A RULE-BASED MEDIATOR IMPLEMENTATION FOR SOLVING SEMANTIC CONFLICTS IN SQL

by

Francisco J. Madero

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL
ENGINEERING AND COMPUTER SCIENCE IN
PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

BACHELOR OF SCIENCE IN ELECTRICAL SCIENCE AND ENGINEERING

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

October 1992

Signature of Author ____________________________
Department of Electrical Engineering and Computer Science
October 9, 1992

Certified by ____________________________
Dr. Michael Siegel
Thesis Supervisor

Accepted by ____________________________
Professor Leonard A. Gould
Chairman, Department Committee on Undergraduate Theses

A RULE-BASED MEDIATOR IMPLEMENTATION FOR SOLVING SEMANTIC CONFLICTS IN SQL

by

Francisco J. Madero

Submitted to the Department of Electrical Engineering and Computer Science on
October 9, 1992 in partial fulfillment of the requirements for the degree of
Bachelor of Science in Electrical Science and Engineering.

Abstract

The implementation of a rule-based algorithm for avoiding semantic conflicts in SQL is discussed. It allows owners of databases and users of application programs that query those databases to define the semantic context of the data that they respectively export or import. They do so by writing a set of rules from which the value of the meta-data that modify the non-primitive attributes can be derived. A *Subsumption Algorithm*, compares the application and database sets of rules, and creates a set of tables that store information regarding when the rules conflict or match. Then, a *Query Processing Algorithm* uses the tables created by the Subsumption Algorithm to make sure that a query made by the application to the database won't return results in the wrong context.

Thesis Supervisor:	Dr. Michael Siegel
Title:	Research Associate, Sloan School of Management

Table of Contents

PART I

INTRODUCTION

Distributed computer systems have become very popular in recent times. The invention of the personal computer, along with advances in networking technology and operating systems have produced a dramatic increase in the number of computer networks. These new sets of interconnected computers provide their users with many novel ways of work and communication. Among the most important of these is the possibility of sharing huge amounts of data. By connecting his or her computer to a network, any user can -if given the necessary permissions- access data from many different sources. This increased ability to share data is already significantly influencing many organizations.

There is, however, a problem that arises when one tries to use data created and maintained by other persons: one might not know very well what that data means. For example, if someone finds out from a database in a network that the price for certain product is 100, that person might not be sure what 100 means. Are they US dollars or deutsche marks? Are taxes included? Is it today's or an obsolete price?, etc. The problem arises because there is a significant amount of information about the price, beyond its magnitude of 100, that is unknown by the user. This information is called the **context** of the price, and it is indispensable to understand what the price means. When someone misunderstands some data because of lack of knowledge of its context, a **semantic conflict** occurs.

In general, semantic conflicts arise because the person who creates and maintains a database (in general its owner) always knows, but does not makes explicit the context of the data that he or she stores. In the time before computer networks facilitated the sharing of data, semantic conflicts were not very important because most of the time people only used their own databases. But today, when anyone can access hundreds of data sources, each with different context specifications which are not made explicit, semantic conflicts are widespread. There is a lot of interest in government and industry in finding ways to avoid these misunderstandings. In particular, there is some research being done at MIT and other places to find ways in which semantic conflicts can be detected automatically by a computer, saving its user the effort of checking that every piece of information that he or she imports has the correct context.

This thesis presents the implementation of a source-receiver system, in which an application retrieves data from database without incurring in semantic conflicts. This is achieved using a **context mediation** method proposed by Madnick and Siegel [SM'91], in which both the application and the database make their context explicit, and all possible semantic conflicts are detected before the application queries the database. The conflicts are found by executing a *Subsumption Algorithm* that compares the application's and the database's context definitions. Figure 1 shows a simple sketch of the elements of the system.

At its highest level, the system consists of an *application* and a *database*. The application can be any program that retrieves information from the database; and the database can have any desired schema. For example, assume that the database is a financial database whose schema contains the following four attributes:

Company_Name, Exchange, Instrument_Type and Trade_Price

A sample tuple from this database could contain the values 'IBM', 'nyse', 'equity', and 30; meaning that IBM's equities were traded in the New York Stock Exchange at a price of 30.

The next level of the system (again shown in Figure 1) consists of *Context Rules* for both the application and the database. These are two sets of instructions for determining the context (meaning) of the attributes whose meaning can be confusing. For example, in the sample database only the value of the Trade_Price attribute is ambiguous: What does a

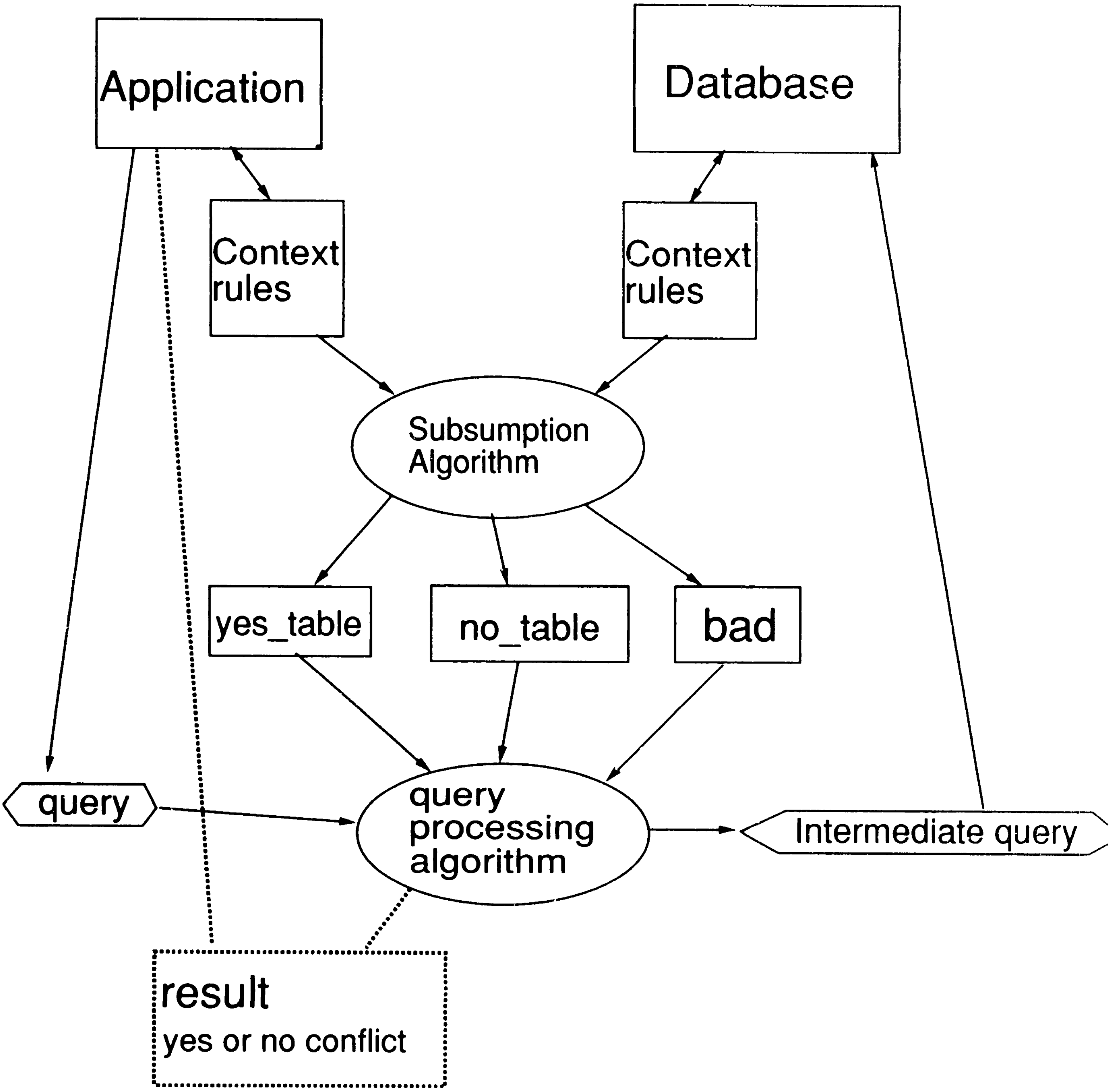

Figure 1 Simple System Diagram

price of 30 means? Today's average price in US dollars? Yesterday's closing price in Deutsche marks?

To write a rule about an attribute, a set of *meta-attributes* is defined. Their values determine the context of the attribute. For example, assume that the context of Trade_Price is determined by the value of the meta-attributes Trade_Price_Currency, and Trade_Price_Status. If someone knows that for a certain Trade_Price, its Trade_Price_Currency is 'USdollars', and its Trade_Price_Status is 'latest_closing_price', then that person knows without ambiguity what the price means: The price is the instrument's price in US dollars when the trading closed for the last time. Suppose that in the sample database, the application's context rules contain only one rule:

i) If Exchange = 'nyse' then Currency = USdollars.

meaning that the application will assume that Currency of the Trade_Price of everything traded in New York is in dollars. Also, suppose that the database's context rules contain the following rule:

i) If Instrument_Type = 'equity' then Currency = 'pesetas'.

meaning that in the database, the Trade_Price of all equities is in Spanish pesetas.

The third level of the system in Figure 1 is a *Subsumption Algorithm* that compares the application's and database's context rules and looks for conflicts between them. (The Subsumption Algorithm stores its results in three tables called the yes_table, no_table, and bad table, shown in Figure 1. They are explained in detail in the next section.) For example, given the previous two context rules, the subsumption finds that there would be a semantic conflict if the application tried to get a Trade_Price from the database where Exchange = 'nyse' and Instrument_Type = 'equity'. The conflict would happen because the application would be expecting a Trade_Price in USdollars, but the database would provide a Trade_Price in pesetas.

The last level of the system in Figure 1 is a *Query Processing Algorithm* that makes sure that no query made by the application to the database will return results in the wrong semantic context, before the query is made. To do this, the query processing algorithm uses the results of the subsumption algorithm. For example, if the application wants to make the following query to the database:

select Trade_Price
where Exchange = 'madrid'

the query processing algorithm check that the query won't return any semantically incorrect value, before allowing the application to perform the query. In this case, the query processing algorithm finds out that the query can be performed because semantic conflicts can only occur when Exchange = 'nyse' and Instrument_Type = 'equity', but not when Exchange = 'madrid'. Therefore, the Query Processing Algorithm allows the application to perform the query on the database. On the other hand, if the application wants to make the following query:

select Trade_Price where Exchange = 'nyse'

then the query processor discovers that there could be a semantic conflict if the database contained a tuple with Exchange = 'nyse' and Instrument_Type = 'equity'. Therefore, before allowing the application to perform the query on the database, the query processing algorithm makes sure that the database does not contain any tuple that satisfies those conditions. It does so by performing one or more *intermediate queries*, (shown in Figure 1) to the database. If the intermediate queries find any tuple in the database that would cause semantic conflicts, then Query Processing Algorithm does not allow the application to perform the query. If, on the other hand, none of the intermediate queries return semantically incorrect results, the Query Processing Algorithm realizes that the query won't return semantically incorrect results, and it allows the application to proceed querying the database.

The method just described prevents the application from retrieving information from the database whose meaning would be misunderstood. Therefore, semantic conflicts are eliminated. Moreover, this is achieved in a very general form: the application user and database owner only have to write their context rules, and the system automatically does the rest of the work. This is much more efficient than the current situation in which the user of the application has to personally check that the data retrieved is in the correct context, or in which no checks are performed at all. Moreover, it provides the base in which to build future systems that would automatically fix semantic conflicts (by automatically changing a price from pesetas to dollars, for example), without making the user of the application write a special purpose conversion algorithm for every database that he or she wants to access.

The following sections will present the system in more detail, and discuss its implementation.

PART II

THE CONTEXT MEDIATOR

An intuitive explanation of what the system does has already been presented in the Introduction. This section gives a more rigorous description of how the context information and queries are stored and manipulated by the system. The emphasis is on explaining in greater detail what the Subsumption and Query Processing Algorithms do, without dwelling on how they are implemented. The two sections after this will discuss the algorithms' implementations, referring constantly to the code in the Appendixes. The Subsumption and Query Processing algorithms were designed in part by Andrew Leung.

STEP 1: Building the Rules Files

As explained in the Introduction, the Context Mediator's work starts by receiving two sets of rules: one from the user and one from the database. In the implementation these rules are written into files with specified format. These files of are parsed by a parser written by Rith Peou into rules tables. For example, assume that the database's schema contains the following attributes:

In general, semantic conflicts arise because the person who creates and maintains a database (in general its owner) always knows, but does not makes explicit the context of the data that he or she stores. In the time before computer networks facilitated the sharing of data, semantic conflicts were not very important because most of the time people only used their own databases. But today, when anyone can access hundreds of data sources, each with different context specifications which are not made explicit, semantic conflicts are widespread. There is a lot of interest in government and industry in finding ways to avoid these misunderstandings. In particular, there is some research being done at MIT and other places to find ways in which semantic conflicts can be detected automatically by a computer, saving its user the effort of checking that every piece of information that he or she imports has the correct context.

This thesis presents the implementation of a source-receiver system, in which an application retrieves data from database without incurring in semantic conflicts. This is achieved using a **context mediation** method proposed by Madnick and Siegel [SM 91], in which both the application and the database make their context explicit, and all possible semantic conflicts are detected before the application queries the database. The conflicts are found by executing a *Subsumption Algorithm* that compares the application's and the database's context definitions. Other useful references are [SSR a 92], and [SSR b 92]. Figure 1 shows a simple sketch of the elements of the system.

At its highest level, the system consists of an *application* and a *database*. The application can be any program that retrieves information from the database; and the database can have any desired schema. For example, assume that the database is a financial database whose schema contains the following four attributes:

Company_Name, Exchange, Instrument_Type and Trade_Price

And, as in the Introduction, only Trade_Price is the only attribute whose meaning can be ambiguous. An example of a file of rules for deriving the application's context is shown in Figure 1.

It is important to point out the structure of this file. First it starts with the keyword `APPLICATION` which specifies that the file refers to the rules for the user (the application) who wants data from the database. Otherwise, if it referred to the rules for the source (the database,) it would start with the keyword `SOURCE`. The keyword is followed by 4 statements: a `define_sem_domain` statement, a `define_assign_domain` statement, and two `add_rule` statements. Each of the statements ends with the keyword `end`.

```
APPLICATION

define_sem_domain Trade_Price
from db_semantics
as { Trade_Price_Status, Currency }
end

define_assign_domain Trade_Price
from db_semantics
as { Instrument_Type, Exchange }
end

add_rule Trade_Price
if Instrument_Type = 'equity' and Exchange = 'madrid'
then   Trade_Price_Status   =   'latest_nominal_price'   and
Currency = 'pesetas'
end

add_rule Trade_Price
if Exchange = 'nyse'
then   Trade_Price_Status   =   'latest_trade_price'   and
Currency = 'US dollars'
end
```

Figure 1. Sample application rules file

All the statements are followed by the name of an attribute in the data base; in this case, all the names are Trade_Price. These are the attributes whose meaning can be ambiguous, and therefore need their context defined. They are called **non_primitive attributes**. Non-primitive attributes have modifiers called **meta-attributes**, which appear in the define_sem_domain statement. In this case, the meta-attributes of Trade_Price are Trade_Price_Status and Currency.

The statement define_assign_domain is used to define the set of attributes from which the values of the meta-attributes of Trade_Price are derived. In this case those attributes are: Instrument_Type and Exchange.

Finally the last two add_rule statements, are used to define the rules for deriving the context of Trade_Price. Note that the rules are *if-then* statements. No else statement is allowed. In the **if-part** (before the then), only attributes from the define_assign_domain statement are allowed. In the **then-part**, only attributes from the define_sem_domain (the meta-attributes) are allowed. Note that *ands* are allowed both in the if-part and the then-part, but *ors* are not permitted in either.

Observe also that in the add_rule statements shown, the conditions in the if-part and then-part only contain the operation =. However, != is also allowed, and >, <, >=, and <= are permitted for those conditions whose values are floating points. In other words, it would be fine if the first rule had said:

```
if Instrument_Type !=  equity' and Exchange != 'madrid'
```

or if another rule said:

```
if Trade_Price > 5.03e2
```

STEP 2: Parsing the Rules Files

The 'human readable' files of rules described a above are parsed into 'machine readable' tables by rule_parser. The parser builds two tables: an **appl_rules** table, that contains the rules from the APPLICATION file, and a **db_rules** table, that contains the SOURCE rules. Figure 2 shows the appl_rules table that results from parsing the file shown in Figure 1.

APPL_RULES

r_number	atr_name	arg_name	operation	value	a_domain	mark
1	Trade_Price	Instrument_Type	=	equity	A	0
1	Trade_Price	Exchange	=	madrid	A	0
1	Trade_Price	Trade_Price_Status	=	latest_nominal	S	0
1	Trade_Price	Currency	=	pesetas	S	0
2	Trade_Price	Exchange	=	nyse	A	0
2	Trade_Price	Trade_Price_Status	=	latest_trade	S	0
2	Trade_Price	Currency	=	USdollars	S	0

Figure 2. appl_rules table created after parsing the rules file from Figure 1.

The appl_rules table shown in Figure 2 separates the different rules by having different values for the r_number field. For example, in Figure 2, all the tuples with r_number = 1 refer to the first add_rule statement of the rules file, and the tuples with r_number = 2 refer to the second add_rule statement. Within a same add_rule statement, the *if*-part is separated from the *then*-part by the domain_type field. Tuples with domain_type = A come from the if-part, and tuples with domain_type = S come from the then-part.

The rest of the fields contain the conditions specified in the rules. To find out the condition specified in the if-part of rule 1, the tuples with r_number = 1 and domain_type = A are ANDed together, to obtain:

```
Instrument_Type = 'equity' AND Exchange = 'madrid'
```

just as in the original rules file. In this way, it is straightforward to reconstruct the rules from the tables.

STEP 3: Performing the Subsumption Algorithm

The Subsumption Program compares the rules from the application and source, to find out under which conditions queries from the application to the database will return results with the correct semantic context, and under which conditions they won't. The idea of the Subsumption Algorithm is to perform the rules comparison only once and store their results, so that they can be used by future queries from the application to the database. This should speed up query processing significantly.

The Subsumption Algorithm produces two tables under which these conditions are stored: the `no_table` and the `yes_table`. It also produces a `bad` table where the non_primitive attributes that may cause semantic conflicts are stored. Figure 3 contains a rules file for a source, and Figure 4 contains the `db_table` that results from parsing it. Figure 5 shows the results returned by the Subsumption Algorithm after comparing the rules in Figure 4 with those shown in Figure 2.

The Subsumption Algorithm performs two steps on every combination of rules from the application and database. First it checks whether their *if* conditions intersect or conflict. If they intersect, it stores their intersection in either the yes_table or the no_table, depending on whether the *then* conditions of the database are equal-or-subset of the *then* conditions of the application. This will become clear with the current example.

```
SOURCE

define_sem_domain Trade_Price
from db_semantics
as { Trade_Price_Status, Currency }
end

define_assign_domain Trade_Price
from db_semantics
as { Instrument_Type, Exchange }
end

add_rule Trade_Price
if Instrument_Type = 'equity' and Exchange = 'madrid'
then Trade_Price_Status = 'latest_nominal' and Currency =
'pesetas'
end

add_rule Trade_Price
if Instrument_Type = 'equity' and Exchange = 'nyse'
then Trade_Price_Status = 'latest_trade' and Currency =
'USdollars'
end

add_rule Trade_Price
if Instrument_Type = 'future'
then Trade_Price_Status = 'latest_closing' and Currency =
'USdollars'
```

Figure 3. Sample source rules file

DB_RULES

r_number	atr_name	arg_name	operation	value	a_domain	mark
1	Trade_Price	Instrument_Type	=	equity	A	0
1	Trade_Price	Exchange	=	madrid	A	0
1	Trade_Price	Trade_Price_Status	=	latest_nominal	S	0
1	Trade_Price	Currency	=	pesetas	S	0
2	Trade_Price	Instrument_Type	=	equity	A	0
2	Trade_Price	Exchange	=	nyse	A	0
2	Trade_Price	Trade_Price_Status	=	latest_trade	S	0
2	Trade_Price	Currency	=	USdollars	S	0
3	Trade_Price	Instrument_Type	=	future	A	0
3	Trade_Price	Trade_Price_Status	=	latest_closing	S	0
3	Trade_Price	Currency	=	USdollars	S	0

Figure 4. db_rules table created after parsing the rules file shown in Figure 3.

Comparing the If Conditions

The if-conditions of rules 2 from the application and 3 from the data source don't conflict. This can be seen by rebuilding the *if* conditions from the tables in Figures 2 and 4, as explained in the previous step. These conditions are:

APPLICATION: Exchange = 'nyse'

SOURCE: Instrument_Type = 'future'

Their intersection (the condition that satisfies both conditions) is:

Exchange = 'nyse' AND Instrument_Type = 'future'

This intersection is stored in either the yes_table or the no_table, depending on the result of comparing the rules' then-conditions. Another pair of rules whose if-conditions intersect are rules 1 and 1 from the application and source:

APPLICATION: Exchange = 'madrid' AND Instrument_Type = 'equity'
SOURCE: Exchange = 'madrid' AND Instrument_Type = 'equity'
In this case finding the intersection is trivial because both conditions are identical. Again, the interaction of the conditions will be stored in either the yes_table or no_table, depending on the next step.

A pair of rules whose if-conditions conflict are rules 2 and 1 from the application and database:

APPLICATION: Exchange = 'nyse'
SOURCE: Exchange = 'madrid' AND Instrument_Type = 'equity'
In this case there is no tuple whose Exchange value can satisfy both conditions above. When the if-parts of two rules conflict, their then-parts are not compared.

Comparing the Then Conditions

Once the intersection of the if-conditions has been found, it is necessary to know if the application's then-condition is automatically satisfied if the source's then-condition is satisfied. In other words we want to know if the source's then-condition is equal or subset of the application's then-condition. If it is, then it is certain that the source will provide data in the correct context to the application if the intersection of the if conditions is satisfied.

For example, see what happens when the then-conditions of rules 1 and 1, and the then-conditions of rules 2 and 3 are compared. (Remember that in the previous step it was found that for either of these combinations of rules their if-conditions intersect).

It is not hard to see that the then-conditions of source rule 3 NOT equal-or-subset of the then-conditions of application rule 2:

APPLICATION: `Trade_Price_Status = 'latest_trade'`
AND `Currency = 'USdollar'`
SOURCE: `Trade_Price_Status = 'latest_closing'`
AND `Currency = 'USdollars`

In other words, satisfying the source's then-conditions means that the application's then conditions are not necessarily satisfied. The attribute that causes the conflict is Trade_Price. Because of this, the intersection these rules' if-conditions that was found in the previous part is stored in the no_table, and the attribute Trade_Price is stored in the bad table.

On the other hand, the then-condition of source rule 1 is equal-or-subset of the then-condition of application rule 1:

APPLICATION: `Trade_Price_status = 'latest_nominal_price'`
AND `Currency = 'pesetas'`
SOURCE: `Trade_Price_status = 'latest_nominal_price'`
AND `Currency = 'pesetas'`

Satisfying the source's then-conditions automatically satisfies the application's then-conditions. Because of this, the intersection of their if-conditions that was found in the previous part is stored in the yes_table. Also the then-condition of rule 2 in the database is equal-or-subset of the then condition of application rule 1. Nothing is stored in the bad table in these two cases. Figure 5 shows the yes_table, no_table, and bad produced by subsuming these two sets of rules. Those tables are used by the query processing algorithm to make sure that the queries made by the application to the database won't return semantically incorrect results.

YES_TABLE

r_number	s_number	atr_name	arg_name	operation	value	mark
1	1	Trade_Price	Instrument_Type	=	equity	0
1	1	Trade_Price	Exchange	=	madrid	0
2	2	Trade_Price	Instrument_Type	=	equity	0
2	2	Trade_Price	Exchange	=	nyse	0

NO_TABLE

r_number	s_number	atr_name	arg_name	operation	value	mark
2	3	Trade_Price	Exchange	=	nyse	0
2	3	Trade_Price	Instrument_Type	=	future	0

BAD

r_number	s_number	np_attr
2	3	Trade_Price

Figure 5. Tables produced by subsuming the tables shown in Figures 2 and 4

STEP 4: Building and Parsing Queries

As in the rules case, queries are written into 'human readable' files which are then parsed into 'machine readable' query tables by a query-parser. The query-parser was written by Rith Peou. The query processing algorithm uses the query tables, along with the yes_table, no_table and bad tables returned by the Subsumption Algorithm, to make sure that the query won't return semantically incorrect results from the database. A sample query file is shown in Figure 6.

```
select TradePrice
from db_table
where Exchange = 'nyse' ;
```

Figure 6 Sample query

The things to note from the query in Figure 6 is that it assumes that the database table is called db_table, and that the query ends with a semicolon. More than one attribute could be in the select clause, and the where clause can contain ORs in addition to the ANDs shown. When the query_parser is given as input the query shown in Figure 6, it produces the 3 tables (called a_list, t_list and c_list) shown in Figure 7.

A_LIST

attrib
TradePrice

T_LIST

tabl
db_table

C_LIST

_number	attr_name	operation	value	mark
1	Exchange	=	nyse	0

Figure 7 Tables produced by the query_parser after processing the file shown in Figure 6.

It is easy to see by comparing Figures 6 and 7 that the attributes in the select list of the query (in this case only Trade_Price) are stored in the a_list table. The name of the database is stored in the t_list table. And the conditions in the where clause of the query are stored in the c_list table. Just as in the case with the rules tables, and the yes_table and no_table, all conditions in the c_list table with the same r_number are ANDed together. By observing the c_list in Figure 7, it is easy to see that the where condition in the query was:

```
where Exchange = 'nyse'
```

If there had been an OR in the where clause of the query, then the c_list would have some tuples with r_number = 2. The tables shown in Figures 5 and 7 are used by the query processing algorithm to decide if the query can be performed by the application.

STEP 5: The Query Processing Algorithm

The purpose of the query processing algorithm is to decide if the application is allowed or not to perform the query on the database. To decide this, the algorithm checks if the query would return tuples in a context different from the one required by the application. If this found to happen, then the application does not receive permission to execute the query. Otherwise it does. The query processing algorithm performs two separate steps to check for semantic inconsistencies:

i) First it checks that the query's where clause is in the correct context.

ii) Second it checks that, if given the conditions in the where clause, the query does not return any attribute from the select clause that is in the wrong context.

For example, for the query shown in Figure 6, the part that checks the where clause would look in the c_table for any attribute that is also in the bad table. As can be seen in Figure 5, the bad table contains only the Trade_Price attribute; and as can be seen in Figure 7, the c_list table only contains the Exchange attribute. Therefore, no attribute is both in the where clause and in the bad table. Because of this, there can be no misunderstanding between the application and the database when deciding what the where clause means. The query processing algorithm then proceeds to check the select list.

In this part, the query processing algorithm would look for an attribute that is in both the select clause and in the bad table. In our example, Trade_Price is in both tables. That means that there is the possibility that the query could return a Trade_Price result in the wrong context. To check if this happens, the query processor has to find out if the database includes a tuple that satisfies the queries where conditions and the conditions in the no_table. For this, the query processor performs the following steps.

First, it copies the where conditions into the 'query_conditions' table, and it copies the relevant conditions from the no_table into the 'no_conditions' table. Figure 8 shows these tables. Note that these two new tables have the same schema than the appl_rules and db_rules tables used in the Subsumption Algorithm, but different names. The reason for this is simple: it is necessary to find the intersection of the where clause and the conditions in the no_table, to see if a tuple in the database can satisfy both at the same time. For this, it is necessary to perform the part if the subsumption algorithm that finds the intersection of the if-clauses of two sets of rules, using the where conditions and the conditions in the no_table as if they were if conditions from rules. This is the reason for which in both the query_conditions and no_conditions tables, the domain_type field contains the only the value "A", and for which the r_number, s_number pair from the no_table has been consolidated into an r_number in the no_conditions table.

QUERY_CONDITIONS

r_number	atr_name	arg_name	operation	value	a_domain	mark
1		Exchange	=	nyse	A	0

NO_CONDITIONS

r_number	atr_name	arg_name	operation	value	a_domain	mark
1		Exchange	=	nyse	A	0
1		Instrument_Type	=	future	A	0

Figure 8. Tables ready to be subsumed produced by the query processing algorithm from the c_list and no_table tables from Figures 5 and 7.

In this case it is easy to see that the condition in the query_conditions table does not conflict with the conditions in the no_conditions table. The intersection of these two sets of conditions is equal to the conditions in the no_conditions table. This intersection is stored by the Subsumption Algorithm in the 'intermediate' table, shown in Figure 9.

INTERMEDIATE

r_number	atr_name	operation	value
1	Exchange	=	nyse
1	Instrument_Type	=	future

Figure 9 Intermediate table contains the intersection of the conditions in the no_conditions and query_conditions tables shown in Figure 8.

The query processing algorithm uses the information in the intermediate table to produce the intermediate query:

```
select TradePrice
from db_table
where Exchange = 'nyse'
    and Instrument_Type = 'future';
```

Any result returned by this query is certain to be semantically incorrect, because the query's where clause intersects (in this case is equal to) the conditions in the no table. Because of that, if the result of the intermediate query is not null, the query processing algorithm would not allow the application to execute the query that it wants to do to the database (query shown in Figure 6). On the other hand, if this intermediate query returns a null result, that means that the database does not contain any attribute in the wrong semantic context for the application, and the application can be permitted to make its query. With this, the procedure is completed.

What is Next?

This section has presented a somewhat sophisticated example of application and database rules, and has followed in general detail the operations performed by the subsumption and query processing algorithms. It has also presented the major storage structures by the algorithms in the system: the appl_rules, db_rules, yes_table, no_table, bad, query_conditions, no_conditions and intermediate tables. The next two sections will respectively discuss the Subsumption Algorithm and the Query Processing Algorithm in exhaustive detail. Their purpose will be to acquaint the reader to the specific implementation of the Algorithms, including their code in the Appendixes, as opposite to the current section and the Introduction which only give a general feel of what the Algorithms do.

PART III

THE SUBSUMPTION ALGORITHM

In this part, a detailed discussion of the Subsumption Algorithm's implementation will be presented. The algorithm's code, which is included in Appendix I, will be constantly referenced. The code consists of 4 major procedures, and 6 subsidiary ones. The names of these major procedures are main, SemanticEqOrSub, CanBeSubsumed, and NotEqStuff. The subsidiary procedures are called DeclareSubsumptionCursors, Number_Application_Rules, Number_Source_Rules, compare_strings, print_status, and str_to_float. The following is a picture of how the procedures call each other:

main

CanBeSubsumed SemanticEqOrSub NotEqStuff Number_Application_Rules
Number_Db_Rules

print_status
str_to_float

In order for the algorithm to work, all the procedures must be in the same .ec file, which must be compiled into an executable. The executable will be called by the user interface program whenever there is need to subsume two sets of rules that have already been parsed. The executable won't work if any of the 10 procedures in the algorithm are copied into different .ec files, which are then linked and compiled together. The reason is that the DeclareSubsumptionCursors procedure declares all the cursors that the other procedures use, and it is an ESQL requirement that a procedure only use cursors which were declared in the file in which it is written. It is also required that DeclareSubsumptionCursors be the procedure at the top of the file.

The next part of this section explains which are the requirements that must be met before calling the Subsumption Algorithm, and then, the following 4 sections explain the code for each of the major programs.

1 Requirements for calling the Subsumption Algorithm

The Subsumption Algorithm should be called after the application and database rules have been parsed into the appl_rules and db_rules tables described in the previous section. There must be at least one rule in each of the tables. The rules in each table must be numbered with increasing integer values, starting from 1. In other words, there must always be a db_rule 1 and an appl_rule 1. There must not be an appl_rule (or db_rule) 3 if there is not an appl_rule (or db_rule) 2, and so forth. This has important repercussions for a future 'rule browser' that might allow the user to delete a rule from a set of rules that have already been parsed. The hypothetical browser will have to make sure that all pertinent rule numbers are changed to fit the specification given above.

Another important specification, is that the mark field of all the tuples in the appl_rules and db_rules tables should be 0 before the subsumption is called. Currently the rules parser always inserts the value of 0 in such a field when the tables are filled.

As explained in the previous section, the rules in either the application or source can only have =, !=, >, <, >=, or <= operations. Also, as explained there, the = and != operations may be followed by any type of string or floating point number, but the other operations only work with floating point numbers.

The final requirement before calling the Subsumption Algorithm is that the bad, yes, and no tables should be empty. The subsumption algorithm doesn't clear them before it starts, so care must be taken of clearing them by the program that calls the subsumption.

2 The main Procedure

The main procedure has two major parts: the first one sets the environment in which the

Subsumption Algorithm is performed, and the second controls the execution of the algorithm. This 2 parts will be explained separately. The two parts are clearly marked in the main procedure's code in Appendix 1.

PART I. Setting the Environment

The first thing that the main procedure does is to set up the environment in which the subsumption algorithm can be performed. It does this in two steps:

STEP 1 sets the 'cdrdb' database as the current database, in which all queries and other ESQL operations will be performed. This is done by executing the statement:

```
$database cdrdb;
```

As explained before, the cdrdb database contains all the tables that are used in the subsumption and Query Processing Algorithms. When the subsumption is compiled and an executable file is created, this executable must be put in the same directory that contains the cdrdb.dbs subdirectory; otherwise the algorithm won't know how to find the cdrdb database.

STEP 2 creates all the 20 cursors used in the subsequent parts of the algorithm by calling the procedure DeclareSubsumptionCursors(). (The reader is referred to the Informix ESQL manual [Informix 86], as well as [Tare 89] for an explanation of what cursors are.) The names of the 20 cursors, followed by the names of the procedures that use them are listed below:

cur1 used by CanBeSubsumed and SemEqualOrSub
cur2 used by CanBeSubsumed
cur_eq used by CanBeSubsumed and SemEqualOrSub
cur_neq used by SemEqualOrSub
cur_gt used by CanBeSubsumed and SemEqualOrSub
cur_lt used by CanBeSubsumed and SemEqualOrSub
cur_get used by CanBeSubsumed

cur_temp1 used by CanBeSubsumed

cur_chk1 used by CanBeSubsumed

cur_chk2 used by CanBeSubsumed

cursor_final used by CanBeSubsumed

cur3 used by NotEqStuff

cur4 used by NotEqStuff

cur5 used by NotEqStuff

cur_Teq used by NotEqStuff

cur_Tgt used by NotEqStuff

cur_Tlt used by NotEqStuff

cur_deltemp used by main

cur_applt used by Number_Application_Rules

*cur_sourc*used by Number_Source_Rules

PART II. Controlling the Execution of the Algorithm

When the environment has been set up, the Subsumption Algorithm can start. Its execution is controlled by PART II of the main procedure. It has 2 steps, which are marked in the code in Appendix 1.

STEP 1 finds out how many rules there are in the appl_rules and db_rules tables. It does so by calling the Number_Application_Rules and Number_Db_Rules procedures. These procedures are so simple that they don't merit any lengthy discussion. Each scans one of the rules tables (the appl_rules and db_rules tables respectively) checking the r_number of the tuples, and returns the largest r_number found.

In STEP 2 the procedure enters a double *for* loop, in which it compares each of the rules in the application with each of the rules in the source. For each combination of application rule and db rule, the procedure does at least the first two of the following 4 operations, which are marked in the Appendix:

FIRST, clean the temporary table because the procedure CanBeSubsumed requires it. This is done by executing the command:

```
$delete from temporary;
```

SECOND, compare the if-conditions of each pair of rules, and find their intersection. (In the language of [SM '91], compares the parts of the rules that deal with assignment domain). This is done in two steps: calling the procedure CanBeSubsumed, and then calling the procedure NotEqStuff, with the r_numbers of the rules that are being compared as arguments. If both procedures return the value 1, then the if-conditions intersect, and the intersection is stored in the temporary table. If, on the other hand, one of them returns 0, the if-conditions conflict, and whatever is stored in the temporary table is garbage.

If the if conditions of the rules are found to conflict, nothing else is done with that combination of rules, and the procedure starts the next loop, in which it compares the next combination of rules. If the if-conditions don't conflict, the procedure proceeds to execute the next two operations.

THIRD, compare the then-conditions of the rules, to find out if the previously found intersection of the if-conditions should be written into the yes_table, or into the no_table. This is done by calling the procedure SemanticEqOrSub, which returns 1 if the then-conditions of the database are equal or subset to the then-condition of the application, and 0 if they are not. The procedure SemanticEqOrSub also updates the bad table when the then-conditions of the database rule are not equal or subset of the then-conditions of the application rule.

FOURTH, copy the contents of the temporary table into the yes_table or no_table, depending on the result of the previous step. This is done by entering a while loop in which the contents of the temporary table are extracted tuple by tuple, and copied into one of the tables. The only thing to note here is that the temporary table does not contain the r_number, s_number, and mark fields that the yes_table and no_table contain. This is not a problem because when a tuple is inserted into the yes_table or no_table, the value of r_number is the number of the application rule that is being examined; the value of s_number is the number of the database rule that is being examined; and the value of mark is always 0.

When STEP II of the main procedure finishes, each combination of rules has been processed, and the Subsumption Algorithm ends. Now, the yes_table and no_table contain the information that say under which conditions the database always returns data in the correct context for the application, and under which conditions it might return data in the wrong context. The bad table contains the information that relates the name of a non-primitive attribute that has different semantic context in the application and the database, with the rule numbers that produced the conflict.

3 The CanBeSubsumed Procedure.

The CanBeSubsumed procedure is called by the main procedure, and receives as arguments the numbers of two rules: one from the application, and one from the database. It then proceeds to compare the *if*-conditions of the application and database rules indexed by these numbers, and finds out if these conditions conflict or intersect. To do this, the procedure gets from the appl_rules and the db_rules tables the tuples that correspond to the if-conditions of these rules. In other words, it gets the tuples from appl_rules and db_rules whose r_numbers are equal to the arguments with which CanBeSubsumed was called, and whose a_domain field contains an A. From these tuples, the procedure ignores all those, whose operation is `!=`, which are later processed by the procedure NotEqStuff. Using the tuples that it keeps, CanBeSubsumed executes a complex algorithm in which it finds the intersection of the if-conditions of the rule from appl_rules, with the if-condition of the rule from db_rules. This intersection is stored in the temporary table. CanBeSubsumed returns 1 when the if-conditions from the two rules compared intersect, and 0 when they conflict.

The CanBeSubsumed algorithm consists of 4 parts, which are performed in strict order, and marked in the code in Appendix 1. If the procedure detects a conflict between the if-conditions of the rules when executing any part, it immediately ends returning the value of 0 to signal that the two rules don't have an intersection. Otherwise, if no conflict is found after part 4 finishes, the procedure returns 1 to signal that the rules intersect, leaving the intersection of the rules in the temporary table.

The 4 parts of the CanBeSubsumed procedure will be presented by going through an example. Suppose that the database had a schema with 5 different fields, which we shall call attribute_A, attribute_B, attribute_C, attribute_D, attribute_E and attribute_F. Suppose that only attribute_F is a non_primitive attribute. In other words, only the value of attribute_F is complex enough to merit defining some meta-attributes to clarify what attribute_F means (We don't give names for the hypothetical meta-attributes of attribute_F here, but the reader can assume that they exist.) Furthermore, suppose that the attributes in the assignment_domain of attribute_F are attribute_A, attribute_B, attribute_C, attribute_D, and attribute_E. In other words, the context of attribute_F is determined by observing the values of attribute_A, attribute_B, attribute_C, attribute_attribute_D, and attribute_E, and applying some rules.

For example, suppose that rule 1 of the application said:

```
add_rule attribute_F
if attribute_A = 50 and attribute_B = 60
     and attribute_C = 1000 and attribute_D >= 300
then ....
```

(The then part of the rule is irrelevant here). Furthermore, suppose that rule 2 of the database said:

```
add_rule attribute_F
if attribute_A = 50 and attribute_B > 30
     and attribute_B < 100
     and attribute_D > 100 and attribute_D <= 300
     and attribute_E = 6
then ....
```

(Again, the *then* part of the rule is irrelevant here.) Obviously the database also has to have a rule 1, but we don't care about it in this example. Immediately after the rules are parsed, the contents of the appl_rules and db_rules tables that refer to the *if*-parts of the rules discussed above are shown in Figure 10. The reader should not have any trouble noting that in Figure 10 not ALL the contents of the appl_rules and db_rules tables are shown, BUT only those that are relevant for the example. For example, only the two rules discussed above are shown, although the application could have more than one rule, and the database at least has to have a rule 1 besides the rule 2 shown. The a_domain field from the tables has been omitted to save space, because all the tuples shown in Figure 10 would have a value of A in that field, because they refer to the conditions in the *if* part of the rules. All the tuples that refer to the *then* parts of the rules have been omitted also because they are irrelevant for the example.

Now, suppose that the main procedure requests the CanBeSubsumed procedure to find the intersection of the *if*-parts of the rules shown above by calling:

```
CanBeSubsumed(1,2)
```

APPL_RULES

r_number	atr_name	arg_name	operation	value	mark
1	attribute_F	attribute_A	=	50	0
1	attribute_F	attribute_B	=	60	0
1	attribute_F	attribute_C	=	1000	0
1	attribute_F	attribute_D	>=	300	0

DB_RULES

r_number	atr_name	arg_name	operation	value	mark
2	attribute_F	attribute_A	=	50	0
2	attribute_F	attribute_B	>	30	0
2	attribute_F	attribute_B	<	100	0
2	attribute_F	attribute_D	<=	300	0
2	attribute_F	attribute_D	>	100	0
2	attribute_F	attribute_E	=	6	0

Figure 10. Sample appl_rules and db_rules tables

Part I, Making Sure that rules refer to the same non-primitive attribute

The first thing that the CanBeSubsumed checks is if the rules refer to the same non-primitive attribute. This is done by checking that the atr_name fields of the db_table and appl_table, indexed by the r_numbers given to CanBeSubsumed, are equal. If this is so, the CanBeSubsumed proceeds with the next part of the procedure; and if if the rules don't refer to the same non-primitive then CanBeSubsumed immediately terminates returning the value of 0. In the example above, both rule 1 of the application and rule 2 of the database refer to the non-primitive attribute attribute_F, so CanBeSubsumed continues to the next Part.

Part II, The >= or <= Special Case

The second part of the algorithm looks for all the tuples from the relevant part of appl_rules whose operation is >= or <=. (For the rest of this discussion, the 'relevant part' of appl_rules or db_rules, or simply the appl_rules and db_rules tables are what is shown in Figure 10.) For each of those tuples, it looks in the relevant part of db_rules for a tuple with the same atr_name, the same value, but 'opposite operation' (the 'opposite' of >= is <= and the other way around.) If it finds such a tuple in db_rules, it is said to 'match' the tuple in appl_rules.

If the procedure doesn't find the match of a tuple, it loops around to look for the 'match' of the the next of the tuples retrieved from appl_rules with operation >= or <=. When there are no more of these tuples left, the procedure goes to Part III. When CanBeSubsumed finds a tuple in db_rules that matches an appl_rules tuple, it executes the following operations. First, it sets the mark to 1 in all the tuples of the relevant part of db_rules whose arg_name is equal to the arg_name of the tuples that matched. Then it copies the arg_name and value from the tuple in appl_rules to the temporary table, using the operation =.

For example, see what Part I does with the tables shown in Figure 10. First, it looks for tuple in the relevant part of appl_rules with operation >= or <=, and finds only one:

`attribute_D >= 300`

Then, it looks in db_rules for the match of the tuple just found, and finds it:

`attribute_D <= 300`

Therefore, the CanBeSubsumed executes the following operations, which have been explained explained above: First, it sets the mark field to 1 in BOTH of the tuples in the db_rules table, whose atr_name is attribute_D:

`attribute_D <= 300` AND `attribute_D > 100`

Then, it inserts into the temporary table the tuple:

`attribute_D = 300`

After this, Part II finishes because there are no more tuples left in appl_rules with operation >= or <=. Figure 11 shows the state of the relevant parts of the appl_rules, db_rules, and temporary tables when Part II terminates. Note that the tuple: `attribute_D = 300` that was inserted into the temporary table, is the intersection of the interval `attribute_D >= 300` from the appl_rules and the interval `100 < attribute_D <= 300` from db_rules. The reasons for which BOTH the tuples in db_rules were marked will become apparent in the next two parts.

Part III, Scanning the appl_rules Tuples

Part III starts with the temporary and db_rules tables as Part II left them. Opposite from Part II, which only uses the tuples in the relevant part of appl_rules with operations >= or <=, Part III uses ALL the tuples in the relevant part of appl_rules, except those whose operation is !=. Part III, treats differently tuples with different values in their operation field. Depending on the value of the operation, Part III classifies tuples into 5 cases: Case 1 for tuples with operation =, Case 2 for operation >, 3 for <, 4 for >=, and 5 for <=. The code for each of these cases is Clearly in Appendix 1. Continuing the example from Figure 10, it is easy to see that `attribute_A`, `attribute_B`, and `attribute_C` in appl_rules are Case 1, and that `attribute_D` is Case 4.

APPL_RULES

r_number	atr_name	arg_name	operation	value	mark
1	attribute_F	attribute_A	=	50	0
1	attribute_F	attribute_B	=	60	0
1	attribute_F	attribute_C	=	1000	0
1	attribute_F	attribute_D	>=	300	0

DB_RULES

r_number	atr_name	arg_name	operation	value	mark
2	attribute_F	attribute_A	=	50	0
2	attribute_F	attribute_B	>	30	0
2	attribute_F	attribute_B	<	100	0
2	attribute_F	attribute_D	<=	300	1
2	attribute_F	attribute_D	>	100	1
2	attribute_F	attribute_E	=	6	0

TEMPORARY

atr_name	operation	value
attribute_D	=	300

Figure 11, State of the appl_rules, db_rules and temporary tables after Part II of CanBeSubsumed finishes. The arrows the tuples in appl_rules [illegible] db_rules that match, from which the tuple in temporary is derived. Note the marks in db_rules.

In Part III, for every tuple in the relevant part of appl_rules, CanBeSubsumed searches in the relevant part of db_rules for tuples with the same arg_name. It checks if the db_rules tuples conflict with the appl_rules tuple, in which case the procedure terminates returning 0 to signal that the if parts of the rules conflict. Otherwise, CanBeSubsumed finds their intersection and writes it in the temporary table. Then, CanBeSubsumed puts some necessary marks on the db_rules tuples that will be explained later.

For example, see what the procedure does with the three Case 1 tuples, and then what it does with the Case 4 one:

Case 1 Tuples

For every tuple that CanBeSubsumed retrieves from appl_rules with operation =, it performs the following steps:

Step 1 looks in the db_table for a tuple with the same arg_name, and operation =. In the example above, it finds such a tuple among the db_table tuples for the tuple with arg_name = `attribute_A`, but not for the tuples with arg_name = `attribute_B` or arg_name = `attribute_C`. For the `attribute_A` case, the procedure executes Step 1.1, but for the `attribute_B` and `attribute_C` cases, it continues with Step 2.

Step 1.1 compares the value of the tuple in appl_rules with that of the tuple in db_rules. In the `attribute_A` case, both values are equal to 50. If the values are equal, it copies the arg_name, operation and value from appl_rules into the temporary table. Then sets the mark in the db_table tuple to 1. If, on the other hand, the values are not equal, the program immediately ends and returns the value of 0, signaling that the if conditions of the two rules are disjoint. After Step 1.1, the procedure is done processing the current tuple from appl_rules, and it loops to the next tuple.

In the example in Figure 11, when the program realizes that the two values of the `attribute_A` tuples are equal, it executes the aforementioned operations, leaving the state of the db_rules and temporary tables as shown in Figure 12. The program proceeds to process the appl_rules tuple with arg_name `attribute_B`.

Step 2 If in Step I CanBeSubsumed couldn't find a tuple in db_rules with the same arg_name as the appl_rules tuple, and with operation =, then it proceeds as follows: It looks for tuples among those retrieved from db_rules whose arg_name matches the arg_name of the tuple from appl_rules, but whose operation is >, <, >=, or <=. In the example in Figure 12, there are two such tuples in db_rules for the `attribute_B` case, but none for the `attribute_C` case. For the `attribute_B` case, the procedure executes step 2.1, but for the `attribute_C` case, it continues with step 3.

Step 2.1 makes sure that none of the tuples retrieved in step 2 conflicts with the tuple from appl_rules. An example of conflict would be the following: If the appl_rules tuple said

```
attribute_B = 60
```

and one of the source_rules tuples said:

```
attribute_B > 100
```

then the first tuple would not be inside the interval defined by the second tuple. Therefore, there would be no intersection between the two if-conditions, and the rules would conflict. The program would then immediately end, returning the value of 0.

In the example from Figure 12, however, there is no such conflict between the `attribute_B` tuple in the appl_rules table and the `attribute_B` tuples in the db_rules table. CanBeSubsumed proceeds as follows. First it sets to 1 the mark field of BOTH of the tuples checked from db_rules discussed above. Then it copies into the temporary table the atr_name, operation, and value of the tuple from appl_rules. After this, the procedure loops around to the following tuple in appl_rules. Figure 13 shows the state of the source_rules and temporary tables after executing step 2.1 for the `attribute_B` tuple from appl_rules.

APPL_RULES

r_number	atr_name	arg_name	operation	value	mark
1	attribute_F	attribute_A	=	50	0
1	attribute_F	attribute_B	=	60	0
1	attribute_F	attribute_C	=	1000	0
1	attribute_F	attribute_D	>=	300	0

DB_RULES

r_number	atr_name	arg_name	operation	value	mark
2	attribute_F	attribute_A	=	50	1
2	attribute_F	attribute_B	>	30	0
2	attribute_F	attribute_B	<	100	0
2	attribute_F	attribute_D	<=	300	1
2	attribute_F	attribute_D	>	100	1
2	attribute_F	attribute_E	=	6	0

TEMPORARY

atr_name	operation	value
attribute_D	=	300
attribute_A	=	50

Figure 12, State of the appl_rules, db_rules and temporary tables after the attribute_A tuple in appl_rules has been processed. The arrow shows which tuple in db_rules matches it. Note the mark in db_rules

APPL_RULES

r_number	atr_name	arg_name	operation	value	mark
1	attribute_F	attribute_A	=	50	0
1	attribute_F	attribute_B	=	60	0
1	attribute_F	attribute_C	=	1000	0
1	attribute_F	attribute_D	>=	300	0

DB_RULES

r_number	atr_name	arg_name	operation	value	mark
2	attribute_F	attribute_A	=	50	1
2	attribute_F	attribute_B	>	30	1
2	attribute_F	attribute_B	<	100	1
2	attribute_F	attribute_D	<=	300	1
2	attribute_F	attribute_D	>	100	1
2	attribute_F	attribute_E	=	6	0

TEMPORARY

atr_name	operation	value
attribute_D	=	300
attribute_A	=	50
attribute_B	=	60

Figure 13, State of the appl_rules, db_rules and temporary tables after the second tuple in appl_rules has been processed. The arrow shows which tuples in db_rules match it. Note the marks there

APPL_RULES

r_number	atr_name	arg_name	operation	value	mark
1	attribute_F	attribute_A	=	50	0
1	attribute_F	attribute_B	=	60	0
1	attribute_F	attribute_C	=	1000	0
1	attribute_F	attribute_D	>=	300	0

DB_RULES

r_number	atr_name	arg_name	operation	value	mark
2	attribute_F	attribute_A	=	50	1
2	attribute_F	attribute_B	>	30	1
2	attribute_F	attribute_B	<	100	1
2	attribute_F	attribute_D	<=	300	1
2	attribute_F	attribute_D	>	100	1
2	attribute_F	attribute_E	=	6	0

TEMPORARY

atr_name	operation	value
attribute_D	=	300
attribute_A	=	50
attribute_B	=	60
attribute_C	=	1000

Figure 14, State of db_rules and temporary tables after the third tuple in appl_rules has been processed. Arrow shows origin of last temporary tuple.

Step 3 If CanBeSubsumed failed to find matches for the appl_rules tuple in the db_rules table in Steps 1 and 2, it jumps to Step 3. Here, CanBeSubsumed simply copies the atr_name, operation, and value from the appl_rules tuple to the temporary table. After Step 3 is performed for the `attribute_C` tuple, the state of the temporary table is shown in Figure 14.

Non Case 1 Tuples

After the `attribute_C` tuple in appl_rules, the procedure reaches the `attribute_D` tuple. For the first time in the example this is a non case 1 tuple: its operation is not = but >=. This time the procedure executes the code marked case 4 in the code (shown in Appendix 1).

The `attribute_D` tuple in appl_rules is a very special case, because it has already been processed in Part II. (Remember that in Part I the procedure found a tuple `attribute_D <= 300` in db_rules that matched the tuple `attribute_D >= 300` in appl_rules, and it stored the tuple `attribute_D = 300` in the temporary table). Because Part II has already taken care of this tuple, all the relevant `attribute_D` tuples is db_rules have already been marked. Part III ignores the `attribute_D` tuple from appl_rules when it finds this marks. In other words, because Part II has already stored the intersection of the application and database `attribute_D` tuples in the temporary table, Part III doesn't do all that work again. Therefore, after processing the `attribute_D` tuple, the tables in Figure 14 are left unchanged. After this, Part III finishes because there are not any new tuples left in the appl_rules tables beyond the `attribute_D` tuple.

Now, suppose that the example shown in Figure 14 had been a little different. Suppose that instead of being the last tuple in appl_rules, there would have been a another tuple after `attribute_D >= 300`; and suppose that this last tuple was: `attribute_D < 500`. In this case, Part III would not have finished after processing the `attribute_D >= 300` tuple, but instead it would have continued with the `attribute_D < 500` tuple. Just as before, Part III would have found this extra tuple irrelevant because the `attribute_D` tuples in db_rules are already marked. Therefore, the presence of this last extra tuple would not make any difference to the db_rules and temporary tables shown in Figure 14.

On the other hand, suppose that the example in Figure 14 had the appl_rules tuples `attribute_D > 200` and `attribute_D < 500`, instead of the `attribute_D >= 300` tuple. This is shown in Figure 15.

APPL_RULES

r_number	atr_name	arg_name	operation	value	mark
1	attribute_F	attribute_A	=	50	0
1	attribute_F	attribute_B	=	60	0
1	attribute_F	attribute_C	=	1000	0
1	attribute_F	attribute_D	>	200	0
1	attribute_F	attribute_D	<	500	0

DB_RULES

r_number	atr_name	arg_name	operation	value	mark
2	attribute_F	attribute_A	=	50	1
2	attribute_F	attribute_B	>	30	1
2	attribute_F	attribute_B	<	100	1
2	attribute_F	attribute_D	<=	300	0
2	attribute_F	attribute_D	>	100	0
2	attribute_F	attribute_E	=	6	0

TEMPORARY

atr_name	operation	value
attribute_A	=	50
attribute_B	=	60
attribute_C	=	1000

Figure 15, New example. The last two appl_rules tuples are different from the original. The state of the temporary and db_rules tables is that after which the third tuple of the appl_rules table has been processed. Note that the attribute_D tuples in db_rules table are not marked anymore. Also, the attribute_D = 300 tuple has disappeared from the temporary table.

Figure 15 shows what would be the state of the tables just before Part III reached the first `attribute_D` tuple in appl_rules. Obviously, in this new example, Part II would not have found any matching pair of tuples with operations >= and <=, as it had done before. Because of this, the temporary table in Figure 15 does not include the `attribute_D` tuple at the top that the table in Figure 14 does. Also because of the same reason, the `attribute_D` tuples in db_rules in Figure 15 are not marked as they are in Figure 14. However, note the non case 1 tuples `attribute_A`, `attribute_B`, and `attribute_C` have been treated in the new example exactly as they had been before. In other words, the temporary table in Figure 15 contains the same `attribute_A`, `attribute_B`, and `attribute_C` as the temporary table in Figure 14, and the `attribute_A` and `attribute_B` tuples in the db_rules table on Figure 15 are marked exactly as the are in Figure 14.

When Part III reaches the first `attribute_D` tuple in appl_rules: `attribute_D > 200`, it finds a non case 1 tuple, whose counterparts in db_rules have not been marked. The processing of this, as any other non Case 1 tuples, is done in four steps:

In Step 1, for each tuple in the appl_rules table with sign > or >= (or <, <=) , the procedure first looks in db_rules for a tuple with the same arg_name and operation =. If the tuple is not found, (as in Figure 14), the procedure continues with step 2. Otherwise, if such a tuple exists, the procedure checks if the tuples conflict, and if they do, it finishes immediately, returning 0. If they don't conflict, the procedure checks if the tuple from db_rules is marked, in which case it doesn't do anything else. If the tuple from db_rules is not marked, it marks it, and copies it into the temporary table. In the example on Figure 15, there are no `attribute_D` tuples in db_rules with operation =, so Part II would continue with step 2.

Step 2 if the tuple from step I is not found, Part III looks for the tuple in db_rules with same arg_name and the 'opposite' operation of the tuple from appl_rules. For example, if the tuple from appl_rules has operation < or <= it looks in db_rules for a tuple with the same arg_name, but operation > or >=, and the other way around. If such a tuple from db_rules does not exist, then that's fine, and the procedure continues with Step 3. Otherwise, the procedure checks that the tuples don't conflict. For example, the tuples in Figure 15:

`attribute_D > 200` from appl_rules, and

`attribute_D <= 300` from db_rules

do not conflict. Also from Figure 15, the tuples:

`attribute_D < 500` from appl_rules, and

`attribute_D > 100` from db_rules

do not conflict either. If, on the other hand, the first `attribute_D` tuple form appl_rules had said `attribute_D > 400`, It would have conflicted with the `attribute_D <= 300` tuple in db_rules. In case of conflict the program immediately ends, returning 0 to indicate the conflict. Otherwise, proceeds to Step 3.

<u>Step 3</u> after making sure that no tuple in db_rules conflicts with our tuple from appl_rules which had operation > or >= (or < or <=), the program looks for the tuple in db_rules with the same atr_name and operation > or >= (or <, <=). If there is no such a tuple, then it goes to Step 4. If the tuple exists, then it marks the tuple in db_rules, picks the most restrictive of the two tuples and copies its arg_name, operation, and primitive into the temporary table. Examples of restrictivity are:

`attribute_D > 400` is more restrictive than `attribute_D > 300`

`attribute_D > 400` is more restrictive than `attribute_D >= 400`

`attribute_D < 400` is more restrictive than `attribute_D < 500`

`attribute_D < 400` is more restrictive than `attribute_D <= 400`

For example, for the tuples in Figure 15:

`attribute_D > 200` from appl_rules, and

`attribute_D > 100` from db_rules

It would insert `attribute_D > 200` into the temporary table because it is more restrictive, and marks the `attribute_D > 100` tuple in db_rules. Also from Figure 15, for the tuples:

`attribute_D < 500` from appl_rules, and

`attribute_D <= 300` from db_rules

It would insert `attribute_D <= 300` into the temporary table because it is more restrictive, and marks the corresponding tuple in the db_rules table. Figure 16 shows the state of the tables after the two `attribute_D` tuples from appl_rules have been processed, at the end of Part III.

Step 4 if none of the tuples from db_rules from steps 1 and 3 are found, then the tuple from appl_rules is inserted into the temporary table.

APPL_RULES

r_number	atr_name	arg_name	operation	value	mark
1	attribute_F	attribute_A	=	50	0
1	attribute_F	attribute_B	=	60	0
1	attribute_F	attribute_C	=	1000	0
1	attribute_F	attribute_D	>=	300	0

DB_RULES

r_number	atr_name	arg_name	operation	value	mark
2	attribute_F	attribute_A	=	50	1
2	attribute_F	attribute_B	>	30	1
2	attribute_F	attribute_B	<	100	1
2	attribute_F	attribute_D	<=	300	1
2	attribute_F	attribute_D	>	100	1
2	attribute_F	attribute_E	=	6	0

TEMPORARY

atr_name	operation	value
attribute_A	=	50
attribute_B	=	60
attribute_C	=	1000
attribute_D	>	200
attribute_D	<=	300

Figure 16, Example from Figure 15 at the end of Part III. See the two last tuples in the temporary table, and the marks on the two last tuples in the db_rules table. Arrows indicate from which tuples in appl_rules and db_rules were derived the two last tuples in temporary.

The final case to treat in the discussion of Part II is the one in which the appl_rules tuples have non Case 1 operators, but the tuple with same atr_name in db_rules has operation =. Figure 17 shows a variation of the example in Figure 15 that satisfies this condition.

The way this cases are handled is simple: for every tuple in appl_rules with operation >, <, >=, or <=, if the procedure finds a tuple in db_rules with the same arg_name and operation =, it performs two steps.

In Step 1, it checks if the values conflict. In Figure 17, for example, the db_rules tuple `attribute_D = 300` doesn't conflict with either of the appl_rules tuples `attribute_D > 200` and `attribute_D < 500`. If, on the other hand, the tuple in db_rules had been `attribute_D = 600`, then it would have conflicted with the second of the appl_rules tuples. In case of conflict, the procedure immediately ends returning 0, to signal that the if conditions conflict. If not, proceeds to Step 2.

In Step 2, the procedure checks if the db_rules tuple has been marked. If it has not been, then it marks it and copies the arg_name, operation and value from the db_table into the temporary table. If it has already been marked, then it doesn't do anything. In our example on Figure 17, only the first of the two appl_rules `attribute_D` tuples marks the db_rule `attribute_D` tuple, although both make sure that they don't conflict with it.

After processing the last two tuples on appl_rules in Figure 17, at the end of Part III, the state of the tables is shown in Figure 18.

APPL_RULES

r_number	atr_name	arg_name	operation	value	mark
1	attribute_F	attribute_A	=	50	0
1	attribute_F	attribute_B	=	60	0
1	attribute_F	attribute_C	=	1000	0
1	attribute_F	attribute_D	>	200	0
1	attribute_F	attribute_D	<	500	0

DB_RULES

r_number	atr_name	arg_name	operation	value	mark
2	attribute_F	attribute_A	=	50	1
2	attribute_F	attribute_B	>	30	1
2	attribute_F	attribute_B	<	100	1
2	attribute_F	attribute_D	=	300	0
2	attribute_F	attribute_E	=	6	0

TEMPORARY

atr_name	operation	value
attribute_A	=	50
attribute_B	=	60
attribute_C	=	1000

Figure 17, Variation of example in Figure 15. The attribute_D tuple in db_table has changed. Note that it is not marked anymore, and the the temporary table doesn't have any attribute_D tuple.

APPL_RULES

r_number	atr_name	arg_name	operation	value	mark
1	attribute_F	attribute_A	=	50	0
1	attribute_F	attribute_B	=	60	0
1	attribute_F	attribute_C	=	1000	0
1	attribute_F	attribute_D	>	200	0
1	attribute_F	attribute_D	<	500	0

DB_RULES

r_number	atr_name	arg_name	operation	value	mark
2	attribute_F	attribute_A	=	50	1
2	attribute_F	attribute_B	>	30	1
2	attribute_F	attribute_B	<	100	1
2	attribute_F	attribute_D	=	300	1
2	attribute_F	attribute_E	=	6	0

TEMPORARY

atr_name	operation	value
attribute_A	=	50
attribute_B	=	60
attribute_C	=	1000
attribute_D	=	300

Figure 18, continues example from Figure 17. State of the tables after the two attribute_D tuples in appl_rules have been processed.

Part IV, Scanning the Unmarked source_rules Tuples

Part IV is much simpler than any of the first two parts. It simply goes through the db_rules table looking for the relevant tuples that have not been marked, marks them, and copies them into the temporary table. Returning to our original example, we can see in Figure 14 (which shows the state of the tables after Part III finishes), that only the `attribute_F = 60` tuple in db_rules is still unmarked. The reason for which it is unmarked is that there are no tuples with arg_name `attribute_E` in appl_rules. (If there had only been a tuple of the form `attribute_E < 100` or `attribute_E <= 100` in appl_rules, then the tuple `attribute_E > 60` would also have remained unmarked in db_rules). Figure 19 shows the state of the tables from Figure 14, after Part IV finishes. With this, the CanBeSubsumed algorithm finishes.

APPL_RULES

r_number	atr_name	arg_name	operation	value	mark
1	attribute_F	attribute_A	=	50	0
1	attribute_F	attribute_B	=	60	0
1	attribute_F	attribute_C	=	1000	0
1	attribute_F	attribute_D	>=	300	0

DB_RULES

r_number	atr_name	arg_name	operation	value	mark
2	attribute_F	attribute_A	=	50	1
2	attribute_F	attribute_B	>	30	1
2	attribute_F	attribute_B	<	100	1
2	attribute_F	attribute_D	<=	300	1
2	attribute_F	attribute_D	>	100	1
2	attribute_F	attribute_E	=	6	1

TEMPORARY

atr_name	operation	value
attribute_D	=	300
attribute_A	=	50
attribute_B	=	60
attribute_C	=	1000
attribute_F	=	6

Figure 19, continued from Figure 14. Tables' state after Part III. Note the mark in the High tuple in db_rules. Arrow indicates how last temporary tuple was obtained.

4 The NotEqStuff Procedure.

As explained in section 3.2, the NotEqStuff procedure is called immediately after the CanBeSubsumed procedure finishes, if there were no conflicts found between the *if*-conditions without operation != of the application rule and database rule that are being compared. NotEqStuff finishes the work that CanBeSubsumed left incomplete. Remember that CanBeSubsumed doesn't find the whole intersection of the two rules; it only finds the intersection of their conditions that don't have operation !=. NotEqStuff takes this intersection and updates it by taking into account the != conditions from both the application rule and from the database rule. Just as CanBeSubsumed did, if NotEqStuff finds out that the two rules conflict because of their != conditions, it returns 0. Otherwise it returns 1, and leaves the (this time final) intersection of the two rules in the temporary table.

Before going into the implementation of NotEqStuff it is important to make clear two important points about how the program works:

First, note that there is only one way in which two rules can conflict because of a condition with != operation in one of them. An example will show this. If among the conditions in the application rule there was:

```
attribute_A != 50
```

and among the conditions in the database rule there was:

```
attribute_A = 50
```

then there would be a conflict, and NotEqStuff would return 0. The same thing would have happened if the first condition was from the data base and the second from the application. However, if the second condition had been anything different from that above, there would not have been a conflict. For example, if the second condition had been:

```
attribute_A >= 50
```

then there would be no conflict. The intersection of the two conditions (what would be written into the temporary table) is:

`attribute_A > 50`

Second note the following tricky point: if there is a condition with `!=` operation in the application, AND if a condition from the database conflicts with, THEN it is possible to find this conflict it by looking ONLY on the temporary table. There is no need to look at the db_table. The same thing happens with a condition with operation `!=` from the database: there is no need to look at the appl_rules table, to find if the application rule conflicts with it; it is only necessary to look at the temporary table. The reason for this can be explained in 3 steps:

Step 1, see that if any of the two rules has a condition:

`attribute_A = 50`

then that condition must be in the temporary table by the time NotEqStuff is called. This is easy to see with an example: suppose that the application rule had the condition shown above, and the data base rule had the condition:

`attribute_A < 100`

Then the intersection of the two conditions would have been `attribute_A = 50`, and CanBeSubsumed would have written it into the temporary table before NotEqStuff was called.

Step 2, now remember that if a rule has the condition:

`attribute_A != 50`

then it can not also have the condition:

`attribute_A = 50`

because the conditions within one rule can not conflict.

Step 3, from steps 1 and 2 it is possible to conclude that:

if either the application rule or the source rule (say application) has the condition:

```
attribute_A != 50
```

and the temporary table has the condition:

```
attribute_A = 50
```

then the condition in the temporary table must have come from the other (say database) rule. Therefore, it is possible to find conflicts by looking at the temporary table. QED.

A clever reader might point out that if the application rule had the condition:

```
attribute_A >= 50
```

and the source rule had the condition:

```
attribute_A <= 50
```

then the temporary table would have the condition:

```
attribute_A = 50
```

That is correct, but in that case, neither of the two rules can have the condition:

```
attribute_A != 50
```

because that would conflict with step 2 above: no rule can have the conditions `attribute_A != 50` and `attribute_A >= 50` or the conditions `attribute_A != 50` and `attribute_A <= 50` because the two conditions contradict each other. Therefore the reader's counter example does not invalidate the conclusion reached above.

Having understood the two points above, it is possible to proceed to discuss NotEqStuff's implementation.

Implementation

NotEqStuff takes the appl_rules, db_rules and temporary tables just as CanBeSubsumed left them. It receives the same two arguments that CanBeSubsumed does: the r_number of the application rule and the r_number of the db_rule that it is going to compare. For example, continuing with the example of section 3.3, after CanBeSubsumed returns from successfully comparing rules 1 and 2 from the application and database, the main program calls:

`NotEqStuff(1,2)`

and the state of the tables is that shown in Figure 19. Obviously that example is a trivial one because none of the two rules contains conditions with operation != . NotEqStuff does not change anything and returns 1, meaning that it did not find any conflict.

NotEqStuff first scans all the tuples with operation != from the appl_rules table, and then scans all the tuples with operation != from the db_rules table that are not also in the appl_rules table. Therefore, there is a slight variation in the way that the conditions from the two tables are treated. Whenever NotEqStuff finds a condition with != operation in the appl_rules table, it looks for a tuple with the same operation and value in the db_rules table. If it finds it, it marks it. For example, if it finds the tuple

`attribute_A != 50`

in the appl_rules table, it looks for that same tuple in the db_rules table, and if found, sets the value of its mark column to 1. The reason for doing this is that the same condition should not be processed twice. In other words, once NotEqStuff has made sure that the condition `attribute_A != 50` from the appl_rules table does not cause a conflict; and once it has updated the temporary table to reflect that condition; it must not perform those operations for a second time when it finds the same condition in the db_rules table. That's why it marks all the conditions in the db_rules table with operation != that are also on the appl_rules table. NotEqStuff processes all the tuples with operation != from appl_rules, but only those with operation != and mark 0 from db_rules.

Besides the previous distinction, NotEqStuff executes the following 3 steps on any tuple that it processes, regardless of which of the rules tables it came from.

Step 1, for every tuple with operation != from either the appl_rules or db_rules tables, it looks in the temporary table for a tuple with operation = and the same value. For example for the tuple `attribute_A != 50`

from any of the rules tables, it looks for the the condition

`attribute_A = 50`

in the temporary table. If it finds it, then that is a conflict between the rules because of the two points made in the Introduction of this subsection, and the procedure finishes immediately returning the value 0. If it doesn't find such a condition in the temporary table, the procedure continues with Step 2.

Step 2, for every tuple with operation != from either the appl_rules of db_rules tables, it looks in the temporary table for a condition with operation >= or operation <= and the same value. For example, for the condition

`attribute_A != 50`

discussed above, it looks for either of the conditions

`attribute_A >= 50` or `attribute_A <= 50`

in the temporary table. If it finds one of them (it can't find both), it updates the temporary table to say:

`attribute_A > 50` or `attribute_A < 50`

and finishes operating on that condition (doesn't do any of the following steps). If it doesn't find either of the >= or <= conditions in the temporary table, continues with step 3. 2~

Step 3 the final step is deciding if the condition should be written into the temporary or if it would be redundant to do so. For example, for our `attribute_A != 50` condition, if the temporary table had the condition:

`attribute_A < 40`

or if it had the condition:

`attribute_A > 60`

then it would be redundant to put the condition `attribute_A != 50` in the temporary table, because other conditions already implied it. If, on the other hand the temporary table had the conditions:

`attribute_A > 40` and `attribute_A < 60`

then the `attribute_A != 50` condition should be added to the others by inserting it into the temporary table.

It is not necessary to give detailed examples of the NotEqStuff program like the ones of the CanBeSubsumed one, because it is much simpler. The three steps described above are clearly marked in NotEqStuff's code in Appendix 1.

5 The SemEqOrSub Procedure.

The structure of the SemEqOrSub procedure is similar to Part III of the CanBeSubsumed procedure described in section 3.3. It one by one selects the conditions from the then part of the application rules and classifies them into 4 different types of cases, depending on their operation field: Case 1 for =, 2 for > or >=, 3 for < or <=, and 4 for !=. For every relevant condition in the application rule, it looks for conditions in the database rule, and sees if they conflict. Whenever a conflict is found, the atr_name of the attribute that caused it, along with the r_numbers of the application and database rules that are being compared are stored into the bad table, and the program returns immediately the result 0. If the program reaches the end without finding a conflict it returns 1. See how the program handles the 4 different cases.

Case 1 Whenever there is a condition in appl_rules like the following:

`meta-attribute_A = 50`

there must be an identical condition in db_rules in order to avoid conflict. This is quite easy to see, because if `meta-attribute_A` had any value in db_rules other than 50, then the condition in db_rules would not be equal or subset of the condition in appl_rules. This is the only thing that has to be checked in Case 1. In case of conflict, the *conflict procedure* is performed: insert attribute_A along with rules numbers into bad, and return 0.

Case 2 Whenever there is a condition in appl_rules like one of the following:

`meta-attribute_A >= 50` or `meta-attribute_A > 50`

there are three steps performed, which are marked in the code:

First, see if a condition like:

`meta-attribute_A = X`

exists, where X is obviously an floating point value. If X is smaller than 50 there is obviously a conflict. A conflicts also occurs if X = 50 and the application condition is `meta-attribute_A > 50`. In case of conflict the same *conflict procedure* as above is performed.

Second, if no `meta-attribute_A = X` condition is found in db_rules, look in that same table for a condition of the form:

`meta-attribute_A > Y` or `attribute_A >= Y`

In any case, if Y is smaller than 50 a conflict occurs. There is also a conflict if the database condition is `meta-attribute_A >= 50`, and the application condition is `meta-attribute_A > 50`. In case of conflict perform the standard *conflict procedure*.

Third, if none of the conditions that were searched in the first and second steps are found, then there is also a conflict. The *conflict procedure* is performed.

Case 3 is very similar to Case 2 above. Wherever there is a condition in appl_rules like:

`meta-attribute_A <= 50` or `meta-attribute_A < 50`

Do the same 3 steps as in Case 2, but substitute all > with < and all >= with <=. The steps are marked in the code.

Case 4 is somewhat more complex. Whenever there is a tuple of the form:

`attribute_A != 50`

in appl_rules, there are 5 steps performed, which are marked in the code:

First, look for a tuple of the form:

`attribute_A = 50`

in db_rules. If it is found then there is immediately a conflict, and the standard *conflict procedure* is performed. There is no need to proceed with the next steps.

Second, look for tuples of the form:

`meta-attribute_A > Y` or `meta-attribute_A >= Y`

where Y is an integer. Obviously if the tuple found is `meta-attribute_A >= 50` there is a conflict and the conflict procedure is performed.

If the value of Y is smaller than 50 there is the potential for a conflict. It can't be known if there is actually a conflict until a tuple with the form `meta-attribute_A < Z` or `meta-attribute_A <= Z` is found. (See next step). For the moment the value of a variable called ne_condition set to 1 to indicate the potential conflict. By the same token, if no tuple of the form `attribute_A > Y` or `meta-attribute_A >= Y` is found, then there is also the potential for a conflict, and the variable ne_condition is set to 1 to indicate this.

If, on the other hand, the value of Y is greater than 50, or the tuple found is `meta-attribute_A > 50` then there is no conflict.

Third, look for tuples of the form:

`meta-attribute_A < Z` or `meta-attribute_A <= Z`

Where Z is an integer. Again, if the tuple found is `meta-attribute_A <= 50` there is a conflict and the conflict procedure is performed.

If Z is greater than 50, nothing is done to the ne_condition variable. It is still unknown if there is a conflict, and step 4 has to be performed to know.

On the other hand, if Z is smaller than 50, or the tuple found is `meta-attribute_A < 50` then there is no conflict, and the value of the variable ne_condition is set to 0.

Fourth, look for the condition: `meta-attribute_A != 50`

in db_rules. If it is found, then there is certainty that there can not be a conflict and ne_condition is turned off. Otherwise proceed to step 5.

Fifth, if the value of ne_condition is 1, or if none of the tuples in the db_rules table that were searched in steps 1 through 4 above was found, then there is conflict, and the standard procedure is applied.

At the end, the variable condition is returned.

6 Conclusion.

In this section the Subsumption Algorithm and the 4 principal procedures that implement it have been described in detail. The reader must now understand not only what the Subsumption Algorithm does, but the code in Appendix 1 works. It is reiterated that after the subsumption finishes, the yes_table contains the intersection of the if parts of all the combinations of application and database rules, in which the database's then part is equal or subset of the application's then part, and the no_table contains the intersection of the if parts of all the combinations of rules whose database's then part is NOT equal or subset of the application's then part. The no_table will later be used by the query processing without conversion algorithm in section 4.

PART IV

THE QUERY PROCESSOR

It is important to check that the application and the database 'understand the same thing' when they interpret the conditions on the where clause of a query. For example, in the query:

```
select Instrument_Name
where Trade_Price > 100
```

the Query Processor must make sure that for all the tuples in the database whose a Trade_Price field's nominal value is greater than a hundred, the application and the database agree on what are the values of the Trade_Price's meta-attributes: Trade_Price_Status, and Trade_Price_Currency. In other words, even if the application and the database are not in disagreement what Instrument_Name means (ie, Instrument_Name is a primitive attribute), it would be incorrect if the application was asking for the names of all instruments that traded at a price greater than 100 USdollars, and the database returned the names of all instruments that traded at Trade_Price greater than 100 pesetas. Therefore, although it is not as obvious as checking for semantic conflicts in the select clause, it important to also check for semantic conflicts in the where clause.

These two operations: checking for conflicts in the where clause, and checking for conflicts in the where clause are done in two steps by the Query Processor. The next two subsections explain them separately.

1 Checking the Where Clause

The where program is the first executed by the Query Processor. It is called by the user interface AFTER the Subsumption Algorithm has been performed, the user has entered a query, and the query has been parsed by the query parser. It has 5 non_trivial procedures:

main, Prepare_Query_Conditions, Arrange_Query_Conditions, ExecuteSubsumpt, QueryRebuild; and 11 trivial ones: Declare_Check_Query_Where_Cursors, Copy_No_To_Intermediate, Copy_C_To_Query_Conditions, Insert_Into_Check_List, Number_Where_Conditions, Copy_No_To_No_Conditions, Declare_Build_Cursor, Clear_Temporary, Clear_No_Conditions, Clear_Tables, and Copy_Temp_To_Interm. Besides, it uses the CanBeSubsumed and NotEqStuff procedures from the Subsumption Algorithm, along with the DeclareSubsumptionCursors procedure explained in Section 3.

The Clear_Temporary, Clear_No_Conditions, and Clear_Tables procedures are used, as their names indicate, to erase all the contents of some tables in the database. The Declare_Check_Query_Where_Cursors, and Declare_Build_Cursor procedures are used to declare the cursors used in the program. Declare_Check_Query_Where_Cursors must be at the top of the file. The following is a picture of how the procedures call each other:

main

Prepare_Query_Conditions Arrange_Query_Conditions

Execute_Subsumpt Query_Rebuild

CanBeSubsumed SemanticEqOrSub NotEqStuff

Their where program starts by calling the main procedure whenever there is need to check the where clause of a query. Suppose the user makes the query:

```
select Instrument_Name
from db_table
where Volume <= 10
      or Trade_Price < 100 AND Volume > 10 AND Exchange = 'nyse'
```

Where Instrument_Name and Exchange are primitive attributes, and Trade_Price and

Volume are non-primitive (the meta-attributes of Trade_Price could be Trade_Price_Currency and Trade_Price_Status; and the meta-attribute of Volume could be Volume_Scale.) The query is parsed by the query parser into the following a_list, t_list and c_list tables:

A_LIST

attrib
Instrument_Name

T_LIST

tabl
db_table

C_LIST

c_number	attr_name	operation	value	mark
1	Volume	<=	10	0
2	Trade_Price	<	100	0
2	Volume	>	10	0
2	Exchange	=	nyse	0

Now, suppose that the bad table contains the following information:

BAD

appl_rule	db_rule	np_attr
1	2	TradePrice
3	3	Volume

The main procedure executes each of the following 4 parts (which are shown in the code in Appendix 2.):

PART 1: Declares the current database and all the cursors necessary for the algorithm.

PART 2. Calls the Prepare_Query_Conditions procedure, which separates the non-primitive attributes that can cause semantic conflicts in the where clause, from the primitive and non-primitive attributes that can not cause semantic conflicts. It does so by filling two new tables: the check_list table, and the quer_cnd_tmp table.

The check_list table is a table with only one field, called attrib, into which the names of all non-primitive attributes in the where clause that also appear in the bad table are inserted. In the example above, both Trade_Price and Volume appear in both the c_list (where clause) and the bad table. Therefore, Prepare_Query_Conditions puts into the check_list table the following information:

CHECK_LIST

attrib
Trade_Price
Volume

The quer_cnd_temp table has exactly the same fields as the query_conditions table (which was discussed in the Context Mediator chapter): r_number, atr_name, arg_name, operation, primitive, domain_type, and mark. Prepare_Query_Conditions copies into this table all the tuples that appear in the c_list which refer to primitive attributes or non-primitive attributes that don't appear in the bad table, according to the following rules: the c_number in c_list becomes the r_number in quer_cnd_tmp; the attr_name in c_list becomes the arg_name in quer_cnd_tmp; the operation is copied to the field with the same name, and the value field is copied into quer_cnd_tmp's primitive field. The other 3 fields in quer_cnd_tmp are filled with an empty string, an A and a 0. For example, in the c_list shown above, there is only one tuple that refers to a non-primitive attribute: Exchange. The rest of the tuple in c_list refer to the non-primitive attributes Trade_Price and Exchange, which also appear in the bad table. Because of that, the following quer_cnd_tmp table is produced:

QUER_CND_TMP

r_number	atr_name	arg_name	operation	primitive	a_domain	mark
2		Exchange	=	nyse	A	0

PART 3. Loops around the check_list table, and performs the following 5 steps for every element in that table:

Step 1. Calls Arrange_Where_Conditions, which copies the contents of the quer_cnd_tmp table into the query_conditions table, making sure that the r_numbers are ordered in consecutive integers, starting from 1. For example, for the quer_cnd_tmp table shown above, the Arrange_Where_Conditions notices that there is no condition with r_number = 1. Therefore, when copying the tuple from quer_cnd_tmp into query_conditions, Arrange_Where_Conditions changes the value of the r_number from 2 to 1, producing the following table:

QUERY_CONDITIONS

r_number	atr_name	arg_name	operation	primitive	a_domain	mark
1		Exchange	=	nyse	A	0

This query_conditions table is ready to be subsumed.

Step 2. Calls the Copy_No_To_No_Conditions to copy the conditions in the no_table that refer to the attribute in the check_list into the no_conditions table. (Remember that the no_conditions table was discussed in the Context Mediator chapter, and has the same schema as the query_conditions table.) For example, suppose that the no_table contains the following information:

NO_TABLE

r_number	s_number	atr_name	arg_name	operation	primitive
1	2	Trade_Price	Instrument_Type	=	future
3	3	Volume	Exchange	=	nyse

Copy_No_To_No_Conditions first checks in the bad table that Trade_Price might conflict because of rules 1 and 2, and then copies the conditions in the bad table indexed by rules 1 and 2 into the no_conditions table, producing the following:

NO_CONDITIONS

r_number	atr_name	arg_name	operation	primitive	a_domain	mark
1		Instrument_Type	=	future	A	0

Which is another table ready for subsumption.

Step 3. Calls Execute_Subsumpt, which finds the intersection of the conditions in the Query_Conditions table and the conditions in the No_Conditions table, and puts it in the Intermediate table. After Step 3, the Intermediate table contains:

INTERMEDIATE

r_number	atr_name	arg_name	operation	primitive
1		Instrument_Type	=	future
1		Exchange	=	nyse

Step 4. Calls Query_Rebuild, which makes an intermediate query to the database based on the conditions in the Intermediate table. For example, the Intermediate table above only has one OR condition (ie, there is only r_number 1), so the Query_Rebuild procedure produces only one intermediate query:

```
select Trade_Price
from db_table
where (Instrument_Type = 'future' and Exchange = 'nyse')
```

If the query returns anything but Null, then a semantic conflict occurs, and the procedure breaks from the loop in Part III, and continues to Part IV (see ahead) where it finishes in failure. Otherwise, if the result of the query is Null, the procedure continues with Step 5.

Step 5. Once that it has been determined that Trade_Price can not cause a semantic conflict in the where clause, all conditions dealing with Trade_Price are copied from the c_list to the quer_cnd_tmp table. This is done by calling the procedure Copy_C_To_Query_Conditions with the argument Trade_Price. This produces the following quer_cnd_tmp table:

QUER_CND_TMP

r_number	atr_name	arg_name	operation	primitive	a_domain	mark
2		Exchange	=	nyse	A	0
2		Trade_Price	<	100	A	0

In this moment, Part III LOOPS AND STARTS AGAIN for the next attribute in the check_list table: Volume. To finish the example, the 4 steps executed will be shown briefly:

Step 1. The Query_Conditions table is erased, and Arrange_Conditions copies the quer_cnd_tmp table into Query_Conditions, making sure that the r_numbers are ordered correctly:

QUERY_CONDITIONS

r_number	atr_name	arg_name	operation	primitive	a_domain	mark
1		Exchange	=	nyse	A	0
1		Trade_Price	<	100	A	0

Step 2. The no_conditions table is erased, and the attributes from the no_table dealing with Volume are copied into it, in the correct r_number ordering (remember the no_table shown above):

NO_CONDITIONS

r_number	atr_name	arg_name	operation	primitive	a_domain	mark
1		Exchange	=	nyse	A	0

Step 3. The Query_Conditions table and no_conditions table are subsumed, and the result put in the Intermediate table:

INTERMEDIATE

r_number	atr_name	arg_name	operation	primitive
1		Trade_Price	<	100
1		Exchange	=	nyse

Step 4. An intermediate query is built to see if Volume can cause a semantic conflict:

```
select Volume
from db_table
where (Trade_Price < 100 and Exchange = 'nyse')
```

If the intermediate query returns Null, then the original query can be made without semantic problems, and if not then it can not. After step 4, because there are no more tuples left in the check_query table, then the procedure continues with Part IV instead of looping around Part III again.

PART IV. If no conflict was found in Part III, Part IV inserts the value 1 in the procedure_result table, and finishes. If a conflict was found, then it inserts 0, and terminates.

This is the end of the where program. After it, if no semantic conflicts were found in the where clause, the Query Processor continues with the select procedure.

2 Checking the Select Clause

The select program is called by the user interface AFTER it has made sure that the application and the database understand the same thing when interpreting the conditions in the query's where clause. It has 4 non_trivial procedures: main, CheckQuerySelect, ExecuteSubsumpt, and QueryRebuild; and 7 trivial ones: Declare_Check_Query_Select_Cursors, Declare_Build_Cursor, Clear_Temporary, Clear_No_Conditions, Clear_Tables, Copy_C_To_No_Conditions, and Copy_Temp_To_Interm. Besides, it uses the CanBeSubsumed and NotEqStuff procedures from the Subsumption algorithm, along with the DeclareSubsumptionCursors procedure explained in Section 3. The Clear_Temporary, Clear_No_Conditions, and Clear_Tables procedures are used, as their names indicate, to erase all the contents of some tables in the database. The Declare_Check_Query_Select_Cursors, and Declare_Build_Cursor procedures are used to declare the cursors used in the program. Declare_Check_Query_Select_Cursors must be at the top of the file.The following is a picture of how the procedures call each other:

main

CheckQuerySelect Execute_Subsumpt Query_Rebuild

CanBeSubsumed SemanticEqOrSub NotEqStuff

Copy_C_To_Query_Conditions is used to copy the contents of the c_list table into the query_conditions table, as explained in Section 1. It returns the number of or conditions in the where clause. The reason for which this is done is that later in the program the where clause is subsumed with the no_table, for which the query_conditions table, and not the c_list table, has the correct schema.. (The query_conditions table has the same schema but different name as the appl_rules and db_rules tables.) Suppose that the user wants to make the query:

```
select TradePrice
from db_table
where Profits > 100 or CEO_pay >= 30000000
```

Then, after parsing, c_list contains:

C_LIST

c_number	attr_name	operation	value	mark
1	Profits	>	100	0
2	CEO_pay	>=	3000000	0

When called, Copy_C_To_No_Conditions puts into the query_conditions table the following information, ready to be subsumed:

QUERY_CONDITIONS

r_number	atr_name	arg_name	operation	value	a_domain	mark
1		Profits	>	100	A	0
2		CEO_pay	>=	3000000	A	0

The Copy_Temp_To_Interm program copies the content of the temporary table into the intermediate table, which is used when rebuilding queries.

The next 4 subsections deal with the 4 non_trivial procedures in the select program.

1 The main Procedure

main is a somewhat simple procedure: it starts by declaring the cursors needed in the select program, and copying the c_list table into the query_conditions table. This copying is necessary because when later in the program the no_table and the c_list are subsumed. They have to be copied to the no_conditions and query_conditions tables to be able to use the old subsumption procedures that were explained in section 3. It has already explained how Copy_C_To_No_Conditions works. It not only copies the c_list to query_conditions, but also tells main how many conditions separated by ORs are there in the c_list.

The next thing that main does is start fetching the attributes from the select statement in the query, which have been put into the a_list table by the query parser. Once it gets one of this attributes it checks to see if it is in the where clause. The reason is that, as the select program is only called after the where program has returned a positive result, no attribute that is in the where clause can possibly produce a semantic conflict. Therefore the program skips the following steps for all the attributes in the select clause that are simultaneously in the where clause.

If an attribute in the select clause is not in the where clause, main calls the procedure CheckQuerySelect with that attribute's name as an argument. CheckQuerySelect returns to main a count of how many times that attribute appears in the bad table. In other words, it tells main how many times that attribute caused a semantic conflict when comparing an application and a source rules. CheckQuerySelect also copies the relevant contents of the no_table into the no_conditions table (see next subsection), preparing the road for a subsumption of the relevant part of the no_table and the c_list to be performed.

If CheckQuerySelect returns a number larger than 0, then that means that a subsumption has to be performed just as explained above, in order to find the intersection of the where table and the relevant part of the no_table. Because of that this subsumption only uses the CanBeSubsumed and NotEqStuff procedures, but not the SemEqOrSub procedure. This last procedure is not used because there is not any then part to worry about here: the Select program does not compare rules, but simply conditions. The intersection of the where clause and no_table is found when main calls the procedure Execute_Subsumpt with two arguments: the number of ORed conditions in the c_list, and the number of ORed conditions in the no_table. Note that this is equivalent to telling the original Subsumption Algorithm how many rules there were in the application and and database repositories. This will be discussed further in the Execute_Subsumption subsection. Execute_Subsumpt puts the intersection of these two sets of conditions in the Intermediate table, from which a query can be built to the database, and returns how many ORed conditions it put into that table.

The main procedure uses the conditions in the Intermediate table to build *intermediate queries*, from which it discovers if the original query will return results in the wrong context. To make this intermediate queries, main calls the procedure Query_Rebuild, which returns 1 if the intermediate query does not find any semantically incorrect data in the database, and 0 if it does. Query_Rebuild makes one intermediate query to the database for each of the ORed conditions in the Intermediate table.

Query_Rebuild returns 1 if all the intermediate queries made to the database return null result, but 0 if any of them doesn't. When a 0 is returned, main immediately terminates putting the value 0 in the procedure_result table. This value can later be seen by the user interface to know that a semantic conflict will occur when making the user's query to the database. The user can still ask the interface to make the original query requested, at the expense of having some semantic conflicts. If, on the other hand, Query_Rebuild returns 1, then that means that there is no tuple in the database in which the attribute from the select clause being checked would return semantically incorrect values. In that case, main can continue fetching the next attribute in the select clause from the a_list table to see if it also won't produce a semantic conflict. When the a_list has been completely scanned, and none of the attributes there has produced a semantic conflict, than the user's query can be made without problems. In that case, main terminates putting the value 1 in the procedure_result table for user interface to see.

2 The CheckQuerySelect Procedure

CheckQuerySelect receives from main the name of an attribute in the select list that is not in the where clause, and returns how many times that attribute appears in the bad table. For example, if main calls:

```
CheckQuerySelect(Trade_Price)
```

and the bad table contains the following information:

BAD

appl_rule	db_rule	np_attr
2	1	Trade_Price
3	3	Trade_Price

CheckQuerySelect returns the value 2 because earnings made the combination of rules 2 and 1 fail, as well as the combination of rules 3 and 3. Suppose that the no_table contains the following information:

NO_TABLE

r_number	s_number	atr_name	arg_name	operation	primitive
2	1	Trade_Price	debt	>	20
2	1	Trade_Price	assets	>=	100
3	3	Trade_Price	debt	<	15
3	3	Trade_Price	CEO_pay	>=	30000000

Then, CheckQuerySelect puts into the no_conditions table the following information:

NO_CONDITIONS

r_number	atr_name	arg_name	operation	primitive	a_domain	mark
1		debt	>	20	A	0
1		assets	>=	100	A	0
2		debt	<	15	A	0
2		CEO_pay	>=	30000000	A	0

which, along with the information put into the query_conditions table by Copy_D_To_Db, leaves everything set up to subsume the rules in both tables. CheckQuerySelect returns to main the value 2, which is the amount of times that the Trade_Price attribute appears in the bad table.

3 The Execute_Subsumption Procedure

The Execute_Subsumption procedure is very similar to the main procedure of the Subsumption Algorithm explained in the subsumption section. It loops through the combination of 'rules' in the query_conditions and no_conditions tables; calls CanBeSubsumed for every combination of rules, and if successful calls NotEqStuff; and copies the result of this subsumption (the temporary table) into a new table called the Intermediate table. However, there are some differences. Remember that the original subsumption's main procedure started by calling the subsumption procedures Number_Application_Rules and Number_Query_Conditions to find out how many rules there were in the application and database. In this case, those two numbers are given to Execute_Subsumption as arguments by main. Also, the Execute_Subsumption does not call SemanticEqOrSub to decide if the contents of the temporary table are copied into the yes_table or no_table, but instead always copies them to the Intermediate table. Finally, Execute_Subsumpt returns a count of how many combinations of conditions from the query_conditions and no_conditions tables did not conflict, as opposite to the original subsumption's main program which does not return any result.

The Execute_Subsumption procedure finds the intersection of the conditions in the no_conditions and query_conditions tables, just as the original Subsumption Algorithm found the intersection of the conditions in the appl_rules and db_rules tables. To do this, Execute_Subsumpt calls CanBeSubsumed for every possible combination of no_conditions condition and query_conditions condition. If CanBeSubsumed returns 1, then Execute_Subsumption calls NotEqStuff for those same rules. If this also returns 1 then the intersection of the rules is in the temporary table, and Execute_Subsumpt copies it into the Intermediate table.

The Intermediate table is like the yes and no tables of the original subsumption, but instead of having two rule number fields (one for the application and one for the database), it only has one such number, that increases every time two rules intersect. When Execute_Subsumption finishes, it returns this number to the main procedure; literally, it means how many intermediate queries will Query_Rebuild have to make to the database (in the next step of the select program.) For example, when Execute_Subsumpt finds the intersection of the query_conditions and no_conditions tables shown above, it leaves in the temporary table the information:

INTERMEDIATE

r_number	atr_name	operation	primitive
.....	.....	.....	.....
1	debt	>	20
1	assets	>=	100
1	Profits	>	100
2	debt	>	20
2	assets	>=	100
2	CEO_pay	>=	30000000
3	debt	<	15
3	CEO_pays	>=	30000000
3	Profits	>	100
4	debt	<	15
4	CEO_pay	>=	30000000
.....	.....	.....	.....

And returns the number 4 that is the number of ORed conditions that the Intermediate table contains. In the next section it will be shown how this information is used to make 4 intermediate queries:

```
select Trade_Price      from query_conditions
                 where (debt > 20 and assets > 100 and Profits > 100)


select Trade_Price      from query_conditions
                 where (debt > 20 and assets >= 100 and CEO_pay >= 30000000)
```

```
select Trade_Price      from query_conditions
            where(debt < 15 and CEO_pay >= 30000000 and Profits > 100)


select Trade_Price      from query_conditions
            where(debt < 15 and CEO_pay >= 30000000)
```

which are used to find out if there are semantic conflicts in the select clause: if any of these queries returns a non-null result, the original query will return semantically incorrect results.

4 The Query_Rebuild Procedure

This is a very simple procedure. It is called by the main program with two arguments: the name of the attribute in the select clause which is being checked to be semantically correct, and the number of ORed conditions in the Intermediate table (the number returned by Execute_Subsumption in subsection 3). This is the same number of intermediate queries that the procedure will make.

What the procedure does is very simple: For every different r_number in the Intermediate table, it builds a string that contains a select statement and the name of the attribute in the select clause of the original query (in this case Earnings):

```
select Earnings
```

which is followed by a from statement indicating the table from which the information should be obtained (in this case the db_table table):

```
from db_table
```

followed by the statements from the Intermediate table with the3 current r_number arranged in the following manner: all statements with the current r_number are ANDed together and put inside parenthesis. For example, for the conditions in the Intermediate table above with r_number = 1, Query_Rebuild produces the string:

```
select Earnings
from query_conditions
        where (debt > 20 and assets > 100 and Profits > 100)
```

Which it later converts to an intermediate query and executes it. If the query returns any result different than NULL, Query_Rebuild returns 0 because the original query would return semantically incorrect results if executed. If, on the other hand, the previous intermediate query returns a null result, similar intermediate queries are built and executed for the conditions in the Intermediate table with r_numbers 2, 3, and 4. If any of these queries returns a non-null result, Query_Rebuild returns 0. If all return null results, Query_Rebuild returns 1.

4.3 Conclusion

We have explained how the Where and Select Programs work. They always are called after the query has been parsed into the a_list, t_list and c_list tables, and they return their result to the jam interface via the procedure_result table.

For every query, the jam interface must first call the Where Program and then the Select Program. If both leave a 1 result in the procedure_result table, than the query won't return any tuple that causes a semantic conflict from the db_table. If the Where Program returns a 0, the query can not be made, because the application and the database would misunderstand each other when interpreting the conditions used to select the tuples that the query returns.. If only the Select program returns 0, then the query also can not be made, because the database would return tuples with different meaning than the application expects.

PART V

CONCLUSION AND FUTURE RESEARCH

This thesis has presented a demonstrable implementation of the ideas proposed in [SM'91]. The system allows the application and database to define independent sets of rules for deriving their semantic contexts, and correctly subsumes them. By taking advantage of the information produced by the Subsumption Algorithm, the Query Processor checks that no query made by the application to the database will return semantically incorrect results, before the query is performed. With this, the primary objective of automatically avoiding semantic conflicts in a source-receiver model has been accomplished.

There are some short comings to this implementation. The most important is that the source and all the system tables have to be in the same database, limiting the usefulness of the system. Ideally, the system tables and the source should be in different parts of a network, in different databases, and some form of Remote Procedure Calls would be used to query the source.

There is also a limit on how many intermediate queries can be done by any call to the Where Program or the Select Program. The problem is that every intermediate query requires building an 'ad-hoc' ESQL cursor, which can not be destroyed. There is a limit to how many cursors can be created in ESQL, and, if a certain query requires many intermediate queries to check either its where clause or its select list, there is the possibility that the system could run out of cursors and crash. I have never seen this problem happen, but would not be surprised if it did.

Another problem is that the system is not optimized, and could be very slow in real applications.

Future research might focus on implementing *conversion functions* and the necessary *ontologies*, which would make possible to convert data from one semantic form to another. A system with these characteristics, would change all the semantically incorrect tuples that a query returns into an equivalent, semantically correct form. This would be a big improvement to the current implementation, which forbids all queries that would produce semantic conflicts.

A further improvement to the system will probably allow rules with other functions besides the current =, !=, >, <, >=, and <=. Longer term research will probably try to implement some form of this system into real life situations, and market it.

PART VI

Appendix 1

```
#include <stdio.h>
#include <string.h>
$include sqlca;
$include sqlda;

/* ******************************************************************   */
/* THIS IS THE CSQL SUBSUMPTION ALGORITHM                               */
/* Algorithms inspired by Professor Madnik and Dr. Siegel, designed by  */
/* Andrew Leung, and implemented by Francisco Madero at MIT's Sloan     */
/* Scool of Management.                         Summer 1992.            */
/* ******************************************************************   */

/* Procedure DeclareSubsumptionCursors declares all the cursors that are */
/* used in the subsumption algorithm. It is called only once by the main */
/* program, before anything else is done.                                */

/* Note, this procedore MUST BE PUT AT THE TOP OF THE FILE, because ESQL */
/* is very picky. It doesn't like a procedure using cursors declared in  */
/* another procedure written after it.                                   */

DeclareSubsumptionCursors()
{
        $char   str1[300];
        $char   str2[300];
        $char   st_fin[300];
        $char   str_eq[300];
        $char   str_gt[300];
        $char   str_get[300];

        $char   check_str1[300];
        $char   check_str2[300];

        $char   str3[300];
        $char   str4[300];
        $char   str5[300];
        $char   st_fin1[300];
        $char   stteq[300];
        $char   sttgt[300];
        $char   str_temporary[300];


        strcpy (check_str1, "select atr_name from appl_rules where r_number = ?");

        strcpy (check_str2, "select atr_name from db_rules where r_number = ?");

        $prepare query_chk1 from $check_str1;
        print_status("Prepare q_chk1");
        $declare cur_chk1 cursor for query_chk1;
        print_status("Declare  q_chk1");

        $prepare query_chk2 from $check_str2;
        print_status("Prepare q_chk2");
        $declare cur_chk2 cursor for query_chk2;
        print_status("Declare  q_chk2");

        strcpy (str1,
                "select atr_name, arg_name, operation, primitive ");
        strcat (str1, " from appl_rules ");
        strcat (str1, " where r_number = ? AND domain_type = ?");

        $prepare query1 from $str1;
        print_status("P1");
        $declare cur1 cursor for query1;
        print_status("A1");

        strcpy (str2,
```

```
        "select atr_name, arg_name, operation, primitive ");
strcat (str2, " from appl_rules ");
strcat (str2, " where r_number = ? AND domain_type = ? ");
strcat (str2, " AND operation = ? OR r_number = ? ");
strcat (str2, " AND domain_type = ? AND operation = ? ");

$prepare query2 from $str2;
 print_status("P2");
$declare cur2 cursor for query2;
print_status("A2");


strcpy (st_fin,
        "select atr_name, arg_name, operation, primitive ");
strcat (st_fin, " from db_rules ");
strcat (st_fin, " where r_number = ? AND mark = 0");
strcat (st_fin, " AND domain_type = ? AND operation != ?");

$prepare query_fin from $st_fin;
print_status("P3");
$declare cursor_final cursor for query_fin;
print_status("A3");

strcpy (str_eq,
        "select primitive, mark from db_rules ");
strcat (str_eq, " where r_number = ? AND atr_name = ? AND ");
strcat (str_eq, " arg_name = ? AND domain_type = ? ");
strcat (str_eq, " AND operation = ?");

$prepare query_eq from $str_eq;
 print_status("P4");
$declare cur_eq cursor for query_eq;
print_status("A4");

$prepare query_neq from $str_eq;
$declare cur_neq cursor for query_neq;
print_status("A55");

strcpy (str_gt,
         "select operation, primitive, mark from db_rules ");
strcat (str_gt, " where r_number = ? AND atr_name = ? AND ");
strcat (str_gt, " arg_name = ? AND domain_type = ? ");
strcat (str_gt, " AND operation = ? OR  ");
strcat (str_gt, " r_number = ? AND atr_name = ? AND ");
strcat (str_gt, " arg_name = ? AND domain_type = ? ");
strcat (str_gt, " AND operation = ?");

$prepare query_gt from $str_gt;
print_status("P5");

$declare cur_gt cursor for query_gt;
print_status("A21");

$prepare query_templ from $str_gt;
$declare cur_templ cursor for query_templ;
print_status("A23");

$prepare query_lt from $str_gt;
$declare cur_lt cursor for query_lt;
print_status("A5");

strcpy (str_get,
         "select operation, primitive, mark from db_rules ");
strcat (str_get, " where r_number = ? AND atr_name = ? AND ");
strcat (str_get, " arg_name = ? AND domain_type = ? ");
strcat (str_get, " AND operation = ? ");
```

```
$prepare query_get from $str_get;
print_status("P6");
$declare cur_get cursor for query_get;
print_status("A24");

strcpy (str3,
        "select atr_name, arg_name, operation, primitive ");
strcat (str3, " from appl_rules ");
strcat (str3, " where r_number = ? AND domain_type = ?");
strcat (str3, " AND operation = ?");

$prepare query3 from $str3;
print_status("P13");
$declare cur3 cursor for query3;
print_status("A13");

strcpy (str4,
        "select atr_name, arg_name, operation, primitive ");
strcat (str4, " from db_rules ");
strcat (str4, " where r_number = ? AND domain_type = ?");
strcat (str4, " AND operation = ? AND mark = 0");

$prepare query4 from $str4;
print_status("P14");
$declare cur4 cursor for query4;
print_status("A14");

strcpy (str5, "select primitive ");
strcat (str5, " from db_rules ");
strcat (str5, " where r_number = ? AND mark = 0 ");
strcat (str5, " AND atr_name = ? AND arg_name = ? ");
strcat (str5, " AND domain_type = ? AND operation = ?");
strcat (str5, " AND primitive = ?");

$prepare query5 from $str5;
print_status("P15");
$declare cur5 cursor for query5;
print_status("A15");

strcpy (stteq, "select primitive from temporary ");
strcat (stteq, " where atr_name = ? AND ");
strcat (stteq, " arg_name = ? ");
strcat (stteq, "AND operation = ? AND primitive = ?");

$prepare query_Teq from $stteq;
print_status("P16");
$declare cur_Teq cursor for query_Teq;
print_status("A16");

strcpy (sttgt, "select operation, primitive from temporary ");
strcat (sttgt, " where atr_name = ? AND ");
strcat (sttgt, " arg_name = ? ");
strcat (sttgt, " AND operation = ? OR  ");
strcat (sttgt, " atr_name = ? AND ");
strcat (sttgt, " arg_name = ? ");
strcat (sttgt, " AND operation = ?");

$prepare query_Tgt from $sttgt;
print_status("P17");
$declare cur_Tgt cursor for query_Tgt;
print_status("A17");


$prepare query_Tlt from $sttgt;
print_status("P18");
```

```
        $declare cur_Tlt cursor for query_Tlt;
        print_status("A18");


         $declare cur_deltemp cursor for
                  select atr_name, arg_name, operation, primitive
                         from temporary;
        print_status("A19");


        $declare cur_applt cursor for select unique r_number from
                                appl_rules;
        print_status("Declare APPLT");

        $declare cur_sourc cursor for select unique r_number from
                                db_rules;
        print_status("Declare SOURC");

 }


main ()
 {
        $char   t_attr[30];
        $char   t_arg[30];
        $char   t_op[5];
        $char   t_val[30];
        $long   a;
        $long   b;
        $long   x;
        $long   y;
        int     sem_case;
        int     result;


        /* PART I, SETTING THE ENVIRONMENT                                */
        /* The fisrt thing that has to be done is to declare the current  */
        /* database and all the cursors that will be used by all the      */
        /* programs in the file. THIS MUST BE DONE ONLY ONCE              */

        $database cdrdb;
        DeclareSubsumptionCursors();


        /* PART II,  SUBSUMPTION ALGORITHM                                */
        /*                                                                */
        /* Step I find out how many rules there are in the application    */
        /*         and in the data base context repositories              */

        a = Number_Application_Rules();
        b = Number_Db_Rules();
        printf("RULES a= %i, b = %i\n", a, b);


        /* Step II perform a double loop to compare every combination of   */
        /*          application rule and source rule                       */

        for(x=1; x<=a; x++)
        {

        for(y=1; y<=b; y++)

         {

                /* FIRST clear temporary table */
```

```
$delete from temporary;
print_status("clear temporary");


/* SECOND Find intersection of if conditions disregarding */
/*        not_equals by calling CanBeSubsumed             */

result = CanBeSubsumed(x,y);
printf ("SUBSUMPT %i, %i, %i\n", result, x, y);


/*       If they are disjoint here don't do anything else.  */
/*       Otherwise find the whole intersection by calling   */
/*       NotEqStuff                                         */

if (result == 1)
  {
        result = NotEqualStuff(x,y);
        printf("NEQ %i\n", result);
  }


/* If not_equals make rules disjoint dont' do anything else.*/
/* If not, temporary table has the intersection of the      */
/* if conditions and we have to figure out if we copy it to */
/* the yes table or to the no table.                        */

if (result == 1)
{

  /* THIRD compare then conditions by calling SemEqOrSub        */
  /*       sem_case is 1 if source's then condition is equal or */
  /*       subset of application's then condition. If it is not, */
  /*       bad table is automatically updated by SemanticEqOrSub */

  sem_case = SemanticEqOrSub(x,y);
  printf("SEMANTIC %i, %i, %i\n", sem_case, x, y);


  /* FOURTH copy temporary table into yes_table if semcase = 1  */
  /*          or into no_table if semcase = 0.                  */

  $open cur_deltemp;
  print_status("3");

  while (1)
    {
     $fetch cur_deltemp
        into $t_attr, $t_arg, $t_op, $t_val;
     print_status("Fetch deltemp");

     if (sqlca.sqlcode == SQLNOTFOUND)
        {
          $close cur_deltemp;
          break;
        }

     if (sem_case == 1)
        {
         $insert into yes_table
            values ($x, $y, $t_attr, $t_arg, $t_op, $t_val, 0);
         print_status("insert into yes");
        }


     else if (sem_case == 0)
```

```
                            {
                             $insert into no_table
                                values ($x, $y, $t_attr, $t_arg, $t_op, $t_val, 0);
                             print_status("insert into no");
                            }
                        }
                   }
              }
         }
}


/* This procedure scans appl_rules and returns how many rules there are in it */

long Number_Application_Rules()
{
        $long ar_num;
        long count;


        count = 0;
        $open cur_applt;
        print_status("Open APPLT");
        while (1)
        {
          $fetch cur_applt into $ar_num;
          printf("FETCH APPLT %i, %i", sqlca.sqlcode, ar_num);
          print_status("Fetch APPLT");
          if (sqlca.sqlcode == SQLNOTFOUND)
                break;
          if (ar_num > count)
                count = ar_num;
        }
        return(count);
}


/* This procedure scans db_rules and returns how many rules there are in it */

long Number_Db_Rules()
{
        $long sr_num;
        long count;


        count = 0;
        $open cur_sourc;
        print_status("Open SOURC");
        while (1)
        {
          $fetch cur_sourc into $sr_num;
           printf("FETCH SOURC %i, %i", sqlca.sqlcode, sr_num);
          print_status("Fetch SOURC");
          if (sqlca.sqlcode == SQLNOTFOUND)
                break;
          if (sr_num > count)
                count = sr_num;
        }
        return(count);
}

int CanBeSubsumed(x,y)
        $long x;
        $long y;
        {
```

```
$char   a_attr[30];
$char   b_attr[30];
$char   a_op[5];
$char   s_op[5];
$char   a_val[30];
$char   s_val[30];
$char   a_nam[30];

$char   str_sul[300];
$char   str_clrtmp[300];
$char   *dom;
$char   dummy[30];
$long   s_mark;
int     condition;
int     comp;
int counter;

int cnt;
int not_found;
int inserted;

cnt = 1;
dom = "A";
condition = 1;

strcpy (str_sul, "update db_rules ");
strcat (str_sul, " set mark = 1 ");
strcat (str_sul, " where r_number = ? AND atr_name = ? AND ");
strcat (str_sul, " arg_name = ? AND domain_type = ? AND ");
strcat (str_sul, " operation = ? AND primitive = ?");

strcpy (str_clrtmp, "update db_rules ");
strcat (str_clrtmp, " set mark = 0 ");
strcat (str_clrtmp, " where r_number = ? ");



$open cur1 using $x, $dom;
print_status("4");

$open cur2 using $x, $dom, ">=  ", $x, $dom, "<=  ";
print_status("5");

$open cursor_final using $y, "A", "!=  ";
print_status("6");

$prepare updte_11 from $str_sul;
$prepare updte_sul from $str_sul;

$prepare update_tmp from $str_clrtmp;
$execute update_tmp using $y;



/*              PART I                                 */
/* USES CUR_CHK1 AND CUR_CHK2 TO SEE IF THE TWO        */
/* RULES REFER TO THE SAME NON-PRIMITIVE ATTRIBUTE     */
/* BY CHECKING THAT THE VALUES OF THEIR ATR_NAME       */
/* FIELDS ARE EQUAL                                    */
$open cur_chk1 using $x;
print_status ("Open cur chk1");
$fetch cur_chk1 into $a_attr;
print_status ("Open cur chk1");
if (sqlca.sqlcode != SQLNOTFOUND)
{
        $open cur_chk2 using $y;
```

```
        print_status ("Open cur chk2");
        $fetch cur_chk2 into $b_attr;
        print_status ("Open cur chk2");
        if (sqlca.sqlcode != SQLNOTFOUND)
        {
          if (strcmp(a_attr, b_attr) != 0)
                return(0);
        }
}



/*                   PART    II                          */
/*                                                       */
/* THE FIRST LOOP IN THE PROGRAM CHECKS FOR THE VERY     */
/* SPECIAL CASES LIKE THE FOLLOWING:                     */
/* APPL_RULES contains a tuple saying:  price >= 50      */
/* AND                                                   */
/* DB_RULES contains the tuple: price <= 50              */
/* IN THAT CASE, STORES INTO THE temporay TABLE THE      */
/* TUPLE:  price = 50,                                   */
/* AND MARKS IN THE db_rules TABLE THE TUPLE SAYING      */
/* price <= 50 AS WELL AS ANY TUPLE THAT SUCH TABLE      */
/* MIGHT CONTAIN SAYING price >= something OR            */
/* price > something                                     */

while (1)
{

        /* cur2 RETRIEVES ALL THE TUPLES IN THE            */
        /* APPL_RULES TABLE WITH OPERARATIONN >= OR <=     */

        $fetch cur2 into $a_nam, $a_attr, $a_op, $a_val;
        if (sqlca.sqlcode)
                break;

        /* FOR EVERY TUPLE WITH OPERATON >= RETIEVED       */
        /* FROM APPL_RULES                                 */
        if (strcmp(a_op, ">=  ") == 0)
         {

            /* cur_get LOOKS FOR A TUPLE WITH THE SAME */
            /* ATTRIBUTE NAME, AND OPERATION <=        */
            $open cur_get
                using $y, $a_nam, $a_attr, $dom, "<=  ";
            print_status("7");
            $fetch cur_get into $s_op, $s_val, $s_mark;

            if (sqlca.sqlcode != SQLNOTFOUND)
             {
                printf("gets %s %s\n ", a_val, s_val);
                if (compare_strings (a_val, s_val) == 0)
                 {
                    printf("OOPS|n");
                    /* IF SUCH TUPLE IS FOUND:                 */
                    /* 1- UPDATES THE TEMPORARY TABLE          */
                     $insert into temporary
                            values ($a_nam, $a_attr,
                                "=   ", $a_val);
                    /* 2- MARKS THE TUPLE FOUND IN THE         */
                    /*    DB_RULES TABLE                       */
                     $execute updte_11 using $y, $a_nam,
                        $a_attr, $dom, $s_op, $s_val;

                    /* 3- cur_templ LOOKS FOR A TUPLE IN       */
                    /*    DB_RULES TABLE WITH SAME ATRIBUTE */
```

```
             /*    NAME AND OPERATOION >= OR >      */
              $open cur_templ using $y, $a_nam,
                 $a_attr, $dom, ">   ", $y, $a_nam,
                 $a_attr, $dom, ">=  ";
              print_status("9");
              $fetch cur_templ into $s_op,
                         $s_val, $s_mark;

              /* 4- IF FOUND MARKS IT              */
              if (sqlca.sqlcode != SQLNOTFOUND)
                 $execute updte_sul using $y, $a_nam,
                         $a_attr, $dom, $s_op, $s_val;
              $close cur_templ;
         }
     }
   $close cur_get;
  }



 /* FOR EVERY TUPLE WITH OPERATON <= RETIEVED    */
 /* FROM APPL_RULES                              */
if (strcmp(a_op, "<=  ") == 0)
  {
     /* cur_get LOOKS FOR A TUPLE WITH THE SAME */
     /* ATTRIBUTE NAME, AND OPERATION <=        */
     $open cur_get
         using $y, $a_nam, $a_attr, $dom, ">=  ";
     print_status("8");
     $fetch cur_get into $s_op, $s_val, $s_mark;

     if (sqlca.sqlcode != SQLNOTFOUND)
        {

           /* IF SUCH TUPLE IS FOUND:              */
           /* 1- UPDATES THE TEMPORARY TABLE       */
          if (compare_strings (a_val, s_val) ==0)

           {
              $insert into temporary
                     values ($a_nam, $a_attr,
                         "=   ", $a_val);
             /* 2- MARKS THE TUPLE FOUND IN THE      */
             /*    DB_RULES TABLE                    */
              $execute updte_11 using $y, $a_nam,
                    $a_attr, $dom, $s_op, $s_val;

             /* 3- cur_templ LOOKS FOR A TUPLE IN    */
             /*    DB_RULES TABLE WITH SAME ATRIBUTE */
             /*    NAME AND OPERATOION <= OR <       */
              $open cur_templ using $y, $a_nam,
                    $a_attr, $dom, "<   ", $y, $a_nam,
                    $a_attr, $dom, "<=  ";
              print_status("10");
              $fetch cur_templ into $s_op,
                           $s_val, $s_mark;

              /* STEP 4 Marks it                 */
              if (sqlca.sqlcode != SQLNOTFOUND)
              $execute updte_sul using $y, $a_nam,
                         $a_attr, $dom, $s_op, $s_val;
              $close cur_templ;

         }
       }
  $close cur_get;
```

```
         }
      }

$close cur2;




/*                   PART    III                       */
/*                                                     */
/* THE SECOND LOOP OF THE PROGRAM DOES THE HEAVY WO    */
/* IT GOES THROUGH ALL THE TUPLES IN THE APPL_RULES    */
/* WITH THE RULE NUMBER GIVEN AS THE FIRST ARGUMENT TO */
/* THE PROGRAM, AND DEPENDING ON THEIR                 */
/* OPERATION DOES ONE OF THE FOLLOWING 5 CASES:        */
/* CASE 1: OPERATION IS =                              */
/* CASE 2: OPERATION IS >                              */
/* CASE 3: OPERATION IS <                              */
/* CASE 4: OPERATION IS >=                             */
/* CASE 5: OPERATION IS <=                             */
/*       (operation != is not handled in this program) */
/*                                                     */
/* THERE ARE VARYING DETAILS THAT WILL BE EXPLAINED    */
/* DOWN BEFORE EACH CASE, BUT THE COMMON CHARACTERISTC */
/* IS THIS:                                            */
/* FOR EVERY CASE OF TUPLE FOUND IN THE                */
/* APPL_RULES, IT LOOKS FOR THE TUPLES IN THE IN THE   */
/* DB_RULES WITH THE RULE NUMBER GIVEN AS THE SECOND   */
/* ARGUMENT TO THE PROGRAM, THAT HAVE THE SAME         */
/* ATTRIBUTE NAME.                                     */
/* IF IT FINDS THEM, CHECKS IF THE THEY CONFLICT WITH  */
/* TUPLE FROM THE APPL_RULES THAT WE ARE OBSERVING     */
/* (in which case the program ends                     */
/* immediately in failure) OR FINDS THEIR INTERSECTION,*/
/* STORES IT IN THE temporary, MARKS THE RELEVANT      */
/* TUPLES IN THE DB_RULES, AND LOOPS TO THE NEXT TUPLE */
/* IN THE APPL_RULES WITH THE GIVEN RULE NUMBER.       */
/* IF IT DOESN'T FIND THEM, COPIES THE TUPLE IN THE    */
/* RULES INTO THE temporary TABLE, AND LOOPS TO THE    */
/* NEXT TUPLE IN APPL_RULES WITH THE GIVEN RULE NUMBER */


while (1)
{
        counter = 1;
        if (condition == 0)
                break;
        $fetch cur1 into $a_nam, $a_attr, $a_op, $a_val;

        if (sqlca.sqlcode)
                break;


        $open cur_eq using $y, $a_nam, $a_attr, $dom, "=   ";
          print_status("11");
        $open cur_gt using $y, $a_nam, $a_attr, $dom, ">   ",
                        $y, $a_nam, $a_attr, $dom, ">=  ";
          print_status("12");
        $open cur_lt using $y, $a_nam, $a_attr, $dom, "<   ",
                        $y, $a_nam, $a_attr, $dom, "<=  ";
          print_status("13");
         not_found = 1;
         inserted = 0
```

```
/* CASE 1: OPERATION  =                                    */
/*    Step 1               LOOKS IN DB_RULES FOR           */
/*      A TUPLE WITH THE SAME ATRIBUTE_NAME AND            */
/*      OPERATION =.                                       */
/*    Step 1.1   IF  IT  FINDS  IT,  COMPARES              */
/*      THEIR VALUES, AND IF THEY ARE EQUAL COPIES IT      */
/*      INTO THE temporary TABLE, AND MAKS THE TUPLE       */
/*      IN THE DB RULES. IF THE VALUES ARE DIFFERENT,      */
/*      THEN THE RULES ARE DISJOINT AND THE PROGRAM        */
/*      ENDS.                                              */
/*    Step 2                                               */
/*      IF IT DOESNT FIND THE TUPLE WITH OPERATION =       */
/*      IN THE DB_RULES, THEN FIRST LOOKS FOR A TUPLE      */
/*      IN DB_RULES WITH THE SAME ATTRIBUTE NAME AND       */
/*      OPERATION > OR >=, AND THEN LOOKS FOR A TUPLE      */
/*      WITH OPERATION <= OR <.
/*    Step 2.1 IF IT FINDS ONE OF THESE TUPLES, NO         */
/*      WHAT COMBINATION IT IS (ie, a <= and >=            */
/*      combination, or a < and >=, or just a > alone,*/
/*       etc)  IT  MAKES  SURE  THAT  THE  VALUE  IN       */
/*      THE  APPL_RULES  DOESN'T CONFLICT WITH  THE        */
/*      VALUE-S IN THE DB_RULES. IF THIS IS THE CASE       */
/*      THE TUPLE-S FOUNF IN THE DB_RULES TABLE, AND       */
/*      COPIES THE TUPLE FOUND IN APPL_RULES INTO THE      */
/*      temporary TABLE. OTHERWISE THE TWO RULES ARE       */
/*      DISJOINT AND THE PROGRAM ENDS.                     */
/*   Step 3  IF NO TUPLE WAS FOUND IN DB_RULES IN          */
/*      STEPS 1 AND 2, COPY THE TUPLE IN APPL_RULES        */
/*      TO  THE TEMPORARY TABLE                            */
        if (strcmp(a_op, "=   ") == 0)
         {

           /* STEP 1                                       */

           $fetch cur_eq into $s_val, $s_mark;
           if (sqlca.sqlcode != SQLNOTFOUND)
            {
              /* STEP 1.1                                  */
              if (strcmp (s_val, a_val) == 0)
                {
                 /* VALUES ARE EQUAL, INSERT INTO          */
                 /* TEMPORSRY                              */
                 not_found = 0;
                 if (s_mark == 0)
                 {
                  $insert into temporary
                        values ($a_nam, $a_attr,
                                $a_op, $a_val);
                  $execute updte_sul using $y, $a_nam,
                        $a_attr, $dom, $a_op, $s_val;

                  }
                 }
                   /* VALUES ARE NOT EQUAL, THER IS          */
                   /* CONFLICT, PROGRAM ENDS IMMEDIATELY */
                   else
                      condition = 0;


            }

            /* STEP 2                                          */
            else
             {
              $fetch cur_gt into $s_op, $s_val, $s_mark;
              if (sqlca.sqlcode != SQLNOTFOUND)
```

```
               /* STEP 2.1                                */
               {
                if
                 (((comp = compare_strings(a_val, s_val)) < 0)
                     || (comp == 0) && (s_op == ">  "))
                   condition = 0;
                   /* CONFLICT OCCURS IF APPL HAS FOR EXAMPLE */
                   /* value = 100 AND DB HAS value > 500      */

                  else
                   {
                     /* IF THERE IS NO CONFLICT INSERT TUPLE  */
                     /* FROM APPLICATION INTO TEMPORARY. THE  */
                     /* VARIABLE inserted IS SET TO MAKE SURE */
                     /* THAT THE TUPLE IS NOT INSERTED TWICE  */
                     $insert into temporary
                        values ($a_nam, $a_attr,
                                $a_op, $a_val);
                     not_found = 0;
                     inserted = 1;
                     $execute updte_sul using $y, $a_nam,
                        $a_attr, $dom, $s_op, $s_val;
                   }
               }
               $fetch cur_lt into $s_op, $s_val, $s_mark;
              if (sqlca.sqlcode != SQLNOTFOUND)

              /* STEP 2.1                                 */
              {
               if
                 (((comp = compare_strings(a_val, s_val)) > 0)
                     || (comp == 0) && (s_op == ">  "))
                   condition = 0;
                   /* CONFLICT OCCURS IF APPL HAS FOR EXAMPLE */
                   /* value = 100 AND DB HAS value < 50       */

                   else
                    {

                     /* IF THERE IS NO CONFLICT AND THE TUPLE   */
                     /* FROM APPLICATION HAS NOT PREVIOUSLY     */
                     /* BEEN INSERTED INTO TEMPORARY. THEN      */
                     /* INSERT IT, OTHERWISE DON'T DO ANYTHING */
                     not_found = 0;
                     if (inserted == 0)
                       $insert into temporary
                            values ($a_nam, $a_attr,
                                $a_op, $a_val);
                       $execute updte_sul using $y, $a_nam,
                                $a_attr, $dom, $s_op, $s_val;
                    }
                 }
               }


             /* STEP 3                                    */
             if (not_found == 1)
                 $insert into temporary
                            values ($a_nam, $a_attr,
                                $a_op, $a_val);
            }


/* CASE 2: OPERATION  >                               */
/*    Step 1               LOOKS IN DB_RULES FOR      */
```

```
/*      A TUPLE WITH THE SAME ATRIBUTE_NAME AND         */
/*      OPERATION =. IF  IT  FINDS  IT,  COMPARES THEM*/
/*      TO SEE IF THEY CONFLICT. IF THEY CONFLICT,      */
/*       PROCEDURE ENDS IMMEDIATELY. IF THEY DON'T      */
/*      CONFLICT, CHECKS IF THE TUPLE FROM DB_RULES IS*/
/*      MARKED, AND IF IT IS NOT, MARKS IT AND COPIES */
/*      INTO TEMPORARY. IF MARKED DON'T DO ANYTHING     */
/*      EXAMPLE: Price > 50 CONFLICTS WITH Price = 30.*/
/*    Step 2                                            */
/*      IF IT DOESNT FIND THE TUPLE WITH OPERATION =    */
/*      IN THE  DB_RULES, THEN  LOOKS FOR A  TUPLE      */
/*      IN DB_RULES WITH THE SAME ATTRIBUTE NAME AND    */
/*      OPERATION < OR <=. IF IT FINDS SUCH A TUPLE,    */
/*      MAKES SURE THAT IT DOESN'T CONFLICT. OTHERWISE*/
/*      RETURNS 0. EXAMPLE: Price > 50 AND Price <= 4 */
/*      CONFLICT. IF THERE'S NO SUCH TUPLE ITS OK       */
/*    Step 3                                            */
/*      IF THERE WAS NO CONFLICT IN  STEP 2, LOOKS      */
/*      IN  DB_RULES  FOR A  TUPLE  WITH  THE  SAME     */
/*      ATTRIBUTE  NAME  AND OPERATION > OR >=. IF IT */
/*      FINDS SUCH A TUPLE, CHECKS IF IT IS MARKED,     */
/*      AND IF NOT, MARKS IT, PICKS THE MOST            */
/*      RESTRICIVE OF THE TWO TUPLES AND INSERTS IN     */
/*      THE TEMPORARY TABLE. IF THE TUPLE IS MARKED     */
/*      DON'T DO ANYTHING. EXAMPLE: Price > 50 IS       */
/*      MORE RESTRICTIVE THAN Price >= 30.              */
/*   Step 4  IF NO TUPLE WAS FOUND IN DB_RULES IN       */
/*      STEPS 1 AND 3, COPY THE TUPLE IN APPL_RULES     */
/*      TO  THE TEMPORARY TABLE                         */

else if (strcmp(a_op, ">   ") == 0)
         {
           /* STEP 1                                    */
           $fetch cur_eq into $s_val;
           if (sqlca.sqlcode != SQLNOTFOUND)
            {
              if (strcmp (s_val, a_val) > 0)
                {
                 not_found = 0;
                  if (s_mark == 0)
                  {
                   $insert into temporary
                        values ($a_nam, $a_attr,
                                "=   ", $s_val);
                   $execute updte_sul using $y, $a_nam,
                        $a_attr, $dom, "=   ", $s_val;
                  }
                 }
                 else
                   condition = 0;
              }

             else
             {

               /* STEP 2                                      */
               $fetch cur_gt into $s_op, $s_val, $s_mark;
              if (sqlca.sqlcode != SQLNOTFOUND)
               {
                not_found = 0;
                if (s_mark == 0)
                {
                 if
                 (((comp = compare_strings(a_val, s_val)) == 0)
                        || (comp < 0))
                    $insert into temporary
```

```
                                values ($a_nam, $a_attr,
                                        $s_op, $s_val);
                          else
                            $insert into temporary
                                values ($a_nam, $a_attr,
                                        $a_op, $a_val);
                         $execute updte_sul using $y, $a_nam,
                                $a_attr, $dom, $s_op, $s_val;
                        }
                       }

                      /* STEP 3                                   */
                      $fetch cur_lt into $s_op, $s_val, $s_mark;
                      if (sqlca.sqlcode != SQLNOTFOUND)
                       {
                        if
                         (((comp = compare_strings(a_val, s_val)) == 0)
                                || (comp > 0))
                           condition = 0;
                        }
                      }

                /* STEP $                                    */
                if (not_found == 1)
                         $insert into temporary
                                    values ($a_nam, $a_attr,
                                        $a_op, $a_val);
            }


        /* CASE 3: OPERATION  <                               */
        /*    Step 1               LOOKS IN DB_RULES FOR       */
        /*      A TUPLE WITH THE SAME ATRIBUTE_NAME AND        */
        /*      OPERATION =. IF  IT  FINDS  IT,  COMPARES THEM*/
        /*      TO SEE IF THEY CONFLICT. IF THEY CONFLICT,     */
        /*       PROCEDURE ENDS IMMEDIATELY. IF THEY DON'T     */
        /*      CONFLICT, CHECKS IF THE TUPLE FROM DB_RULES IS*/
        /*      MARKED, AND IF IT IS NOT, MARKS IT AND COPIES */
        /*      INTO TEMPORARY. IF MARKED DON'T DO ANYTHING    */
        /*      EXAMPLE: Price < 50 CONFLICTS WITH Price = 70.*/
        /*    Step 2                                           */
        /*      IF IT DOESNT FIND THE TUPLE WITH OPERATION =   */
        /*      IN THE  DB_RULES, THEN  LOOKS FOR A  TUPLE     */
        /*      IN DB_RULES WITH THE SAME ATTRIBUTE NAME AND   */
        /*      OPERATION > OR >=. IF IT FINDS SUCH A TUPLE,   */
        /*      MAKES SURE THAT IT DOESN'T CONFLICT. OTHERWISE*/
        /*      RETURNS 0. EXAMPLE: Price > 50 AND Price >= 60*/
        /*      CONFLICT. IF THERE'S NO SUCH TUPLE ITS OK      */
        /*    Step 3                                           */
        /*      IF THERE WAS NO CONFLICT IN  STEP 2, LOOKS     */
        /*      IN  DB_RULES  FOR A  TUPLE  WITH  THE  SAME    */
        /*      ATTRIBUTE  NAME  AND OPERATION < OR <=. IF IT */
        /*      FINDS SUCH A TUPLE, CHECKS IF IT IS MARKED,    */
        /*      AND IF NOT, MARKS IT, PICKS THE MOST           */
        /*      RESTRICIVE OF THE TWO TUPLES, AND INSERTS IN   */
        /*      THE TEMPORARY TABLE. IF THE TUPLE IS MARKED    */
        /*      DON'T DO ANYTHING. EXAMPLE: Price < 50 IS      */
        /*      MORE RESTRICTIVE THAN Price <= 60.             */
        /*   Step 4  IF NO TUPLE WAS FOUND IN DB_RULES IN      */
        /*      STEPS 1 AND 3, COPY THE TUPLE IN APPL_RULES    */
        /*      TO  THE TEMPORARY TABLE                        */

else if (strcmp (a_op, "<   ") == 0)
                 {

                   /* STEP 1                                  */
```

```
   $fetch cur_eq into $s_val, $s_mark;
   if (sqlca.sqlcode != SQLNOTFOUND)
    {
      if (strcmp (s_val, a_val) < 0)
        {
          not_found = 0;
          if (s_mark == 0)
           {
            $insert into temporary
                values ($a_nam, $a_attr,
                        "=   ", $s_val);
            $execute updte_sul using $y,$a_nam,
                $a_attr, $dom, "=   ", $s_val;
           }
         }
      }

     else
     {

      /* STEP 3                                  */
      $fetch cur_lt into $s_op, $s_val, $s_mark;
      if (sqlca.sqlcode != SQLNOTFOUND)
       {
        not_found = 0;
        if (s_mark == 0)
        {
         if
         (((comp = compare_strings(a_val, s_val)) == 0)
        || (comp < 0))
            $insert into temporary
                values ($a_nam, $a_attr,
                        $a_op, $a_val);
          else
             $insert into temporary
                values ($a_nam, $a_attr,
                        $s_op, $s_val);

         $execute updte_sul using $y, $a_nam,
                $a_attr, $dom, $s_op, $s_val;
        }
      }


      /* STEP 2                                      */
      $fetch cur_gt into $s_op, $s_val, $s_mark;
      if (sqlca.sqlcode != SQLNOTFOUND)
       {
        if
        (((comp = compare_strings (a_val, s_val)) == 0)
                || (comp < 0))
           {
                condition = 0;
           }
        }
      }

      /* STEP 4                                        */
     if (not_found == 1)
         $insert into temporary
                    values ($a_nam, $a_attr,
                        $a_op, $a_val);


}
```

```
/* CASE 4: OPERATION  >=                             */
/*    Step 1                    LOOKS IN DB_RULES FOR */
/*      A TUPLE WITH THE SAME ATRIBUTE_NAME AND       */
/*      OPERATION =. IF  IT  FINDS  IT,  COMPARES THEM*/
/*      TO SEE IF THEY CONFLICT. IF THEY CONFLICT,    */
/*       PROCEDURE ENDS IMMEDIATELY. IF THEY DON'T    */
/*      CONFLICT, CHECKS IF THE TUPLE FROM DB_RULES IS*/
/*      MARKED, AND IF IT IS NOT, MARKS IT AND COPIES */
/*      INTO TEMPORARY. IF MARKED DON'T DO ANYTHING   */
/*      EXAMPLE: Price >=50 CONFLICTS WITH Price = 30.*/
/*    Step 2                                          */
/*      IF IT DOESNT FIND THE TUPLE WITH OPERATION =  */
/*      IN THE  DB_RULES, THEN  LOOKS FOR A  TUPLE    */
/*      IN DB_RULES WITH THE SAME ATTRIBUTE NAME AND  */
/*      OPERATION < OR <=. IF IT FINDS SUCH A TUPLE,  */
/*      MAKES SURE THAT IT DOESN'T CONFLICT. OTHERWISE*/
/*      RETURNS 0. EXAMPLE: Price >= 50 AND Price <= 6*/
/*      CONFLICT. IF THERE'S NO SUCH TUPLE ITS OK     */
/*    Step 3                                          */
/*      IF THERE WAS NO CONFLICT IN  STEP 2, LOOKS    */
/*      IN  DB_RULES  FOR A  TUPLE  WITH  THE  SAME   */
/*      ATTRIBUTE  NAME  AND OPERATION > OR >=. IF IT */
/*      FINDS SUCH A TUPLE, CHECKS IF IT IS MARKED,   */
/*      AND IF NOT, MARKS IT, PICKS THE MOST          */
/*      RESTRICIVE OF THE TWO TUPLES, AND INSERTS IN  */
/*      THE TEMPORARY TABLE. IF THE TUPLE IS MARKED   */
/*      DON'T DO ANYTHING. EXAMPLE: Price >= 50 IS    */
/*      MORE RESTRICTIVE THAN Price >= 40.            */
/*   Step 4  IF NO TUPLE WAS FOUND IN DB_RULES IN     */
/*      STEPS 1 AND 3, COPY THE TUPLE IN APPL_RULES   */
/*      TO  THE TEMPORARY TABLE                       */

else if (strcmp(a_op, ">=  ") == 0)
         {

           /* STEP 1                                  */
           $fetch cur_eq into $s_val, $s_mark;
            if (sqlca.sqlcode != SQLNOTFOUND)
            {
              if (((comp = compare_strings (s_val, a_val)) > 0)
                 || (comp == 0))
                {
                  not_found = 0;
                  if (s_mark == 0)
                  {
                    $insert into temporary
                        values ($a_nam, $a_attr,
                                        "=   ", $s_val);
                    $execute updte_sul using $y, $a_nam,
                        $a_attr, $dom, "=   ", $s_val;
                  }
                }
                 else
                   condition = 0;
              }

         else
             {

             /* STEP 2                                  */
             $fetch cur_lt into $s_op, $s_val, $s_mark;
              if (sqlca.sqlcode != SQLNOTFOUND)
                {
                 comp = compare_strings (a_val, s_val);
                 not_found = 0;
                 if (((strcmp(s_op, "<   ") == 0) &&
```

```
                          (comp == 0)) || (comp > 0))
                          {
                            condition = 0;
                          }
                  }

                /* STEP 3                                 */
                $fetch cur_gt into $s_op, $s_val, $s_mark;
                print_status("Fetch cur_gt");
printf("\nGT1 %i\n", sqlca.sqlcode);
                if (sqlca.sqlcode != SQLNOTFOUND)
                 {
                  not_found = 0;
                  comp = compare_strings (a_val, s_val);
  printf("\nGT %s %s %i\n", a_val, s_val, comp);
                  if (s_mark == 0)
                  {
                    if
                     ((comp == 0) || (comp < 0))
                      $insert into temporary
                          values ($a_nam, $a_attr,
                                  $s_op, $s_val);
                     else
                       $insert into temporary
                          values ($a_nam, $a_attr,
                                  $a_op, $a_val);


                   $execute updte_sul using $y, $a_nam,
                          $a_attr, $dom, $s_op, $s_val;
                  }
                 }
              }

            /* STEP 4                                     */
            if (not_found == 1)
                   $insert into temporary
                                  values ($a_nam, $a_attr,
                                  $a_op, $a_val);
          }


  /* CASE 4: OPERATION  <=                              */
  /*    Step 1               LOOKS IN DB_RULES FOR      */
  /*      A TUPLE WITH THE SAME ATRIBUTE_NAME AND       */
  /*      OPERATION =. IF  IT  FINDS  IT,  COMPARES THEM*/
  /*      TO SEE IF THEY CONFLICT. IF THEY CONFLICT,    */
  /*       PROCEDURE ENDS IMMEDIATELY. IF THEY DON'T    */
  /*      CONFLICT, CHECKS IF THE TUPLE FROM DB_RULES IS*/
  /*      MARKED, AND IF IT IS NOT, MARKS IT AND COPIES */
  /*      INTO TEMPORARY. IF MARKED DON'T DO ANYTHING   */
  /*      EXAMPLE: Price <=50 CONFLICTS WITH Price = 70.*/
  /*    Step 2                                          */
  /*      IF IT DOESNT FIND THE TUPLE WITH OPERATION =  */
  /*      IN THE  DB_RULES, THEN  LOOKS FOR A  TUPLE    */
  /*      IN DB_RULES WITH THE SAME ATTRIBUTE NAME AND  */
  /*      OPERATION > OR >=. IF IT FINDS SUCH A TUPLE,  */
  /*      MAKES SURE THAT IT DOESN'T CONFLICT. OTHERWISE*/
  /*      RETURNS 0. EXAMPLE: Price <= 5 AND Price > 5  */
  /*      CONFLICT. IF THERE'S NO SUCH TUPLE ITS OK     */
  /*    Step 3                                          */
  /*      IF THERE WAS NO CONFLICT IN  STEP 2, LOOKS    */
  /*      IN  DB_RULES  FOR A  TUPLE  WITH  THE  SAME   */
  /*      ATTRIBUTE  NAME  AND OPERATION < OR <=. IF IT */
  /*      FINDS SUCH A TUPLE, CHECKS IF IT IS MARKED,   */
```

```
/*      AND IF NOT, MARKS IT, PICKS THE MOST              */
/*      RESTRICICE OF THE TWO TUPLES, AND INSERTS IN      */
/*      THE TEMPORARY TABLE. IF THE TUPLE IS MARKED       */
/*      DON'T DO ANYTHING. EXAMPLE: Price <= 50 IS        */
/*      MORE RESTRICTIVE THAN Price <= 4000.              */
/*   Step 4  IF NO TUPLE WAS FOUND IN DB_RULES IN         */
/*      STEPS 1 AND 3, COPY THE TUPLE IN APPL_RULES       */
/*      TO  THE TEMPORARY TABLE                           */
else if (strcmp(a_op, "<=  ") == 0)
         {

           /* STEP 1                                 */
           $fetch cur_eq into $s_val, $s_mark;
           if (sqlca.sqlcode != SQLNOTFOUND)
            {
              comp = compare_strings (s_val, a_val);
              if ((comp > 0)
                 || (comp == 0))
                {
                  not_found = 0;
                  if (s_mark == 0)
                  {
                    $insert into temporary
                        values ($a_nam, $a_attr,
                                "=   ", $s_val);

                    $execute updte_sul using $y, $a_nam,
                        $a_attr, $dom, "=   ", $s_val;

                  }
                 }
                 else
                   condition = 0;
              }

         else
             {

             /* STEP 2                                    */
             $fetch cur_gt into $s_op, $s_val, $s_mark;
              if (sqlca.sqlcode != SQLNOTFOUND)
               {
                 comp = compare_strings (a_val, s_val);
                 not_found = 0;

                 if (((strcmp (s_op, ">   ") == 0) &&
                        (comp == 0)) || (comp < 0))
                    {

                         condition = 0;
                    }
                }

              /* STEP 3                                   */
              $fetch cur_lt into $s_op, $s_val, $s_mark;
              if (sqlca.sqlcode != SQLNOTFOUND)
               {
                not_found = 0;
                comp = compare_strings (a_val, s_val);
                if (s_mark == 0)
                {
                  if
                   ((comp == 0) || (comp > 0))
                   $insert into temporary
                        values ($a_nam, $a_attr,
                                $s_op, $s_val);
```

```
                                   else
                                     $insert into temporary
                                        values ($a_nam, $a_attr,
                                                $a_op, $a_val);

                                   $execute updte_sul using $y, $a_nam,
                                        $a_attr, $dom, $s_op, $s_val;
                                }
                              }
                          }

                          /* STEP 4                               */
                          if (not_found == 1)
                                 $insert into temporary
                                            values ($a_nam, $a_attr,
                                                $a_op, $a_val);
                        }

        $close cur_eq;
        $close cur_gt;
        $close cur_lt;

}
        $close curl;



                /*             PART  IV                                 */
                /*                                                      */
                /* COPY THE UNMARKED TUPLES FROM DB_RULES INTO TEMPORARY */
                while (1)
                 {
                  $fetch cursor_final into $a_nam, $a_attr, $a_op, $a_val;
                  if (sqlca.sqlcode)
                        break;

                  $insert into temporary
                        values ($a_nam, $a_attr, $a_op, $a_val);
                }

        $close cursor_final;
        return(condition);


}



print_status(statmt)
char *statmt;
{
 if (sqlca.sqlcode < 0)
        {
 fprintf (stderr, "=================================================\n");
 fprintf (stderr, "SQLCA After %s\n", statmt);
 fprintf (stderr, "sqlcode :   %ld\n", sqlca.sqlcode);
 fprintf (stderr, "Error character :   %ld\n", sqlca.sqlerrd[4]);
}}



long str_to_float(j)
 char *j;
{
         float   jjj;
```

```
int i;
float divider;
int cas;
int i_case;
int exponent;
int expo_case;
float result;
float intermediate;

i_case = 1;
expo_case = 1;
exponent = 0;
cas = 0;
result = 0;
divider = 10;
i = 0;
while (1)
{

 if (j[i] == '\0' || j[i] == '\n' || j[i] == ' ')
       break;


 if ( j[i] == '.' )
    cas = 1;

 else if ((j[i] == 'e') || (j[i] == 'E'))
        cas = 2;

  else if (cas == 0)
          {
            if (j[i] == '-')
                 {
                   result = result * -1;
                   i_case = -1;
                 }
               else result = (result * 10) + i_case * (j[i] - '0');
           }
          else if (cas == 1)
                 {
                  intermediate = (j[i] - '0') / divider;
                  result = result + (i_case * intermediate);
                  divider = divider * 10;
                 }
                else if (cas == 2)
                         {
                           if (j[i] == '-')
                                expo_case = 0;
                               else exponent = (exponent * 10) + j[i] - '0
                          }

i++;
}

while (exponent > 0)
 {
  exponent = exponent - 1;
  if (expo_case == 1)
        result = result * 10;
       else result = result / 10;
  }
printf("Res %f\n", result);
return(result);
}
```

```
int SemanticEqOrSub(x,y)
        $long x;
        $long y;
        {

                $char   a_attr[30];
                $char   a_op[5];
                $char   s_op[5];
                $char   a_val[30];
                $char   s_val[30];
                $char   a_nam[30];


                $char   *dom;
                $char   dummy[30];
                int     s_mark;

                int     ne_condition;
                int     condition;
                int     comp;
                int not_found;

                dom = "S";
                condition = 1;

                $open cur1 using $x, $dom;
                print_status("14");

                /*            PART I                                        */
                /* THIS PROCEDURE ONLY HAS ONE PART IN WHICH IT SACANS EVERY */
                /* TUPLE FROM THE THEN CONDITION IN THE APPL_RULES. EACH OF  */
                /* THOSE TUPLES IS CLASIFIED INTO 4 CASES, DEPENDING ON ITS  */
                /* OPERATION:                                                */
                /* CASE 1 FOR OPERATION =.                                   */
                /* CASE 2 FOR OPERATION > OR >=                              */
                /* CASE 3 FOR OPERATION < OR <=                              */
                /* CASE 4 FOR OPERATION !=                                   */
                /* EACH CASE VARIES, BUT IN ALL OF THEM IT LOOKS FOR TUPLES  */
                /* IN THE DB_RULES TABLE THAT MIGHT CONFLICT WITH THE TUPLE  */
                /* FROM APPL_RULES. IF ANY CONFLICT HAPPENS, THE ATR_NAME OF */
                /* THE TUPLE THAT CAUSED IT IS STORED IN THE BAD TABLE, AND  */
                /* THE PROGRAM, ALTHOUGH NOT FORCED TO FINISH IMMEDIATELY,   */
                /* IS SET TO RETURN 0. THIS IS DONE BY SETTING THE condition */
                /* VARIABLE TO 0. condition IS ALWAYS INITIATED WITH VALUE   */
                /* 1, BUT WHENEVER IT IS CHANGED TO 0, IT CAN NEVER BE 1     */
                /* AT THE END, THE PROCEDURE RETURNS THE VALUE OF condition. */
                /* THE  PROCEDURE  FINISHES  WHEN  ALL  TUPLES               */
                /* FROM APPL_RULES HAVE BEEN SCANED                          */

                while (1)
                {

                        $fetch cur1 into $a_nam, $a_attr, $a_op, $a_val;
                        if (sqlca.sqlcode)
                                /* ONLY WAY TO FINISH IS WHEN THERE ARE NO   */
                                /* TUPLES LEFT IN APPL_RULES                 */
                                break;


                        $open cur_eq using $y, $a_nam, $a_attr, $dom, "=   ";
                          print_status("15");
                        $open cur_neq using $y, $a_nam, $a_attr, $dom, "!=  ";
                          print_status("16");
                        $open cur_gt using $y, $a_nam, $a_attr, $dom, ">   ",
                                        $y, $a_nam, $a_attr, $dom, ">=  ";
```

```
  print_status("17");
$open cur_lt using $y, $a_nam, $a_attr, $dom, "<   ",
                $y, $a_nam, $a_attr, $dom, "<=  ";
  print_status("18");
not_found = 1;


/* CASE 1                                                   */
/* AFTER FINDING A TUPLE Price = 50 IN APPL_RULES,          */
/* THERE MUST BE A TUPLE IDENTICAL TO IT IN DB_RULES */
/* OR THERE IS A CONFLICT.                                  */
if (strcmp(a_op, "=   ") == 0)
 {
      $fetch cur_eq into $s_val, $s_mark;
      if (sqlca.sqlcode == SQLNOTFOUND)
          {
                $insert into bad
                values($x, $y, $a_nam);
                 condition = 0;
          }
      if (strcmp (s_val, a_val) != 0)
           {
                $insert into bad
                values($x, $y, $a_nam);
                condition = 0;
           }
}

/* CASE 2: OPERATION > OR >=                            */
/*    FIRST, FOR EVERY TUPLE Price > or >= A            */
/*      LOOK FOR TUPLE Price = B IN DB_RULES, AND */
/*      SEE IF THEY CONFLICT. EXAMPLE: PRICE > 50 */
/*      CONFLICTS WITH Price = 30                       */
/*    SECOND, IF NO TUPLE WAS FOUND IN STEP I, LOOK*/
/*      FOR TUPLE Price > C IN DB_RULES AND SEE IF*/
/*      IT IS LESS RESTRICTIVE THAN THE TUPLE FROM*/
/*      APPL_RULES, IN WHICH CASE THEY CONFLICT.  */
/*      FOR EXAMPLE Price >= 50 CONFLICTS WITH     */
/*      Price > 20.                                     */
/*    THIRD IF NO TUPLE WAS FOUND IN THE TWO STEPS */
/*      ABOVE, THERE IS A CONFLICT                      */




else if ((strcmp(a_op, ">   ") == 0)
         || (strcmp(a_op, ">=  ") == 0))
 {

      /* FIRST                                            */
      $fetch cur_eq into $s_val, $s_mark;
      if (sqlca.sqlcode != SQLNOTFOUND)
      {
       not_found = 0;
       if
         (compare_strings(a_val, s_val) > 0)
         {
           $insert into bad
                values($x, $y, $a_nam);
           condition = 0;
          }
       else if ((strcmp(a_op, ">   ") == 0)
             && (compare_strings(a_val, s_val) == 0))
           {
            $insert into bad
```

```
               values($x, $y, $a_nam);
          condition = 0;
          }
       }

         else
       {

         /* SECOND                                        */
         $fetch cur_gt into $s_op, $s_val, $s_mark;
         if (sqlca.sqlcode != SQLNOTFOUND)
          {
             not_found = 0;
             if (compare_strings(a_val, s_val) > 0)
                {
                       $insert into bad
                       values($x, $y, $a_nam);
                       condition = 0;
                }
             if ((compare_strings(a_val, s_val) == 0)
                && (strcmp(a_op, ">   ") == 0)
                && (strcmp(s_op, ">=  ") == 0))
               {
                $insert into bad
                 values($x, $y, $a_nam);
                condition = 0;
               }
          }

        }

        /* THIRD                                           */
       if (not_found == 1)
        {
          $insert into bad
                values($x, $y, $a_nam);
          condition = 0;
        }

}

/* CASE 3: OPERATION < OR <=                         */
/*    FIRST, FOR EVERY TUPLE Price < or <= A         */
/*      LOOK FOR TUPLE Price = B IN DB_RULES, AND    */
/*      SEE IF THEY CONFLICT. EXAMPLE: PRICE < 50    */
/*      CONFLICTS WITH Price = 80                    */
/*   SECOND, IF NO TUPLE WAS FOUND IN STEP I, LOOK*/
/*      FOR TUPLE Price < C IN DB_RULES AND SEE IF*/
/*      IT IS LESS RESTRICTIVE THAN THE TUPLE FROM*/
/*      APPL_RULES, IN WHICH CASE THEY CONFLICT.  */
/*      FOR EXAMPLE Price <= 50 CONFLICTS WITH    */
/*      Price < 700.                              */
/*   THIRD IF NO TUPLE WAS FOUND IN THE TWO STEPS */
/*      ABOVE, THERE IS A CONFLICT                */

else if ((strcmp(a_op, "<   ") == 0)
         || (strcmp(a_op, "<=  ") == 0))
{

      /* FIRST                                          */
      $fetch cur_eq into $s_val, $s_mark;
      if (sqlca.sqlcode != SQLNOTFOUND)
      {
       not_found = 0;
       if
         (compare_strings(a_val, s_val) < 0)
```

```
          {
           $insert into bad
                values($x, $y, $a_nam);
           condition = 0;
          }
       else if  ((strcmp(a_op, "<   ") == 0)
              && (compare_strings(a_val, s_val) == 0))
            {
                $insert into bad
                values($x, $y, $a_nam);
                condition = 0;
             }
       }

      else

        /* SECOND                                   */
        {
          $fetch cur_lt into $s_op, $s_val, $s_mark;

          if (sqlca.sqlcode != SQLNOTFOUND)
          {

            not_found = 0;
            if (compare_strings(a_val, s_val) < 0)
              {
                $insert into bad
                values($x, $y, $a_nam);
                condition = 0;
              }
            if ((compare_strings(a_val, s_val) == 0)
                && (strcmp(a_op, "<   ") == 0)
                && (strcmp(s_op, "<=  ") == 0))
             {
              $insert into bad
                values($x, $y, $a_nam);
              condition = 0;
             }
          }


       }

       /* THIRD                                     */
       if (not_found == 1)
         {
           $insert into bad
                 values($x, $y, $a_nam);
            condition = 0;
         }
 }


/* CASE 4: OPERATION !=                              */
/*   FIRST FOR EVERY TUPLE OF THE FORM Price != 50   */
/*       FOUND IN APPL_RULES, LOOK FOR TUPLE         */
/*       Price = 50 IN DB_RULES, IN WHICH CASE THERE */
/*       IS A CONFLICT.                              */
/*   SECOND FOR THE TUPLE Price != 50, IF THE        */
/*       Price = 50 WAS NOT FOUND IN FIRST STEP, LOOK*/
/*       IN DB_RULES FOR TUPLE Price > OR >= A, AND  */
/*       SEE IF THERE IS POTENTIAL CONFLICT. FOR     */
/*       EXAMPLE, THE TUPLE Price >= 30 POTENTIALLY  */
/*       CONFLICTS WITH Price != 50. ON THE OTHER    */
/*       HAND, Price > 70 COMPLETELY RULES OUT ANY   */
/*       CONFLICT                                    */
```

```
/*   THIRD IF CONFLICT IS POTENTIAL LOOK                */
/*       IN DB_RULES FOR TUPLE Price < OR <= B, AND     */
/*       SEE IF IT RULES OUT CONFLICT. FOR              */
/*       EXAMPLE, THE TUPLE Price <= 30 COMPLETELY      */
/*       RULES OUT CONFLICT WITH Price != 50, BUT       */
/*       Price < 70 LEAVES THE POSSIBILITY OPEN FOR     */
/*       CONFLICT                                       */
/*   FOURT IF CONFLICT HAS NOT YET BEEN RULED OUT,      */
/*       LOOK IN DB RULES FOR THE TUPLE Price != 50     */
/*       WHICH COMPLETELY RULES OUT CONFLICT            */
/*   FIFTH IF CONFLICT WAS NOT RULED OUT IN ANY OF      */
/*       THE THREE STEPS BEFORE, SET THE condition      */
/*       VARIABLE TO 0, AND UPDATE THE BAD TABLE        */
else if (strcmp(a_op, "!=  ") == 0)
 {
       /* FIRST                                          */
       $fetch cur_eq into $s_val, $s_mark;
       if (sqlca.sqlcode != SQLNOTFOUND)
       {
           not_found = 0;
           if (compare_strings(a_val, s_val) == 0)
            {
                 $insert into bad
                 values($x, $y, $a_nam);
                  condition = 0;
             }
         }

   else
       {
        ne_condition = 1;

        /* THIRD                                    */
        $fetch cur_lt into $s_op, $s_val, $s_mark;
        if (sqlca.sqlcode != SQLNOTFOUND)
          {
            not_found = 0;
           if
           (((comp = compare_strings(a_val, s_val)) > 0)
            || (comp == 0))

               ne_condition = 0;
            if ((strcmp(s_op, "<=  ") == 0) &&
                 (compare_strings(a_val, s_val) == 0))
             {
               $insert into bad
                 values($x, $y, $a_nam);
               condition = 0;
              }
           }

        /* SECOND                                    */
        $fetch cur_gt into $s_op, $s_val, $s_mark;
         if (sqlca.sqlcode != SQLNOTFOUND)

          {
           not_found = 0;
           if
           (((comp =compare_strings(a_val, s_val)) < 0)
             || (comp == 0))
               ne_condition = 0;
            if ((strcmp(s_op, "<=  ") == 0) &&
                 (compare_strings(a_val, s_val) == 0))
               {
                 $insert into bad
                 values($x, $y, $a_nam);
```

```
                                            condition = 0;
                                        }
                                    }

                                  /* FOURTH                                       */
                                  while (1)
                                   {
                                     $fetch cur_neq into $s_op, $s_val, $s_mark;
                                     if (sqlca.sqlcode)
                                          break;
                                     if (compare_strings(a_val, s_val) == 0)
                                       {
                                         ne_condition = 0;
                                         not_found = 0;
                                         break;
                                       }
                                   }
                               }

                             /* FIFTH                                             */
                             if ((not_found == 1) || (ne_condition == 1))
                                {
                                  $insert into bad
                                        values($x, $y, $a_nam);
                                  condition = 0;
                                 }
                          }
       $close cur_eq;
       $close cur_neq;
       $close cur_gt;
       $close cur_lt;
        }
       $close cur1;
       return (condition);
}



int NotEqualStuff(x,y)
       $long x;
       $long y;
       {

              $char   a_attr[30];
              $char   a_op[5];
              $char   s_op[5];
              $char   a_val[30];
              $char   s_val[30];
              $char   a_nam[30];

              $char   str_sul[300];
              $char   sttup[300];

              $char   *dom;
              $char   dummy[30];
              $long   s_mark;
              int     cond;
              int     ne_condition;

              int     cas;


              strcpy (str_sul, "update db_rules ");
              strcat (str_sul, " set mark = 1 ");
```

```
        strcat (str_sul, " where r_number = ? AND atr_name = ? AND ");
        strcat (str_sul, " arg_name = ? AND domain_type = ? AND ");
        strcat (str_sul, " operation = ? AND primitive = ?");

        strcpy (sttup, "update temporary ");
        strcat (sttup, " set operation = ? ");
        strcat (sttup, " where atr_name = ? AND ");
        strcat (sttup, " arg_name = ?  AND ");
        strcat (sttup, " operation = ? AND primitive = ?");
        dom = "A";

        print_status("QQQQ");
        $open cur3 using $x, $dom, "!=  ";
        print_status("19");

        $open cur4 using $y, $dom, "!=  ";
          print_status("20");

        $prepare updte_suN from $str_sul;
        $prepare updte_Tup from $sttup;

        cond = 1;
        cas = 1;

/* THE SAME LOOP APPLIES TO FETCHING THE ATTRIBUTES WITH OPERATION   */
/* != FROM APPL_RULES AND FROM DB_RULES. IT STARTS WITH THE VARIABLE */
/* cas = 1, FETCHING TUPLES FROM APPL_RULES. WHEN THERE ARE NO MORE  */
/* TUPLES WITH OPERATION != FROM APPL_RULES, cas IS SET TO 2, AND    */
/* THE PROCEDURE STARTS FETCHING TUPLES FROM DB_RULES. WHEN ALL THESE*/
/* ARE GONE IT FINISHES. ALSO FINISHES WHEN A CONFLICT OCCURS        */

        while (1)
        {
        if (cond == 0)
           /* break BECAUSE OF CONFLICT */
           break;

        ne_condition = 0;
        if (cas == 1)
          {
           $fetch cur3 into $a_nam, $a_attr, $a_op, $a_val;
           print_status("FETCH CUR3");
printf("CUR3 %i %s %s\n", sqlca.sqlcode, a_op, a_val);
           if (sqlca.sqlcode)
                 cas = 2;
           }
        if (cas == 2)
          {
           $fetch cur4 into $a_nam, $a_attr, $a_op, $a_val;
           if (sqlca.sqlcode)
                /* break BECAUSE NO TUPLES LEFT IN EITHER TABLE */
                break;
           }


        $open cur5 using $y, $a_nam, $a_attr, $dom, "!=  ", $a_val;
                  print_status("21");
        $open cur_Teq using $a_nam, $a_attr, "=   ";
                  print_status("22");
        $open cur_Tgt using $a_nam, $a_attr, ">   ",
                                $a_nam, $a_attr, ">=  ";
                  print_status("23");
        $open cur_Tlt using $a_nam, $a_attr, "<   ",
                                $a_nam, $a_attr, "<=  ";
                  print_status("24");
```

```
   $fetch cur5 into $s_val;
   if (sqlca.sqlcode != SQLNOTFOUND)
           $execute updte_suN using $y, $a_nam,
                   $a_attr, $dom, "!=  ", $s_val;

   /* STEP 1 FOR EVERY TUPLE Price != 50 FROM APPL_RULES OR */
   /*         DB_RULES LOOKS IN TEMPORARY FOR Price = 50 IN  */
   /*         WHICH CASE THERE IS THE ONLY POSSIBLE CONFLICT */
   /*         IN THIS PROCEDURE                              */

   $fetch cur_Teq into $s_val;
   if (sqlca.sqlcode != SQLNOTFOUND)
           if (strcmp (s_val, a_val) == 0)
               cond = 0;
               /* THIS IS THE ONLY CONFLICT. cond = 0 SIGNALS IT*/
           else ne_condition = 1;
    else
     {
           $fetch cur_Tgt into $s_op, $s_val;
           if (sqlca.sqlcode != SQLNOTFOUND)
           {
           if ((compare_strings(a_val, s_val) == 0)
                   && (strcmp(s_op, ">=  ") == 0))
               {

                     /* STEP 2 IF WE HAVE TUPLE Price != 50    */
                     /*   AND IN TEMPORARY WE HAVE Price >= 50 */
                     /*   UPDATE TEMPORARY TO SAY Price > 50.  */
                     $execute updte_Tup using ">   ", $a_nam,
                           $a_attr, $s_op, $s_val;
                                     ne_condition = 1;

               }
           else if ((compare_strings(a_val, s_val) < 0)
                    || ((compare_strings(a_val, s_val) == 0)
                        && (strcmp(s_op, ">   ") == 0)))
                     ne_condition = 1;
           }

           $fetch cur_Tlt into $s_op, $s_val;
           if (sqlca.sqlcode != SQLNOTFOUND)
           {
            if ((compare_strings(a_val, s_val) == 0)
                   && (strcmp(s_op, "<=  ") == 0))
                   {

                     /* STEP 2 IF WE HAVE TUPLE Price != 50    */
                     /*   AND IN TEMPORARY WE HAVE Price <= 50 */
                     /*   UPDATE TEMPORARY TO SAY Price < 50.  */
                     $execute updte_Tup using "<   ", $a_nam,
                           $a_attr, $s_op, $s_val;
                    ne_condition = 1;

                   }
           else if ((compare_strings(a_val, s_val) > 0)
                    || ((compare_strings(a_val, s_val) == 0)
                        && (strcmp(s_op, "<   ") == 0)))

                     ne_condition = 1;
           }
   }


/* STEP 3 IF IT IS NOT REDUNDANT TO WRITE THE != TUPLE INTO THE */
/* TEMPORARY TABLE, ne_condition IS 0, AND THE TUPLE IS WRITTEN */
```

```
             if (ne_condition == 0)
                $insert into temporary
                        values ($a_nam, $a_attr,
                                $a_op, $a_val);
            $close cur5;
            $close cur_Teq;
            $close cur_Tgt;
            $close cur_Tlt;
            }
           $close cur3;
           $close cur4;
           return(cond);
        }



int compare_strings(f, g)
        char *f;
        char *g;
        {
        float k;
        float l;
        int  res;
        k = str_to_float (f);
        l = str_to_float (g);
        printf("RETURNS %f %f\n", k, l);
        if (k > l)
                res = 1;
        if (l > k)
                res = -1;
        if (l == k)
                res = 0;
        return (res);
        }
```

PART VII

Appendix 2

```
#include <stdio.h>
#include <string.h>
$include sqlca;
$include sqlda;


/* Procedure Declare_Check_Query_Select_Cursor */
/* Is called only once at the statrt of query processing program */

Declare_Check_Query_Where_Cursors()
        {
          $char  a_str[300];
          $char  c_str[300];
          $char  sel_c_str[300];
          $char  d_str[300];
          $char  e_str[300];
          $char  number_no_str[300];
          $char  bad_str[300];
          $char  where_str[300];
          $char  scan_check_str[300];

          strcpy (c_str, "select unique attr_name from c_list ");

          strcpy (a_str, "select unique attrib from a_list ");

          strcpy (scan_check_str, "select unique attrib from check_list ");

          strcpy (sel_c_str, "select c_number, operation, value, mark ");
          strcat (sel_c_str, " from c_list ");
          strcat (sel_c_str, " where attr_name = ? ");


          strcpy (bad_str, "select unique appl_rule, db_rule ");
          strcat (bad_str, " from bad ");
          strcat (bad_str, " where np_attr = ? ");

          strcpy (number_no_str, "select atr_name, arg_name, operation, primitive,
          strcat (number_no_str, " from no_table ");
          strcat (number_no_str, " where r_number = ? AND s_number = ? ");

          strcpy (where_str, "select unique r_number ");
          strcat (where_str, " from quer_cnd_tmp ");

          strcpy (d_str, "select unique r_number ");
          strcat (d_str, " from quer_cnd_tmp ");
          strcat (d_str, " where r_number = ? ");

          strcpy (e_str,
                "select atr_name, arg_name, operation, primitive, domain_type ");
          strcat (e_str, "  from quer_cnd_tmp ");
          strcat (e_str, " where r_number = ? ");

          $prepare C_query from $c_str;
          print_status("Prepare C");
          $declare C_cur cursor for C_query;
          print_status("Declare C");

          $prepare sel_C_query from $sel_c_str;
          print_status("Prepare sel_C");
          $declare sel_C_cur cursor for sel_C_query;
          print_status("Declare sel_C");

          $prepare A_query from $a_str;
          print_status("Prepare A");
          $declare A_cur cursor for A_query;
          print_status("Declare A");
```

```
          $prepare where_query from $where_str;
          print_status("Prepare where");
          $declare where_cur cursor for where_query;
          print_status("Declare where");

          $prepare scan_check_query from $scan_check_str;
          print_status("Prepare Scan_Check");
          $declare scan_check_cur cursor for scan_check_query;
          print_status("Declare Scan_Check");

          $prepare D_query from $d_str;
          print_status("Prepare D");
          $declare D_cur cursor for D_query;
          print_status("Declare D");

          $prepare E_query from $e_str;
          print_status("Prepare E");
          $declare E_cur cursor for E_query;
          print_status("Declare E");


          $prepare BAD_query from $bad_str;
          print_status("Prepare BAD");
          $declare BAD_cur cursor for BAD_query;
          print_status("Declare BAD");

          $prepare num_NO_query from $number_no_str;
          print_status("Prepare num_NO");
          $declare num_NO_cur cursor for num_NO_query;
          print_status("Declare num_NO");

          $declare copyC_cur cursor for select * from c_list;
          $declare copyTemp_cur cursor for select * from temporary;
          $declare copy_no_cur cursor for select * from no_conditions;
        }

main ()
     {
        long    conditions;
        $char   c_attr[30];
        $char   a_attr[30];
        $char   chk_attr[30];
        long    no_count;
        long    c_count;
        int     result;

        result = 1;
        conditions = 0;
        no_count = 0;

        /*              PART I                                    */
        /* PREPARES THE ENVIRONMENT FOR EXECUTING THE ALGORITHM   */
        /* BY DECLARING THE CURSORS (INCLUDING THE SUBSUMPTION'S) */

        $database cdrdb;
        Declare_Check_Query_Where_Cursors();
        DeclareSubsumptionCursors();
        Declare_Build_Cursor();

        $delete from procedure_result;
        print_status("Clear procedure_result"
        $update no_table set mark = 0;
        print_status("Update NO_TABLE");

        /*            PART II                                     */
```

```
/* CALLS Prepare_Query_Conditions TO SEPARATE THE        */
/* CONDITIONS IN THE WHERE CLAUSE IN TWO: THOSE DEALING  */
/* WITH NON-PRIMITIVE ATTRIBUTES, AND THOSE DEALING WITH */
/* PRIMITIVE ATTRIBUTES. Prepare_Query_Conditions COPIES */
/* ALL CONDITIONS DEALING WITH NON-PRIMITIVE ATTRIBUTES  */
/* DIRECTLY INTO THE quer_cnd_tmp TABLE; AND COPIES ALL  */
/* THE NAMES OF THE NON-PRIMITIVE ATTRIBUTES APPEARING IN */
/* THE WHERE CLAUSE INTO THE check_list TABLE            */

Prepare_Query_Conditions();
Clear_Tables();

/*           PART III                                    */
/* PERFORMS THE FOLLOWING LOOP FOR EVERY NON-PRIMITIVE   */
/* ATTRIBUTE IN THE WHERE CLAUSE (ie. every attribute in */
/* the check_list table.):                               */
/*    STEP I                                             */
/*      CALLS Arrange_Query_Conditions, WHICH            */
/*      COPIES THE quer_cnd_tmp TABLE INTO THE           */
/*      query_conditions TABLE, WHERE THE r_numbers ARE  */
/*      SET UP IN INCREASING NUMBERS, AS THE SUBSUMPTION'S*/
/*      SPECIFICATIONS REQUIRE.                          */
/*    STEP II                                            */
/*      COPIES ALL THE CONDITIONS IN THE NO_TABLE WHICH  */
/*      CAN MAKE THE CURRENT NON_PRIMITIVE ATTRIBUTE FROM */
/*      THE WHERE CLAUSE PRODUCE A SEMANTIC CONFLICT INTO */
/*      THE CONDITIONS, LEAVING THE STAGE READY FOR A    */
/*      SUBSUMPTION.                                     */
/*    STEP III                                           */
/*      CALLS Execute_subsumpt,  WHICH                   */
/*      SUBSUMES  THE NO_CONDITIONS TABLE WITH THE       */
/*      QUERY_CONDITIONS TABLE, AND LEAVES THE RESULT IN */
/*      THE Intermediate TABLE. THIS TABLE NOW CONTAINS  */
/*      THE CONDITIONS UNDER WHICH THE CURRENT           */
/*      NON-PRIMITIVE FROM THE WHERE CLAUSE CAN PRODUCE  */
/*      SEMANTIC CONFLICTS                               */
/*    STEP IV                                            */
/*      CALLS THE Query_Rebuild PROCEDURE, WHICH,        */
/*      FOR EVERY OF THE OR CONDITIONS IN THE Intermediate*/
/*      TABLE MAKES A QUERY TO THE DATABASE. IF ALL OF THE*/
/*      RETURN NULL RESULTS, THEN THE CURRENT ATTRIBUTE IN*/
/*      THE WHERE CLAUSE DOES NOT CAUSE A SEMANTIC       */
/*    STEP V                                             */
/*      COPIES ALL CONDITIONS DEALING WITH THAT ATTRIBUTE */
/*      FROM THE c_list INTO THE quer_cnd_tmp TABLE.     */

$open scan_check_cur;
print_status ("Open Scan_Check");
while (1)
  {


        $fetch scan_check_cur into $c_attr;
        print_status("Fetch Scan_Check");
        if (sqlca.sqlcode == SQLNOTFOUND)
                break;

        /*      STEP I                                  */
        Clear_Tables();
        c_count = Arrange_Query_Conditions();
        printf ("C_count = %i\n", c_count);
        /*      STEP II                                 */
        no_count = Copy_No_To_No_Conditions(c_attr);
        printf ("No_count = %i\n", no_count);

        if (no_count > 0)
```

```
                {
                     /* IF NO_COUNT = 0, IT MEANS THAT THERE ARE NO */
                     /* POSSIBILITIES OF HAVING A SEMANTIC CONFLICT,*/
                     /* SO THERE IS NO NEED TO MAKE THE INTERMEDIATE*/
                     /* QUERY                                       */
                 if (c_count == 0)
                  {
                    conditions = no_count;
                    Copy_No_To_Intermediate();
                   }
                  else
                        {
                         /*     STEP III                         */
                         conditions = Execute_Subsumpt(no_count, c_count);
                        }
                 printf("Conditions = %i\n", conditions);
                 /*     STEP IV                                  */
                 result = Query_Rebuild(c_attr, conditions);
                 if (result == 0)
                        break;
                }
                /*              STEP V                           */
                Copy_C_to_Query_Conditions(c_attr);

         }


        /*              PART IV                                   */
        /* IF THERE IS NO SEMANTCI CONFLICT IN THE WHERE CLAUSE, */
        /* INSERTS THE VALUE OF 1 IN THE procedure_result TABLE. */
        /* OTHERWISE INSERTS THE VALUE OF 1 IN THAT TABLE.       */
        printf("RESULT = %i\n", result);
        $insert into procedure_result values(result);
}

Copy_No_To_Intermediate()
{
  $long r_n;
  $char at_n[30];
  $char ar_n[30];
  $char op[5];
  $char prim[30];
  $char domn[3];
  $long mk;


        $open copy_no_cur;
        print_status("OPen Copy No");
        while (1)
        {
         $fetch copy_no_cur into $r_n, $at_n, $ar_n, $op, $prim, $domn, $mk;
         print_status ("Fetch Copy No");
         if (sqlca.sqlcode == SQLNOTFOUND)
                break;
         $insert into intermediate
                values ( $r_n, $at_n, $ar_n, $op, $prim );
         print_status("Insert from no to intermediate");
        }
}



Copy_C_to_Query_Conditions (c_attr)
   $char c_attr[30];
   {
        $long cnum;
```

```
        $char c_op[5];
        $char cval[30];
        $long cmark;

        $open sel_C_cur using $c_attr;
        print_status("Open Sel_C");
        while (1)
        {
          $fetch sel_C_cur into $cnum, $c_op, $cval, $cmark;
          print_status("Fetch Sel_C");
          if (sqlca.sqlcode == SQLNOTFOUND)
                break;
          $insert into quer_cnd_tmp
                values ( $cnum,
                         "",
                         $c_attr,
                         $c_op,
                         $cval,
                         "A",
                         $cmark );
         print_status("Insert into query_cnd_tmp");
        }
   }


Prepare_Query_Conditions()
     {
          $long a;
          $long b;
          $char c_attr[30];

          $delete from quer_cnd_tmp;
          print_status(" Clear Quer_Cnd_Tmp");

          $open C_cur;
          print_status("Open C");
          while (1)
            {

                $fetch C_cur into $c_attr;
                print_status("Fetch C");
                if (sqlca.sqlcode == SQLNOTFOUND)
                        break;

                $open BAD_cur using $c_attr;
                print_status("Open BAD I");
                $fetch BAD_cur into $a, $b;
                print_status("Fetch BAD I");
                if (sqlca.sqlcode == SQLNOTFOUND)
                    Copy_C_to_Query_Conditions(c_attr);
                else Insert_Into_Check_List(c_attr);
                $close BAD_cur;
                print_status("Close BAD I");
             }
          $close C_cur;
          print_status("Close C");
     }

Insert_Into_Check_List (c_attr)
   $char c_attr[30];
    {
        $insert into check_list
                values ( $c_attr );
    }

long Number_Where_Conditions ()
```

```
   {
     $long r_n;
     long   conds_count;

     conds_count = 0;
     $open where_cur;
     print_status("Open where I");
     while (1)
     {
        $fetch where_cur into $r_n;
        print_status("Fetch where I");
        if (sqlca.sqlcode == SQLNOTFOUND)
               break;
        conds_count++;
     }
     return(conds_count);
    }

long Arrange_Query_Conditions ()
   {
        $long cond_count;
        $long current_count;
        $long actual_count;
        $long  dmb;
        $char tat_name[30];
        $char tar_name[30];
        $char top[5];
        $char tprim[30];
        $char tdom[3];

        cond_count = Number_Where_Conditions();
        current_count = 1;
        actual_count = 1;
        while (current_count <= cond_count)
          {
            while (1)
            {
             $open D_cur using $actual_count;
             print_status ("Open D");
             $fetch D_cur into $dmb;
             print_status ("Fetch D");
             if (sqlca.sqlcode != SQLNOTFOUND)
                break;
             else actual_count++;
            }

           $open e_cur using $actual_count;
           print_status ("Open E");
           while (1)
            {
              $fetch e_cur into $tat_name, $tar_name, $top, $tprim, $tdom;
              print_status("Fetch E");
              if (sqlca.sqlcode == SQLNOTFOUND)
                break;
              $insert into query_conditions
                values($current_count,
                        $tat_name,
                        $tar_name,
                        $top,
                        $tprim,
                        $tdom,
                        0);
               print_status("Insert into query_conditions");
             }

           current_count++;
```

```
        return(cond_count);
   }


long Copy_No_To_No_Conditions(c_attr)
  $char c_attr[30];
  {
        $long no_count;
        $long s_n;
        $long r_n;
        $char nat_name[30];
        $char nar_name[30];
        $char nop[5];
        $char nprim[30];

        no_count = 0;
        $open BAD_cur using $c_attr;
        print_status("Open BAD II");
        while (1)
        {
          $fetch BAD_cur into $s_n, $r_n;
          print_status("Fetch BAD II");
          if (sqlca.sqlcode == SQLNOTFOUND)
                break;

          no_count++;
          $open num_NO_cur using $s_n, $r_n;
          print_status("Open NO");
          while (1)
           {
                $fetch num_NO_cur into $nat_name, $nar_name, $nop, $nprim;
                print_status("Fetch NO");
                if (sqlca.sqlcode == SQLNOTFOUND)
                        break;
                $insert into no_conditions
                    values ( $no_count,
                                "",
                                $nar_name,
                                $nop,
                                $nprim,
                                "A",
                                0 );
                print_status("Insert into No_conds");
           }
        }
        return(no_count);

   }




long Execute_Subsumpt (a_num, s_num)
        long  a_num;
        long  s_num;

        {
         int result;
         long count;
         long x;
         long y;
```

```
         count = 0;
         result = 0;
         printf("ex subs %i %i\n", a_num, s_num);

         /* ENTERS DOUBLE LOOP TO FIND EVERY POSSIBLE COMBINATION OF CONDITIONS */
         /* FROM THE NO_CONDITIONS AND DB_RULES TABLE                           */
         for (x=1; x<=a_num; x++)
         {
          for (y = 1; y <= s_num; y++)
          {
            Clear_Temporary();


           /* FIND INTERSECTION OF CONDITIONS BY CALLING CanBeSubsumed AND     */
           /* NotEqStuff.                                                      */
            result = CanBeSubsumed(x,y);
            if (result == 1)
             {
                result = NotEqualStuff(x,y);
               if (result == 1)
                  {
                     /* IF CONDITIONS INTERSECT INCREMENT count AND COPY TO    */
                     /* intermediate TABLE                                     */
                     count++;
                     Copy_Temp_To_Interm(count);
                  }
             }
          }
        }
        /* RETURN COUNT OF SUCCESSFULL INTERSECTIONS TO FINISH                 */
        return(count);
       }

Declare_Build_Cursor()
        {
         $char bui_str[300];

         strcpy (bui_str,
                 "select atr_name, arg_name, operation, primitive ");
         strcat (bui_str, " from intermediate ");
         strcat (bui_str, "where r_number = ? ");

         $prepare BUI_query from $bui_str;
         print_status("Prepare BUI");
         $declare BUI_cur cursor for BUI_query;
         print_status("Declare BUI");
        }


int Query_Rebuild(c_attr, conditions)
        $char  c_attr[30];
        long    conditions;
        {
           $long x;
           $char i_attr[30];
           $char i_arg[30];
           $char i_op[30];
           $char i_val[30];
           $char   dbquery_str[600];
           int     nonecond;
           int     result;
           int     s_cas;


           result = 1;
           printf("Q_REBUILD %i %s", conditions, c_attr);
```

```
           for (x = 1; x <= conditions; x++)
           {

                strcpy (dbquery_str, "select  ");
                strcat (dbquery_str, c_attr );
                strcat (dbquery_str, "  from db_table ");
                strcat (dbquery_str, " where \n  ( ");

                $open BUI_cur using $x;
                print_status("Open BUI");
                nonecond = 0;


                while (1)
                  {

                    $fetch BUI_cur into $i_attr, $i_arg, $i_op, $i_val;
                    printf("BUI x %i sql %i\n", x, sqlca.sqlcode);
                    print_status("Fetch BUI");
                    if (sqlca.sqlcode == SQLNOTFOUND)
                        break;
                    if (nonecond == 0)
                        nonecond++;
                      else  strcat (dbquery_str, "\n AND \n ");

                    s_cas = is_string(i_val);

                    strcat (dbquery_str, i_arg);
                    strcat (dbquery_str, "  ");
                    strcat (dbquery_str, i_op);
                    strcat (dbquery_str, "  ");
                    if (s_cas == 0)
                        strcat (dbquery_str, "'");
                    strcat (dbquery_str, i_val);
                    if (s_cas == 0)
                        strcat (dbquery_str, "'");
                 }

              strcat (dbquery_str, " )");

              printf("%s\n", dbquery_str);
              $prepare DBSEL_query from $dbquery_str;
              print_status("Prepare DBSEL_query");
              $declare DBSEL_cur cursor for DBSEL_query;
              print_status("Declare DBSEL_cur");
              $open DBSEL_cur;
              print_status("Open DBSEL_cur");
              $fetch DBSEL_cur into $c_attr;
              print_status("Fetch DBSEL_cur");
              if (sqlca.sqlcode != SQLNOTFOUND)
                {
                 result = 0;
                 break;
                }
            }
          return(result);
        }


int is_string (j)
     char *j;
     {
        int i;
        int val;
```

```
        int result;

        result = 0;
        i = 0;
        while (1)
         {
          if (j[i] == '\0' || j[i] == '\n' || j[i] == ' ')
                break;
          if ((j[i] != '-') && (j[i] != '.'))
             {
                if (j[i] == 'e')
                   result = 0;
                else
                {
                  val = j[i] - '0';
                  if ((val < 0) || (val > 9))
                   {
                      result = 0;
                      break;
                   }
                  else
                   result = 1;
                }
             }
           i++;
          }
     return(result);
     }



Clear_No_Conditions()
        {
          $delete from no_conditions;
           print_status("delete NO_CONDS");
        }
Clear_Temporary()
        {
          $delete from temporary;
           print_status("delete temp");
        }

Clear_Tables ()
        {
           $delete from intermediate;
           print_status("delete intermediate");
           $delete from temporary;
           print_status("delete temp");
           $delete from no_conditions;
           print_status("delete no_conditions");
           $delete from query_conditions;
           print_status("delete query_conditions");
        }


Copy_Temp_To_Interm(t_num)
        $long   t_num;
        {
         $char t_attr[30];
         $char t_arg[30];
         $char t_op[5];
         $char t_val[30];

        printf("COPY TEMP %i", t_num);
        $open copyTemp_cur;
        print_status("open copyTemp_cur");
```

```
    while (1)
    {
     $fetch copyTemp_cur into $t_attr, $t_arg, $t_op, $t_val;
     printf("COPY TEMP TO I %i", sqlca.sqlcode);
     print_status("fetch copyTemp_cur");
     if (sqlca.sqlcode == SQLNOTFOUND)
            break;
     $insert into intermediate
            values ($t_num, $t_attr, $t_arg, $t_op, $t_val);
     print_status("insert into intermediate");
    }
    $close copyTemp_cur;
    print_status("close copyTemp_cur");
   }




/* *******************************************************************   */
/* HERE GO THE CanBeSubsumed, NotEqStuff, DeclareSubsumptionCursors      */
/* procedures. Omitted in this print out.                                */
```

```
#include <stdio.h>
#include <string.h>
$include sqlca;
$include sqlda;


/* Procedure Declare_Check_Query_Select_Cursor */
/* Is called only once at the statrt of query processing program */

Declare_Check_Query_Select_Cursors()
       {
         $char  a_str[300];
         $char  check_c[300];
         $char  bad_str[300];
         $char  no_str[300];

          strcpy (a_str, "select attrib from a_list ");

         strcpy (check_c,
               "select attr_name ");
          strcat (check_c, " from c_list where attr_name= ?");

         strcpy (bad_str, "select unique appl_rule, db_rule ");
         strcat (bad_str, " from bad ");
         strcat (bad_str, " where np_attr = ? ");

         strcpy (no_str, "select atr_name, arg_name, operation,primitive, mark ");
         strcat (no_str, " from no_table ");
         strcat (no_str, " where r_number = ? AND s_number = ? ");


         $prepare A_query from $a_str;
         print_status("Prepare A");
         $declare A_cur cursor for A_query;
         print_status("Declare A");

         $prepare checkC_query from $check_c;
         print_status("Prepare check_C");
         $declare checkC_cur cursor for checkC_query;
         print_status("Declare C");

         $prepare BAD_query from $bad_str;
         print_status("Prepare BAD");
         $declare BAD_cur cursor for BAD_query;
         print_status("Declare BAD");

         $prepare NO_query from $no_str;
         print_status("Prepare NO");
         $declare NO_cur cursor for NO_query;
         print_status("Declare NO");

         $declare copyC_cur cursor for select * from c_list;
         $declare copyTemp_cur cursor for select * from temporary;
       }
/* THE MAIN PROCEDURE DOES THE FOLLOWING OPERATIONS          */
/* SATRTS BY CREATING THE CURSORS USED IN THE PROGRAM,       */
/* DECLARING THE CURRENT DATABASE, AND COPYING THE C_LIST    */
/* INTO THE QUERY_CONDITIONS TABLE BY CALLING Copy_C_To_Db. THIS     */
/* RETURNES THE NUMBER OF ORED CONDITIONS IN THE QUERY_CONDITIONS    */
/* TABLE THAT ARE GOING TO BE SUBSUMED.                      */
/* THEN, FOR EVERY ATTRIBUTE IN THE SELECT CLAUSE PERFORMS   */
/* THE FOLLOWING 5 STEPS:                                    */
/* FIRST ERRASE TEMPORARY, INTERMEDIATE, AND ALL TUPLES      */
/*       WITH IN QUERY_CONDITIONS, TO SET THE     */
/*       STAGE FOR PERFORMING THE SUBSUMPTION OF STEP 4      */
/* SECOND CHECK IF THE TUPLE IS ALSO IN THE WHERE CLAUSE.    */
```

```
/*        IF IT IS, SKIP THE NEXT STEPS, AND IF IT IS NOT     */
/*        CONTINUE WITH STEP 3.                               */
/* THIRD CALL CheckQuerySelect WITH THE ATTRIBUTE NAME AS     */
/*        ARGUMENT. THIS PREPARES THE APPL_RULES TABLE BY     */
/*        COPYING THE RELEVANT CONDITIONS FROM THE NO_TABLE   */
/*        FOR THE SUBSUMPTION OF STEP 4. THE RESULT RETURNED  */
/*        BY CheckQuerySelect IS HOW MANY ORED CONDITIONS     */
/*        ARE IN THE NO_TABLE FOR THE NEXT STEP.              */
/* FOURTH IF BOTH CheckQuerySelect AND Copy_C_To_Db           */
/*        RETURNED VALUES GREATER THAN 0, Execute_Subsumpt    */
/*        IS CALLED TO PUT THE INTERSECTION OF THE C_LIST     */
/*        AND RELEVANT PART OF NO_TABLE IN THE INTERMEDIATE   */
/*        TABLE.                                              */
/* FIFTH THE PROCEDURE Query_Rebuild IS CALLED TO TAKE THE    */
/*        CONTENTS OF THE INTERMEDIATE TABLE, AND PERFORM A   */
/*        QUERY TO DB_TABLE, TO SEE IF SOME ATTRIBUTES IN THE*/
/*        SELECT CLAUSE ARE IN THE WRONG CONTEXT.             */
/*                                                            */
/* FINALLY, IF Query_Rebuild RETURNS 0, THEN A CONFLICT       */
/* OCCURS. THEN THE VALUE OF 0 IS STORED IN THE MESSAGE       */
/* TABLE, AND THE PROGRAM ENDS. IF ON THE OTHER HAND THE      */
/* PROGRAM REACHES THE END WITHOUT HAVING FOUND A CONFLICT    */
/* IN ANY ATTRIBUTE ON THE SELECT CLAUSE, THEN IT STORES 1    */
/* IN THE MESSAGE TABLE AND FINISHES.                         */

main ()
    {
       long    conditions;
       $char   c_attr[30];
       $char   chk_attr[30];
       long    no_count;
       long    c_count;
       int     result;
       $char   up_db[300];

       $database cdrdb;
       strcpy (up_db, "update query_conditions set atr_name = ? ");
       $prepare db_update from $up_db;
       print_status("Prepare dbup");

       result = 1;
       conditions = 0;
       no_count = 0;

       /* SET UP ENVIRONMENT                                  */

       Declare_Check_Query_Select_Cursors();
       DeclareSubsumptionCursors();
       Declare_Build_Cursor();
       Clear_Query_Conditions();
       $delete from procedure_result;

       /* COPY C_LIST TO QUERY_CONDITIONS AND FIGURE HOW MANY CONDITIONS */
       /* ARE IN IT                                                      */
       c_count = Copy_C_To_Query_Conditions();
       $update no_table set mark = 0;
       print_status("Update NO_TABLE");
       $open A_cur;


         /* THE FOLLOWING LOOP IS PERFORMED FOR EVERY TUPLE IN    */
         /* THE SELECT CLAUSE                                     */
         while (1)
           {

              $fetch A_cur into $c_attr;
```

```
                print_status("Fetch A");
                if (sqlca.sqlcode == SQLNOTFOUND)
                        break;

                /* STEP 1                                          */
                Clear_Tables();
                printf("A selects %s\n", c_attr);

                /* STEP 2                                          */
                $open checkC_cur using $c_attr;
                print_status("Open CheckC");
                $fetch checkC_cur into $chk_attr;
                print_status("Fetch CheckC");
                if (sqlca.sqlcode == SQLNOTFOUND)
                {
                   $execute db_update using $c_attr;
                   /* STEP 3                                       */
                   no_count =  CheckQuerySelect(c_attr);
                    printf ("No_count = %i\n", no_count);
                   if (no_count > 0)
                   {
                     /* STEP 4                                     */
                     conditions = Execute_Subsumpt(no_count, c_count);
                     printf("Conditions = %i\n", conditions);
                     /* STEP 5                                     */
                     result = Query_Rebuild(c_attr, conditions);
                     if (result == 0)
                        break;
                   }
                 }
             }

        printf("RESULT = %i\n", result);
        $insert into procedure_result values(result);
}

/* CHECKQUERYSELECT GETS FROM MAIN THE NAME OF AN ATTRIBUTE WHICH IS IN THE  */
/* SELECT CLAUSE BUT NOT IN THE WHERE CLAUSE. IT LOOKS IN THE BAD TABLE FOR  */
/* EVERY APPEARANCE OF THAT ATTRIBUTE, AND FOR EACH PERFORMS THE FOLLOWUING  */
/* 4 STEPS:                                                                  */
/* FIRST CHECK IF THE TUPLES WITH ATRIBUTE NAME AND RULE NUMBERS IN THE      */
/*     NO_TABLE EQUAL TO THOSE FETCHED FROM THE BAD TABLE ARE ALREADY MARKED.*/
/*     IN WHICH CASE NOTHING IS DONE WITH THEM. OTHERWISE INCREMENT THE      */
/*     COUNT OF TUPLES GOT FROM THE BAD TABLE (cont) AND GOTO SET 2.         */
/* SECOND COPY THE TUPLES IN THE NO TABLE INDEXED BY THE ATRIBUTE NAME AND   */
/*     RULE NUMBERS FROM THE BAD TABLE INTO THE NO_CONDITIONS TABLE, AND MARK   */
/*     THEM.                                                                 */
/* THIRD WHEN THERE ARE NO MORE TUPLES TO FETCH FROM THE BAD TABLE, FINISH   */
/*     RETURNING THE VALUE count.                                            */

long CheckQuerySelect (c_attr)
        $char   c_attr[30];
        {
          $long  rule_num;
          $long  a_r;
          $long  s_r;
          $long  n_value;
          $char  updte_no[300];
          $char  s_attr[30];
          $char  s_arg[30];
          $char  s_op[5];
          $char  s_val[30];
          long   count;

          count = 0;
```

```
       strcpy (updte_no, "update no_table ");
       strcat (updte_no, " set mark = 1 ");
       strcat (updte_no, " where r_number = ? AND s_number = ? ");
       strcat (updte_no, " AND atr_name = ? and arg_name = ? ");
       strcat (updte_no, " AND operation = ? AND primitive = ? ");


      $prepare upno_query from $updte_no;
      print_status("Prepare upno");

       $open BAD_cur using $c_attr;
       print_status("Open BAD");
       while (1)
       {

          /* THE FOLLOWING STEPS ARE PERFORMED ON ALL THE TUPLE IN THE BAD */
          /* TABLE WITH np_attr FIELD = c_attr.                            */
          $fetch BAD_cur into $a_r, $s_r;
          print_status("Fetch BAD");
          printf("BAD %i\n", sqlca.sqlcode);
          if (sqlca.sqlcode == SQLNOTFOUND)
                    break;

          count++;
          rule_num = count;
          $open NO_cur using $a_r, $s_r;
          print_status("Open NO");

          while (1)
          {

              /* STEP 1, CHECK IF NO TABLE ATTRIBUTES */
              /* ARE MARKED, IN WHICH CASE   */
              /* n_value IS 1. IF MARKED, DON'T COPY OR */
              /* COUNT THEM: (no_cond ==0) */
              $fetch NO_cur into $s_attr, $s_arg, $s_op, $s_val, $n_value;
              print_status("Fetch NO");
              if (sqlca.sqlcode == SQLNOTFOUND)
                    break;
              if (n_value == 1)
                {
                    count = count - 1;
                    break;
                }
              /* STEP 2 IF ATTRIBUTES FROM NO_TABLE ARE */
              /* NOT MARKED, COPY THEM INTO */
              /* NO_CONDITIONS AND MARK THEM      */
              $insert into no_conditions
                     values($rule_num, $s_attr, $s_arg, $s_op,
                            $s_val,"A", 0);
              print_status("Insert A_RULES");
              $execute upno_query using $a_r, $s_r, $s_attr, $s_arg,
                                        $s_op, $s_val;
              print_status("Execute UPNO");
          }
          $close NO_cur;
          print_status("Close NO");

      }
      $close BAD_cur;
      print_status("Close BAD");

     /* STEP 3 RETURN count.                          */
      return(count);
}
```

```
/* EXECUTE_SUBSUMPT RECEIVES ARE ARGUMENT HOW MANY OR CONDITIONS ARE IN THE    */
/* NO_CONDITIONS AND QUERY_CONDITIONS, AND FINDS THEIR INTERSECTION */
/* BY CALLING THE PROCEDURES CanBeSubsumed AND NotEqStuff, FOR EVERY POSSIBLE */
/* COMBINATION OF CONDITIONS FROM THE NO_CONDITIONS AND QUERY_CONDITIONS.
/* IF BOTH PROCEDURES RETURN 1 FOR A COMBINATION OF CONDITIONS, INCREMENTS     */
/* count, AND COPIES THE CONDITIONS INTO THE INTERMEDIATE TABLE.              */
/* WHEN THERE ARE NO MORE COMBINATIONS OF CONDITIONS LEFT TO SUBSUME RETURNS  */
/* count.                                                                     */

long Execute_Subsumpt (a_num, s_num)
        long  a_num;
        long  s_num;

        {
         int result;
         long count;
         long x;
         long y;

         count = 0;
         result = 0;
         printf("ex subs %i %i\n", a_num, s_num);

         /* ENTERS DOUBLE LOOP TO FIND EVERY POSSIBLE COMBINATION OF CONDITIONS */
         /* FROM THE NO_CONDITIONS AND QUERY_CONDITIONS TABLE
         for (x=1; x<=a_num; x++)
         {
          for (y = 1; y <= s_num; y++)
          {
            Clear_Temporary();

           /* FIND INTERSECTION OF CONDITIONS BY CALLING CanBeSubsumed AND     */
           /* NotEqStuff.                                                      */
            result = CanBeSubsumed(x,y);
            if (result == 1)
             {
                result = NotEqualStuff(x,y);
               if (result == 1)
                  {
                     /* IF CONDITIONS INTERSECT INCREMENT count AND COPY TO    */
                     /* INTERMEDIATE TABLE                                     */
                     count++;
                     Copy_Temp_To_Interm(count);
                  }
             }
          }
         }
        /* RETURN COUNT OF SUCCESSFULL INTERSECTIONS TO FINISH                 */
        return(count);
       }

Declare_Build_Cursor()
        {
         $char bui_str[300];

         strcpy (bui_str,
                 "select atr_name, arg_name, operation, primitive ");
         strcat (bui_str, " from intermediate ");
         strcat (bui_str, "where r_number = ? ");

         $prepare BUI_query from $bui_str;
         print_status("Prepare BUI");
         $declare BUI_cur cursor for BUI_query;
         print_status("Declare BUI");
        }
```

```
int Query_Rebuild(c_attr, conditions)
        $char  c_attr[30];
        long    conditions;
        {
           $long x;
           $char i_attr[30];
           $char i_arg[30];
           $char i_op[30];
           $char i_val[30];
           $char   dbquery_str[600];
           int     nonecond;
           int     result;
           int     s_cas;


           result = 1;
           printf("Q_REBUILD %i %s", conditions, c_attr);

           strcpy (dbquery_str, "select  ");
           strcat (dbquery_str, c_attr );
           strcat (dbquery_str, "  from db_table ");
           strcat (dbquery_str, " where \n  ( ");

           for (x = 1; x <= conditions; x++)
           {
                $open BUI_cur using $x;
                print_status("Open BUI");
                nonecond = 0;


                while (1)
                  {

                    $fetch BUI_cur into $i_attr, $i_arg, $i_op, $i_val;
                    printf("BUI x %i sql %i\n", x, sqlca.sqlcode);
                    print_status("Fetch BUI");
                    if (sqlca.sqlcode == SQLNOTFOUND)
                        break;
                    if (nonecond == 0)
                        nonecond++;
                      else  strcat (dbquery_str, "\n AND \n ");

                    s_cas = is_string(i_val);

                    strcat (dbquery_str, i_arg);
                    strcat (dbquery_str, "  ");
                    strcat (dbquery_str, i_op);
                    strcat (dbquery_str, "  ");
                    if (s_cas == 0)
                        strcat (dbquery_str, "'");
                    strcat (dbquery_str, i_val);
                    if (s_cas == 0)
                        strcat (dbquery_str, "'");
                 }

              strcat (dbquery_str, " )");

              printf("%s\n", dbquery_str);
              $prepare DBSEL_query from $dbquery_str;
              print_status("Prepare DBSEL_query");
              $declare DBSEL_cur cursor for DBSEL_query;
              print_status("Declare DBSEL_cur");
              $open DBSEL_cur;
              print_status("Open DBSEL_cur");
              $fetch DBSEL_cur into $c_attr;
```

```
              if (sqlca.sqlcode != SQLNOTFOUND)
                {
                 result = 0;
                 break;
                }
            }
          return(result);
        }



Clear_Temporary()
        {
          $delete from temporary;
           print_status("delete temp");
        }

Clear_Query_Conditions ()
        {
          $delete from query_conditions;
         print_status("delete query_conditions");
        }

Clear_Tables ()
        {
           $delete from intermediate;
           print_status("delete Intermediate");
           $delete from no_conditions;
            print_status("delete no_conditions");
           $delete from temporary;
           print_status("delete temp");
        }


long Copy_C_To_Query_Conditions()
        {
         $long c_num;
         $long ci_num;
         $long dum;
         long  total;
         $char c_arg[30];
         $char c_op[5];
         $char c_val[30];

        total = 0;
        $open copyC_cur;
        print_status("open copyC_cur");
        while (1)
        {
         $fetch copyC_cur into $c_num, $c_arg, $c_op, $c_val, $dum;
         printf("c_Cur, %i, cenum %i\n", sqlca.sqlcode, c_num);
        printf("c_attr, %s, c_op %s\n", c_arg, c_op);
        printf("c_val, %s, dum %s\n", c_val, dum);
         print_status("fetch copyC_cur");
         if (sqlca.sqlcode == SQLNOTFOUND)
                break;
         if (c_num > total)
                total = c_num;
         ci_num = c_num;
         $insert into query_conditions
                values ($ci_num, "", $c_arg, $c_op, $c_val, "A", 0);
         print_status("insert into db");
        }
        $close copyC_cur;
        print_status("close copyC_cur");
```

```
        return(total);
       }

Copy_Temp_To_Interm(t_num)
        $long   t_num;
        {
         $char t_attr[30];
         $char t_arg[30];
         $char t_op[5];
         $char t_val[30];

        printf("COPY TEMP %i", t_num);
        $open copyTemp_cur;
        print_status("open copyTemp_cur");
        while (1)
        {
         $fetch copyTemp_cur into $t_attr, $t_arg, $t_op, $t_val;
         printf("COPY TEMP TO I %i", sqlca.sqlcode);
         print_status("fetch copyTemp_cur");
         if (sqlca.sqlcode == SQLNOTFOUND)
                break;
         $insert into Intermediate
                values ($t_num, $t_attr, $t_arg, $t_op, $t_val);
         print_status("insert into intermediate");
        }
        $close copyTemp_cur;
        print_status("close copyTemp_cur");
       }




/* Here go Can BeSubSumed, NotEqStuff, and DeclareSubsumptionCursors    */
/* Omitted from printout                                                */
```

References

[Informix 86] *Informix ESQL User's Manual*
Informix, 1986.

[SM 91] M. Siegel and S. Madnick.
Metadata requirements for resolving semantic heterogeneities.
Sigmod Record Special Isuue on Semantic Heterogeneity , 1991.

[SSR a 92] E. Sciore, M. Siegel and A. Rosenthal.
Context interchange using meta-attributes.
ISSM First International Conference on Information and Knowledge Management , November, 1992.

[SSR b 92] E. Sciore, M. Siegel and A. Rosenthal.
Using semantic values for semantic interoperability.
In In Submittion to Seond Workshop on Information Knowledge Management , November, 1992.

[Tare 89] R. S. Tare.
Data Processing in the UNIX Environment.
MacGraw-Hill, 1989.