expansion is required to insert domain x's syntax tree into where node x resides in the syntax tree of domain y. As illustrated in the figure, there is no virtual domain node in the final syntax tree of virtual domain z.

Figure 4.29 gives a more complex example in which relational operations such as projection, selection and joins are involved. Although the syntax trees for relational operations are much more complicated than those of normal arithmetic operations, the basic idea for tree expansion

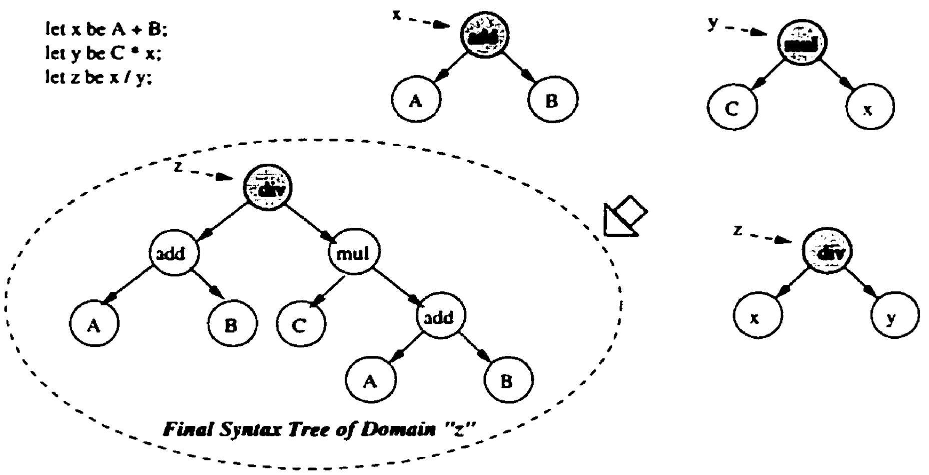


Figure 4.28: Example of Tree Expansion

remains the same. Therefore, detailed explanations for this example are omitted.

Tree expansion occurs when the *buildTree()* method finds that nodes passed to it are virtual domain nodes (of type *OP_IDENTIFIER*). The first thing it will do is to duplicate the syntax tree of that virtual domain. This is important since further operations including tree expansion might be performed on the syntax tree of that virtual domain, and the original syntax tree of the virtual domain maintained in the jRelix system table should not be modified. *buildTree()* calls *SimpleNode*'s *jjtDuplicate()* method to create a copy of the original syntax tree node. Any further operations will only be applied to this copy.

Secondly, the syntax tree of the virtual domain in question needs to be inserted where the virtual domain node was residing. *buildTree()* handles this by calling *SimpleNode*'s *jjtReplaceChild()* or *jjtReplace()* methods. After this, the syntax tree of the top-level virtual domain is expanded to become bigger.

Due to its recursive nature, *buildTree()* continues to analyze the lower-level syntax tree just inserted, and performs further tree expansion when necessary. Finally, the so-called "*big-tree*" is generated, which only contains node of actual domains.

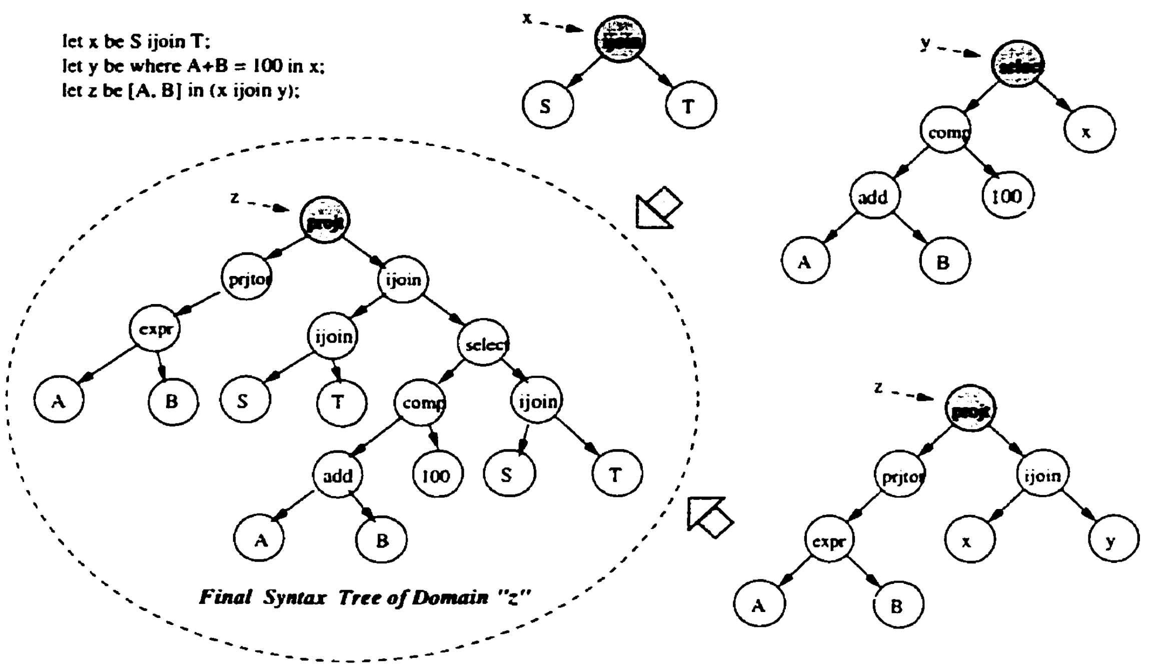
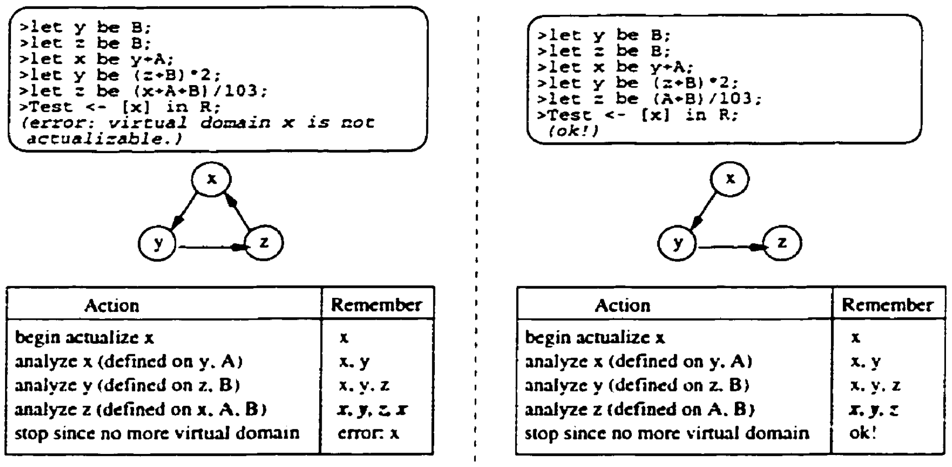


Figure 4.29: More Complex Example of Tree Expansion

Recursive Loop Detection

As illustrated in Figure 3.40, a virtual domain is regarded *unactualizable* if it is recursively defined on itself, i.e. there is a recursive loop in the definition of the virtual domain in question. The mechanism used to detect a recursive loop is quite simple. It is exemplified in Figure 4.30.

Since *buildTree()* is a recursive method which analyzes each node of the syntax tree that is passed to it, it sees all the domain nodes (either virtual or actual) contained in the final syntax tree. *buildTree()* remembers the domains it has seen so far, therefore, when it starts to analyze a new domain, recursive definitions can be detected. In the jRelix implementation, a vector object *curpath* is used to store all the domains (actually, only the virtual domains are relevant) *buildTree()* has seen along the path to the current node. If the current virtual domain node already exists in the *curpath* vector, it means that this domain is somehow (recursively) defined on itself, and an error message should be generated to notify you of the problem. To make this mechanism work, a virtual domain node is inserted into the vector before *buildTree()* recursively calls itself, and the same domain node is removed from the vector after *buildTree()* returns from



```
>let y be B;
>let z be B;
>let x be y+A;
>let y be (z+B)*2;
>let z be (x+A+B)/103;
>Test <- [x] in R;
(error: virtual domain x is not
 actualizable.)
```

Action	Remember
begin actualize x	x
analyze x (defined on y, A)	x, y
analyze y (defined on z, B)	x, y, z
analyze z (defined on x, A, B)	x, y, z, x
stop since no more virtual domain	error: x

```
>let y be B;
>let z be B;
>let x be y+A;
>let y be (z+B)*2;
>let z be (A+B)/103;
>Test <- [x] in R;
 (ok!)
```

Action	Remember
begin actualize x	x
analyze x (defined on y, A)	x, y
analyze y (defined on z, B)	x, y, z
analyze z (defined on A, B)	x, y, z
stop since no more virtual domain	ok!

Figure 4.30: Examples of Detecting Recursive Loop

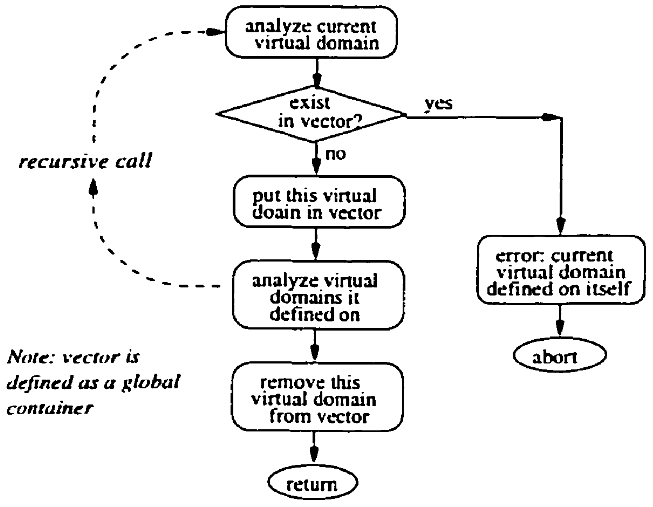

analyze current virtual domain
exist in vector?
yes
no
put this virtual doain in vector
analyze virtual domains it defined on
remove this virtual domain from vector
return
error: current virtual domain defined on itself
abort
recursive call
Note: vector is defined as a global container

Figure 4.31: Procedure for Recursive Loop Detection

its lower-level recursive call. Figure 4.31 illustrates this procedure.

Virtual Tree Truncation

Apart from expanding a syntax tree for a virtual domain actualization, (as described in previous sub-sections), it is sometimes necessary to truncate a syntax tree in order to fulfill the actualization. There are certain situations in which the actualizer cannot actualize a syntax tree using only one pass. In this case, the virtual tree has to be truncated and separated into several sub-trees, each sub-tree being actualized in seperate passes. Two examples were given in Figure 4.25 and 4.26 of section 4.5.3.

There are altogether three situations when syntax tree truncation must be performed, two of which are mentioned in section 4.5.3. The first case is when a virtual domain is defined on multiple vertical operations (i.e. reductions), and so the vertical operations must be separated and actualized in seperate passes. This is illustrated in Figure 4.25, where intermediate virtual domains "*0*" and "*1*" are generated and are responsible for actualizing the reduction operations *red+* and *equiv** respectively. In addition, the sequence of actualization is of significant importance, i.e. domain "*0*" must be actualized prior to the actualization of domain "*1*", since domain "*1*" depends on the value of domain "*0*".

The second situation is when the *top-level approach* is used to actualize a nested virtual domain. The virtual tree is truncated and separated into sub-trees when multiple joins are involved, with each sub-tree in charge of the actualization of a single join. This is because only one join can be performed by the top-level approach of actualization. An example was given in Figure 4.26.

The third case is that the *by-list* (refer to section 3.5.4) in equivalence reduction contains (arithmetic) expressions rather than domains. Since tuples must be sorted according to the *by-list*, all expressions in the *by-list* must be evaluated before the reduction operation. This requires that the expression tree in *by-list* be truncated and be associated with an intermediate virtual domain, which must be actualized first. This also poses a sequence problem. Figure 4.32 illustrates this situation.

In this example, the sub-tree for the expression "*A+B*" in the *by-list* is truncated and asso-

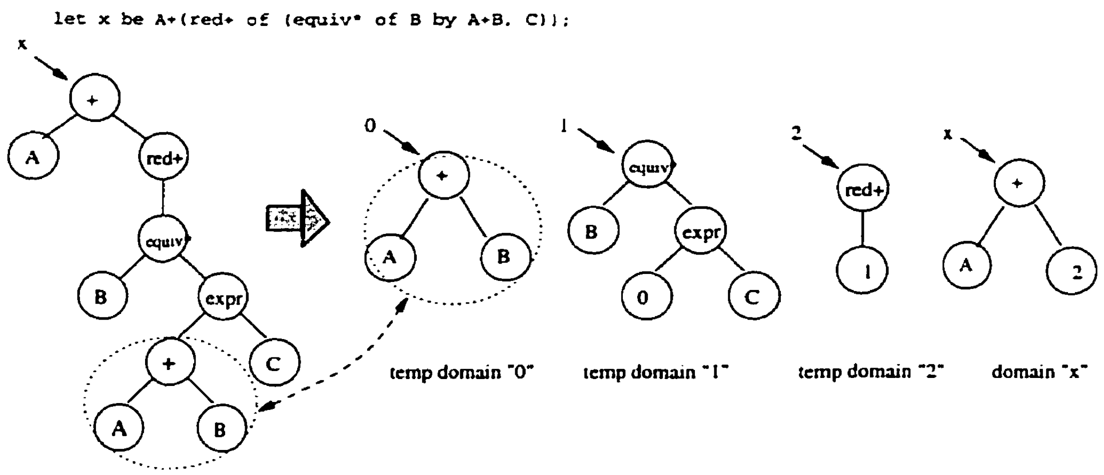


Figure 4.32: Virtual Tree Truncation

ciated with a temporary domain "*0*", which is subsequently used in the temporary domain "*1*" etc. Needless to say, the sequence for (intermediate) domain actualization is *0*, *1*, *2* and finally domain x.

In the jRelix implementation, syntax tree truncation for the first and third cases is handled by the *buildTree()* method, while the second case is dealt with by a *processIDListDom()* method. Both methods generate intermediate domains which are inserted into system *domtable*. Therefore, clean-up is required to remove these temporary domains after actualization. This clean-up procedure is done by a *cleanup()* method, as described in section 4.5.2.

To secure the sequence of actualization, three vectors are used for vertical operation domains, nested domains and normal virtual domains respectively. Vertical operation domains are actualized first, then normal virtual domains (including nested domains when the *tuple-by-tuple approach* is applied), and finally nested domains (in case the *top-level approach* is applied). In addition, an integer-typed *level-tag* is associated with each vertical operation domain. Domains whose level-tags are larger are always actualized first.

4.5.5 Actualization by Tuple-by-Tuple Approach

As mentioned at the beginning of this chapter, a virtual domain is usually actualized on a tuple-by-tuple level, which means that the relation on which the virtual domain is to be actualized is scanned from the first tuple to the last one. The virtual domain value is calculated according to the data in each tuple. This is particularly true with the horizontal operations of domain algebra. For vertical operations, the relation is still scanned and relevant tuple data is stored somewhere for final vertical calculation. This section describes the tuple-by-tuple approach used by the actualizer.

Horizontal Operations

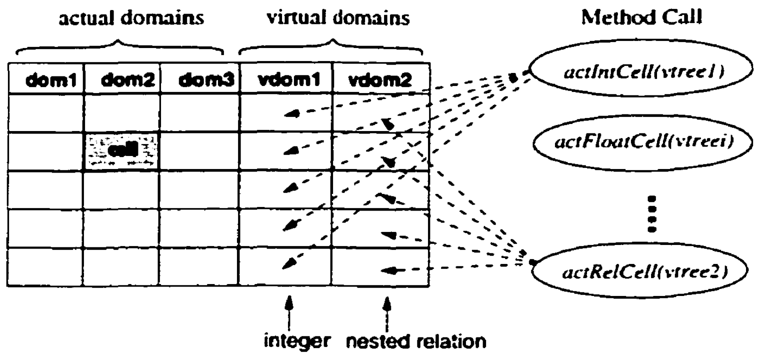


Figure 4.33: Actualization: Fill the Cells

As illustrated in Figure 4.33, to actualizing a virtual domain is similar to filling calculated values into corresponding positions in a table. These positions are termed "*cells*" in the jRelix implementation. The actualized value of a virtual domain occupies a column of *cells* in the table, i.e. relation. The task of actualization is to calculate each virtual domain cell's data by using cell data of actual domains in the same row (or tuple).

In the jRelix implementation, several methods are developed for cell data calculation. These methods are called "*cell-methods*" and are listed in Figure 4.34.

Cell-methods basically accept a syntax tree as their only input parameter and return a calculated value corresponding to their types. Figure 4.35 gives an example of actualizing an

Method	Type of Actualized Domain
actIntCell()	integer, short
actBoolCell()	boolean
actLongCell()	long integer
actDoubleCell()	float, double
actStrCell()	string
actRelCell()	nested domain

Figure 4.34: Methods for Actualizing Cells

integer cell. In this example, a virtual domain *"x"* is defined as *"let x be A+B;"*. The cell-method *actIntCell* takes a virtual tree as input and realizes that domains *A* and *B* are to be involved in the actualization. It then goes through the source relation, grabs the values for *A* and *B* from each tuple and performs the computation according to the syntax tree. For example, for the first tuple, the values of *A* and *B* are 1 and 2 respectively, which are retrieved by *actIntCell* method who subsequently produces *3* as the result.

When multiple types are involved (e.g. *actDoubleCell()* method takes care of both *double* and *float* cells), an explicit type cast is necessary to avoid ambiguity. For each virtual domain actualization, a corresponding cell-method is called in a scanning loop from the first tuple of the source relation to the last. For each tuple of the source relation, a cell-method is firstly invoked by being passed the virtual domain's syntax tree which has been preprocessed by the tree expansion procedure previously described. Therefore, the cell-method knows there will be no problem by using this tree as it was cleaned up at previous stage.

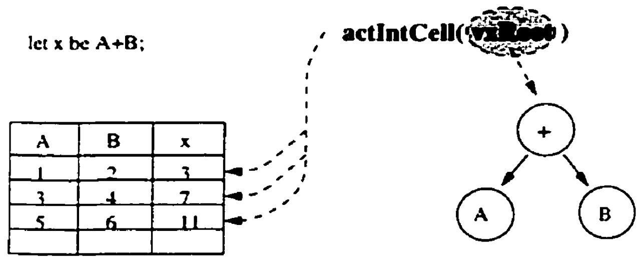
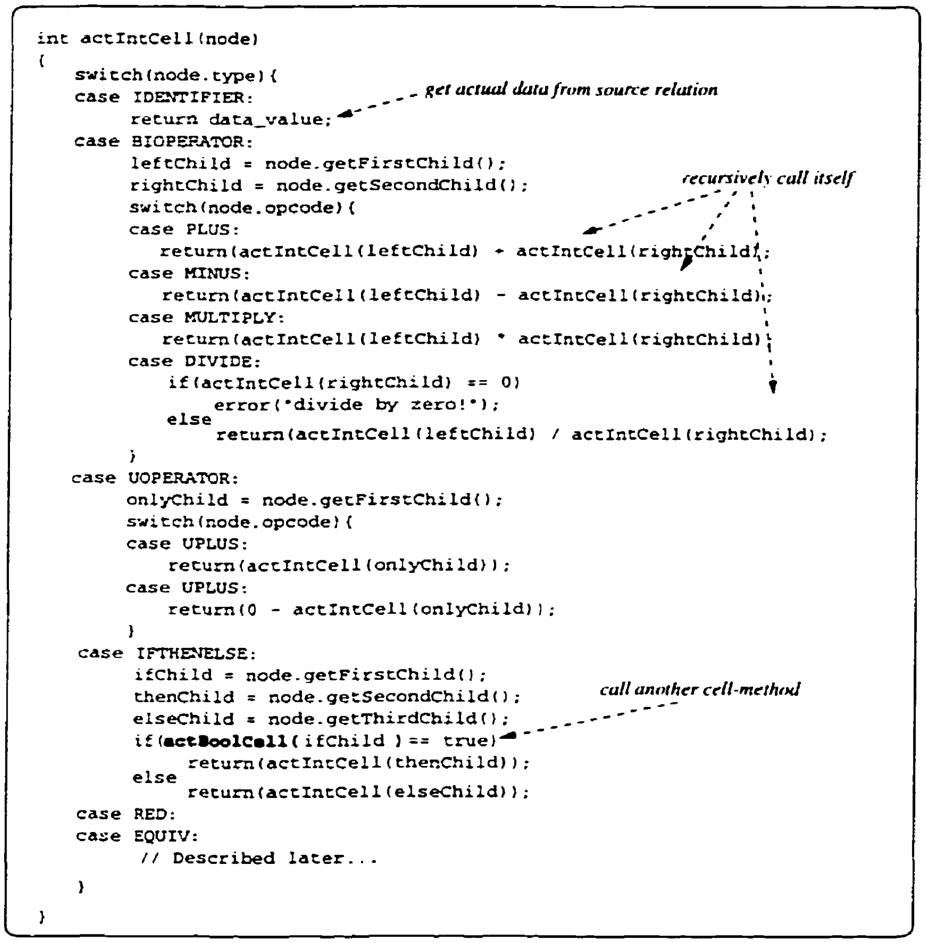


Figure 4.35: Actualizing Integer Cells (Horizontal Operations)

The cell-methods analyze the syntax tree by parsing its structure; access the actual cells' data in the source relation, and calculate the virtual cell values. Sometimes cell-methods may cut off a sub-tree from the *big-tree*, and recursively pass the sub-tree to themselves in order to perform a part of the calculation. This means that cell-methods are basically recursive calls. In addition, cell-methods may call other cell-methods when necessary. Figure 4.36 gives a simplified pseudo code for the *actIntCell()* method implementation (horizontal operations only).



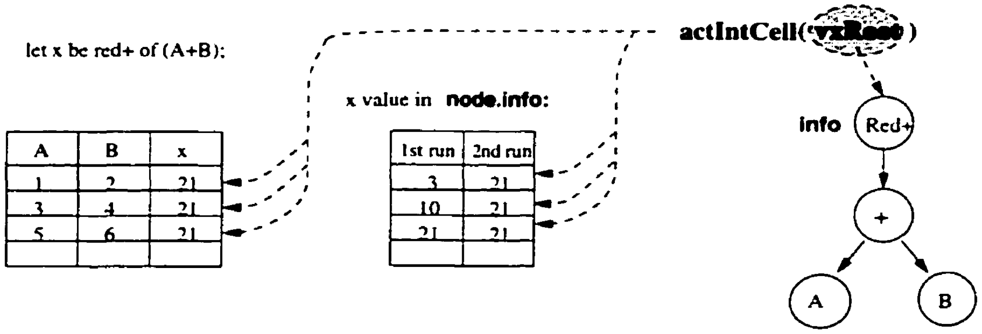
```
int actIntCell(node)
{
   switch(node.type){
   case IDENTIFIER:
        return data_value;
   case BIOPERATOR:
        leftChild = node.getFirstChild();
        rightChild = node.getSecondChild();
        switch(node.opcode){
        case PLUS:
           return(actIntCell(leftChild) + actIntCell(rightChild));
        case MINUS:
           return(actIntCell(leftChild) - actIntCell(rightChild));
        case MULTIPLY:
           return(actIntCell(leftChild) * actIntCell(rightChild));
        case DIVIDE:
            if(actIntCell(rightChild) == 0)
                error("divide by zero!");
            else
                return(actIntCell(leftChild) / actIntCell(rightChild);
        }
   case UOPERATOR:
        onlyChild = node.getFirstChild();
        switch(node.opcode){
        case UPLUS:
            return(actIntCell(onlyChild));
        case UPLUS:
            return(0 - actIntCell(onlyChild));
        }
   case IFTHENELSE:
        ifChild = node.getFirstChild();
        thenChild = node.getSecondChild();
        elseChild = node.getThirdChild();
        if(actBoolCell( ifChild ) == true)
            return(actIntCell(thenChild));
        else
            return(actIntCell(elseChild));
   case RED:
   case EQUIV:
        // Described later...
   }
}
```

Figure 4.36: Pseudo Code for actIntCell() Method (Horizontal Operations)

Vertical Operations

Cell-methods are also responsible for vertical operations. Each node in a syntax tree has a field called "*info*", which is designed as an *Object* type and is supposed to store any possible intermediate values. Naturally, this *info* field is used by cell-methods to save the temporary results of vertical operations. Figure 4.37 illustrates an example of actualizing an integer virtual domain x, defined as "*let x be red+ of (A+B);*". Actualization of virtual domains with vertical operations takes a two-run procedure, i.e. the method needs to go through the relation twice. In the first run, the cell-method scans the relation tuple by tuple, retrieves the tuple values and performs computations according to the syntax tree. The intermediate result calculated for the vertical operations is stored in the *info* field of the current node. In the second run, the final result in the node's *info* field is retrieved by the cell-method and is saved in each tuple of the relation.

In Figure 4.37, method *actintCell* keeps on modifying the "*info*" field of the "*Red+*" node in the first run, i.e. upon scanning the first tuple, the result of A+B (i.e. 3) is stored in field *info*; after scanning the second tuple, the result of A+B plus the old *info* value (i.e. 10) is stored again. When the end of a relation is reached, the final result of the vertical operation (i.e. 21) is stored again. In the second loop, the final result in the *info* field is fetched from the node and stored in the relation.



A	B	x
1	2	21
3	4	21
5	6	21

1st run	2nd run
3	21
10	21
21	21

Figure 4.37: Actualizing Integer Cells (Vertical Operations)

Equivalent operations have similar behavior, except that the source relation is firstly sorted by the cell-method on the "*by-list*" of the equivalent expression, and the cell-methods keep track of the change of the *by-list* value in order to decide when to store the calculated result in the source relation. Detailed descriptions for equivalent operations are omitted due to complexity.

Figure 4.38 illustrates the pseudo code of the *actIntCell()* method for vertical operations.

```
int actIntCell(node)
{
    switch(node.type){
    case IDENTIFIER:
         . . . . .
    case BIOPERATOR:
         . . . . .
    case UOPERATOR:
         . . . . .
    case IFTHENELSE:
         . . . . .
    case RED:
    case EQUIV:
       int actValue= actIntCell((SimpleNode)node.jjtGetChild(0));
       switch(node.opcode){
       case OP_PLUS:
          if(node.info == null)  node.info = new Integer(0);
          node.info = new Integer(((Integer)node.info).intValue()+actValue);
          break;
       case OP_MUL:
          if(node.info == null)  node.info = new Integer(1);
          node.info = new Integer(((Integer)node.info).intValue()*actValue);
          break;
       case OP_MUL:
          if(node.info == null)  node.info = new Integer(Integer.MAX_VALUE);
          node.info = new Integer(
                      Math.min(((Integer)node.info).intValue(), actValue));
          break;
       case OP_MUL:
          if(node.info == null)  node.info = new Integer(Integer.MIN_VALUE);
          node.info = new Integer(
                     Math.max(((Integer)node.info).intValue(), actValue));
          break;
       }
       return actValue;
   }
}
```

Figure 4.38: Pseudo Code for actIntCell() Method (Vertical Operations)

4.5.6 Actualization by Top-Level Approach

As mentioned before, the tuple-by-tuple approach has efficiency problems since a loop within the entire relation is involved. This poses a even more serious problem when actualizing a virtual domain with relational operations on a nested relation, because, for example, joins on a tuple level are supposed to slow down the whole actualization procedure, as highly time-consuming sorting and disk I/O are involved with multiple joins. The top-level approach described in this section deals with this problem by going another way, i.e. it joins two top level nested relations directly, and then does some kind of post-processing which results in the same result as tuple-level actualization.

However, the top-level approach has several limitations which are listed as follows:

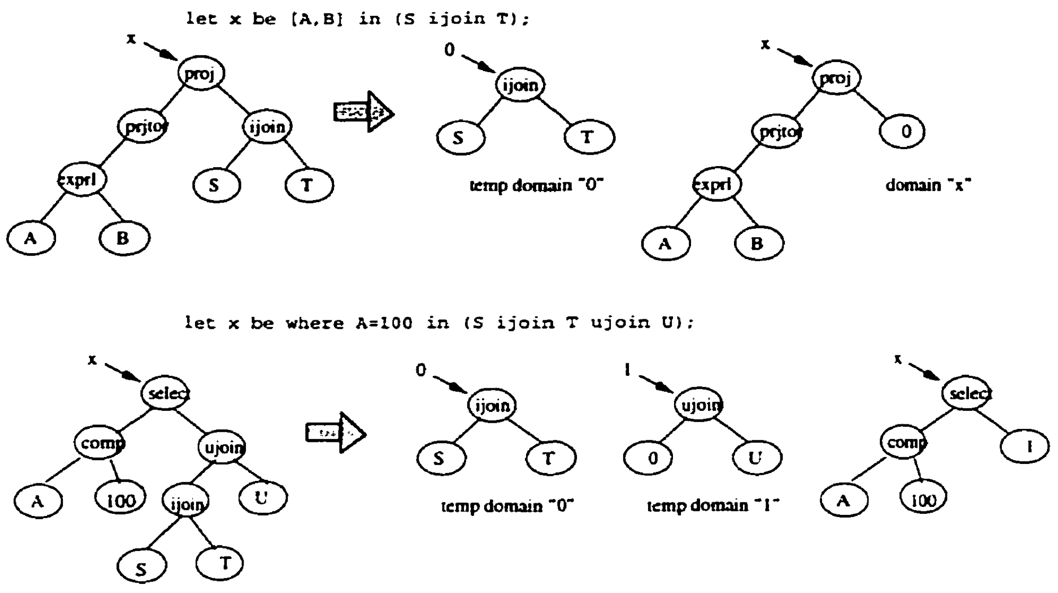
- It can only be applied to actualize nested relational domains in a nested relation. For non-nested virtual domain actualization, the tuple-by-tuple approach is unavoidable.
- The algorithm for the top-level approach can only work for certain relational operations without foreseeable problems. Although three types of relational operations i.e. ijoin. ujoin and sjoin can guarantee that the post-processing combined with top-level joins is able to produce the same result as tuple-level actualization, it is not sure that other types of operations will achieve the same results.
- Actualization of virtual domains with multiple relational operations (e.g. let x be S ijoin T ujoin U) requires additional reorganization (i.e. truncation) of the syntax tree before performing the actualization (refer to next subsection).

Pre-processing of Syntax Trees

In the *top-level approach*, after finishing the basic "*building tree*" procedure described in section 4.5.4, the syntax tree of a virtual domain is passed to a *processIDListDomain()* method for additional processing, as mentioned in the "*Virtual Tree Truncation*" of section 4.5.4 as well as in section 4.5.3.

As declared before, the reason for this additional pre-processing is that only one relational operation can be performed at a time by the top-level approach due to its algorithm. Therefore, tree truncation or reorganization occurs when composition of relational operations exists in the syntax tree of a virtual domain, and intermediate domains are generated for those truncated sub-trees (refer to "*Virtual Tree Truncation*" in section 4.5.4).

Figures 4.26 and 4.39 give some examples of this kind of pre-processing of syntax trees by the top-level approach.



```
let x be [A,B] in (S ijoin T);
let x be where A=100 in (S ijoin T ujoin U);
```

Figure 4.39: Pre-processing of Syntax Trees by the Top-level Approach

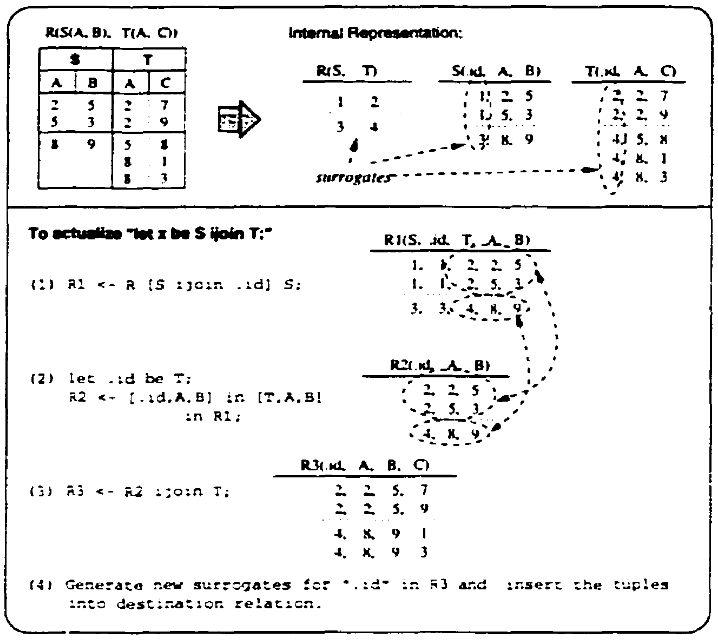
It is clear from the figure that the result of the pre-processing is that the original syntax tree is separated into a set of sub-trees corresponding to a set of intermediate virtual domains. Note that this kind of tree separation or truncation is not necessary in the tuple-by-tuple approach, since in that case, the whole syntax tree (along with tuple data) is passed to the relational processor, which is in charge of the calculation which is transparent to the virtual domain actualizer.

The subsequent implementation of horizontal and vertical operations of domain algebra in the top-level approach is based on the result of this pre-processing, i.e. only one operation at a time is involved in the actualization.

Horizontal Operations

The basic idea of the top-level approach can be summarized as "*lift the lower-level nested domain data up to the top level and join them as top-level relations*". This section describes the procedure of implementing horizontal operations, i.e. joins, selection, and projection.

Given a sample relation R defined on nested domains *S(A, B)* and *T(A, C)*, Figure 4.40 illustrates the procedure used to actualize a virtual domain x that is defined on "*S ijoin T*".



```
(1) R1 <- R [S ijoin .id] S;

(2) let .id be T;
    R2 <- [.id,A,B] in [T.A.B]
              in R1;

(3) R3 <- R2 ijoin T;

(4) Generate new surrogates for ".id" in R3 and  insert the tuples
    into destination relation.
```

Figure 4.40: Actualizing a Virtual Domain with **ijoin** by the Top-level Approach

As described in the above figure, the steps used to actualize x are as follows:

1. Extract the relevant tuple data in nested relation S.

   As described in "*Declare and Initialize Nested Relations*" of section 3.2.3 and illustrated by Figure 3.12, lower-level nested relation $.S$ is associated with top-level relation R by means of surrogates, which is indicated by the internal representation of the nested relation R in Figure 4.40. In order to extract the actual tuples in S that are connected with relation R, a natural join between top-level and lower-level relations R and S is performed, and the join attributes are the surrogates' names.

   In the jRelix implementation, this join requires a syntax tree as depicted in Figure 4.41 to be created first, and then both the syntax tree and the nested relations R and $.S$ are passed to the top-level evaluator which subsequently evaluates and passes the same information to the relational processor to fulfill the join. This is procedure (1) in Figure 4.40.

2. Prepare the next join between the resulting relation from step 1 and the nested relation T.

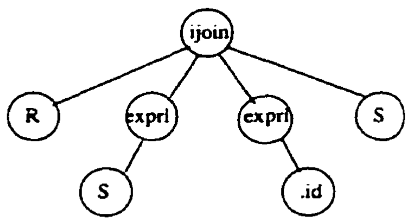


Figure 4.41: Syntax Tree for Natural Join Between *R* and *.S*

The previous join generates a relation *R1* which contains domain *T*. In order to join relation *R1* with the nested relation *T* on domain T which represents the surrogate name, the domain name *T* in *R1* must be changed to "*.id*". This is done by renaming and certain projections. A new relation is generated, i.e. relation *R2* as illustrated by step (2) in Figure 4.40.

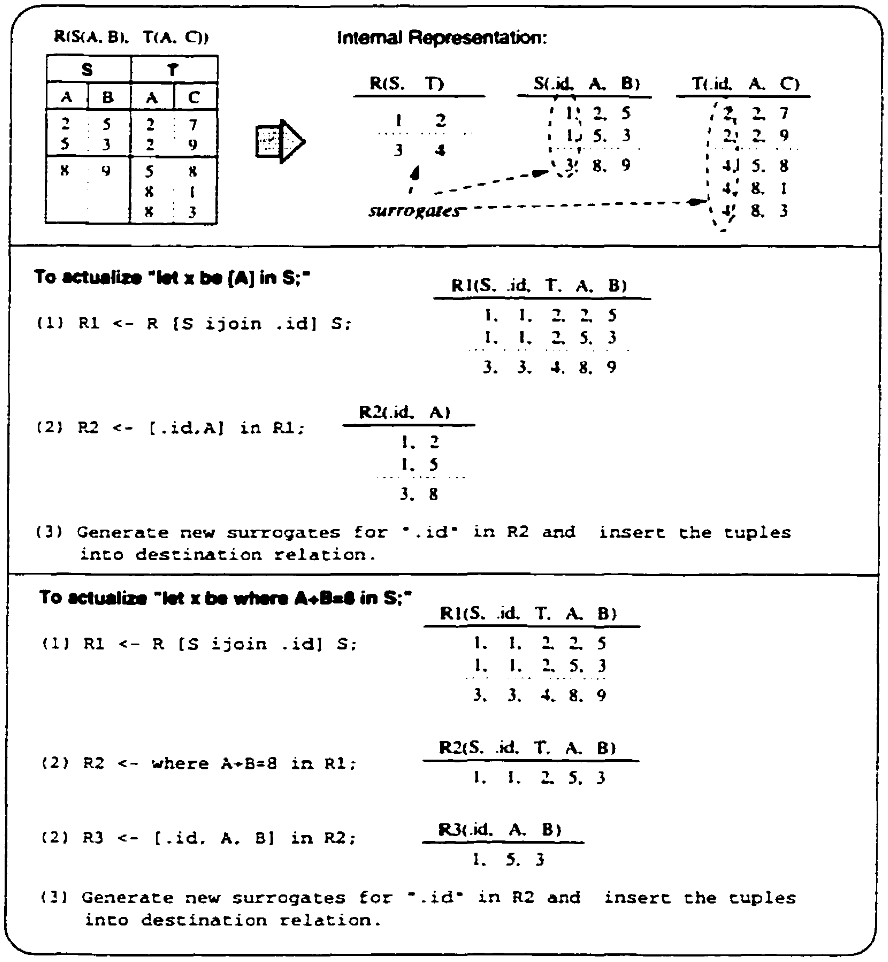
3. Join the result relation from step 2 with the nested relation *T*.

   The join is performed on the common attributes of the two source relations *R2* and *T*. This way, the tuple data of nested relation *T* is safely associated with the tuple data of nested relation *S*. The result relation *R3* is what is expected apart from the surrogate values. This is step (3) in figure 4.40.

4. Finally, change the surrogate values in the result relation from step 3, and append the new tuple data to the nested relation *x*. This concludes the actualization of virtual domain *x*, as illustrated in step (4) in Figure 4.40.

In jRelix, the above-mentioned procedure is implemented by an "*actualizeNestedJoinDom()*" method in the actualizer.

The cases of selection and projection are quite similar, or even simpler. In jRelix, they are implemented by an "*actualizeNestedPrjSelDom()*" method. Figure 4.42 illustrates the procedure to actualize virtual domains defined by projections and selections. Due to its self-describing nature, detailed explanations are omitted.



R(S(A, B), T(A, C))

S		T	
A	B	A	C
2	5	2	7
5	3	2	9
8	9	5	8
		8	1
		8	3

Internal Representation:

```
R(S.  T)        S(.id. A. B)     T(.id. A. C)
 1    2           1.  2.  5        2.  2.  7
 3    4           1.  5.  3        2.  2.  9
                  3.  8.  9        4.  5.  8
                                   4.  8.  1
                                   4.  8.  3
```

To actualize "let x be [A] in S;"

```
(1) R1 <- R [S ijoin .id] S;
```

```
R1(S. .id. T. A. B)
   1.  1.  2. 2. 5
   1.  1.  2. 5. 3
   3.  3.  4. 8. 9
```

```
(2) R2 <- [.id,A] in R1;
```

```
R2(.id. A)
    1. 2
    1. 5
    3. 8
```

```
(3) Generate new surrogates for ".id" in R2 and  insert the tuples
    into destination relation.
```

To actualize "let x be where A+B=8 in S;"

```
(1) R1 <- R [S ijoin .id] S;
```

```
R1(S. .id. T. A. B)
   1.  1.  2. 2. 5
   1.  1.  2. 5. 3
   3.  3.  4. 8. 9
```

```
(2) R2 <- where A+B=8 in R1;
```

```
R2(S. .id. T. A. B)
   1.  1.  2. 5. 3
```

```
(2) R3 <- [.id, A, B] in R2;
```

```
R3(.id. A. B)
    1.  5. 3
```

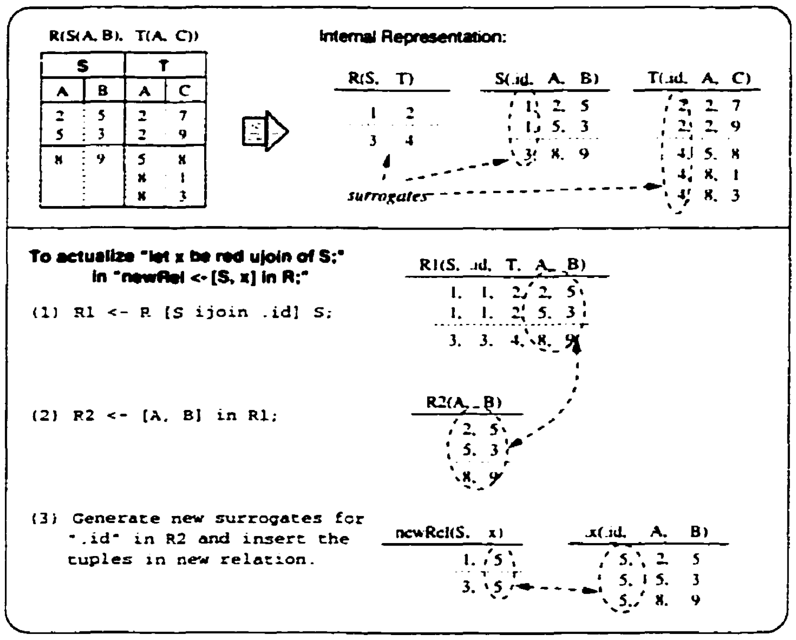
```
(3) Generate new surrogates for ".id" in R2 and  insert the tuples
    into destination relation.
```

Figure 4.42: Actualize Virtual Domains by Top-level Approach

Vertical Operations

Similar to horizontal operations, vertical operations are also implemented by "*lifting*" lower-level nested relations to an upper level, and then doing corresponding vertical operations.

Given the same sample relation R defined in Figure 4.40, Figure 4.43 illustrates the procedure to actualize a virtual domain x that is defined on "*red ujoin of S*".



To actualize "let x be red ujoin of S;" in "newRel <- [S, x] in R;"

```
(1) R1 <- R [S ijoin .id] S;

(2) R2 <- [A, B] in R1;

(3) Generate new surrogates for
    ".id" in R2 and insert the
    tuples in new relation.
```

Figure 4.43: Actualize a Virtual Domain with Reduction by Top-level Approach

As described in the figure, the steps used to actualize x are as follows:

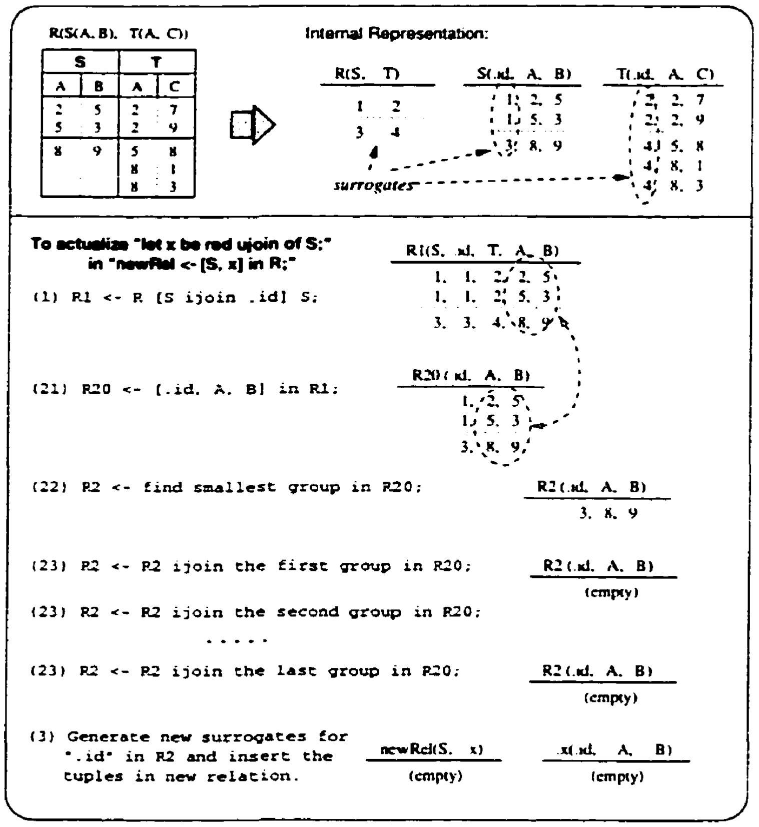
1. Extract the relevant tuple data in the nested relation S. This is exactly the same operation as in the horizontal operation described in the previous section. The result is a new relation *R1* as illustrated in step (1), Figure 4.43.

2. Project the same attributes as in nested domain S from relation *R1*. Since reduction of **ujoin** on S produces all tuples of S (associate with the top relation R without duplicate tuples), the projection is in charge of removing the duplicate tuple.

3. Finally, generate a new surrogate. This surrogate is for the result relation of ujoin, i.e. for

all the newly generated tuples in the nested domain *.x*. Needless to say, the result tuples are appended to the invisible relation *.x*. This finishes the actualization of virtual domain *x*, as in step (3) Figure 4.43.

For the reduction of **ijoin**, step 2 in the above procedure will be slightly different. As the operation *ijoin* calculates the "*minimum common set*" between two operand relations, a "*red ijoin*" operation is equivalent to calculating the minimum common set among a set of operand relations. Therefore, in the case of "*red ijoin*", step 2 in Figure 4.43 is modified and extended by the following steps, which are also illustrated in Figure 4.44.



```
(1) R1 <- R [S ijoin .id] S;
(21) R20 <- [.id, A, B] in R1;
(22) R2 <- find smallest group in R20;
(23) R2 <- R2 ijoin the first group in R20;
(23) R2 <- R2 ijoin the second group in R20;
          .....
(23) R2 <- R2 ijoin the last group in R20;
(3) Generate new surrogates for
    ".id" in R2 and insert the
    tuples in new relation.
```

Figure 4.44: Actualize a Virtual Domain with Reduction of ijoin by Top-level Approach

1. Project the same attributes as in the nested domain *S* including the *.id* domain from the relation *R1*. The result relation is assigned to *R20*. Note that the *.id* domain in relation *R20* serves to categorize different groups of tuples which will be used in the next step.

2. According to the value of domain *.id*, find the group of tuples in relation *R20* that has the minimum number of tuples, and put them in relation *R2*.

3. According to the value of domain *.id*, do a natural join between relation *R2* and the first group of tuples in relation *R20*, and overwrite the result in relation *R2*. Note that relation *R2* is now the minimum common set of the two operand relations that participated in the natural join.

4. Do the above natural join with the next group of tuples in relation *R20*. Continue until getting to the last group of tuples in relation *R20*. Note that relation *R2* is now the minimum common set of all groups of tuples in relation *R20*.

As mentioned above, the result relation (i.e. *R2*) in the above procedure is the minimum common set of operand relations involved in the natural join, i.e. the result of "vertical ijoin". This is exactly what the "*reduction of ijoin*" operation is supposed to accomplish.

Chapter 5

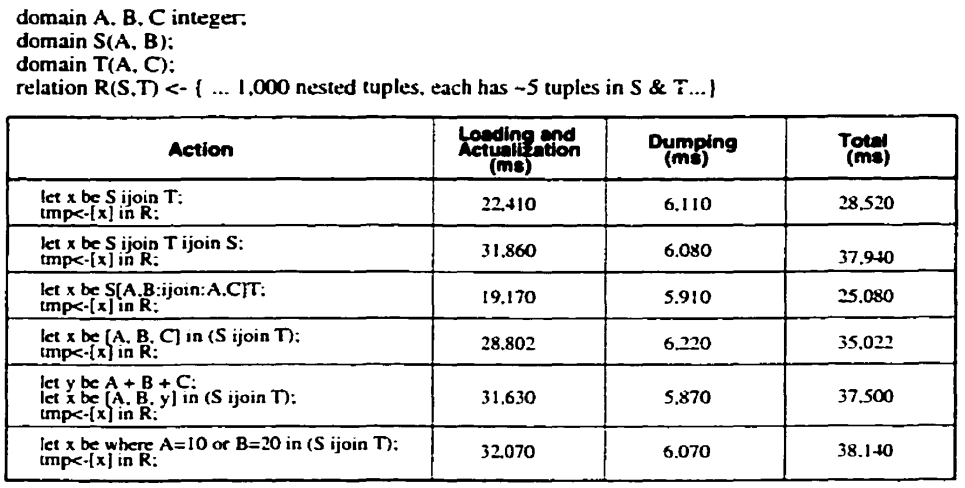
Conclusion: Results and Future Work

This chapter summarizes the result of the jRelix implementation and discusses some future work that may be done to improve the functionality of the current jRelix system. Section 5.1 focuses on the performance issue - a big concern of the current system. Testing procedures and results are described. As well, the potential of performance improvement is discussed. Section 5.2 discusses the possibility of multi-threading control in jRelix, which may hopefully result in certain performance improvements. Also discussed in this section is the potential of building a client/server jRelix system. In section 5.3, some aspects of converting the current Java application into a jRelix applet which can be displayed in a web browser such as Netscape Navigator are discussed. Some graphical user interface (GUI) samples are also presented.

5.1 Performance Issue

During the development of jRelix, system performance has been taken care of and the performance problem has been kept under control. The major method used to measure the system performance was to time the joins on nested relations. A join on nested relations theoretically consists of multiple sub-joins on the nested attributes which are also relations (or even nested relations). During the process of join operations, CPU-bound sorting as well as access to the secondary storage is intensively involved. Therefore, the speed/time of joins becomes a major performance issue. In order to trace the performance of domain algebra in jRelix, the actualization time of a nested virtual domain is regarded as the indicator of the join speed since joins on nested attributes are performed during actualization.

In order to test the speed of joins, two sample nested relations were created and used. As illustrated in Figure 5.1, relation *R* is defined on domains *S* and *T*, which are nested and are defined on 2 integer attributes (A, B) and (A, C) respectively. Relation *R* consists of 1,000 nested tuples (of domains *S* and *T*), in each tuple, both domains *S* and *T* further contain about 5 tuples on the nested level. On the other hand, as illustrated in Figure 5.2, relation *W* is defined on integer domains *A*, *B* and *C* (hence a flat relation) and contains 10,000 tuples.



```
domain A. B. C integer:
domain S(A. B):
domain T(A. C):
relation R(S.T) <- { ... 1,000 nested tuples. each has ~5 tuples in S & T...}
```

Action	Loading and Actualization (ms)	Dumping (ms)	Total (ms)
let x be S ijoin T: tmp<-[x] in R:	22,410	6,110	28,520
let x be S ijoin T ijoin S: tmp<-[x] in R:	31,860	6,080	37,940
let x be S[A.B:ijoin:A.C]T: tmp<-[x] in R:	19,170	5,910	25,080
let x be [A. B. C] in (S ijoin T): tmp<-[x] in R:	28,802	6,220	35,022
let y be A + B + C: let x be [A. B. y] in (S ijoin T): tmp<-[x] in R:	31,630	5,870	37,500
let x be where A=10 or B=20 in (S ijoin T): tmp<-[x] in R:	32,070	6,070	38,140

Figure 5.1: Actualization on Nested Relations

Figure 5.1 gives the timing results of actualizing virtual domains on relation *R*. For example, to actualize a virtual domain "*x*" defined on "*let x be S ijoin T*", 22,410 milli-seconds were used to load the relations (i.e. R, S and T) from the hard-disk into memory as well as the actualization of domain "*x*"; 6,110 milli-seconds were used to dump the resulting relation "*tmp*" onto the disk. This gives the total time consumed as 28,520 milli-seconds. Note that the dumping times are almost constant.

Figure 5.2 illustrates the timing results of actualizing virtual domains when top-level relations are involved in the virtual domain declaration. Figure 5.3 gives a comparison of actualizations between using the top-level approach and the tuple-by-tuple approach. Note that when using

```
domain A. B. C integer;
domain S(A. B);
domain T(A. C);
relation R(S.T) <- { ... 1,000 nested tuples. each has ~5 tuples in S & T...}
relation W(A.B.C) <- { ... 10,000 tuples. flat relation ... }
```

Action	Loading and Actualization (ms)	Dumping (ms)	Total (ms)

let x be S ijoin W;
tm-<-[x] in R;
	18,620	6,130	24,750

let x be S ijoin W ijoin S;
tm-<-[x] in R;
	28,450	5,680	34,130

let x be S[A,B:ijoin:A,B]W;
tm-<-[x] in R;
	17,740	5,910	23,650

let x be [A. B. C] in (S ijoin W);
tm-<-[x] in R;
	25,760	6,220	31,980

let y be A + B + C;
let x be [A. B. y] in (S ijoin W);
tm-<-[x] in R;
	25,980	6,310	32,290

let x be where A=10 or B=2 in (S ijoin W); tm-<-[x] in R;	27,290	6,510	33,800

Figure 5.2: Actualization on Nested Relation joined with Top-level Relation

```
domain A. B. C integer;
domain S(A. B);
domain T(A. C);
relation R(S.T) <- { ... 1,000 nested tuples. each has ~5 tuples in S & T...}
```

Approach / Action	Top-level (ms)			Tuple-by-Tuple (ms)		
	Loading	Actualization	CLean-up	Loading	Actualization	CLean-up

let x be S ijoin T;
tm-<-[x] in R;
	2,420	20,870	-	2,530	22,300	-

let x be S ujoin T;
tm-<-[x] in R;
	2,220	43,060	-	2,100	56,850	-

Figure 5.3: Comparison of Different Approaches

the tuple-by-tuple approach, the actualization time is only a little longer than when using the top-level approach.

Noticed that during join time measurements, the sort procedure involved in the join operations consumes a large portion of the actualization time. Sorting is an intensive memory-bound process; if the basic sort routine that resides in the Relation Algebra could be given more fine-tuning, the performance of joins is expected to be enhanced significantly. The same is true with the secondary storage access problem. In the current implementation, intermediate relations are frequently saved to, and retrieved from the hard-disk. Should a mechanism be developed that utilizes the capability of memory instead of secondary storage for join operations, the performance of jRelix will also be improved remarkably.

5.2 Multi-Threading and Client-Server Model

The current jRelix implementation did not take advantage of the multi-threading construction provided by the Java programming language. Multi-threading mechanism can be introduced in future jRelix designs and implementations in order to improve performance and functionality.

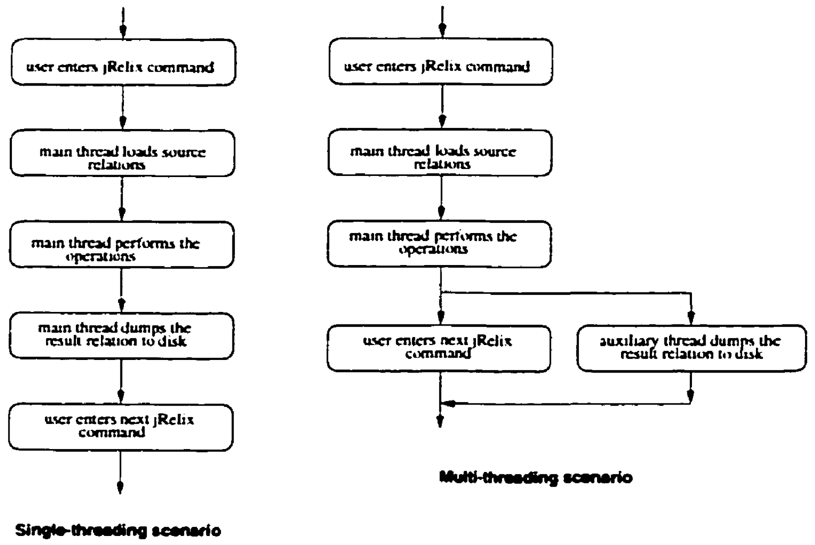

user enters jRelix command
main thread loads source relations
main thread performs the operations
main thread dumps the result relation to disk
user enters next jRelix command
Single-threading scenario
user enters jRelix command
main thread loads source relations
main thread performs the operations
user enters next jRelix command
auxiliary thread dumps the result relation to disk
Multi-threading scenario

Figure 5.4: Multi-threading in jRelix

For example, when performing relational or domain algebra operations, apart from the main operational thread which handles the major task, an additional auxiliary thread can always be used to dump the results onto the secondary storage after the main thread finishes the major task. Why use an auxiliary thread specifically? This is because when a task is entered by an end-user, the main thread performs the operation and returns control (i.e. the system prompt) to the end-user when the task is finished. In this scenario, the response time includes the time used for dumping the result relations to the hard-disk. By using an auxiliary thread to dump the result, the main thread can return control to the end-user without having to wait until the results are saved to the disk. As displayed in Figure 5.1 or Figure 5.2, the dumping time is usually several seconds; while the time consumed by the end-user to type the next jRelix command is also that long (or longer). This gives the "*dumping*" thread enough time to finish its job silently in the background without attracting the end-user's attention. Figure 5.4 illustrates this improvement.

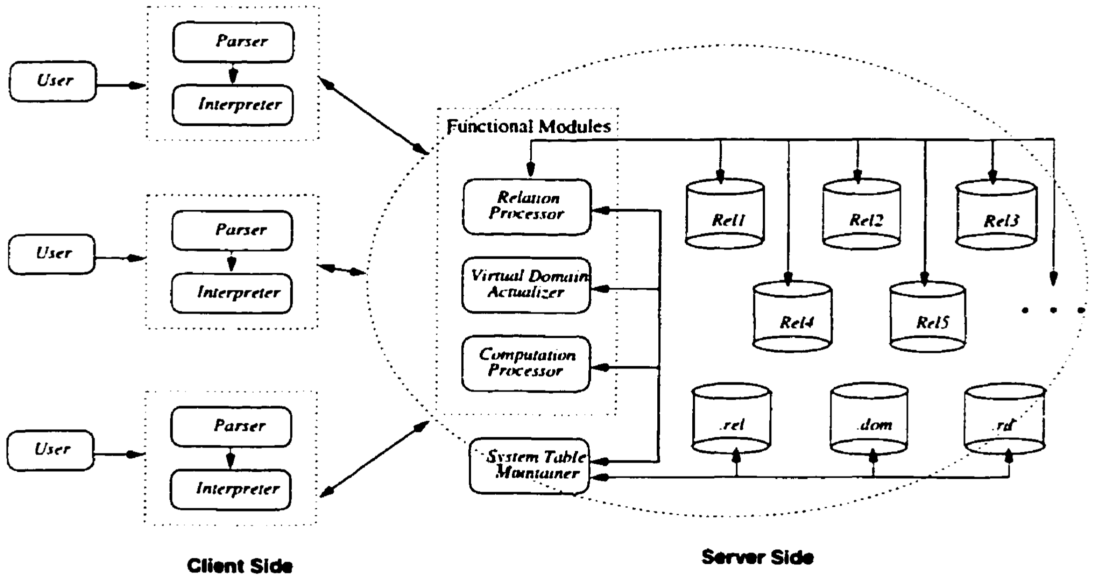
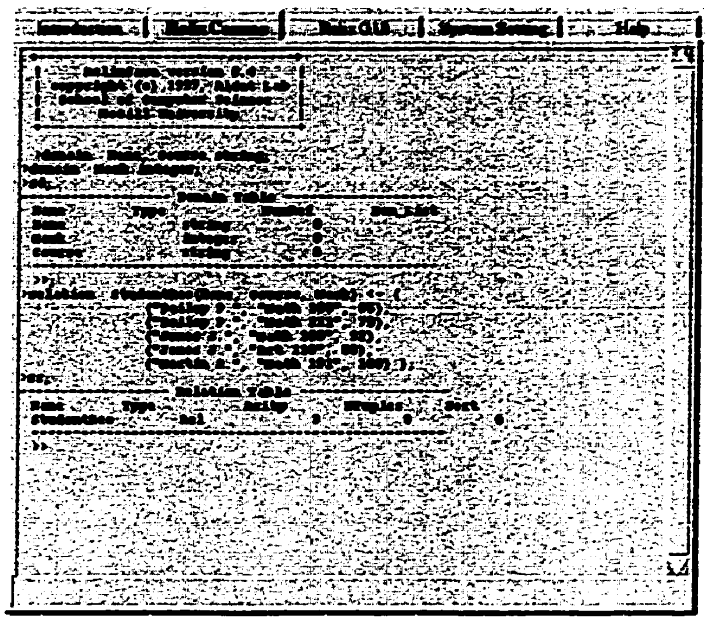


Figure 5.5: A Client/Server Model in jRelix

On the other hand, a client/server jRelix can also be built in future work in order to release a certain amount of work from the server system to its clients, especially when the jRelix functionality becomes more complex or the central databases become much larger. A client/server model implementation might involve a redesign of the current jRelix architecture, which is beyond the scope of this thesis. The basic idea in this regard is to move the front-end interpreter etc. to the client side. The server would only contain the service logic i.e. the logic providing the operations of relation algebra, domain algebra and computations, as illustrated in Figure 5.5.

5.3 Migration to Internet: Applet and GUI

The current jRelix system is a Java application program which runs on the command line of the operating system. It is easy to convert a graphical Java application into an applet which can be embedded in a web page by following some general steps listed below:



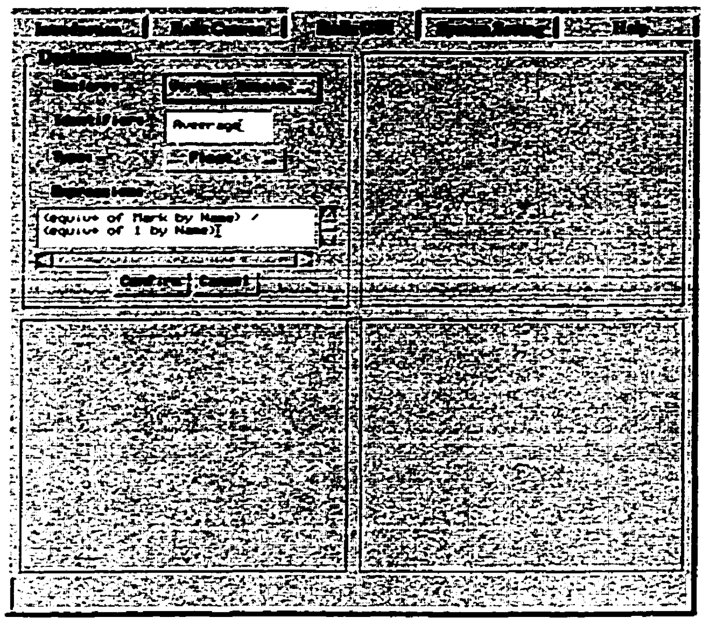
```
  relixgui version [illegible]
  copyright (c) 1999, Aldat Lab
  School of Computer Science
  McGill University

>domain Name, Course string;
>domain Mark integer;
>dd;
                 Domain Table
 Name        Type       RefCnt      [illegible]
 Name        string     [illegible]
 Mark        integer    [illegible]
 Course      string     [illegible]

>>
>relation StudentMarks(Name, Course, Mark) <- {
             [illegible]
             [illegible]
             [illegible]
             [illegible]
             [illegible] };
>dr;
                 Relation Table
 Name          Type     Arity     NTuples     Sort
 StudentMarks  rel      3         [illegible] [illegible]

>>
```

Figure 5.6: Sample GUI Design for jRelix Applet

1. Make an HTML page with an APPLET tag.
2. Derive the starting class of the application (`Interpreter` class in our case) from the `Applet` class provided by the Java Development Kit.
3. Eliminate the *main()* method in the starting class, and move the major functioning code into a method called `init()`. When the browser creates an object of the applet class, it calls the init() method.

4. Create a proper layout manager to organize the graphical components of the applet.

As the current jRelix system is not a graphical application, the graphical layout needs to be designed for the applet version of jRelix. An example of the design is illustrated in Figure 5.6.



```
(equive of Mark by Name) /
(equive of 1 by Name)
```

Figure 5.7: New GUI Design for jRelix Applet (Incomplete)

In addition, a completely new version of the user interface dealing with all operations and functionalities of jRelix can be developed, from which the user can perform jRelix operations by interacting with certain graphical components of the applet. For instance, during declaration, the end-user can inform the system what he/she wants to declare (e.g. domain, relation etc.) by selecting the items from a list box. He/she then types the identifier for the new declaration and chooses the type of the identifier (e.g. integer, boolean or relation etc.) from another list box. Expressions for the new declaration can be entered in a text area. When the user clicks the "*Confirm*" button, the system checks the validity of the declaration and accepts the new declaration if everything is OK. Figure 5.7 illustrates the implementation of such an idea (incomplete).

Finally, an auxiliary user interface can be designed to support the main working applet illustrated in Figures 5.6 and 5.7. Figure 5.8 gives an example of a user interface used to

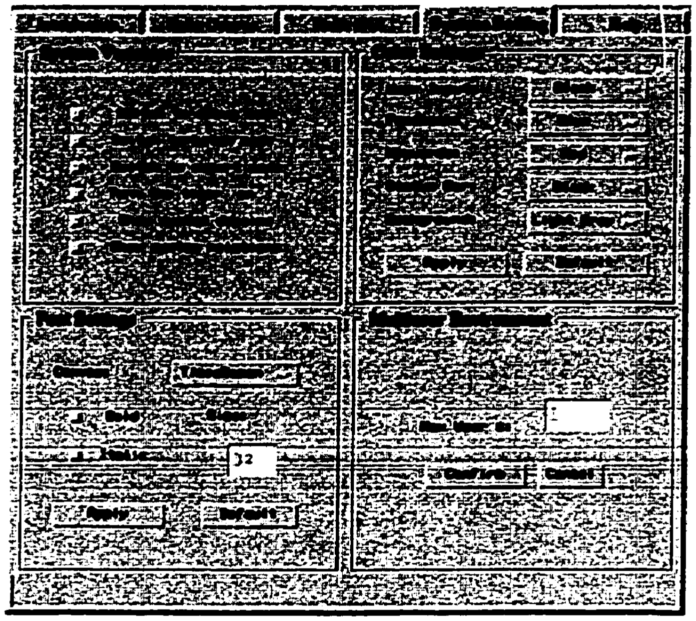
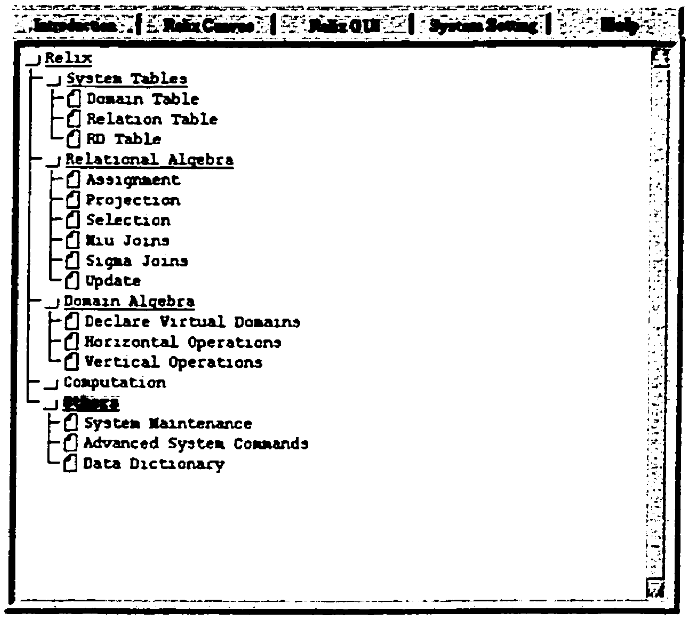


Figure 5.8: GUI Design for jRelix System Environment Settings



```
Relix
├ System Tables
│ ├ Domain Table
│ ├ Relation Table
│ └ RD Table
├ Relational Algebra
│ ├ Assignment
│ ├ Projection
│ ├ Selection
│ ├ Miu Joins
│ ├ Sigma Joins
│ └ Update
├ Domain Algebra
│ ├ Declare Virtual Domains
│ ├ Horizontal Operations
│ └ Vertical Operations
├ Computation
└ Others
  ├ System Maintenance
  ├ Advanced System Commands
  └ Data Dictionary
```

Figure 5.9: jRelix Help Information

manage the system environment; the system variables, the text display's color and font settings etc. are managed. Figure 5.9 illustrates a user interface that provides end-users with help information; where definitions of various concepts involved in the jRelix system (e.g. relation algebra and domain algebra etc.) as well as the usage information of the jRelix applets are provided in detail.

Appendix A

Backus-Naur Form for jRelix Commands

This appendix describes jRelix grammar/syntax in the Backus-Naur Form (BNF) format. The convention of this BNF definition is explained in table A.1.

Form	Meaning
<SYMBOL>	SYMBOL is a definition of token and must be substituted
"SYMBOL"	SYMBOL is reserved word or symbol and must be typed as it is
S1 | S2	either S1 or S2 can be used
(SYMBOL)?	SYMBOL is optional
(SYMBOL)*	SYMBOL may appear zero or more times
(SYMBOLS)	grouping SYMBOLS as one unit for high precedence

Table A.1: BNF convention.

The grammar is created from the grammar specification (in file Parser.jjt), using the JavaCC documentation generator called `jjdoc`. Because JavaCC is a top-down parser, left-recursion is not allowed in the grammar specification. Therefore the grammar looks different from that of the former Relix which is intended for the bottom-up parser generator Yacc.

There are five token definitions: <EOF> for end-of-file; <IDENTIFIER> for identifier; <INTEGER_LITERAL> for integer constants; <FLOAT_LITERAL> for floating constants; and <STRING_LITERAL> for string constants.

```
Start := Command ";" | Statement ";" | ";" | <EOF>

Command := "help" (<IDENTIFIER>)?
    | "quit" | "input" FilePath | "debug" | "batch" | "expert"
    | "time" | "deld" IDList | "delr" IDList | "pr" Expression
    | "sd" (<IDENTIFIER>)? | "sr" (<IDENTIFIER>)? | "srd"
    | "ssd" | "ssr" | "print" <STRING_LITERAL>
```

```
Statement := SequentialStatement

SequentialStatement := ParallelStatement ("--" ParallelStatement)*

ParallelStatement := ChoiceStatement ("||" ChoiceStatement)*

ChoiceStatement := PrimaryStatement ("??" PrimaryStatement)*

PrimaryStatement := Declaration | Assignment | Update
    | ComputationCall | Conditional | ForLoop | WhileLoop
    | Exit | DeadLock | Exec | StatementBlock

StatementBlock := "{" Statement (";" Statement)* (";")? "}"

Conditional := "if" Expression "then" Statement ("else" Statement)?

ForLoop := ("for" Identifier)? ("from" Expression)?
           ("to" Expression)? ("by" Expression)?
           ("do" | "loop") Statement

WhileLoop := "while" Expression ("do" | "loop") Statement

Exit := "exit"

DeadLock := "deadlock"

Exec := "exec" Identifier

Declaration := "relation" IDList "(" IDList ")" (Initialization)?
    | Identifier ("initial" Expression)? "is" Expression
      ("target" Expression)?
    | "domain" IDList Type
    | "let" Identifier ("initial" Expression)? "be" Expression
    | ("computation" | "comp") Identifier
      "(" (ParameterList)? ")" "is" ComputationBody

Initialization := "<-" ("{" ConstantTupleList "}" | Identifier)

ConstantTupleList := ConstantTuple ("," ConstantTuple)*

ConstantTuple := "(" Constant ("," Constant)* ")"

Constant := Literal | "{" ConstantTupleList "}"
```

```
Identifier := <IDENTIFIER>

FilePath := <STRING_LITERAL>

Assignment := Identifier
    ( ("<-" | "<+") Expression
      | "[" IDList ("<-" | "<+") ExpressionList "]" Expression
    )

Update := "update" Identifier
    ( ("add" | "delete") Expression
      | "change" (StatementList)? (UsingClause)?
      | "[" IDList ("add" | "delete") ExpressionList "]" Expression
    )

StatementList := Statement ("," Statement)*

UsingClause := "using"
    ( JoinOperator Expression
      |
      "[" ExpressionList ":" JoinOperator (":")?
      ExpressionList "]" Expression
    )

IDList := Identifier ("," Identifier)*

ExpressionList := Expression ("," Expression)*

Expression := Disjunction

Disjunction := Conjunction (("or" | "|") Conjunction)*

Conjunction := Comparison (("and" | "&") Comparison)*

Comparison := Concatenation (ComparativeOperator Concatenation)?

Concatenation := MinMax ("cat" MinMax)*

MinMax := Summation (("min" | "max") Summation)*

Summation := JoinExpression (("+" | "-") JoinExpression)*

JoinExpression := Projection
    ( JoinOperator Projection
       | "[" ExpressionList ":" JoinOperator (":")?
       ExpressionList "]" Projection
      )*

Projection := Projector (("in" | "from") Projection | Selection) | Selection

Projector := (QuantifierOperator)? "[" (ExpressionList)? "]"

Selection := Selector | QSelector | Term

Selector := ("where" | "when") Expression ("in" | "from") Projection
    | "edit" (Projection)? | "zorder" Projection

QSelector := "quant" QuantifierList (("where" | "when") Expression)?
             ("in" | "from") Projection

QuantifierOperator := "." | "%" | "#"

QuantifierList := Quantifier ("," Quantifier)*

Quantifier := "(" Expression ")" Expression

Term := Factor (("*" | "/" | "mod") Factor)*

Factor := ("+" | "-" | "not" | "!") Factor | Power

Power := Primary ("**" Power)*

Primary := Literal | QuantifierOperator | ArrayElement
    | PositionalRename | Identifier | Cast | "(" Expression ")"
    | Pick | Eval | Function | IfThenElseExpression | VerticalExpression

ArrayElement := Identifier "[" ArrayIndexList "]"

ArrayIndexList := (Expression)? ("," (Expression)?)*

PositionalRename := Identifier "(" (IDList)? ")"

Cast := "(" Type ")" Primary

Pick := "pick" Selection

Eval := "eval" Expression
```

```
Function := FunctionOperator "(" Expression ")"

Literal := "null" | "dc" | "dk" | "true" | "false"
    | ("+" | "-")? (<INTEGER_LITERAL> | <FLOAT_LITERAL>)
    | <STRING_LITERAL>

IfThenElseExpression := "if" Expression "then" Expression
                        "else" Expression

VerticalExpression := "red" AssoCommuOperator "of" Expression
    | "equiv" AssoCommuOperator "of" Expression
      "by" ExpressionList
    | "fun" OrderedOperator "of" Expression
      "order" ExpressionList
    | "par" OrderedOperator "of" Expression
      ( "order" ExpressionList "by" ExpressionList
        | "by" ExpressionList "order" ExpressionList
      )

Type := ("boolean" | "bool") | "short"
    | ("integer" | "intg") | "long"
    | ("float" | "real") | "double"
    | ("string" | "strg") | "text"
    | ("statement" | "stmt")
    | ("expression" | "expr")
    | ("computation" | "comp") "(" IDList ")"
    | "(" IDList ")"

AssoCommuOperator := ("or" | "|")
    | ("and" | "&") | "min" | "max" | "+" | "*"
    | ("ijoin" | "natjoin") | "ujoin" | "sjoin"

OrderedOperator := AssoCommuOperator
    | "cat" | "-" | "/" | "mod" | "**" | "pred" | "succ"

ComparativeOperator := "substr" | "=" | "!=" | ">" | "<" | ">=" | "<="

JoinOperator := "nop" | MuJoin
    | (("not" | "!"))? SigmaJoin

MuJoin := ("ijoin" | "natjoin")
    | "ujoin" | "sjoin" | "ljoin" | "rjoin"
    | ("dljoin" | "djoin") | "drjoin"
```

```
SigmaJoin := ("icomp" | "natcomp") | "eqjoin"
    | ("gejoin" | "sup" | "div") | "ltjoin"
    | ("lejoin" | "sub") | ("iejoin" | "sep")

FunctionOperator := "abs"
    | "sqrt" | "sin" | "asin" | "cos" | "acos" | "tan"
    | "atan" | "sinh" | "cosh" | "tanh" | "log" | "log10"
    | "round" | "ceil" | "floor" | "isknown" | "chr" | "ord"

ParameterList := Parameter ("," Parameter)*

Parameter := <IDENTIFIER> (":" "seq")?

ComputationBody := ComputationDeclarationAndInitialization
                   ComputationBlock ("alt" ComputationBlock)*

ComputationBlock := "{" ComputationStatements "}"

ComputationDeclarationAndInitialization :=
    ( LocalVariableDeclaration
      | StateVariableDeclaration
      | ComputationInitialization
    )*

LocalVariableDeclaration := "local" IDList Type
                            (VariableInitialization)? ";"

StateVariableDeclaration := "state" IDList Type
                            (VariableInitialization)? ";"

ComputationInitialization := IDList "<-" Expression ";"

VariableInitialization := "<-" Expression

ComputationStatements := Statement (";" Statement |
                         "also" Statement)* (";")?

ComputationCall := Identifier "(" (CallParameterList)? ")"

CallParameterList := CallParameter ("," CallParameter)*

CallParameter := ("in" | "out") <IDENTIFIER>
```

Bibliography

[AB84] S. Abiteboul and N. Bidoit. Non first normal form relations to represent hierarchically organized data. In *Proceedings of 2nd ACMSIGACT/SIGMOD Symposium on Principles of Database Systems*, pages 191–200, 1984.

[AG96] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[AMM83] H. Arisawa, K. Moriya, and T. Miura. Operations and the propertie: on non-first-normal-form relational databases. In *Proceedings of the 9th International Conference on Very Large Data Bases*, pages 197–204, 1983.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Bak98] Patrick Baker. *Java Implementation of Computations in a Database Programming Language*. 1998. Master's thesis, McGill University.

[BK86] F. Bancilhon and S. Khoshafian. A calculus for complex objects. In *Proceedings 5th of ACM SIGACT-SIGMOD Symposium on Principles of Database System*, pages 53–59, 1986.

[BNR+87] C. Beeri, S. Naqvi, R. Ramakrishnan, O. Shmueli, and S. Tsur. Sets and negation in logic database language (ldl1). In *Proceedings 6th PODS, San Diego*, pages 21–37, 1987.

[BRS82] F. Bancilhon, P. Richard, and M. Scholl. On line processing of compacted relations. In *Proceedings 8th VLDB*, pages 263–269, 1982.

[CG86] S. Ceri and G. Gottlob. Normalization of relations and prolog. *Communications of ACM*, 29(6):524–544, 1986.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[Cod71] E. F. Codd. A data base sublanguage founded on the relational calculus. In *Proceedings of ACM SIGFIDET Workshop on Data Description, Access and Control*, Los Angeles, 1971.

[Cod72a] E. F. Codd. *Database Systems: Further Normalization of the Data Base Relational Model.* Prentice-Hall, Edited by R. Rustin, 1972.

[Cod72b] E. F. Codd. *Database Systems: Relational Completeness of Data Base Sublanguages.* Prentice-Hall, Edited by R. Rustin, 1972.

[Dat81] C. J. Date. *An Introduction to Database Systems, 3rd Edition.* Addison-Wesley, Reading, MA, 1981.

[DG88] A. Deshpande and D. Van Gucht. An implementation of nested relational database. In *Proceedings of the 14th International Conference on Very Large Data Bases*, pages 266–274, 1988.

[DKA+86] P. Dadam, K. Kuespert, F. Andersen, H. Blanken, R. Erbe, J. Guenauer, V. Lum, P. Pistor, and G. Walch. *A DBMS Prototype to Support Extended NF2 Relations: An Integrated View on Flat Tables and Hierarchies.* 1986.

[DNS91] D. DeWitt, J. Naughton, and D. Schneider. An evaluation of non-equijoin algorithms. In *vldb*, pages 443–452, 1991.

[Dou91] Samir Douik. *Implementation of Delay and Nondeterminism in a Database Programming Language.* Montreal, 1991.

[DPS86] U. Deppisch, H. B. Paul, and H. J. Schek. A storage system for complex objects. In *Proceedings of the International Workshop on Object-Oriented Database System*, pages 183–195, 1986.

[FG85] P. C. Fisher and D. Van Gucht. Determining when a structure is a nested relation. In *Proceedings of the 11th International Conference on Very Large Data Bases*, pages 171–180, 1985.

[FT83] P. C. Fisher and S. J. Thomas. Operations on non-first-normal-form relations. In *Proceeding of IEEE COMPSAC '83*, pages 464–475, 1983.

[Ger75] R. Gerritsen. *The Relational and Network Models of Databases: Bridging the Gap, Second U.S.A.* Japan Computer Conference, 1975.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[Gos96] J. Gosling. *Java Programming Language.* SunSoft Press, 1996.

[Gra85] P. Gray. *Logic, Algebra and Databases.* Ellis Horwood Limited, West Sussex, 1985.

[Hao98] Biao Hao. *Implementation of The Nested Relational Algebra in Java.* 1998. Master's thesis, McGill University.

[He97] Hongbo He. *Implementation of Nested Relations in a Database Programming Language.* Montreal, 1997.

[HP87] G. Houben and J. Paredaens. The r-algebra: An extension of an algebra for nested relations. *Technical Report*, 1987.

[ICRR81] Thompson III, William C., Ries, and Daniel R. A multiprocessor sort-merge join algorithm for relational data bases. *rcvd*, February 1981.

[JS82] G. Jaeschke and H. J. Schek. Remarks on the algebra of non-first-normal-form relations. In *Proceedings of the First ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 124–138, Los Angeles, 1982.

[Ken83] W. Kent. A simple guide to five normal forms in relational database theory. *Communications of ACM*, 26(2):120–125, 1983.

[KK89] H. Kitagawa and T. L. Kunii. *The Unnormalized Relational Data Model for Office Form Processor Design*. Springer-Verlag, Tokyo, Japan, 1989.

[KO90] Masaru Kitsuregawa and Yasushi Ogawa. Bucket spreading parallel hash: A new, robust, parallel hash join method for data skew in the super database computer (SDC). In *vldb*, pages 210–221, 1990.

[KR89] H. F. Korth and M. A. Roth. Query languages for nested relational databases. In *Nested Relations and Complex Objects in Databases, Lecture Notes in Computer Science 361*. Springer-Verlag, Berlin, 1989.

[KS86] H. F. Korth and A. Silberschatz. *Database System Concepts*. McGraw-hill Book Company, New York, 1986.

[Lal86] N. Laliberté. *Design and Implementation of a Primary Memory Version of Aldat*. Montreal, 1986.

[LS88] R. Lorie and H. J. Schek. On dynamically defined objects and SQL. In *Proceedings of 2nd Workshop on Object-Oriented Database Systems*, Bad Münster, 1988.

[LT95] Hongjun Lu and Kian-Lee Tan. On sort-merge algorithm for band joins. *IEEE TKDE*, 7(3):508–510, June 1995.

[Mak77] A. Makinouchi. A consideration on normal form of not-necessarily-normalized relation in the relational data model. In *Proceedings of 3rd International Conference on Very Large Data Bases*, pages 447–453, Tokyo, 1977.

[Mer77] T. H. Merrett. Relations as programming language elements. *Information Processing Letters*, 6(1):29–33, 1977.

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

[MRS88] H. Korth M. Roth and A. Silberschatz. Extended algebra and calculus for nested relational database. *ACM Transactions on Database Systems*, 13(4):389–417, 1988.

[OH86] S. Osborne and T. E. Heaven. The design of a relational database system with abstract data types for domains. *ACM Trans. on Database Systems*, 11(3):357–73, Sept. 1986.

[OY87] Z. M. Ozsoyoglu and L. Y. Yuan. A new normal form for nested relations. *ACM Transactions on Database Systems*, 12(1):111–136, 1987.

[PA86] P. Pistor and F. Anderson. Designing a generalized NF^2 model with an SQL-type language interface. In *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 278–285, 1986.

[PT86] P. Pistor and R. Traunmueller. A database language for sets, lists and tables. *Information Systems*, 11(4):323–336, 1986.

[RS95] Nick Ryan and Dan Smith. *Database Systems Engineering*. International Thomson Computer Press, Boston, MA, 1995.

[SAB$^+$89] M. H. Scholl, S. Abiteboul, F. Bancilhon, N. Bidoit, S. Gamerman, D. Plateau, P. Richard, and A. Verroust. VERSO: A database machine based on nested relations. In *Nested Relations and Complex Objects in Databases, Lecture Notes in Computer Science 361*. Springer-Verlag, Berlin, 1989.

[Sal86] B. J. Salzberg. Third normal form made easy. *SIGMOD Record*, 15(4):2–17, 1986.

[SC90] Eugene J. Shekita and Michael J. Carey. A performance evaluation of pointer-based joins. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 19(2):300–311, June 1990.

[SDV96] Sriram Sankar, Rob Duncan, and Sreenivasa Viswanadha. Java compiler compiler (javacc)-the java parser generator. JavaCC web site at: http://www.suntest.com/JavaCC/, 1996. the web site contains documentations, FAQs, newsgroups, and softwares for JavaCC and JJTree.

[SM94] D. K. Shin and A. C. Meltzer. A new join algorithm. *SIGMOD Record*, 23(4):13–18, December 1994.

[SP82] H. Schek and P. Pistor. Data structures for an integrated data base management and information retrieval system. In *Proceedings of 8th International Conference on Very Large Data Bases*, pages 197–207, 1982.

[SPS87] M. H. Scholl, H. B. Paul, and H. J. Schek. Supporting flat relations by a nested relational kernel. In *Proceedings 13th VLDB, London*, 1987.

[SS86] H. Schek and M. Scholl. The relational model with relation-valued attributes. *Information Systems*, 11(2):137–147, 1986.

[Tak89] K. Takeda. On the uniqueness of nested relations. In *Nested Relations and Complex Objects in Databases, Lecture Notes in Computer Science 361*. Springer-Verlag, Berlin, 1989.

[TF86] S. J. Thomas and P. C. Fischer. Nested relational structures. In P. C. Kanellakis, editor, *Advances in Computing Research III, The Theory of Databases*, pages 269–307. JAI Press, 1986.

[TPB87] Vinay K. Chaudhri Tapan P. Bagchi. *Interactive Relational Database Design: A Logic Programming Implementation.* Lecture Notes in Computer Science 402. Springer-Verlag, Berlin, 1987.

[Ull82] J. D. Ullman. *Principles of Database Systems, 2nd Edition.* Computer Science Press, Rockville, MD, 1982.

[Yan86] C. C. Yang. *Relational Databases.* Prentice-Hall, Englewood Cliffs, NJ, 1986.

[Yao85] S. Bing Yao. *Principles of Database Design, Vol. 1.* Prentice-Hall, Englewood Cliffs, NJ, 1985.

[ZJM94] X. Zhao, R. G. Johnson, and N. J. Martin. Dbj — a dynamic balancing hash join algorithm in multiprocessor database systems. *Information Systems*, 19(1):89–100, 1994.