8-9-2005

Global Semantic Integrity Constraint Checking for a System of Databases

Praveen Madiraju

GLOBAL SEMANTIC INTEGRITY CONSTRAINT CHECKING FOR A SYSTEM OF DATABASES

by

PRAVEEN MADIRAJU

Under the Direction of Rajshekhar Sunderraman

ABSTRACT

In today's emerging information systems, it is natural to have data distributed across multiple sites. We define a System of Databases (*SyDb*) as a collection of autonomous and heterogeneous databases. *R-SyDb* (System of Relational Databases) is a restricted form of SyDb, referring to a collection of relational databases, which are independent. Similarly, X-SyDb (System of XML Databases) refers to a collection of XML databases.

Global integrity constraints ensure integrity and consistency of data spanning multiple databases. In this dissertation, we present (i) C*onstraint Checker*, a general framework of a mobile agent based approach for checking global constraints on R-SyDb, and (ii) *XConstraint Checker*, a general framework for checking global XML constraints on X-SyDb. Furthermore, we formalize multiple efficient algorithms for varying semantic integrity constraints involving both arithmetic and aggregate predicates. The algorithms take as input an update statement, list of all global semantic integrity constraints with arithmetic predicates or aggregate predicates and outputs sub-constraints to be executed on remote sites. The algorithms are efficient since (i) constraint check is carried out at compile time, i.e. before executing update statement; hence we

save time and resources by avoiding rollbacks, and (ii) the implementation exploits parallelism. We have also implemented a prototype of systems and algorithms for both R-SyDb and X-SyDb. We also present performance evaluations of the system.

INDEX WORDS: Multidatabases, Global Semantic Integrity Constraints, XML Databases, XML Constraints

GLOBAL SEMANTIC INTEGRITY CONSTRAINT CHECKING FOR A SYSTEM OF DATABASES

by

Praveen Madiraju

Presented in Partial Fulfillment of Requirements for the Degree of

Doctor of Philosophy

Georgia State University

2005

GLOBAL SEMANTIC INTEGRITY CONSTRAINT CHECKING FOR A SYSTEM OF DATABASES

by

PRAVEEN MAIDRAJU

Major Professor:	Rajshekhar Sunderraman
Committee:	Anu G. Bourgeois
	Jeff Qin
	Yanqing Zhang

Electronic Version Approved:

Office of Graduate Studies
College of Art and Sciences
Georgia State University
August 2005

ACKNOWLEDGEMENTS

During the process of completing my Ph.D., many people have helped me in realizing my goal.

First, and foremost, I would like to thank my advisor, Dr. Raj Sunderraman for introducing me to the area of databases and the research problems in this area. As a researcher, he is sharp, quick and smart. As a person, he is very humble, easily approachable, plain hearted, and straightforward. One can learn lot of fascinating things, both technical and non-technical issues from him. I hope I have learnt at least some of them. I would also like to thank my other Ph.D committee members - Dr. Anu Bourgeois, Dr. Yanqing Zhang, and Dr. Jeff Qin for advising me and reviewing the document. I sincerely appreciate the advice and help I have received from Dr. Yi Pan on many issues during the course of my study.

This course of study would not have been pleasant and enjoyable without the love of my life – my Bujji. I also would like to thank my friends – Abbi, Ajay, Ramu, Amarnath, Arthi, Arun, Ayyappa, Ramna, Bhanu, Shilpa, Harshi, Laxmikanth, Arvind, Bindu, Varsha, Mugdha, Janaka, Praveena, Naresh, Swaroop, Somu, Amar, Venu, Srikanth, Smitha, Vijay, Vamshi, Anna, Vinod, Rhony and others.

Finally, definitely not the least, without the loving support and patience of my family – mummy, daddy, and Sugandhi, I would not have realized my goals.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ACRONYMS

System of Databases	SyDb
System of Relational Databases	R-SyDb
System of XML Databases	X-SyDb
Constraint Planning Algorithm	CPA
System of Databases Query Language	SyDbQL
Constraint Data Source Table	CDST
Database Object List	DOL
XConstraint Source Table	XCST
XML Node Value List	XNVL

1. INTRODUCTION

A multidatabase system consists of autonomous component heterogeneous database systems. Multidatabase research is an important field in the area of database systems. Some of the important sub problems in this area are: (i) global schema mapping and integration ([1], [2], [13], [34]); (ii) global query decomposition and optimization ([43]); and (iii) global constraint checking ([26], [28], [39], [40]).

We consider a restricted form of multidatabase system; we call it System of Databases (*SyDb*). SyDb consists of autonomous multiple database systems which are homogeneous. Data is distributed among multiple sites. The reasons for data distribution may be the inherent nature of the data, performance reasons, or individual sites being incapable of hosting large amounts of data (mobile environment). Data distribution is quite natural in a healthcare database system. Say for instance that patient information is stored at site S_1. Insurance company stores patient's claim information at site S_2 and a different agency stores doctor's information at site S_3. It is difficult to enforce a centralized scheme as we have different agencies operating at their own rules. In some cases, where large volumes of data with millions of records are stored, it is just not possible to have centralized data due to performance factors. Data at these individual sites are not necessarily independent, but may participate in a relationship with data from other sites. Integrity constraints are valuable tools for enforcing consistency of data in a database ([51]). Global integrity constraints ensure integrity and consistency of data spanning multiple databases.

1.1 Motivation

In the general setting of a multidatabase, when multiple database systems interoperate, there is a very large likelihood of global constraints to be violated. Global constraints specify and enforce that a particular database state is consistent and ensures integrity of data across multiple databases. Much of the previous research and commercial database systems consider integrity constraint checking at run time and are inefficient as they suffer from rollbacks. An update statement issued on a single site might cause a global constraint to be violated, essentially endangering the consistency of the database. Frequent changes in data causes frequent global constraint violations causing the system to rollback frequently. Such systems are inherently inefficient as they consume lot of resources for rolling back the database state. Hence, we need a complete, standalone system that enables *efficient* and *speedy* checking of global constraint violations. Efficiency of the system needs to be achieved at both analytical and implementation level that avoids rollback situations.

1.2 Problem Statement

In a SyDb, given an update[1] statement and list of global constraints, the proposed system should check if the update statement violates any of the global constraints. The proposed system needs to check for global constraint violations for both relational and XML databases before updating the database i.e. at compile time. Compile time checking

---

[1] An update statement could be either insert or delete or update in a database

of constraints avoids time and resources spent on rolling back the database state in case of constraint violation.

1.3 Contributions

Here, we give the overall contributions of this dissertation. We also cite the papers that resulted from this dissertation work.

Constraint Checker ([39]): None of the literature so far has considered using mobile agents for global constraint checking in multidatabases. We introduce a general framework of a mobile agent based constraint checker in our research. The motivations for using mobile agents in our context are given in Section 3.6. The constraint checker has five major modules: update parser, metadatabase extractor, constraint planner, constraint optimizer, and constraint executor.

Constraint Planning Algorithm ([40]**):** The constraint planning algorithm is the algorithmic back bone for the constraint checker system. Given an update statement and list of global constraints, constraint planning algorithm decomposes global constraints into a set of sub-constraints based on the locality of the sites. More formally, the approach of the constraint planning algorithm (*CPA*) is to scan through the global constraint C_i, update statement U and then generate the conjunction of sub-constraints, C_{ij}'s[2], based on the locality of the sites. The value of each conjunct (C_{ij}) is either 0 or 1 and if the overall value of the conjunction is 1, constraint is violated, otherwise not.

---

[2] C_{ij} indicates the sub constraint corresponding to a global constraint C_i on site S_j

Agent Based Execution Engine ([38]**):** An agent based execution engine aids in creation, management and destruction of agents on remote sites.

Implementation of Constraint Checker: We have implemented the constraint planning algorithm using JAVA. We used a java based mobile agent framework – aglets ([32]) for implementing the overall constraint checker system

Constraint Checking for XML databases ([41]**,** [42]**):** Very few research results exist in the area of integrity constraint checking on a single XML database. To our knowledge, we have not come across any research on semantic integrity constraint checking for multiple XML databases. We have introduced notations for representing XML constraints. We proposed a general framework and algorithmic description of constraint checking for multiple XML databases. We have also implemented a prototype of the system.

1.4 Research Path

Here, we discuss our overall vision for carrying out the entire research. Initially, we started with the literature survey on integrity constraint checking for multidatabases, its motivation, why it is needed, what has been done, and what needs to be done. We quickly found out that research on multidatabase systems started in the early 1990's ([1], [9], [48]). In the mid 1990's and in later part of the 1990's, an abundant body of literature started concentrating on transaction aspects of multidatabases ([7]), global querying ([43]), schema mapping and integration ([2], [5], [19]). However, surprisingly not much research concentrated on global constraint checking for multidatabases ([28]) under

updates. The motivation for our research is well known as global integrity constraints are valuable tools that preserve the consistency of a multidatabase system. Only when all the integrity constraints are satisfied (not violated), the multidatabase is consistent and reliable. Hence efficient and speedy checking for constraint violations is an important area of research. In our research, we consider a restricted form of multidatabases; we call it System of Databases (*SyDb*). We also introduce *R-SyDb* (System of Relational Databases) and *X-SyDb* (System of XML Databases). R-SyDb considers collection of autonomous component database systems, which are all relational database systems and X-SyDb contains a collection of autonomous XML database systems.

During the process of analysing what is needed in this area, we identified a number of interesting research problems. First, we realized that global integrity constraint checking at compile time (before updating) for relational databases is a very interesting idea. At the same time, we also recognized mobile agents from AI field have been recently used for distributed information processing, distributed computing and intersection of agent approach with global constraint checking is definitely new and promising field. Second, we outlined important sub problems related to global constraint checking under updates to R-SyDb. We made a classification of integrity constraints and narrowed our problem to semantic integrity constraint violations. Third, we further drilled down semantic integrity constraints and classified them in to two major groups: semantic integrity constraints with simple arithmetic predicates and semantic integrity constraints with aggregate predicates. Therefore, we summarized that global semantic integrity constraint checking at compile time with arithmetic predicates and aggregates

predicates, with algorithmic description would make the semantic integrity constraint checking complete. Fourth, we proposed an architecture identifying other major modules for a constraint checker such as constraint optimiser. However, at this point we observed that our approach of semantic integrity constraint checking for R-SyDb can be extended to X-SyDb. The effort has been to expand our research ideas to XML databases (new area) instead of being confined to only one area. Fifth, we started undertaking a literature survey on global semantic integrity constraints for XML databases in an effort to broaden our impact and coverage of research areas. We understood that none of the research results exist in the area of semantic integrity constraint checking for XML databases. Sixth, we have come across integrity constraints in logic databases, relational databases and XML databases. We identified that a mapping of constraints from one to another is another interesting research issue.

1.5 Organization

In Chapter 2, we discuss our System of Databases architecture. Throughout the dissertation, we use a sample healthcare multidatabase system as a running example to illustrate our ideas. We present the sample healthcare multidatabase system, the architecture and algorithms for constraint checking for a System of Relational Databases in Chapter 3. In Chapter 4, we summarize our research on constraint checking in a System of XML Databases. We discuss related work in Chapter 5. Finally, we present the conclusions and future work in Chapter 6.

2. SYSTEM OF DATABASES

A *System of Databases (SyDb)* refers to a collaborating set of heterogeneous data resources. Global querying refers to the problem of information retrieval from heterogeneous and distributed sources. Global updates allow the user to store data to a set of heterogeneous and distributed sources. The significant problems related to Global queries and updates are:

(i) Global schema mapping and integration ([1], [2], [13], [34]);

(ii) Global query decomposition and optimization ([43]); and

(iii) Global constraint checking ([26], [28], [39], [40]).

However, the application developer should not be bogged down with the above problems. Just as JDBC (Java Database Connectivity) hides connection and retrieval details from the user, SyDb provides a framework/API that would hide all the complexities and empower the user with a ready to use package for Global queries and updates. Earlier, we proposed a principled extension to the current SQL standard, SyDbQL ([49]). The SyDbQL allows a user to specify global queries and global updates. We have also set forth the design and implementation of SyDbQL Java API that allows global queries and updates through a Java JDBC program.

Assumptions of SyDb

The assumptions of the system under consideration are:

(i) A system of collaborating heterogeneous databases,

(ii) Schema integration and schema mapping of the individual component databases is resolved using the techniques such as [1], [2], [13], and [34], and

(iii) An application that needs to query/update the component databases.

2.1 SyDb Architecture

The System of Databases (SyDb) architecture and the detailed steps involved in executing a global query/update is presented in [49]. Figure 1 gives the overall architecture of SyDb.

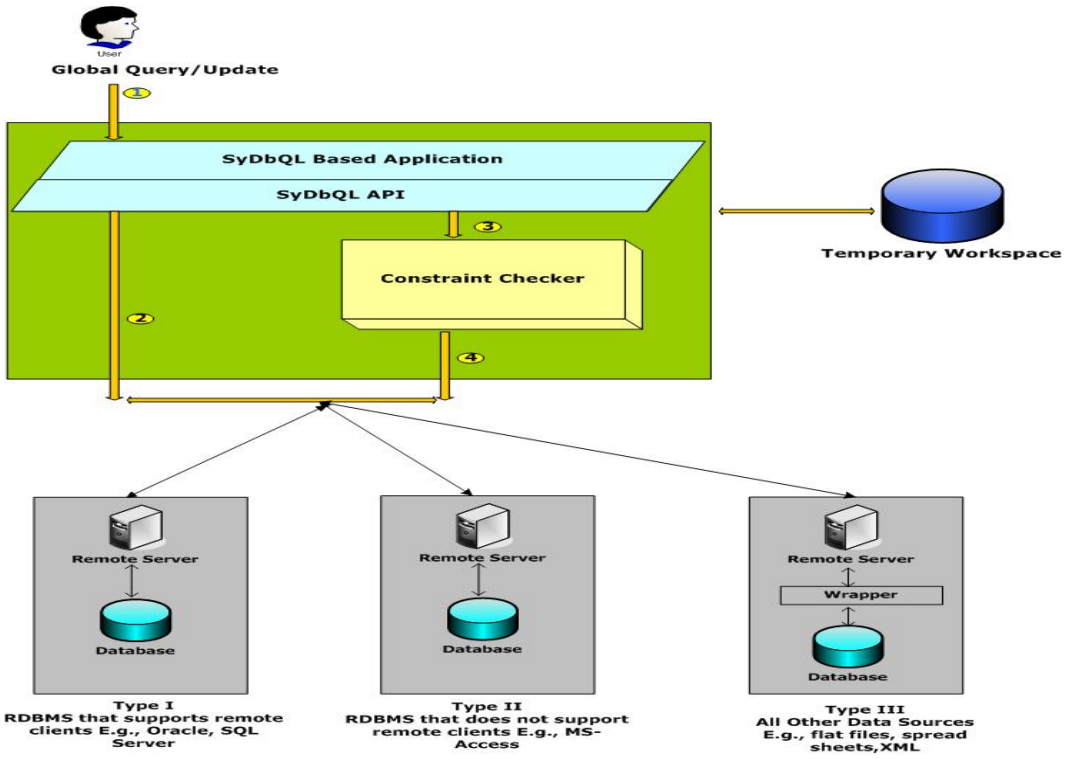


Figure 1 : System of Databases Architecture

The SyDb architecture has three layers. At the lowest layer, we have multiple heterogeneous databases. The middle layer consists of constraint checker (details in Chapter 3) and any SyDbQL based application. The top layer consists of the client making a global query/update request.

Step 1: Using the SyDbQL API, the user issues a global query/update

Step 2: If the user issued a global query, the query is parsed, decomposed into a set of sub queries and sent to the component databases. The results obtained from the component databases are gathered, modified and finally the output is displayed.

Step 3: If the user issued a global update, the update is input to the Constraint checker module. A general framework and an algorithm for the constraint checker module are given in Chapter 3. Given an update statement U and the list of all global constraints C, the constraint checker module checks if any global constraints are violated without actually updating the database (compile time). In the current set up, global constraints can be stored in the temporary workspace.

Step 4: Constraint Checker generates sub constraint checks on to the component databases, gathers results and finally makes a decision if a constraint is violated. In case of non constraint violation, the global update statement is executed. The temporary workspace shown in Figure 1 is a local temporary workspace that a SyDbQL-based application can interact with.

We broadly categorize data resources into three groups: (1) Type I: Relational databases that support remote clients, where data can be retrieved in Java by using JDBC

drivers. (2) Type II: Relational Databases that do not support remote clients, where data can be retrieved by using either JDBC-ODBC bridge or by using JDBC drivers. (3) Type III: Remaining data resources, such as object-oriented databases, flat files, and XML data, where data can be retrieved using a wrapper that would convert the specific data format to relational tables and vice versa. The wrapper is data-source type dependent.

2.2 Global Queries and SyDbQL Syntax

Consider the personal database which several individuals keep in their personal computers/personal digital assistants (PDAs). Typical data kept in these databases are appointments, addresses of acquaintances, etc. Let us assume there are three individuals, John, Tony and Aaron who maintain such data in their PCs/PDAs. John's database may have the following schema:

```
schedule (date, startTime, endTime, event)
addressBook (name, email, address, wphone, hphone, cphone, fax)
```

Tony's database may have the following slightly different schema:

```
schedule (date, startTime, endTime, event)
addressBook (name, email)
```

Aaron's database may have the following schema, similar to John's except email addresses are not kept:

```
schedule (date, startTime, endTime, event)
addressBook (name, address, wphone, hphone, cphone, fax)
```

These three databases are assumed to be located on different nodes of a network and are assumed to be autonomous. Let us assume that these three individuals work together and hence a need for their databases to collaborate exists.

2.2.1 Global Queries

Global queries allow users in one database to extract information from their local database as well as remote databases. Such queries will have references to remote database objects. As an example of a global query, consider Aaron's problem of locating email addresses of all individuals in his addressBook. To accomplish this task, Aaron may execute the following global query in his database:

```
SELECT  t.email
FROM    tony.addressBook t, addressBook a
WHERE   t.name = a.name
```

Notice that in the above global query which executes within Aaron's database, there are references to tables in Tony. We are assuming the name of Tony's database is tony. Aaron is joining his addressBook table with that of Tony's to obtain email addresses.

2.2.2 SyDbQL Syntax

SyDbQL extends SQL by allowing tables to be referenced by the databases they are located in. A database naming mechanism is introduced, and tables in SyDbQL queries are identified by the databases they belong to. This is accomplished by preceding each table name with the database name of the table in the form of *database.table*. Following is a list of standard SyDbQL statements and their syntax:

Creating tables

```
CREATE TABLE [dbname.]<tablename>[, [dbname.]<tablename>]* (
col-def, …, col-def, table-constr, …, tab-constr);
```

where col-def is:

```
<column-name><data-type>[DEFAULT <expr>][<column-constraints>]
```

and tab-constr is:

```
[CONSTRAINT <constraint_name>] [NOT] NULL | CHECK (<condition>) |
UNIQUE | PRIMARY KEY | REFERENCES
[dbname.]<table_name>[(<column_name>)] [ ON DELETE CASCADE]
```

This statement allows one or more tables with the same schema to be created in remote databases. Constraints on table(s) can also be specified after CONSTRAINT keyword such as primary key, foreign key (REFERENCES), and cascading delete constraints.

Deleting tables

```
DROP TABLE [dbname.]<tablename> [, [dbname.]<tablename>]*
[CASCADE CONSTRAINTS];
```

This statement allows one or more tables to be deleted from multiple database schemas. CASCADE CONSTRAINTS allows the user to delete referenced tables as well.

Inserting rows into table

```
INSERT INTO [dbname.]<tablename>[, [dbname.]<tablename>]*
[(column {,column})] VALUES (expression, {,expression});
```

Same row is inserted to one or more distributed database tables using SyDbQL INSERT statement.

Selecting rows from table(s)

```
<sub-select>
```

```
{ UNION [ALL] <sub-select> }[ ORDER BY result_column [ASC | DESC
]
{ , result_column [ASC | DESC ]}]
where <sub-select> is:
SELECT  [DISTINCT] <expression> {,<expression>}
FROM [dbname.]<tablename>[<alias>]
{,dbname.]<tablename>[<alias>]}
[WHERE <search_condition>]
[GROUP BY <column> {,<column>}]
[HAVING <condition>]
```

SyDbQL SELECT statement is similar to standard SQL SELECT statement, but it allows querying tables from distributed databases. UNION statement provides a mechanism to get the union of the results of two SELECT statements. ORDER clause allows sorting the results.

3. CONSTRAINT CHECKING IN A SYSTEM OF RELATIONAL DATABASES

We present a general framework of an agent based *Constraint Checker* module. We then, give an efficient algorithm, *CPA* (Constraint Planning Algorithm) for decomposing a global constraint into conjunction of sub-constraints based on the locality of sites. CPA forms the algorithmic backbone for the constraint checker module. We also discuss the implementation and performance results of constraint checker.

3.1 Preliminaries

Here, we give an example healthcare multidatabase system that will be referred throughout this chapter. We then present constraint representation notations and their classification. We also give a brief overview of agents.

3.1.1 Example Database

Consider a typical health care multidatabase management system as an example. It is a very natural scenario to have patient's information distributed across multiple sites. In a multidatabase system, we can have the same predicate names at two different sites. Hence, we need a notation that distinguishes one predicate from the other. We use the notation of: (S_i: table t), where t is the name of the table stored on site S_i.

At site S1: Patient information is stored. A *PATIENT* relation with attributes *name* and type of *healthplan* is recorded. S_1*:PATIENT (name, healthplan)*. A *PATIENTDETAILS* relation with attributes *name*, *address* where the patient lives, *employer* name and *salary* of the patient is recorded. S_1*:PATIENTDETAILS (name,address,employer,salary).*

At site S2: Health insurance companies store patient's claim information. A *CLAIM* relation with attributes *name* (patient name), *amount* of claim, date of claim and *type* of claim is recorded. *S_2:CLAIM (name, amount, claimdate, type).* A *CLAIMREVIEW* relation records patient's *name*, *date* of claim and *reviewer* name. *S_2:CLAIMREVIEW (name, claimdate, reviewer).*

At site S3: Doctor's office maintains patient's name, doctor treating the patient and disease for which the patient is being diagnosed. A DOCTOR relation with attributes *name* (patient name), *doctorname* and *disease* is recorded. *S_3:DOCTOR (name, doctorname, disease).*

3.1.2 Constraints

In order to represent integrity constraints in the context of a database as query evaluation in the database, we consider integrity constraints in the form of range-restricted denials.

$\leftarrow L_1 \wedge L_2 \wedge \ldots \wedge L_n$

where each L_i is a literal or an aggregate literal involving a base predicate and global variables are assumed to be universally quantified over the whole formula ([18]).

Say integrity constraint C_0 states, the sum of all claim amounts of a patient with healthplan 'C' may not be more than 200000.

```
← S1:PATIENT (name,'C'),Sum(amount,S2:CLAIM(name,amount,-,-),s),
  s > 200000.
```

This can be conveniently represented using the approach of [27]. A constraint is a query whose result is either 0 or 1(Gupta and Widom ([28]) call it "panic"). If the query produces 0 on the multidatabase D, then D is said to satisfy the constraint, or the constraint is violated on D.

```
PanicC0 :- S1:PATIENT(name,'C'),Sum(amount,S2:CLAIM(name,amount,-,-),s),
           s > 200000.
```

For convenience, we will refer to *PanicC*$_0$ as just C_0.

3.1.3 Integrity Constraint Classification

Integrity constraints can be classified into six major categories. They are:

- **Domain constraints:** They are the most primitive form of integrity constraints and they make sure that the comparisons and the values inserted into the database are logical. For example, if we try to test the name of a person to digit 10, domain constraints are violated as name of the person is *varchar* and 10 is numeric.
- **Key constraints:** These are the unique/primary key constraints.

  Every patient name is unique.

  ```
  C1:- S1:PATIENT (name, X), S1:PATIENT (name, Y), X<>Y.
  ```

- **Referential integrity constraints:** They ensure that values that appear in one relation also appear in another relation.

  Every name referenced by CLAIM relation exists in PATIENT relation

  ```
  C2:- S2:CLAIM (name, -, -, -), not PATIENTNAMES (name).
  ```

```
PATIENTNAMES (name): - S1:PATIENT (name, -).
```

- **Semantic integrity constraints (general form of assertions):** They specify a general condition in a database that needs to be always true. Integrity constraints of this type deal with information in a single state of the world

  Any patient with healthplan 'B' may not file a claim type of 'emergency'

```
C3:- S1: PATIENT (name, healthplan),
     S2:CLAIM (name,_,_,'emergency'), healthplan = 'B'.
```

- **State transition constraints**: These constraints deal with two consecutive database states. Example of such a constraint would be: when a claim is updated, the new claimdate must be greater than the older claimdate.

```
C4:- CLAIM_new(_,_,cd1,_),CLAIM_old(_,_,cd2,_),cd1 < cd2.
```

- **State sequence (temporal constraints):** These constraints refer to more than two database states (not necessarily consecutive database states). "An employee salary must never decrease"([51]) is an example of such a constraint.

Our constraint checking procedure is limited to only the class of semantic integrity constraints.

3.1.4 Mobile Agents

Mobile agents can be considered as an incremental evolution of the earlier idea of "process migration". A mobile agent is an autonomous, active program that can move both data and functionality (code) to multiple places within a distributed system. The

state of the running program is saved and transported to the new host, allowing the program to continue execution from where it left off before migration ([29]). Mobile agents require two components for their successful execution. The first component is the agent itself. The second component being the place where in an agent can execute. This is often referred to as the software agent framework. It provides services and primitives that help in the use, implementation and execution of systems deploying mobile agents. This generic framework allows the developers to focus on the logic of the application being implemented, instead of focusing on the implementation details of the mobile agent system. Specifically, it should support the creation, activation, deactivation and management of agents, which include mechanisms to help in the migration, communication, persistence, failure recovery, management, creation and finalization of agents. Additional services as naming and object persistence can also be provided. This environment must also be safe, in order to protect the resources of the machine from malicious attacks and possible bugs in the implementation of the agent code. Some of the popular examples are: IBM's Aglets ([32]), Mitsubishi Electric ITA's Concordia ([33]) and Object Space's Voyager ([23]).

3.2 Constraint Checker Overview

In this section, we discuss details of the overview of the system, constraint checking architecture and constraint checking procedure. Figure 2 shows an outline of our approach. Using the database description of remote database objects, global metadatabase is constructed. Global constraints to be enforced are also stored in the global metadatabase. We provide a design of constraint checker module that accepts an

insert/update/delete request from a user and considers one constraint at a time from global metadatabase and decides if any constraint is violated.

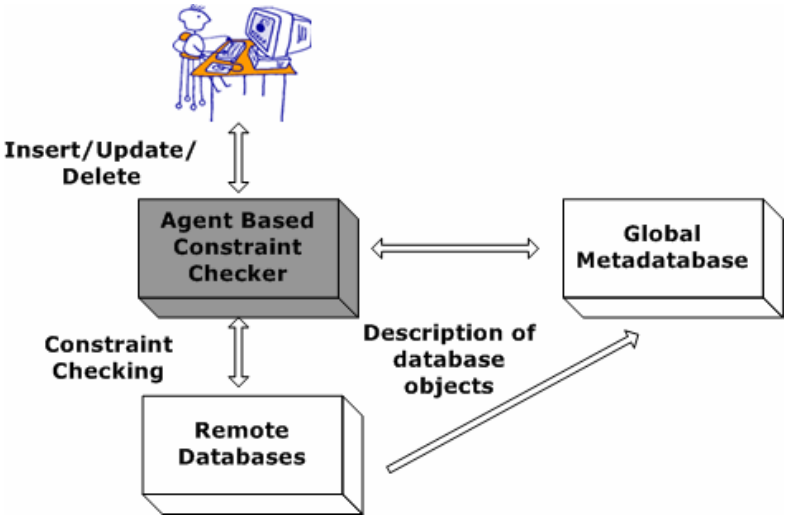


Figure 2: Overview of Constraint Checking System

The system architecture is shown in Figure 3. R-SyDb (System of Relational Databases) consists of (i) collection of data sources on multiple sites that are all autonomous relational databases and (ii) global metadatabase. The global metadatabase is a repository of site and domain information. Site information gives description of sites where data sources reside. Domain information gives metadata description of database objects of all data sources and global constraints, say $C_1 \ldots C_n$.

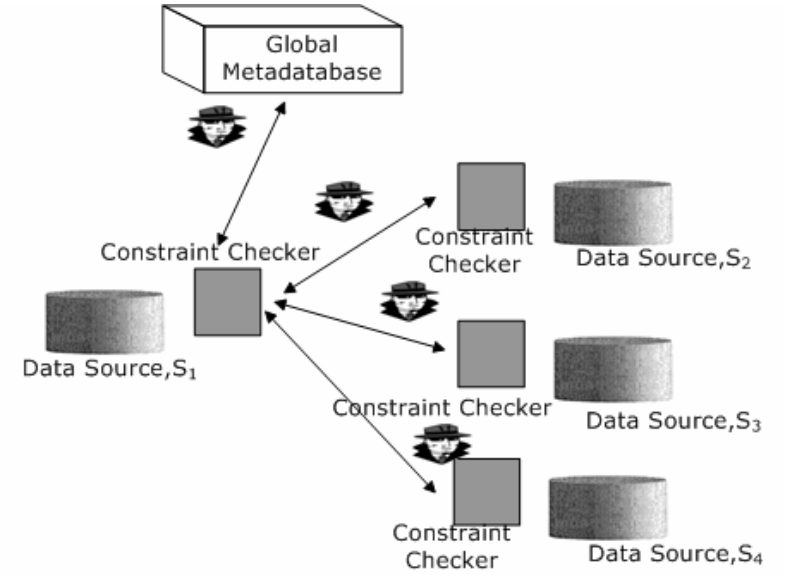


Figure 3: Constraint Checking Architecture for R-SyDb

A *constraint checker* module resides on each of the data sources. This module is responsible for interfacing with the global metadatabase. In Figure 3, say, an update statement U_1 is issued on site S_1. It modifies/updates some of the database objects. Constraint checker on S_1 sends out mobile agent on to the global metadatabase. The mobile agent at the global metadatabase is equipped with the knowledge of database objects being modified and also data processing code. The mobile agent computes the list of global constraints being affected by U_1, say $C_1 \ldots C_m$. The mobile agent returns this list to the constraint checker. Constraint checker takes as input one global constraint at a time, C_1. For each global constraint, sub-constraints corresponding to remote sites are generated. Mobile agents $rmagent_2$, $rmagent_3$, $rmagent_4$ are spawned to individual sites S_2, S_3, S_4. Constraint checker gathers results from these mobile agents and makes a

decision if a global constraint is violated. This process is repeated for all remaining constraints $C_2 \ldots C_m$.

3.3 Constraint Checker Internal Architecture

The internal architecture of the constraint checker and the overall procedure of constraint checking are explained using Figure 4. The constraint checker has five major modules: update parser, metadatabase extractor, constraint planner, constraint optimizer, and constraint executor.

Update parser: parses an update statement input by the user and identifies the database objects involved in the update statement.

Metadatabase extractor: extracts all the global constraints being affected by the update statement.

Constraint planner: devises an effective plan for generating sub-constraints based on the locality of the sites.

Constraint optimizer: optimizes sub-constraints for efficient constraint checking.

Constraint executor: generates and spawns mobile agents. The mobile agents execute the sub-constraints and with the summarized information gathered from all the mobile agents, a decision is made if a global constraint is violated.

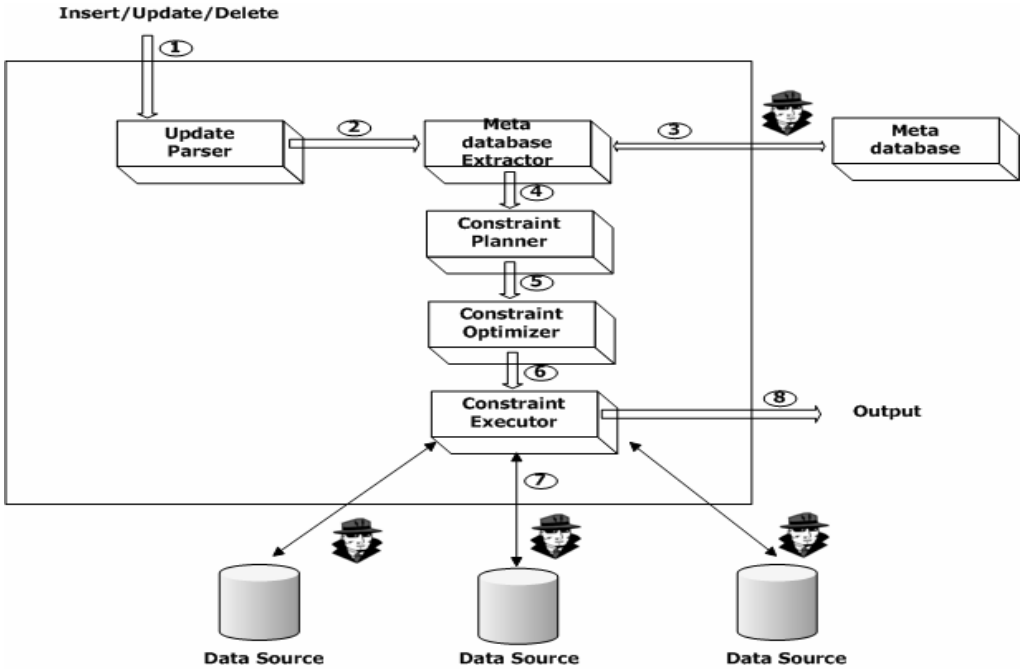


Figure 4: Constraint Checker Internal Architecture

STEP 1

The user issues an update statement onto his local data source. For example, the user issues an update statement U_1 on site S_2. Let

```
U1 = insert into S2:CLAIM values
     (5, to_date ('02/20/2005','MM/DD/YYYY'),25000,'Emergency');
```

STEP 2 (Update Parser)

The update parser parses the given update statement and identifies database objects being modified. The output from this step is the *database object list (DOL)*. For the running example,

```
DOL = {S2:CLAIM (CaseId=5,ClaimDate='02/20/2005',amount=25000,
      type='Emergency'}.
```

STEP 3 (Metadatabase Extractor)

The metadatabase extractor takes as input a database object list. It contacts the metadatabase and gets the list of constraints being affected by the update statement and also the list of sites involved for each such constraint. The metadatabase extractor constructs the Constraint Data Source Table (*CDST*) as shown in Figure 5.

CDST (C_i) = <C_i, list (S_j)> where

C_i is the global constraint identifier

list(S_j) is the list of data sources being affected by C_i

C_i	list(S_j)
C_5	(S_1,S_2,S_3)
C_6	(S_1,S_2)

Figure 5 : The Constraint Data Source Table

```
C5:-S1:PATIENT(name, 'B'), S2:CLAIM(name, amount, _, _),
    S3:DOCTOR(name, _, 'smallpox'), amount > 20000.
C6:-S1:PATIENT(name, healthplan),
    S2:CLAIM(name, _, _, 'emergency'), healthplan = 'B'.
```

Constraint c_5 states that a patient with *healthplan* 'B' diagnosed with 'smallpox' may not claim more than 20000 dollars. Constraint c_6 states that a patient with *healthplan* 'B' may not file a claim of type 'emergency'.

STEP 4

The metadatabase extractor sends CDST to the constraint planner module.

STEP 5 (Constraint Planner)

The *constraint planner* takes as input *DOL* (Database Object List) and *CDST* (Constraint Data Source Table). It outputs the list of sub-constraints *list* (C_{ij}) for each global constraint. *list*(C_{ij}) is the list of sub-constraints corresponding to each C_i and site S_j. The value of each C_{ij} is either 0 or 1. The constraint planning algorithm given in the next sub section decomposes a global constraint C_i into a set of sub-constraints C_{ij} to be executed locally on remote sites (decomposition is based on locality of sites).

For the running example, for C_5, the corresponding sub-constraints generated are: C_{51}, C_{52}, C_{53} and for C_6, the sub-constraints generated are: C_{61}, C_{62}. The algorithmic procedure for generating these sub-constraints can be found in Section 3.4. However, to preserve the flow of the dissertation, the values of these sub-constraints are given below:

```
C51 = select 1 from dual where exists (select * from patient where
      name = 'john' and healthplan = 'B')
C52 = return 1 if {'john' = 'john' and 25000 > 20000} else
      return 0
C53 = select 1 from dual where exists (select * from DOCTOR
      Where name='john' and disease = 'smallpox')
C61 = select 1 from dual where exists (select * from patient where
      name = 'john' and healthplan = 'B')
C62 = return 1 if {'john'='john' and 'emergency' = 'emergency'}
               else return 0
```

STEP 6 (Constraint Optimizer)

The *constraint optimizer* optimizes the constraint checking process. The constraint optimizer generates constraint optimized table (COT), as shown in Figure 6. Any optimizations that increase efficiency of the constraint checking process are carried out here. The parameters considered are: number of sites accessed, locality of sites, and history of constraint failures on a site. For the running example, c_6 involves accessing two sites s_1 and s_2, where as c_5 involves accessing sites s_1, s_2, and s_3. Constraint optimizer orders the execution of the constraints and also sub-constraints.

COT

C_i	list(C_{ij})	cot-list(C_i)
C_6	(C_{61}, C_{62})	(C_{62}, C_{61})
C_5	(C_{51}, C_{52}, C_{53})	(C_{52}, C_{51}, C_{53})

Figure 6 : The Constraint Optimized Table

Observe that if c_6 is violated, we will not check c_5 and since the constraint checking is much faster doing c_6 first and then c_5 (C_6 involves accessing lesser number of sites), we have gained efficiency. Hence, in Figure 6, the row for c_6 occurs before c_5, indicating order of execution of the constraints. Also, the *cot-list* is ordered for each sub constraint. In the running example, since u_1 is initiated on s_2, we have ordered the *cot-list* (C_6) in the order of c_{62} and c_{61}. The idea is to first check for local sub-constraints (local to S_2) and then any remote sub-constraints. The reason is, if one of c_{62} or c_{61} returns "false" or "no rows returned", then constraint c_6 is satisfied. In a similar way *cot-list* (C_5) is also

ordered. Further optimization is possible by keeping track of the history information of constraint violations on every site.

Constraint Optimizations

A classic optimization strategy that could be employed for checking global constraints is the Local Verification of Global Integrity Constraints ([28]). For each site s_j and global constraint c_i, whenever possible, a local test condition is checked instead of having to check for sub-constraints on remote data. Say integrity constraint c_7 requires that every "name" referenced by a tuple in CLAIM relation exists in the PATIENT relation.

```
C7:- S2:CLAIM(name,_,_,_), not PATIENTNAMES(name).

PATIENTNAMES(name) :- PATIENT(name,_);
```

Let us consider, we have an update U_2 on s_2

```
U2 = insert into S2:claim values
     ('john',10000,'06/10/2003','prescription');
```

Traditionally, we will have to check for the occurrence of name *'john'* in the PATIENT relation on s_1. However, if we can first do a local test condition such as

```
t1 = select * from S2:CLAIM where name = 'john';
```

If the above query has a non-empty answer, then we can conclude that U_2 does not violate c_3. This is the basic idea suggested in [28]. We are saving time spent on accessing remote data and also any issues related to data transfer through the network are nullified. We are proposing to expand this basic idea to the next level.

Consider a particular database state *D* where the CLAIM relation has a tuple with name *'john'*. This state *D* satisfies c_7. At this point if we delete tuple with name *'john'* from CLAIM, at all subsequent database states, whenever an insert on CLAIM relation with name *'john'* is performed, the local test condition t_1 fails. Our belief is referential integrity constraints need to be checked often. Since, for a scenario described like above, if the local verification approach fails then we will have to check for data at remote site. We are also spending extra time in checking for local test t_1 and then doing the remote check. We are proposing that, when a delete statement is issued on s_2, we do not actually delete the tuple with name 'john', we instead *"mark it for deletion"*. For normal queries and other database related tasks, CLAIM relation with name 'john' does not exist, however, the constraint checker on s_2, *knows* john was *marked for deletion* and existed before. With this approach, in a scenario like above, we do not have to do a remote constraint check. However for this approach to work, the parent relation PATIENT needs to be monitored whenever *'john'* gets deleted. Constraint checker or s_1, can monitor for such a deletion and in the event of deletion, it can inform the constraint checker on s_2 that *'john'* no longer exists in the PATIENT relation. Constraint checker on s_2 can then completely delete it from its database. The only extra burden is the monitoring step of constraint checker on s_1. We believe that this is reasonable as most of the times, parent keys are not deleted from the database. Hence, our approach adds efficiency to the local verification approach by extending it by one more step.

STEP 7 (Constraint Executor)

The constraint executor reads the *COT* and spawns mobile agent for each C_{ij}. The results are gathered from the mobile agents and the constraint executor makes a decision if a constraint has been violated. For the running example, $c_6 = c_{61} \wedge c_{62}$. We observe from step 5, $c_{62} = 1$ (true) and $c_{61} = 1$ (true). Hence, c_6 = true ^ true, implies c_6 = true. Therefore, c_6 is violated. In this case, we do not have to check for c_5, because, if one of the constraints is violated, the update statement is rejected.

STEP 8

The results are sent to the user.

3.4 Constraint Planning Algorithm

The basic idea of constraint planning is to decompose a global constraint into a conjunction of sub-constraints, where each conjunct represents constraint check as seen from each individual database ([26]). Given an update statement, a brute force approach would be to go ahead and update the database state from D to D' and then check for constraint violation. However, we want to be able to check for constraint violation with out updating the database. Hence, the update statement is carried out only if it is a non constraint violator.

The approach of the constraint planning algorithm (*CPA*) is to scan through the global constraint C_i, update statement U and then generate the conjunction of sub-

constraints, C_{ij}'s[3]. The value of each conjunct (C_{ij}) is either 0 or 1 and if the overall value of the conjunction is 1, the constraint is violated, otherwise not. An update U can be an update involving an insert or a delete or a modify statement. Hence, we have three different cases for the algorithm. They are given in the following sections: 3.4.1, 3.4.2, and 3.4.3.

3.4.1 CPA-insert

Algorithm `CPA-insert` (constraint planning algorithm for an insert statement) shown in Figure 7 gives constraint decompositions (C_{ij}'s), corresponding to global constraint C_i and an update statement involving an insert statement. Algorithm `CPA-insert` takes as input the update statement U and the list of all global constraints C and outputs the list of sub-constraints (C_{ij}) for each C_i being affected by U.

Algorithm **`CPA-insert`**

1: **INPUT**: (a) U: insert S_m:R($t_1,\ldots,t_n$)
(b) C: list of all global constraints /* Note: insert is occurring on site S_m */

2: **OUTPUT**: list of sub-constraints < C_{i1} ,…,C_{ik_i} > for each C_i affected by U

3: DOL (U) = < R (a_1= t_1,…,a_n= t_n) >

4: CDST(C,DOL(U)) = < <C_1, (S_{11},…,S_{1n_1})>,…,<C_q, (S_{q1},…,S_{qn_q})> >

5: *let* θ = {$x_1 \leftarrow t_1,\ldots,x_n \leftarrow t_n$}be obtained from DOL(U) where $x_1 \ldots x_n$ are variables corresponding to the columns of table R

6: **for** each *i* in {*1*... *q*} **do**

7: **for** each *j* in {*1*...n_i} **do**

8: *let* S_j:$p_1(X_1)$,$p_2(X_2)$,…,$p_r(X_r)$ be the sub goals of C_i associated with S_j and A be all arithmetic sub goals associated with S_j

9: **if** (j <> m) **then** /* site where update is not occurring */

[3] Recall that C_{ij} indicates the sub constraint corresponding to a global constraint C_i on site S_j

10: Cij = select 1 from dual where exists
(select * from p1 …pr where <cond1>)
11: <cond1> is obtained from $X_1 \ldots X_r$ using standard method of joining tables. It also
includes any arithmetic sub goal conditions
12: **else if** (j=m) **then** /* site where update is occurring */
13: **if** (there exists variables in A that do not appear among $X_1 \ldots X_r$) **then**
14: **for** each variable ν in A that do not appear among $X_1 \ldots X_r$ **do**
15: *let* k be the site where ν appears in a sub goal, S:t(X) in C_i
16: IP_{ikd} = (select *Col(v)* from S:t where <cond2>)
17: *Col (v)* is the column name corresponding to ν
18: <cond2> is obtained from $X_1 \ldots X_r$ and X. d is nth *intermediate predicate*
19: **end for**
20: **end if**
21: C_{ij} = return 1 if (<*cond3*> and *A′*) else return 0.
22: <*cond3*> is obtained from θ and $X_1 \ldots X_r$ and *A′* is A with IP's replacing corresponding
variables
23: **end if**
24: **end for**
25: **end for**
26: apply the substitution θ(U) to all C_{ij}

Figure 7 : Algorithm CPA-insert

Database Object List (DOL) identifies the database objects being modified by the update statement, U. *DOL* (*line 3*) identifies, the table R with attributes (column names) $a_1 \ldots a_n$ inserted with values $t_1 \ldots t_n$. *CDST* (*line 4*) gives the list of sites involved, for each constraint being affected by the update statement. The outer for loop variable *i* (*line 6*) loops through all the constraints $C_1 \ldots C_q$ affected by the update U. The inner for loop variable *j* (*line 7*) loops through each site ($<S_{11},\ldots,S_{1n_1}>,\ldots,<S_{q1},\ldots,S_{qn_q}>$) for each constraint *i*. Inside the for loop (*lines 6-25*), all the sub-constraints C_{ij}'s are generated. $S_j{:}p_1(X_1), p_2(X_2), \ldots, p_r(X_r)$ (*line 8*) denotes, for a particular site S_j, $X_1 \ldots X_r$ is the vector of

variables corresponding to the predicates (table names), $p_1 \ldots p_r$. A critical feature of the algorithm is the generation of *intermediate predicates* (IP). *IP*'s are generated only at the site where update is occurring. In concept, *IP*'s represent information that needs to be shared from a different site. Implementation wise, *IP* is a SQL query returning value of the variable, ν (*line 14*) from a different site. IP_{ikd} (*line 16*) means the *d*th intermediate predicate corresponding to constraint C_i and site S_K. The table dual (*line 10*) is like a "dummy" table provided by the oracle. It is a convenience table provided by Oracle that has exactly one column and only one row.

Theorem 3.1: *The conjunction of sub-constraints C_{ij}'s, generated from Algorithm CPA-insert conclusively determines, if an update statement involving an insert statement violates a global constraint C_i.*

Proof:

Consider an update statement on site S_m, global constraint C_i and the list of sub-constraints, C_{ij}'s generated from algorithm CPA-insert. The generation of each C_{ij} needs to achieve the same affect as sub goal corresponding to S_j. Let S_j:$P_1(X_1),P_2(X_2),\ldots,P_r(X_r)$ be the sub goals of C_i associated with S_j and A be all arithmetic sub goals associated with S_j. At this point each C_{ij} falls in one of the two cases. We will show that each C_{ij} in both the cases achieves the same affect as the sub goal corresponding to site S_j.

Case I (j<>m): This is the case where sub goal is associated with a site other than where update is occurring (*lines 9-11*). The generation of C_{ij} in this case is rather straight forward as it generates a sub constraint check from all the predicates involved on site *j*

using appropriate join conditions and it also includes any arithmetic sub goal conditions. Hence C_{ij} naturally achieves the exact same result as the sub goal corresponding to site S_j.

Case II (j=m): This is the case where sub goal is associated with a site where update is occurring (*lines 12-23*). The generation of C_{ij}'s in this case consists of two parts. Part 1 consists of information from the same site – trivial case (just as Case I). Part 2 relates to information acquired from a remote site. For each such variable a unique *intermediate predicate* is generated. IP's are SQL queries returning values of such variable by computing appropriate joins and arithmetic conditions involved with such variables. Hence, IP's guarantees correct exchange of information from a different site. The reason we are generating unique IP's is we can either store all the IP's at a global directory such as the metadatabase or we can generate IP's at run time.

Hence, from both the cases, we observe that the conjunction of C_{ij}'s entails the original global constraint, C_i. Therefore, if C_i determines whether an update involving an insert statement violates a global constraint C_i, then the conjunction of its sub-constraints C_{ij}'s also determines if the constraint C_i is violated. In other words, if the conjunction of C_{ij}'s evaluates to 0 (false), constraint Ci is not violated, otherwise Ci is violated. ■

We show the working of the Algorithm `CPA-insert` on some sample examples given below:

Example 3.1

This example considers constraint defined on the healthcare multidatabase system from sub section 3.1.1. It showcases how sub-constraints are generated in a simple case, when intermediate predicates are not involved.

Input:

U = insert into S_2:CLAIM values ('John', 25000, '06/10/2003', 'emergency')

C = list of all global constraints

Output:

List of sub-constraints $<C_{i_1} \ldots C_{ik_i}>$ for each C_i affected by U

DOL(U) = S_2:CLAIM{name='john',amount=25000,
claimdate='06/10/2003',type='emergency'}

CDST(C,DOL(U)) = $<C_5,(S_1,S_2,S_3)>$

where

C_5:-S_1:PATIENT(name, 'B'),S_2:CLAIM(name, amount, _, _),
S_3:DOCTOR(name, _, 'smallpox'), amount > 20000.

/* C_5 states that "A patient with health plan 'B' diagnosed with 'smallpox' may not claim more than 20,000 dollars". */

θ = {S_2:CLAIM(name1='john', amount1=25000, claimdate1='06/10/2003' ,type1='emergency') }

/* C_{51} is generated from algorithm CPA-insert (lines 9-11)*/

C_{51} = select 1 from dual where exists (select * from patient where name = name1 and healthplan = 'B')

/* C_{52} is generated from algorithm CPA-insert (lines 12-23) */

C_{52} = return 1 if {name=name1 and amount1 > 20000} else return 0.

/* C_{53} is generated from algorithm CPA-insert (lines 9-11) */

```
C53 = select 1 from dual where exists (select * from DOCTOR where
     name=name1 and disease = 'smallpox')
```

apply θ to each of the sub-constraints

```
θ(C51) = select 1 from dual where exists (select * from patient
where name = 'john' and healthplan = 'B')

θ(C52)= return 1 if {'john' = 'john' and 25000 > 20000}
                else return 0

θ(C53)= select 1 from dual where exists (select * from DOCTOR
                  where name='john' and disease = 'smallpox')
```

$C_5 = C_{51} \wedge C_{52} \wedge C_{53}$. In this example, $\theta(C_{51}) = 1$ (true), $\theta(C_{52}) = 1$ (true) and $\theta(C_{53}) = 1$ (true). The conjunction of C_{51}, C_{52} and C_{53} evaluates to true. Hence, C_5 is violated (*from Theorem 3.1*)

Similarly, for the example constraint C_6 from sub section 3.3, we generate:

```
θ(C61) = select 1 from dual where exists (select * from patient
                      where name = 'john' and healthplan = 'B')

θ(C62)= return 1 if {'john'='john' and 'emergency' = 'emergency'}
                else return 0
```

$C_6 = C_{61} \wedge C_{62}$. In this example, $\theta(C_{61}) = 1$(true), $\theta(C_{62}) = 1$(true). The conjunction of C_{61} and C_{62} evaluates to true. Hence, C_6 is also violated (*from Theorem 3.1*). Note that we do not need to evaluate other constraints if one of the constraints is violated by an update statement. In this example, since C_5 is violated, we do not need to evaluate/check for C_6. We show the evaluation of C_6 simply for illustrative purposes.

3.4.2 CPA-delete

Here, we make an important observation that an update statement involving a delete can only violate referential integrity constraints, semantic integrity constraints involving aggregate predicates (sum, max, min, avg and count), state transition and state

sequence constraints involving aggregate predicates. It does not violate semantic integrity constraints involving arithmetic predicates considered in this sub section.

3.4.3 CPA-modify

The constraint planning algorithm for a modify statement can be modeled as a delete followed by an insert statement.

3.4.4 Discussion

The CPA considers only elementary update statements. The elementary update statements are statements affecting only one row of a table at a time. However, note that any update statement can be translated equivalently to a set of elementary updates. Hence the generality of the CPA is not lost.

We have not considered the issue of constraint checking in the presence of transactions. Let a transaction T change the current database state D to D'. A naïve approach would be to check for constraint violations in D' and if any constraints are violated, we rollback to the previous state D.

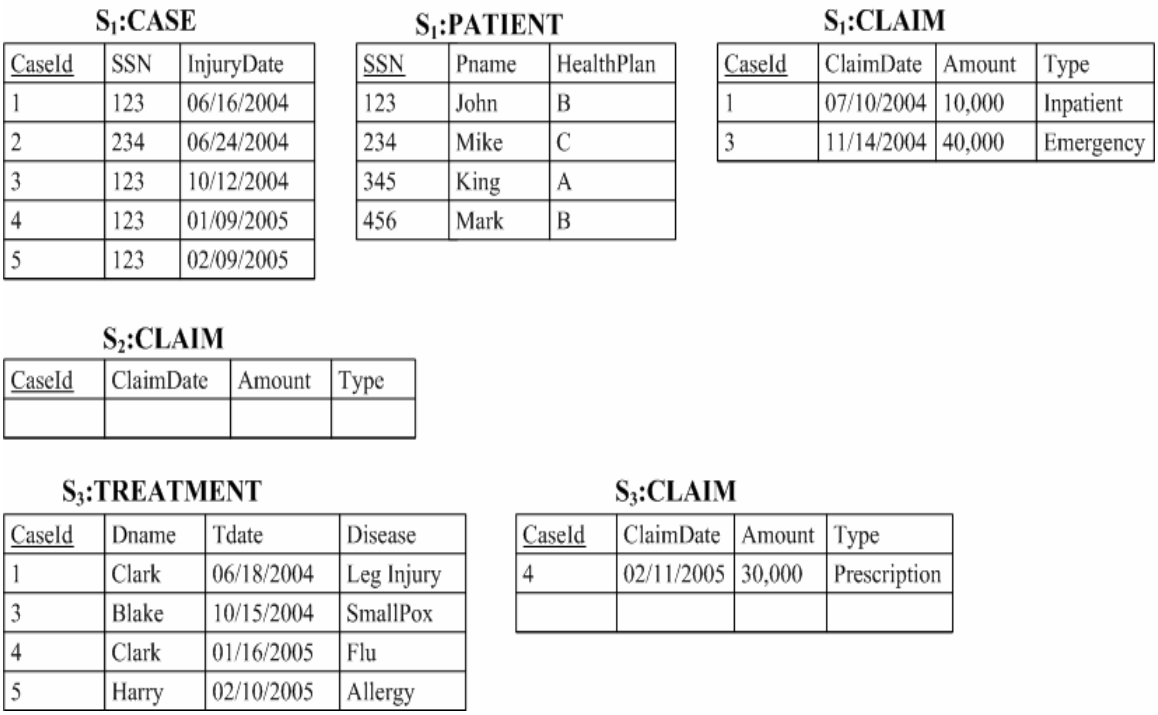
The CPA can generate sub-constraints for constraints having universally quantified variables over a simple conjunction of predicates. We extend our work on CPA for sub goals of the global constraint involving aggregate predicates (sum, max, min, avg and count) in the next section. The extensions are: a) modified Algorithm CPA-insert to deal with aggregates; b) a new algorithm CPA-delete (constraint planning algorithm for a delete statement).

3.5 Constraint Planning Involving Aggregates

Similar to concepts in Section 3.4, here, we present constraint checking algorithms involving aggregate predicates. For the aggregate predicates, we extend the example database from before to having both horizontal and vertical partitioning as given in the next sub section.

3.5.1 Example Database

To make the problem interesting and generic, we consider both vertical and horizontal distribution of data (see Figure 8). *CLAIM* table is horizontally distributed across all the three sites, S_1, S_2 and S_3. A patient can make multiple claims uniquely identified by their *CaseId*. For example, John is associated with multiple claims (with CaseId's - 1, 3, and 4) on sites S_1 and S_3. We avoid the description of the tables and columns as they are self explanatory from their names.



S_1:CASE

CaseId	SSN	InjuryDate
1	123	06/16/2004
2	234	06/24/2004
3	123	10/12/2004
4	123	01/09/2005
5	123	02/09/2005

S_1:PATIENT

SSN	Pname	HealthPlan
123	John	B
234	Mike	C
345	King	A
456	Mark	B

S_1:CLAIM

CaseId	ClaimDate	Amount	Type
1	07/10/2004	10,000	Inpatient
3	11/14/2004	40,000	Emergency

S_2:CLAIM

CaseId	ClaimDate	Amount	Type

S_3:TREATMENT

CaseId	Dname	Tdate	Disease
1	Clark	06/18/2004	Leg Injury
3	Blake	10/15/2004	SmallPox
4	Clark	01/16/2005	Flu
5	Harry	02/10/2005	Allergy

S_3:CLAIM

CaseId	ClaimDate	Amount	Type
4	02/11/2005	30,000	Prescription

Figure 8 : Example Multidatabase Involving Both Horizontal and Vertical Partitions

3.5.2 Aggregate Constraints

In order to represent integrity constraints in the context of a database as query evaluation in the database, we consider integrity constraints in the form of range-restricted denials (datalog style notation).

$\leftarrow A_1 \wedge A_2 \wedge \ldots \wedge A_n$

Where each A_i is a literal or an aggregate literal involving a base predicate and global variables are assumed to be universally quantified over the whole formula ([18]). An aggregate literal is expressed as

$A_i(\hat{s}, \alpha(y):v) :- B$

Where

(i) *B* is a conjunction of predicate atoms that represent relations,

(ii) $\hat{s}$ is the grouping list of attributes that must appear some where in the body of the rule – B,

(iii) α is aggregate function such as *avg, count, max, and min*,

(iv) *y* is the aggregate variable, and

(v) *v* is the result of applying the aggregate function.

We assume that the aggregate literals are not recursive.

For example integrity constraint C_1 states “the sum of claim amounts for each patient with healthplan 'B' may not be more than 100000”. This can be conveniently represented using the approach of [28]. A constraint is a query whose result is either 0 or 1 (Gupta and Widom ([28]) calls it "panic"). If the query produces 0 on the multidatabase D, then D is said to satisfy the constraint, or the constraint is violated on D.

```
A(SSN,SUM(Amount):v1) :- S1:PATIENT(SSN,-,'B'),
                         S1:CASE(CaseId,SSN,-),
                         S1:CLAIM(CaseId,-,Amount,-).
B(SSN,SUM(Amount):v2):-  S1:PATIENT(SSN,-,'B'),
                         S1:CASE(CaseId,SSN,-),
                         S2:CLAIM(CaseId,-,Amount,-).
```

```
C(SSN,SUM(Amount):v3):-  S1:PATIENT(SSN,-,'B'),
                         S1:CASE(CaseId,SSN,-),
                         S3:CLAIM(CaseId,-,Amount,-).
```

$PanicC_1$ ← `A(SSN,v1),B(SSN,v2),C(SSN,v3),v1+v2+v3 >10000.`

For convenience, we will refer to *Panic*C_1 as just C_1.

3.5.3 CPAggreg-insert

Algorithm `CPAggreg-insert` (constraint planning involving aggregates for an insert statement) shown in Figure 9 gives constraint decompositions (C_{ij}'s), corresponding to global constraint C_i (involving aggregates) and an insert statement (decomposition is based on the locality of sites). Algorithm `CPAggreg-insert` takes as input the insert statement U and the list of all global constraints C and outputs the list of sub-constraints (C_{ij}) for each C_i being affected by U.

*DOL (*database object list) identifies the database objects being modified by the update statement, U. *DOL* (*line 3*) identifies, the table R with attributes (column names) $a_1 \ldots a_n$ inserted with values $t_1 \ldots t_n$. The constraint data source table, *CDST* (*line 4*) gives the list of sites involved, for each constraint being affected by the update statement. The outer for loop variable *i* (*line 6*) loops through all the constraints $C_1 \ldots C_q$ affected by the update U. The inner for loop variable *j* (*line 7*) loops through each site ($<S_{11},\ldots,S_{1n_1}>,\ldots,<S_{q1},\ldots,S_{qn_q}>$) for each constraint *i*. Inside the for loop (*lines 6-40*), all the sub-constraints C_{ij}'s are generated. $S_j{:}p_1(X_1), p_2(X_2), \ldots, p_r(X_r)$ (*line 8*) denotes, for a particular

site S_j, $X_1 \ldots X_r$ are the vector of variables corresponding to the predicates (table names), $p_1 \ldots p_r$.

Algorithm `CPAggreg-insert`

1: **INPUT**: (a) U: insert S_m:R($t_1,\ldots,t_n$)
(b) C: list of all global constraints /* Note: insert is occurring on site S_m */

2: **OUTPUT**: list of sub-constraints < $C_{i_1},\ldots,C_{ik_i}$ > for each C_i affected by U

3: DOL (U) = < R (a_1= $t_1,\ldots,a_n$= t_n) >

4: CDST(C,DOL(U)) = < <C_1, ($S_{11},\ldots,S_{1n_1}$)>,…,<C_q, ($S_{q1},\ldots,S_{qn_q}$)> >

5: *let* θ = {$x_1 \leftarrow t_1,\ldots,x_n \leftarrow t_n$}be obtained from DOL(U) where $x_1 \ldots x_n$ are variables corresponding to the columns of table R

6: **for** each *i* in {*1*... *q*} **do**

7: **for** each *j* in {*1*...n_i} **do**

8: *let* `A` be all arithmetic sub goals associated with S_j , `Aggreg` be all Aggregate literals associated with site S_j (atleast one of the predicates in the body of aggregate literal belongs to S_j) and S_j: $p_1(X_1)$, $p_2(X_2)\ldots$ $p_r(X_r)$ be sub goals of C_i associated with S_j

9: **if** (j <> m) **then** /* site where update is not occurring */

for each Aggregate literal, aggreg(ŝ,α(y):v):- B **do**

A_{ijd} = select ŝ,α(y)
from predicates in the Body B
where <cond1>
group by ŝ

10: if all the predicates in B belong to same site S_j, <cond1> is obtained by standard joining of tables from B using variables from θ; else semi-join operation is employed for distributed tables. It also includes any arithmetic sub goal conditions. A_{ijd} is the value of the aggregate literal corresponding to constraint C_i, site S_j and d is the nth such literal. V_{ijd} is the value of aggregate operation corresponding to A_{ijd}

11: **end for**

12: **else if** (j=m) **then** /* site where update is occurring */

13: **for** each Aggregate literal, aggreg(ŝ,α(y):v):- B **do**

14: A_{ijd} = select ŝ,α(y)
from predicates in the Body B
where <cond2>
group by ŝ /* this step is similar to line 10 */

15: if α = "sum"

16: v_{ijd} = θ(y)+ v_{ijd} /* vijd is the value calculated from Aijd of line 14 */

17: else if α = "min"

18: v_{ijd} = min(θ(y),v_{ijd})
19: else if α = "max"
20: v_{ijd} = max(θ(y),v_{ijd})
21: else if α = "count"
22: if θ(y) is not null then v_{ijd} = v_{ijd} + 1 /* we are assuming single row inserts */
23: else if α = "avg"
24: add θ(y)to the sum aggregate and divide by total count
25: **end if**
26: **end for**
27: **if** (there exists variables in A that do not appear in Aggreg or θ) **then**
28: **for** each variable *var* in A that do not appear in Aggreg or θ **do**
29: *let* k be the site where *var* appears in a sub goal, S:t(X) in C_i
30: IP_{ikd} = (select Col(νar) from S:t where <cond3>)
31: Col(νar) is the column name corresponding to νar
32: <*cond3*> is obtained from joining X and θ . d is nth *intermediate predicate*
33: **end for**
34: **end if**
35: C_{ij} = return 1 if (<*cond4*> and (*logical and*) *A′*) else return 0.
36: <*cond4*> is obtained from θ and $X_1 \ldots X_r$. *A′* is A with IP's replacing corresponding variables and v_{ijd}'s replacing corresponding aggregate values
37: **end if** /* end of the "else if" on line 12 */
38: **end for**
39: **end for**
40: apply the substitution θ(U) to all C_{ij}

Figure 9 : Algorithm CPAggreg-insert

A critical feature of the algorithm is the generation of v_{ijd}'s (lines 15-28) at the site where update is happening. Also, an *intermediate predicate* (IP) is generated only at the site where update is occurring. In concept, *IP*'s represent information that needs to be shared from a different site. Implementation wise, *IP* is a SQL query returning value of

the variable, var (*line 30*) from a different site. IP_{ikd} (*line 32*) means the *d*th intermediate predicate corresponding to constraint C_i and site S_K.

Theorem 3.2: *The conjunction of sub-constraints C_{ij}'s, generated from Algorithm CPAggreg-insert conclusively determines, if an insert statement violates a global constraint C_i involving aggregates.*

Proof: The proof is similar to the proof of Theorem 3.1. The idea is to prove that conjunction of C_{ij}'s generated from *CPAggreg-insert* entails the original global constraint C_i. Hence, it logically follows that if C_i is violated by an insert statement, so is the conjunction of Cij's. ■

Example 3.2

Here, we show the working of the algorithm `CPAggreg-insert` on the example database and constraints introduced in Chapter 3.5.1. Consider the initial multidatabase state as shown in Figure 8.

Input: U_1 = insert into S_2:CLAIM values

(5,'02/20/2005',25000,'Emergency');

C = list of all global constraints

Output: list of sub-constraints Ci_1 ,…,Cik_i for each C_i affected by U_1

DOL = {S_2:CLAIM (CaseId=5,ClaimDate='02/20/2005',

Amount=25000,Type='Emergency'}.

CDST = <C_1, (S_1, S_2, S_3)> /* C_1 is given in Section 2.2 */

θ = {S_2:CLAIM(CaseId1=5,ClaimDate1='02/20/2005',

Amount1 = 25000,Type1 = 'emergency') }

/* A_{111} and A_{112} are generated from CPAggreg-insert from line 11 */

```
A111 = select PA.SSN,sum(CL.Amount) "v111"
       from S1_PATIENT PA, S1_CASE CA, S1_CLAIM CL
       where PA.SSN = CA.SSN and PA.HealthPlan = 'B'
       and CA.CaseId = CL.CaseId and CA.CaseId = CaseId1
       group by PA.SSN;
```

A_{112} = select PA.SSN,sum(CL.Amount) "v_{112}"

from S1_PATIENT PA, S1_CASE CA, S3_CLAIM CL

where PA.SSN = CA.SSN and PA.HealthPlan = 'B'

and CA.CaseId = CL.CaseId and CA.CaseId = CaseId1

group by PA.SSN;

/* A_{121} is generated from CPAggreg-insert from line 16 */

A_{121} = select PA.SSN,sum(CL.Amount) "v_{121}"

from S1_PATIENT PA, S1_CASE CA, S2_CLAIM CL

where PA.SSN = CA.SSN and PA.HealthPlan = 'B'

and CA.CaseId = CL.CaseId and CA.CaseId = CaseId1

group by PA.SSN;

V_{121} = amount1 + v_{121}; /* from line 18 */

C_{12} = return 1 if {V_{111}+V_{112}+V_{121} > 100000} /* line 36 */

$\theta(C_{12})$ = return 1 if { $\theta(V_{111})+\theta(V_{112})+\theta(V_{121})$ > 100000 }

/* $\theta(V_{111})$ is obtained by substituting CaseId1=5 in A_{111} and similarly we calculate $\theta(V_{112})$ and $\theta(V_{121})$ */

Hence, $\theta(C_{12})$ = return 1 if (50000+30000+25000 > 100000)

Therefore, $C_1 = C_{12} = 1$ (true). Hence, constraint C_1 is violated by the given update statement.

3.5.4 CPAggreg-delete

CPAggreg-delete (Constraint Planning involving Aggregates for a delete) proceeds in a similar way as the CPAggreg-insert. We identify major differences from the previous algorithm. The first part of CPAggreg-delete contains almost same logic as lines 1-13 of CPAggreg-insert. The only difference is that input is a delete statement as opposed to insert. The calculation of aggregate literals at the site(s) where delete is not occurring is similar to the insert algorithm. In the second part of the algorithm, the site where delete is occurring, line 16 of CPAggreg-insert is modified in the where clause and <cond2> is obtained by negating the variables from θ (negation is done because it is a delete statement). To illustrate the negation idea, let us consider a delete statement on Site S_1, where we delete all claims, where amount < 5000. The calculation of aggregate literals on S_1 would then consider only amounts > 5000, if the delete were to happen. Lines 17-27 of insert algorithm are not necessary for the delete case.

Theorem 3.3: *The conjunction of sub-constraints C_{ij}'s, generated from Algorithm CPAggreg-delete conclusively determines, if a delete statement violates a global constraint C_i involving aggregates.*

Proof: similar to the proof of Theorem 3.2. ■

3.5.5 CPAggreg-modify

The constraint planning algorithm for a modify statement can be modeled as a delete followed by an insert statement.

3.5.6 Discussion

The constraint planning algorithm considers only elementary update statements. The elementary update statements are statements affecting only one row of a table at a time. However, note that any update statement can be translated equivalently to a set of elementary updates. Hence the generality of the algorithm is not lost. Also, note that we have not considered the issue of constraint checking in the presence of transactions. Hence, the issues regarding deferred or immediate constraint checking does not apply. Although it is trivial, we can say, by default, we use immediate constraint checking. It would be challenging to extend the constraint checking algorithms involving transactions without allowing the update to occur.

The aggregate literals of the constraints are executed in an order which respects dependencies among them. This order can be computed from a dependency graph of literals by evaluating bottom up in such a graph. The graph is acyclic, as we do not consider recursion for aggregate literals.

NULL values are automatically handled by the system by conforming to the ANSI SQL standard. ANSI SQL standard specifies that a constraint (CHECK (<*searchcondition*>)) is violated only when <*searchcondition*> evaluates to false. In the other cases (true or unknown), constraint is satisfied. In our context, when <*searchcondition*> is false, conjunction of sub-constraints evaluates to true; hence, constraint is violated. Otherwise, constraint is satisfied.

When we compare approach of constraint checking after update vs. constraint checking before update, the only extra time we are spending is the time spent in the part of the algorithm, where the site is the updating site. Even at this site, performance gain can be obtained by carrying out most of the steps at compile time. If we have a template of possible update statements, most of the steps of the algorithm can be executed in compile time and when an actual update statement is given, a template match can occur and only the last line of the algorithm (line 41 of `CPAggreg-insert`) happens at run time. By pushing most of the processing at compile time, we gain efficiency at run time. Hence, constraint checking before the update statement saves lot of time and resources that are spent on rollbacks and also uses very less time at run time.

Once the decomposition of each constraint into sub-constraints happens, any optimizations that increase the efficiency of the constraint checking process can be employed. The parameters we consider are: number of sites accessed by a sub constraint, locality of sites, and, history of constraint failures on a site. Constraint optimizations are part of our on-going future work.

3.6 Implementation

The constraint planning algorithms discussed earlier have been implemented using JDK version 1.3 and the system UI is designed using javax.swing package. We use aglets agent framework [35] for implementing agents. A prototype of the system implementation is given in Figure 10

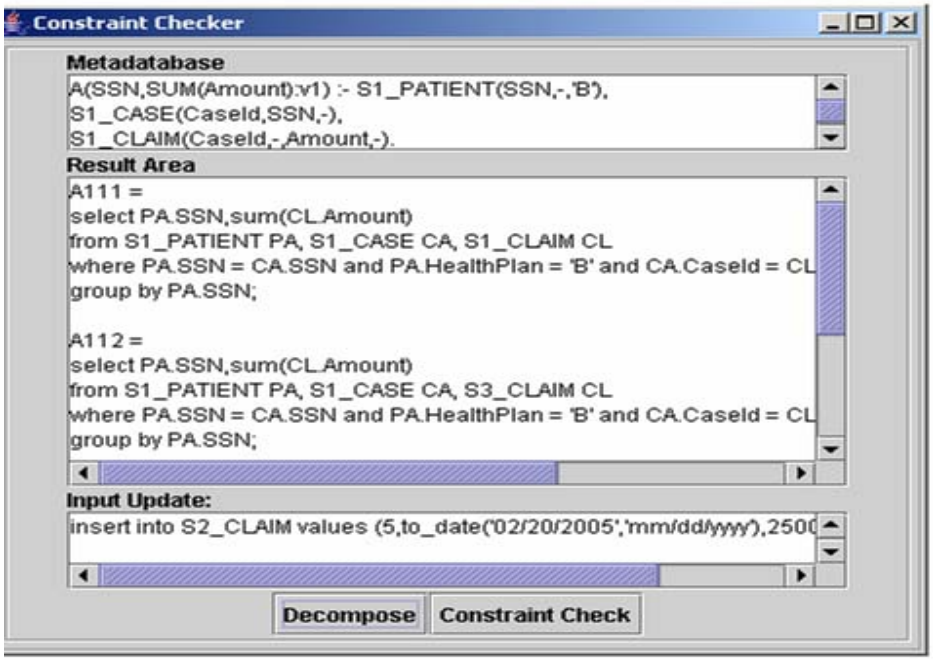


Figure 10 : Constraint Checker Implementation

When the user clicks “Decompose”, sub-constraints are generated and displayed in the “Result Area”. The resulting sub-constraints are executed by mobile agents on remote sites, when the user clicks “Constraint Check”.

The motivation for using mobile agents are: (i) For each sub constraint generated from CPA, a mobile agent would carry the data processing code and execute the sub

constraint check on the remote site. Agents on the remote site process the data and only filtered data is transported to the base site. Thus we save on the network bandwidth. (ii) Constraint checking mechanism is much faster as the sub constraint checks on remote sites are executed in parallel by mobile agents. 3) Since the mobile agent framework is inherently asynchronous, the algorithm can be extended to carry sub constraint checks on mobile multidatabases.

The constraint executor module inside constraint checker interfaces with agent based execution engine. The agent based execution engine is responsible for creating, dispatching, managing and terminating of agents. The constraint executor gathers the results obtained by dispatching agents using execution engine and makes a decision if a global constraint is violated.

A prototype of an agent execution engine ([38]) has been implemented in the context of *System of Mobile Devices (SyD)* middleware ([44, [45], [46]]. SyD is a new middleware that enables rapid application development for heterogeneous, autonomous and mobile devices. More details on the SyD and our agent based execution engine can be found in [38], [44], [45], and [46].

3.7 Performance Evaluations

We calculate the time constraint checker takes to check the Global constraint for C_1 and C_2 that we mentioned earlier in Section 3.1.2 separately and then we exclude the time the remote aglets itself use for communication. We calculate this timing by repeating the experiment over a number of times and taking the average of all the timings

obtained. Also, we experiment with different timings by allowing rollback on the database, and without the need for rollback.

In Figure 11, we summarize the time taken by the system in 3 cases, which are total time to check constraints without using the algorithm (allowing rollback), total time to check constraints using CPA-insert algorithm and time for aglet communication. Constraint C_1 involves sites S_1, S_2, and S_3 and constraint C_2 involves S_1 and S_2.

	Total time to check constraints without using algorithm (allowing rollback)	**Total time to check constraints using CPA-insert algorithm**	**Time for aglet communication**
C_1	9884 miliseconds	328 miliseconds	235 miliseconds
C_2	671 miliseconds	266 miliseconds	172 miliseconds

Figure 11: Time Consumed By Using CPA-insert And Without Using It

The first column is the time to check constraints without using the CPA-insert algorithm. In this case, the constraint checker will go ahead and insert the insert statement after getting it from user on local source. If the constraint checker detects that the insert statement is violated, the system will rollback the update statement to the previous database state.

The second column is the time to check constraints using the CPA-insert algorithm. The system will start by waiting for the insert statement from user on local source. After that, the system will follow the same steps as the first case, but it will not execute the insert statement at first. Also, the system will use the CPA-insert algorithm to decide and construct the sub constraint for the constraint planner.

The third column is the time for aglet communication. We calculate the time from when the constraint checker spawns all the remote agents until all the results are obtained.

We can see from the table that the constraint checker with CPA-insert algorithm saves lot of time.

From the given experiments, we can comfortably generalize that as the number of constraint violations increases, our system performs better as we do not incur the overhead of time spent on rolling back the database state. Our future additions to these sets of experiments would be to undertake an exhaustive list of performance evaluations for insert/delete/modify statements. Also, we would like to generate random sets of update statements and then check for the system behavior.

4. CONSTRAINT CHECKING IN A SYSTEM OF XML DATABASES

Consider a scenario wherein two or three different companies host XML data (native XML database management system) at different and independent sites. Data at these sites is not necessarily independent, but may participate in a relationship with data from other sites. A single XUpdate ([50], [36]) on one site might cause a global constraint (*global XConstraint*[4]) to be violated. Hence we need an approach to check for such constraint violations. In the XML database setting, the majority of the times, users are interested in generating (updating), integrating and exchanging data. So, frequent updates on XML data may cause frequent global constraint violations. *Hence we need a plan that will efficiently and speedily check for such global constraint violations.*

Plan *A* would be to translate the XML document to relational data using methods such as those found in [14] and [47 , and then, map the updates and constraints on the XML data to corresponding updates and constraints on the relational data ([15]). Now the problem of constraint checking on XML data is pushed to the problem of constraint checking on relational data. There are well established models for constraint checking in the relational world. However, this approach suffers from the overhead cost involved in transforming and storing XML data to relational data ([31]). Plan *B* would be to check for constraint violations on the XML data without transforming to relational data. It should

---

[4] By global XConstraints we mean global semantic integrity constraints affecting multiple XML databases.

be noted that using plan *A* vs. plan *B* depends on the application being considered. If the application contains millions of records and if it benefits to use relational database features such as querying, fast indexing, etc., it is worth while to consider plan A; otherwise plan B suffices for a normal sized application. In this paper, we consider the plan B route.

A brute force approach would first update the XML document and then check for constraint violations. If a constraint is violated, we can rollback. However, such a brute force approach suffers from the overhead of time and resources spent on rollback. Hence, we need an approach that would check for constraint violations before updating the database and therefore obviates the need for rollback situations.

In our constraint checking procedure, constraint violations are checked at compile time, *before* updating the database. Our approach centers on the design of the *XConstraint Checker*. Given an XUpdate statement and a list of global XConstraints, we generate *sub XConstraint*[5] checks corresponding to local sites. The results gathered from these sub XConstraints determine if the XUpdate statement violates any global XConstraints. Our approach is *efficient*; since we do not require the update statement to be executed before the constraint check is carried out and hence, we avoid any rollback situations. Our approach achieves *speed* as the sub constraint checks can be executed in parallel.

---

[5] Sub XConstraint is a XML constraint, expressed as an XQuery, local to a single site (more details in Section 4).

4.1 Overview of XConstraint Checking

Figure 12 gives overview of the system. We propose three-tier architecture.

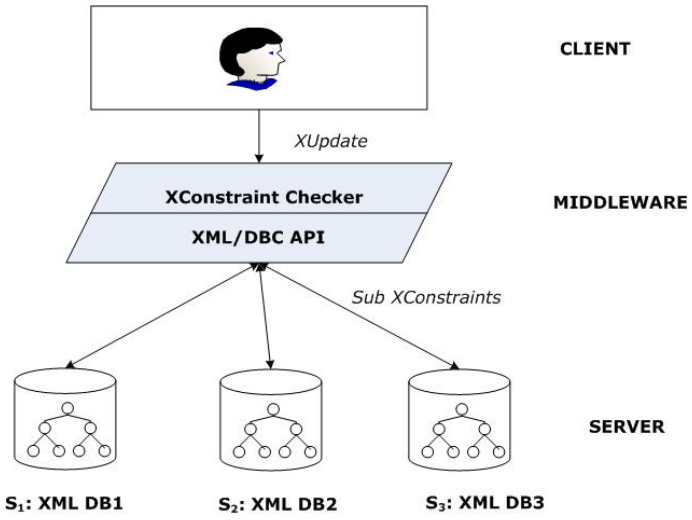


Figure 12 : Overview Of XConstraint Checking System

The server side consists of two or more sites hosting native XML databases. In Figure 12 we show three sites S_1, S_2 and S_3. The client makes an XUpdate request through the middleware. The middleware consists of *XConstraint Checker* and the XML/DBC API ([22]). We have introduced our notations for representing XConstraints and proposed architecture for XConstraint Checker. One of the important modules in XConstraint Checker is the *XConstraint Decomposer.* Furthermore, we (i) give the algorithmic description for the XConstraint Decomposer, (ii) illustrate the algorithm with clear examples, and (iii) implement the system. The XConstraint Decomposer takes as input a global XUpdate and a list of global XConstraints and outputs sub XConstraints to be executed on remote sites. XML/DBC is the standard XML XQuery API that facilitates access to XML based data products. The XML/DBC API consists of two API's: 1) The

Java API is a JDBC extension to query XML collections using XQuery. 2) The web services API is designed to provide a SOAP style server interface to clients. In our case, XML/DBC API executes sub XConstraints corresponding to remote sites. The XConstraint Checker gathers results obtained from sub XConstraints and makes a decision whether a constraint is violated. Only in the event of no constraint being violated, the XUpdate statement is executed.

The rest of the chapter is organized as follows: In Section 4.2, we give example XML databases that will be referred to throughout the paper. We also give the syntax of XUpdate language and introduce our notations for defining global XConstraints. In Section 4.3, we give the internal architecture of the XConstraint Checker. In Section 4.4, we present the algorithmic description of the XConstraint Decomposer that decomposes a global XConstraint into a conjunction of sub XConstraints. In Section 4.5, we give implementation details.

4.2 Preliminaries

Here we give an example healthcare XML database and explain the notations of XUpdate. We also introduce our notation for defining XConstraints.

4.2.1 Example XML Database

Consider a sample *healthdb.xml* represented in a tree form in Figure 13. Figure 13 gives the logical representation of the HEALTHDB XML databases. Physically, information is distributed across multiple sites:

Site S_1: PATIENT information such as *SSN (primary key)*, *PName* and *HealthPlan* is stored. CASE information with *CaseId (primary key – like a sequence number)*, *SSN*, and *InjuryDate* is also stored.

Site S_2: patient's CLAIM information such as *CaseId (primary key)*, *ClaimDate*, *Amount* and *Type* is recorded.

Site S_3: TREATMENT information such as *CaseId (primary key)*, *DName* (doctor name), *TDate* (Treatment Date), and *Disease* is stored.

Note that a patient can suffer multiple injuries uniquely identified by their *CaseId* at Site S_1, and can also make multiple claims identified by their *CaseId* at site S_2.

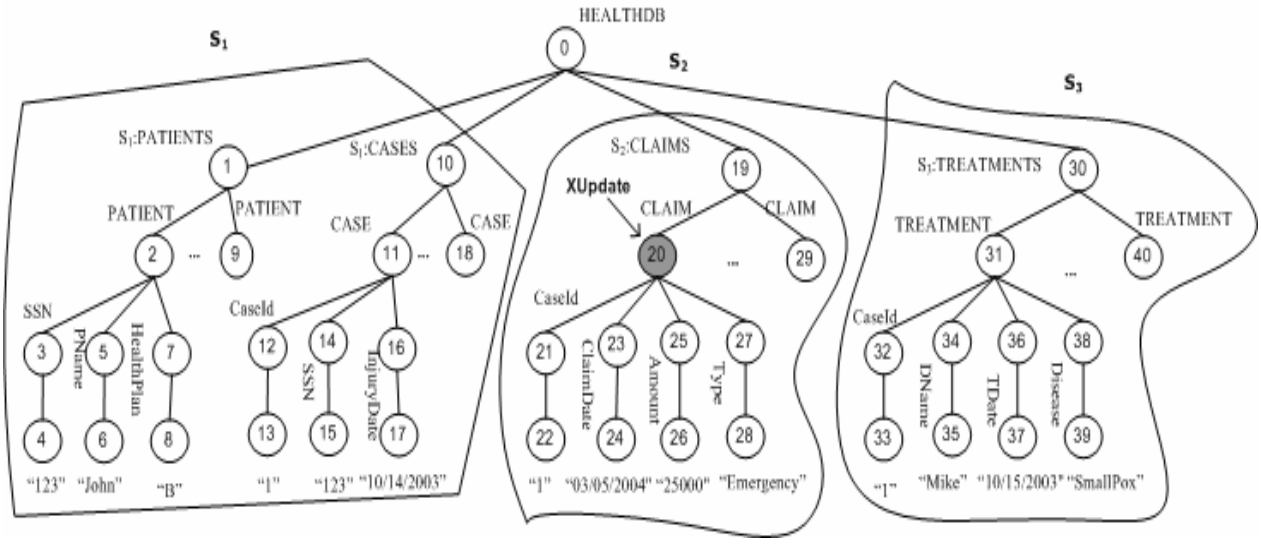


Figure 13: Tree Representation of Healthdb.xml

4.2.2 XUpdate

XUpdate is the language extension to XQuery to accommodate insert, replace, delete and rename operations. Tatarinov *et al*. ([50]) gives XUpdate language syntax and

semantics. For purpose of better presentation, we give brief description and syntax of XUpdate. The syntax of XUpdate is given below.

```
FOR $binding1 IN XPath-expr, ...
LET $binding: = XPath-expr, ...
WHERE predicate1, ...
updateOP, ...

where updateOP is defined in EBNF as :

UPDATE $binding { subOP {,subOP}* }

where subOP is defined as :

DELETE $child |
RENAME $child TO name |
INSERT content [BEFORE | AFTER $child] |
REPLACE $child with $content |
FOR $binding IN XPath-subexpr, ...
    WHERE predicate1, ... updateOP
```

The semantics of the FOR, LET, WHERE clauses (FLW) are taken from XQuery, while the updateOP clause specifies a sequence of update operations to be executed on the target nodes identified by FLW clause. Here, we note that, in our context, the XPath-expr from the FOR clause can only refer to nodes from a single site, restricting the updates to only a single site. This is a reasonable assumption, as an XUpdate on a single site might cause one or more global XConstraints to be violated and we want to check for such constraint violations at compile time (before the XUpdate is executed). Below, we show a sample XUpdate occurring on the XML tree (node 20) of Figure 13.

```
FOR $cl in document("healthdb.xml")/HEALTHDB/S2:CLAIMS
UPDATE $cl
{
INSERT <CLAIM>
          <CaseId>1</CaseId>
          <ClaimDate>03/05/2004</ClaimDate>
          <Amount>25000</Amount>
          <Type>Emergency</Type>
       </CLAIM>
}
```

For a detailed description of the XUpdate language, readers are referred to [36] and [50].

4.2.3 XML Constraint Representation

Semantic integrity constraints can be considered as a general form of assertions. They specify a general condition in the database which needs to be true always. Constraints of this type deal with information in a single state of the world. Throughout the paper, we denote semantic integrity constraints for XML database as *XConstraints*. *Global XConstraints* are the constraints spanning multiple XML databases. Here we give the constraint representation for global XConstraints.

A datalog rule (expressed as Head ← Body) without a Head clause is referred to as a denial. It is customary to represent integrity constraints in the logic databases as range restricted (safe or allowed) denials.

Definition 4.1: In order to represent global XConstraint in the context of XML database as query evaluation, we consider global XConstraint in the form of range restricted denials (datalog style notation) given below:

$c \leftarrow X_1 \wedge X_2 \wedge, \ldots, X_n$, where c is the name of the global XConstraint and each X_i is either an XML literal or Arithmetic literal ■

We define both XML literal and arithmetic literal below. The definition of XML literal is chiefly inspired from [11] and [15]. Semantics for representing key constraints for a single XML database are given there. We extend their semantics by introducing user defined variables, term paths and XML literals for representing global XConstraints for multiple XML databases.

Definition 4.2: *XML literal* is defined as follows:

X_i : (Q_i , (Q_i' , [$V_{i1} = t_{i1}$, $V_{i2} = t_{i2}$,…, $V_{ik_i} = t_{ik_i}$]))

Using the syntax from [11], [15], Q_i , Q_i' and t_{i1}, t_{i2} ,…, t_{ik_i} are path expressions corresponding to X_i . V_{i1}, V_{i2} ,…, V_{ik_i} are user defined variables corresponding to t_{i1}, t_{i2} ,…, t_{ik_i} . Q_i is called the context path, Q_i' the target path and t_{i1}, t_{i2} ,…, t_{ik_i} are the term paths. Context path Q_i identifies the set of context nodes, *c* and for each *c*, V_{i1}, V_{i2} ,…, V_{ik_i} are the set of user defined variables corresponding to the term paths, t_{i1}, t_{i2} ,…, t_{ik_i} reachable from *c* via Q_i'. ■

Definition 4.3: *Arithmetic literal* is defined as: *expression θ expression,* where expression – is a linear expression made of variables occurring in *XML literals,* integer constants, and the four arithmetic operator +, -, *, /; *θ* – is a comparison operator (=, <, >, <=, >=, <>). Joins between nodes are expressed either as an equality (=) between two variables in an arithmetic literal or by having the same variable name appear in different XML literals within the same global XConstraint. Note that variables with the same name cannot appear in the same XML literal. ■

Now, we are ready to define the satisfiability of a global semantic integrity constraint (global XConstraint), C.

Definition 4.4: A XML tree T is said to satisfy a global integrity constraint (global XConstraint), C, if and only if the conjunction of X_1, X_2 ,…, X_n evaluates to *false* ■

The motivation behind using our constraint representation and negative semantics for checking the satisfiability of a global semantic integrity constraint are: 1) constraint representation using our approach resembles query evaluation for heterogeneous databases (logic, relational, XML) and hence is very generic due to the inherent logic based approach used in representing the XConstraints. 2) Global XConstraints decomposed using Algorithm 4.1 (Section 4) are much easier using our XConstraint representation, as the sub XConstraints generated are XQueries evaluated against local database and can return a true/false. Hence the overall conjunction (which is also true/false) of sub XConstraints determines the satisfiability of a global XConstraint.

Note that each Qi ,Qi', user defined variables and the term paths corresponding to each XML literal - Xi has the site information referred to as Sj and can only refer to a single site. However, a global XConstraint has one or more XML literals and hence can refer to multiple XML databases. In case of Arithmetic literal, *expression θ expression,* the variables in the expression could belong to different sites. If two variables are not the leaf nodes, the equality join among the two variables is similar to the node equality considered in [11].

Example 4.1: Consider two global XConstraints c_1 and c_2 defined on *healthdb.xml*. Constraint c_1 states that a patient with HealthPlan 'B' diagnosed with 'SmallPox' may not claim more than 40000 dollars. Constraint C_2 states that a patient with HealthPlan 'B' may not file a claim of type 'Emergency'.

```
C1:-
  (//S1:PATIENTS,
  (./PATIENT,[ssn=./SSN,healthplan=./HealthPlan])),
```

```
  (//S1:CASES,(./CASE,[caseid=./CaseId,ssn=./SSN])),
  (//S2:CLAIMS,(./CLAIM,[caseid=./CaseId,amount=./Amount])),
  (//S3:TREATMENTS,
  (./TREATMENT,[caseid=./CaseId,disease=./Disease])),
   healthplan = 'B',disease = 'SmallPox',amount > 40000.
```

```
C2:-
    (//S1:PATIENTS,
    (./PATIENT,[ssn=./SSN,healthplan=./healthplan])),
    (//S1:CASES,(./CASE,[caseid=./CaseId,ssn=./SSN])),
    (//S2:CLAIMS,(./CLAIM,[caseid=./CaseId,type=./type])),
    healthplan = 'B',type = 'Emergency'.
```

For the example contained in Figure 13, C_1 is satisfied, but C_2 is violated. C_1 is satisfied for the healthdb.xml as one of the arithmetic literals *amount (node 25, value = 25000) > 40000* returns false and hence the whole conjunction for c_1 evaluates to false. c_2 is violated as the conjunction for c_2 evaluates to true. Arithmetic literal, *healthplan (node 7, value = 'B') = 'B'* evaluates to true and similarly, *type (node 27, value='Emergency') = 'Emergency'* evaluates to true and hence the whole conjunction for c_2 evaluates to true.

We also note that keys introduced in [15], can be expressed using our representation. Consider a key constraint, c_3, which states that within the context of PATIENTS, a PATIENT is uniquely identified by SSN. Using the notation of [15], c_3 can be expressed as follows:

```
C3:- (/HEALTHDB/S1:PATIENTS,(./PATIENT,{./SSN}))
```

A key constraint such as c_3 could be expressed in our notation (a functional dependency) as follows:

```
C3:-
    (//S1:PATIENTS,(./PATIENT,[ssn=./SSN,name1=./PName])),
    (//S1:PATIENTS,(./PATIENT,[ssn=./SSN,name2=./PName])),
     name1 <> name2.
```

This has some similarity with the notion of template dependencies ([20]), wherein we can represent any general constraints in relations.

4.3 XConstraint Checker

We first give the assumptions of the system and then present the detailed architecture of the XConstraint Checker.

4.3.1 Assumptions

XConstraint Checker relies on the fundamental concepts (XConstraint, XUpdate) introduced in Chapter 4.2. The assumptions we make for the XConstraint Checker are:

1. A restricted set of XUpdate language is considered without losing the generality of the approach. We permit the following SubOP's: DELETE $child, INSERT content [BEFORE | AFTER $child] and REPLACE $child with $content. The optional [BEFORE | AFTER $child] is applicable for an ordered execution model of XML tree. Also, we restrict the updates to *elementary updates*. The elementary update considers: (i) updates occurring only on one single node of an XML tree and (ii) updates with only one SubOP at a time. However, note that any update can be equivalently transformed to a set of elementary updates; therefore, we do not lose the generality of the approach.
2. XML constraint representation follows from Section 4.2.3. We consider semantic integrity constraints with arithmetic literals affecting multiple XML databases. A topic for future work would be to consider XML constraints with aggregate literals (sum, max, min, avg and count).

4.3.2 XConstraint Architecture

The internal architecture of the XConstraint Checker is presented in Figure 14. The XConstraint checker interfaces with the rest of the system as shown in Figure 12. The XConstraint Checker consists of the following modules.

- **XUpdate Parser**: parses a XUpdate statement input by the user and identifies the XNode Value List (*XNVL*), involved in the XUpdate.
- **XMetadatabase**: stores and acts as a repository of global XConstraints.
- **XMeta Extractor**: extracts only the global XConstraints being affected by the XUpdate.
- **XConstraint Decomposer**: decomposes a global XConstraint into a set of sub XConstraints to be validated locally on remote sites.

The overall process of constraint checking is explained in the following four steps (see Figure 14).

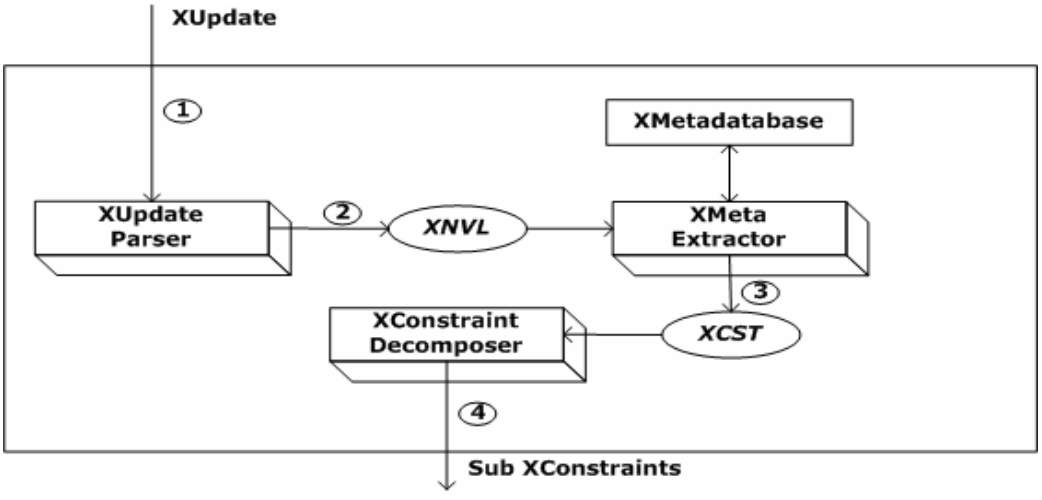


Figure 14: XConstraint Architecture

STEP 1

The user issues a XUpdate statement on one of the sites. **Figure 15** gives the initial XML database state before the XUpdate statement is executed. For example, user issues a XUpdate statement, XU_1 on site S_2.

```
XU1 =
FOR $cl in document("healthdb.xml")/HEALTHDB/S2:CLAIMS
UPDATE $cl
{
INSERT <CLAIM>
          <CaseId>1</CaseId>
          <ClaimDate>03/05/2004</ClaimDate>
          <Amount>25000</Amount>
          <Type>Emergency</Type>
       </CLAIM>
}
```

Figure 16 gives the modified tree representation of the *healthdb.xml*, if the update is successful. The nodes affected by the XUpdate are shown in filled circles.

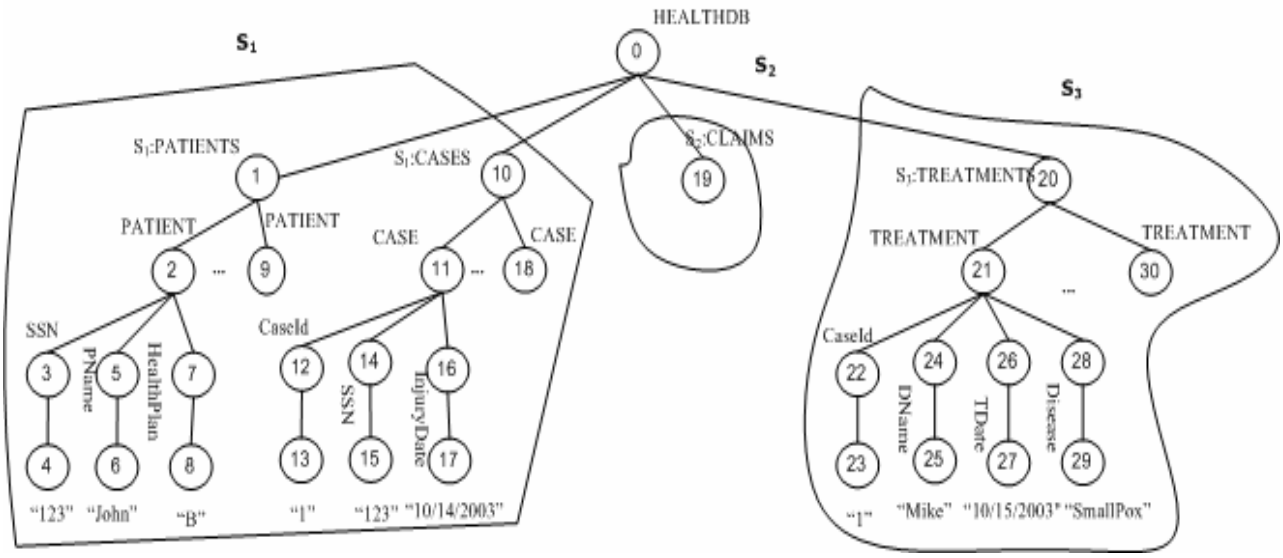


Figure 15: Tree Representation of Healthdb.xml before XUpdate

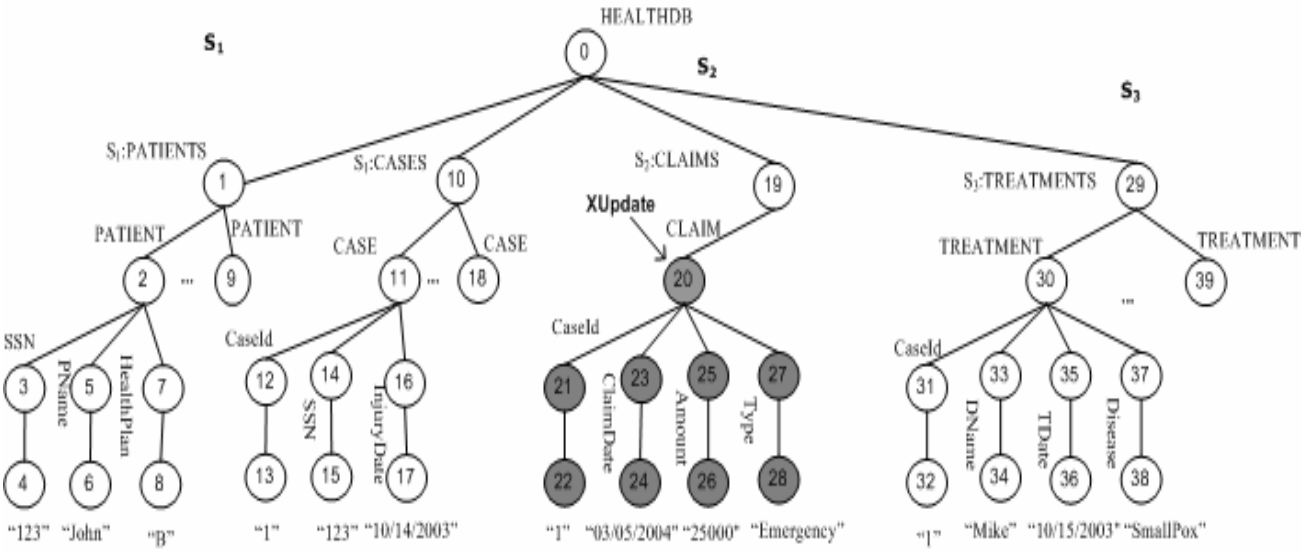


Figure 16: Modified Tree Representation, If XUpdate Is Successful

STEP 2 (XUpdate Parser)

The XUpdate Parser parses the given XUpdate statement and identifies the XML node being modified. The output from this step is the XML Node Value List (*XNVL*).

XNVL = $N(a_1=v_1,a_2=v_2,\ldots,a_n=v_n)$, where N is the node being updated and is obtained from the $binding in the XUpdate syntax, $v_1,v_2,\ldots,v_n$ are the values being updated corresponding to the attributes $a_1,a_2, \ldots ,a_n$. $a_1,a_2, \ldots,a_n$ are either the XML sub elements or XML attributes being updated and are obtained from the content of the XUpdate statement (Section 2.2). For the running example,

```
XNVL = {/HEALTHDB/S2:CLAIMS/CLAIM( CaseId = 1,
       ClaimDate = '03/05/2004', Amount = 25000,
       Type='Emergency')}
```

STEP 3 (XMeta Extractor)

Let XU↓ denote the path involved in executing the XUpdate statement, XU on the XML tree T. Similarly, C↓ denotes path in defining the constraint C. We say that a XUpdate, XU might violate a constraint C if, XU↓ ∩ C↓ is not empty. For the running example, XU_1↓ corresponds to the following nodes: {20,21,22,23,24,25,26,27,28}, C_1↓ matches {3,4,7,8,12,13,14,15,21,22,25,26,31,32,37,38} and C_2↓ matches {3,4,7,8,12,13,14,15, 21,22,27,28} (refer to Figure 16). $XU_1 \cap C_1$↓ is not empty and $XU_1 \cap C_2$ is also not empty; hence, both the constraints might be violated by the update statement. If a global schema or a global DTD (Document Type Definition) is given, we can identify the list of global XConstraints that might be violated by simply consulting the global DTD.

The *XMeta Extractor* identifies the list of constraints being affected by the XUpdate and constructs the XConstraint Source Table (*XCST*). $XCST(C_i) = < C_i, list(S_j) >$, where C_i is the constraint identifier and $list(S_j)$ is the list of sites being affected by C_i. For the running example, *XCST* is given in Figure 17. The XMeta Extractor sends the *XCST* to the XConstraint Decomposer.

C_i	list(S_j)
C_1	(S_1,S_2,S_3)
C_2	(S_1,S_2)

Figure 17: XCST

STEP 4 (XConstraint Decomposer)

The XConstraint Decomposer generates a set of sub XConstraints, C_{ij} on the basis of locality of sites. C_{ij} is the sub XConstraint corresponding to constraint - C_i and site - S_j. We present algorithmic description of generating C_{ij}'s in the next section. For the running example, c_{11}, c_{12}, c_{13}, c_{21} and c_{22} are generated. The values of the sub XConstraints are also given in the next section.

4.4 XConstraint Decomposer

The basic idea of XConstraint Decomposer is to decompose a global constraint into a conjunction of sub XConstraints, where each conjunct represents the constraint check as seen from each individual site. Given an XUpdate statement, a brute force approach would be to go ahead and update the XML document and then check for constraint violations. However, we want to be able to check for constraint violations without updating the database. In other words, the XUpdate is carried out only if it is a non constraint violator. Thus, we avoid any potential rollbacks.

Our idea here is to scan through a global XConstraint C_i, XUpdate U and then generate conjunction of sub XConstraints, C_{ij}'s. The value of each conjunct (each C_{ij}) is either 0 or 1. If the overall value of conjunction is 1, constraint C_i is violated (*from Theorem 4.1*).

Algorithm 4.1 presented in Figure 18 gives the constraint decompositions (C_{ij}'s) corresponding to a global constraint C_i and an XUpdate statement involving an insert statement.

Algorithm 4.1

1: **INPUT** : (a) *XNVL* = S_m:N(a_1=v_1,a_2=v_2,…,a_n=v_n) on XML tree T

// Note: insert is occurring on Site S_m

2: (b) *XCST* = < <C_1,(S_{11},S_{12},…,S_{1n_1})>,…,<C_q,(S_{q1},S_{q2},…,S_{qn_q} > >

3: **OUTPUT**: list of sub XConstraints <C_{i1},C_{i2},…,C_{ik_i} > for each C_i affected by XUpdate, XU

4: **for** each i in {1…q} **do**

5: **for** each j in {1…n_i} **do**

6: *let* S_j:(Q_1,(Q_1',[X_1])),…, S_j:(Q_r,(Q_r',[X_r])) be *XML literals* and A be all *arithmetic literals* associated with S_j

7: **if** (j <> m) **then**

8: C_{ij} = for var_1$ in document("T")Q_1.Q_1' ,

9: for var_2$ in document("T")Q_2.Q_2', …,

10: for var_r$ in document("T")Qr.Qr'

11: where <*cond1*>

12: return 1

13: <*cond1*> is obtained by joining variables with same name appearing in *XML literals* and including any arithmetic conditions

14: **else if** (j = m) **then** /* site where update is occurring */

15: **if** (there exists variables in A that do not appear among X_1…X_r) **then**

16: **for** each variable, ν in A that do not appear among X_1…X_r **do**

17: *let* k be the site where ν appears as one of the *XML literals*, (S_k:Q(Q'[X]))

18: IP_{ikd} = for $ν in document ("T")Q.Q'

19: where <*cond2*>

20: return {$ν /$t_ν$ }

21: $t_ν$ is the path expression corresponding to $ν in *XML literal* and <*cond 2*> is obtained from X_1…X_r and X and d is the nth *intermediate predicate*

22: **end for**

23: **end if**

24: C_{ij} = return 1 if (<*cond3*> and *A'*) else return 0

25: <*cond3*> is obtained from XNVL *and (logical and)* X_1…X_r

A' is A with *IP's* replacing corresponding variables in A

```
26:      end if
27:    end for
28: end for
```

Figure 18 : XML Constraint Checker Algorithm

Algorithm 4.1 takes as input XML Node Value List, *XNVL* (STEP2, Section 3.2) and XConstraint Source Table - *XCST* (STEP3, Section 3.2) and gives as output the sub XConstraints. *XNVL* (line 1) identifies the node N being inserted with the values $v_1 \ldots v_n$ corresponding to attribute names, $a_1 \ldots a_n$ (similar to XUpdate syntax). The update is occurring on site S_m. The outer for loop variable *i* (line 4) loops through all the constraints $C_1 \ldots C_q$ affected by the XUpdate. The inner for loop variable *j* (line 5) loops through each site $< (S_{11},S_{12},\ldots,S_{1n_1}),\ldots,(S_{q1},S_{q2},\ldots,S_{qn_q})>$ for each constraint C_i. Inside the for loop (lines 4-28), all the sub-constraints C_{ij}'s are generated. $X_1 \ldots X_r$ (line 6) denotes vector of user defined variable *v* = path expression *t* in a *XML literal* (Definition 2.2). $Q_1.Q_1'$ (line 8) denotes the conjunction of path expressions Q_1 and Q_1'. A critical feature of the algorithm is the generation of *intermediate predicate*, *IP* (line 18). *IP*'s are generated only at the site where update is occurring. For each variable that occurs in a different site, we generate *IP*. Conceptually, *IP* denotes information that needs to be shared from a different site; implementation wise, *IP* is an XQuery returning the value of the variable from a different site. IP_{ikd} means the *d*th *intermediate predicate* corresponding to constraint C_i and site S_k.

Theorem 4.1: *The conjunction of sub XConstraints, C_{ij}'s generated from Algorithm 4.1 conclusively determines if a XUpdate statement violates a global XConstraint, C_i.*

Proof sketch:

1. Given a XUpdate statement occurring on site S_m and a global constraint C_i, C_i can be written as conjunction of *XML literals* and *arithmetic literals*. If the whole conjunction evaluates to *false*, C_i is satisfied (*from Definition 2.4*).
2. Each sub XConstraint C_{ij} needs to achieve the exact same result as the *XML literal* and A*rithmetic literals* corresponding to site S_j.
3. At this point C_{ij} falls in one of the two cases depending on the site S_j :

Case 1: (j <> m) - This is the case where C_{ij} corresponds to a site other than where update is occurring. The generation of C_{ij} in this case involves computing appropriate join conditions and applying arithmetic conditions on *XML literals* and A*rithmetic literals* associated with S_j. Hence C_{ij} naturally achieves the exact same result as the *XML literals* and A*rithmetic literals* associated with S_j.

Case 2: (j = m) - This is the case where C_{ij} corresponds to the site where update is occurring. The generation of C_{ij} in this case consists of two parts. Part 1 consists of information from the same site S_j – trivial case (just like Case 1). Part 2 consists of acquiring information from a different site. For each such variable, a unique *intermediate predicate* is generated. *IP*'s are XQueries that return the values of such variables by computing appropriate joins and arithmetic conditions involved with such variables. Hence, *IP*'s guarantee correct information exchange from a different site. The reason, we generate unique *IP*'s is we can either store all the IP's at a global directory such as the XMeta database or we can generate *IP*'s at run time.

From steps 2 and 3 we observe that the conjunction of sub XConstraints C_{ij}'s, entails the global XConstraint, C_i. Hence, if C_i determines whether a XUpdate violates the constraint, then conjunction of its C_{ij}'s also determines if the constraint C_i is violated. In other words, if the whole conjunction of C_i evaluates to false, constraint C_i is not violated, otherwise C_i is violated. ■

Example 4.2

We illustrate the working of the algorithm on the example from Chapter 4.2.1. Here, we illustrate the sub XConstraints generated when intermediate predicates are not involved.

```
XNVL = {/HEALTHDB/S2:CLAIMS/CLAIM( CaseId = 1,
       ClaimDate = '03/05/2004', Amount = 25000,
       Type='Emergency')}
```

$CDST\ (C_1) = <C_1, (S_1, S_2, S_3)>$

where

C_1:-

```
  (//S1:PATIENTS,
  (./PATIENT,[ssn=./SSN,healthplan=./HealthPlan])),
  (//S1:CASES,(./CASE,[caseid=./CaseId,ssn=./SSN])),
  (//S2:CLAIMS,(./CLAIM,[caseid=./CaseId,amount=./Amount])),
  (//S3:TREATMENTS,
  (./TREATMENT,[caseid=./CaseId,disease=./Disease])),
   healthplan = 'B',disease = 'SmallPox',amount > 40000.
```

```
/* C11 is generated from Algorithm 4.1 (lines 7-13) */
C11 = for $var1 in document("healthdb.xml")//S1_PATIENTS/PATIENT,
      for $var2 in document("healthdb.xml")//S1_CASES/CASE,
      where $var1/SSN = $var2/SSN and $var2/CaseId = 1 and
      $var1/HealthPlan = "B"
      return 1

/* C12 is generated from Algorithm 4.1 (lines 14-26) */
C12 = return 1 if {1 = 1 and 25000 > 40000}
      else return 0

/* C13 is generated from Algorithm 4.1 (lines 7-13) */

C13 = for $var1 in
      document("healthdb.xml")//S3_TREATMENTS/TREATMENT
```

```
      where $var1/CaseId = 1 and $var1/Disease = "SmallPox"
      return 1
```

So, $C_1 = C_{11} \wedge C_{12} \wedge C_{13}$. In this example, C_{11} = 1(true), C_{12} = 0(false) and C_{13} = 1(true). The conjunction of C_{11}, C_{12} and C_{13} evaluates to *false*. Hence the update statement does not violate constraint C_1 (*from Theorem 4.1*)

```
Similarly,
C21 = for $var1 in document("healthdb.xml")//S1_PATIENTS/PATIENT,
      for $var2 in document("healthdb.xml")//S1_CASES/CASE,
      where $var1/SSN = $var2/SSN and $var2/CaseId = 1 and
      $var1/HealthPlan = "B"
      return 1
C22 = return 1 if {1 = 1 and "Emergency" = "Emergency"}
                else return 0
```

So, $C_2 = C_{21} \wedge C_{22}$. In this example, C_{21} = 1(true), C_{22} = 1(true). The conjunction of C_{21} and C_{22} evaluates to *true*. Hence the update statement violates constraint C_2 (*from Theorem 4.1*)

Example 4.3

Here, we illustrate the generation of sub-constraints when intermediate predicates are involved. For the example database given in Chapter 4.2.1, consider C_4, which states "A patient's date of claim may not be earlier than his/her injury date". Constraint C_4 can be expressed as:

```
C4:-  (//S1:PATIENTS,(./PATIENT,[ssn=./SSN])),
      (//S1:CASES,
      (./CASE,[caseid=./CaseId,ssn=./SSN,idate=./InjuryDate])),
      (//S2:CLAIMS,(./CLAIM,[caseid=./CaseId,cdate=./ClaimDate])),
      cdate<idate.
```

We also assume date arithmetic is available for both XConstraints and sub XConstraints represented as XQueries.

Say, an update statement XU_2 is occurring on site S_2 of the XML tree given in **Figure 15**.

```
XU2 =
FOR $claim in document ("healthdb.xml")/HEALTHDB/S2:CLAIMS
UPDATE $claim
{
INSERT <CLAIM>
          <CaseId>1</CaseId>
          <ClaimDate>09/14/2003</ClaimDate>
          <Amount>25000</Amount>
          <Type>emergency</Type>
       </CLAIM>
}
```

Applying steps 1-4 from Chapter 4.3, we obtain

```
XNVL = {/HEALTHDB/S2:CLAIMS/CLAIM( CaseId = 1,
       ClaimDate = '09/14/2003',Amount = 25000,
       ,Type='Emergency')}

CDST (C4) = <C4, (S1, S2)>

IP411= for $var1 in document("healthdb.xml")//S1_PATIENTS/PATIENT,
      for $var2 in document("healthdb.xml")//S1_CASES/CASE,
      where $var1/SSN = $var2/SSN and $var2/CaseId = 1
      return $var2/InjuryDate

C42 = return 1 if (1 = 1 and (09/14/2003 < IP411) )
              else return 0
```

$C_4 = C_{42}$. C_{42} evaluates to true. Hence, C_4 is violated (*from Theorem 4.1*).

Discussion

Algorithm 4.1 considers elementary XUpdate statements involving an insert statement. The elementary XUpdate statements are statements affecting only one node of an XML tree. However, note that any XUpdate statement can be translated equivalently to a set of elementary updates; hence, the generality of the algorithm is not lost. Also, we do not consider the issue of transactions. Hence, rollbacks caused by failed transactions can not be avoided.

Here, we make an important observation that a XUpdate statement involving a delete can only violate referential integrity constraints, semantic integrity constraints

involving aggregate predicates (sum, max, min, avg and count), state transition and state sequence constraints involving aggregate predicates. It does not violate semantic integrity constraints involving arithmetic predicates considered in this paper. XUpdate statement involving a modify can be modeled as a delete followed by insert. Hence, we have presented a complete model for global semantic integrity constraint checking for XML databases with arithmetic predicates under insert/delete/modify statements.

Let m be the number of global constraints, n is the number of sites, and p is the number of tables at the site where update is occurring. The time complexity of Algorithm 4.1 is $O(m*n)$. If we have a template of possible XUpdate statements, note that all the steps of the algorithm can be carried out during compile time and we can generate sub-constraints for each such template. However, at run time, when an actual XUpdate statement is given, a template match can occur and the corresponding sub-constraints, which are already decomposed at compile time, can be executed in parallel at the corresponding sites. Hence, the run time complexity is $O(p)$ plus the communication time required for executing at the corresponding sites. P is usually a smaller number and is usually much smaller than $m*n$. Hence, we say the run time complexity is $O(1)$. If we did not execute sub-constraints in parallel, the run time complexity would be $O(m*n)$. Hence, by pushing most of the processing at compile time, we gain efficiency at run time.

Algorithm 4.1 considers global XConstraints involving a simple conjunction of XML literals and arithmetic literals. We will extend our semantic integrity constraint checking for global XConstraints involving *aggregate literals* (sum, count, max, min and avg).

4.5 Implementation

The XConstraint Checker architecture and Algorithm 4.1 have been implemented using JDK version 1.3 and the system UI is designed using javax.swing package. A prototype of the system implementation is given in Figure 19. The XMetadatabase panel (top left panel) stores global XConstraints, result area (centre panel) displays the results, XUpdate panel (lower left panel) gives the user to input XUpdate statement and XML database panel (right most panel) shows the xml files of two or more different sites.

The GUI has two buttons, “Decompose” and “XConstraint Check”. When the user clicks “Decompose”, sub XConstraints are generated and displayed in the result area panel, shown in Figure 20. The resulting sub XConstraints need to be executed on their corresponding remote XML database sites using the XML/DBC API ([22]), when “XConstraint Check” button is clicked. However, for our system implementation, we are not considering the action of XConstraint Check, as we have not seen a working version of the XML/DBC kind of products. We have checked for the validity of the sub XConstraints by executing them on the Galax XQuery interpreter version 0.3.5 ([21]) using the sample healthdb.xml file.

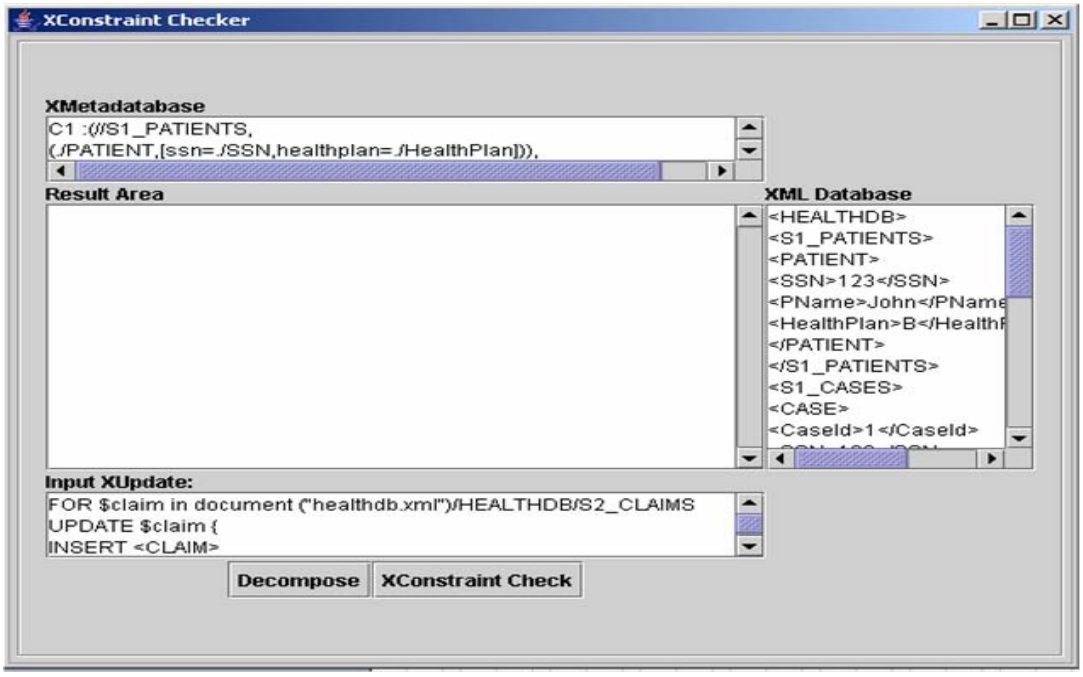


Figure 19 : XConstraint Checker GUI

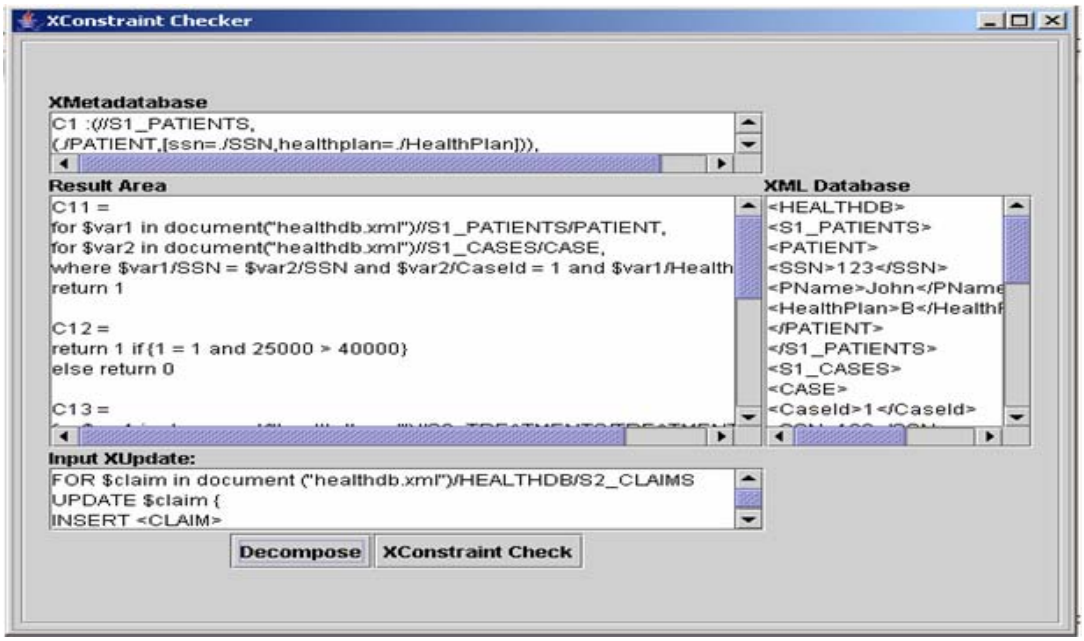


Figure 20: XConstraint Checker GUI After Decompose

5. RELATED WORK

Our related work section broadly spans three areas: constraint checking in relational databases, constraint checking in XML databases and mobile agents for constraint checking.

5.1 Constraint Checking in Relational Databases

Much of the research concerning integrity constraint checking has been done in the area of relational database systems. Grefen and Apers ([24]) provide an excellent survey of constraint checking and enforcement methods in relational database systems. Grefen and Widom ([25]) give an exhaustive survey of protocols for integrity constraint checking in federated database systems. Gupta and Widom ([28]) give approaches for constraint checking in distributed databases at a single site. They show how a class of distributed constraints can be broken down into local update checks. Some of the approaches for distributed databases and federated databases can be easily applied to multidatabases with some minor changes. Ceri and Widom ([12]) propose inter-database triggers for maintaining equality constraints between heterogeneous databases. Their approach relies on active rules and assumes a persistent queue facility between sites. Widom and Ceri ([52]) mention research on active databases and constraints.

Grufman *et al*. ([26]) provide a formal description of distributing a constraint check over a number of databases. They propose that the problem of generating sub-constraints from a global constraint is the same as rewriting a predicate calculus expression of the constraint check into a form in which the distribution of the data is

respected. The rewritten predicate can be seen as a conjunction of sub-constraints, where each sub constraint may be visualized as the constraint check as seen from each individual database. During the process of rewriting the constraint check predicate, they introduce the concept of intermediate predicates. We use the idea of intermediate predicates in our constraint planning algorithm discussed in Section 3.4. In their constraint distribution model, an update statement is first carried out and the new database state is checked for constraint violation. If the constraint is violated, the update is rolled back. Our work differs from theirs by giving an algorithm that automatically decomposes a global constraint in to a conjunction of sub-constraints. Our approach is much more sophisticated, as we check for constraint violation without actually updating the database. The update is executed only when there are no constraint violations. Hence our algorithm is efficient as there are no problems involved with rollbacks as such. Also, the overhead introduced from our algorithms are very negligible as the only extra overhead is the time required for constraint checking on the site where update is happening. At all the remaining sites, constraint check takes the same time.

Ibrahim ([30]) proposes a strategy for constraint checking in distributed database where data distribution is transparent to the application domain. They propose an algorithm for transforming a global constraint into a set of equivalent fragment constraints. However, our algorithm coverage is much broader as we can have different tables on different sites. In our approach, the constraint planning algorithm generates the sub-constraints, which can be readily implemented on Oracle database system. With minor changes, it can be implemented on any commercial database.

5.2 Constraint Checking in XML databases

Constraint checking in XML databases is very new and very few research results exist in this area. Here, literature survey spans two major topics: constrains for XML and constraint checking in XML.

5.2.1 Constraints for XML

The idea of keys and foreign keys for XML was introduced in [11] and [15]. The basic approach is to express constraints using path expressions. We also study constraint representation in distributed databases. In [28], a constraint is treated as query whose result is either 0 or 1.If the query produces 0 on the database D, D is said to satisfy the constraint. Otherwise, constraint is violated (Gupta and Widom ([28]) call it "panic"). We have extended the approach of [11] and [15] with datalog style notations and also used the concepts from [28] in representing XConstraints. Our XConstraint representation is limited to only semantic integrity constraints involving *arithmetic literals*. We plan to extend the representation to *aggregate literals*.

5.2.2 Constraint Checking in XML

Our approach of constraint checking for multiple XML databases is novel as we have not seen any research on semantic integrity constraint checking for multiple XML databases. Research on validating keys for XML can be found in [3], [6], and [15]. To our knowledge, the only work closest to ours is from Kane *et al.* ([31]). Kane et al. execute only those XUpdates that would preserve the consistency of the XML document with respect to a particular schema. The underlying idea is to generate constraint check sub

queries. The constraint check sub queries check if the given XUpdate statement violates the consistency of the XML document. The XUpdate statement is executed only if it is safe. Hence they avoid any potential rollbacks. We also take a similar route. However, they do not consider semantic integrity constraint checking for multiple XML databases.

5.3 Agent Based Approach

Mobile agents have been recently recognized as an efficient means for distributed information retrieval ([8]). Recent research has considered using mobile agents for global querying, but none of the literature so far has looked in to the aspect of using mobile agents for global constraint checking. We intend on using a suitable mobile agent platform for implementing our constraint checker system.

ACQUIRE ([17]), an agent based complex query and information retrieval engine considers an agent-based approach for information retrieval from distributed data sources. ACQUIRE translates each user query into a set of sub queries by employing a combination of planning and traditional database query optimisation techniques. For each sub query ACQUIRE then sends a corresponding mobile agent which does the computation work and retrieves the result. When all the agents have returned, ACQUIRE filters and merges retrieved data and the results are displayed to the user. MOMIS ([4]) gives a framework for information integration that deals with the integration and query of multiple, heterogeneous information sources. MOMIS (Mediator environment for multiple information sources) uses agent-based approach, where in they have multiple agents doing different kinds of tasks. A Global virtual view of all the sources is generated

using XML as the basis. A Global schema is generated from the individual source sites (wrapper agent). The wrapper agent resides at each of the individual source sites and monitors for any changes in the data structure of the sources. The Query Manager agent is responsible for querying information from all the source sites. Similar to ACQUIRE sub queries are generated and Query Manager Agent is responsible for querying from individual data sources. Our intent is also similar to the above, however they are using mobile agents in a different context of global querying and we intend on using mobile agents for global constraint checking.

6. CONCLUSIONS

It is well understood that constraint checking for a System of Databases is an important area of research. We have made contributions primarily along two lines of research: constraint checking for a System of Relational Databases (R-SyDb) and constraint checking for a System of XML Databases (X-SyDb).

Chapter 3 summarized our research results in the area of semantic integrity constraint checking for R-SyDb. We have designed and implemented a general framework of an agent based *constraint checker* for checking constraint violations in a System of Relational Databases. We have also proposed constraint planning algorithms that form as an algorithmic backbone for constraint checker. The constraint planning algorithms take as an input an update statement, a list of global constraints and make a decision, if a constraint has been violated. The performance results have shown that constraint planning algorithm shows better timing as compared to the other approaches.

Figure 21 gives the constraint violation chart under insert/update/delete statement. An X indicates a possible constraint violation corresponding to the column. Research on Row ID of "1" is trivial and Row ID 2 is a special case of Row ID 4, which we have already completed. Research on Row ID's 2 and 5 is a major component of our research, which has been summarized in Chapter 3. We intend on proposing algorithms in the future for checking constraint violations for semantic integrity constraints involving state transition, state sequence and referential integrity constraints.

Row ID	Constraint	Insert	Delete	Modify	Research
1	Domain Constraint	X		X	trivial
2	Key Constraint	X		X	special case of Row ID 4
3	Referential Integrity Constraint	X	X	X	future
4	Semantic Integrity Constraint with Arithmetic Predicates	X		X	done
5	Semantic Integrity Constraint with Aggregate Predicates	X	X	X	done
6	State Transition and State Sequence Constraints	X		X	future

Figure 21: Constraint Violation Chart for Insert/Update/Delete

We have proposed solutions for semantic integrity constraint checking for multiple XML databases (refer Chapter 4). As stated earlier, none of the research has considered the issue of semantic integrity constraint checking for multiple XML databases. Although, native XML databases are not being used very much for commercial purposes, we believe that with the growing popularity of XQuery coupled with efficient storage and indexing techniques for native XML databases, multiple XML databases will be a norm. With this goal in mind, we have presented the architecture of XConstraint Checker. XConstraint Checker is part of a middleware module that determines if an XUpdate statement violates any global XConstraints. In the area of X-SyDb, we have:

(i) introduced a notation for representing XConstraints,

(ii) proposed architecture for XConstraint Checker,

(iii) formalized an algorithm for XConstraint Decomposer, and

(iv) implemented a prototype of the system with the ideas discussed in Chapter 4.

Given an XUpdate statement and a list of global XConstraints, XConstraint Decomposer (A*lgorithm 4.1*) generates sub XConstraints to be validated locally on remote sites. Since most of the steps of the algorithm can be carried out at compile time, we achieve efficiency at run-time.

Future Work

In the near future, we would like to pursue research by extending on the current work and possibly work in new emerging areas in databases.

Hybrid Execution Engine Module

As stated earlier in Chapter 3.6, we implemented an agent based execution engine module and applied it in the context of SyD Middleware. We propose to implement a hybrid engine module for system on mobile devices middleware. Hybrid engine module exploits the best of the features of Asynchronous RMI and mobile agents. When the user on a mobile device tries to execute a method call on another device, the hybrid engine module can automatically switch between agent approach and RMI approach based on a decision algorithm.

R-SyDb and X-SyDb

We are interested in extending the dissertation topic to develop new algorithms and systems for checking integrity constraint violations for state transition and state sequence constraints. We would like to tailor the existing algorithms to work for state transition and state sequence constrains. XML database is a new research area and we are

keenly interested in checking for all types of constraint violations for XML databases. We have been considering constraint checking for homogeneous databases. We aim to pursue research for constraint checking in heterogeneous databases.

Constraint Optimizations

So far, for both R-SyDb and X-SyDb, we have only looked at finding correctly and efficiently, if a constraint is being violated by an update statement. However, we have left out the issue of constraint optimizations. For each global XConstraint (or constraint) that could be violated, multiple sub-XConstraints (or constraints) are generated. Hence, we have a large number of sub XConstraints (or constraints) when we consider all the set of global XConstraints (or constraints). All this process can be done in compile time. Therefore, efficient ordering of sub XConstraints (or constraints) for executing on remote sites would optimize the constraint checking mechanism. To achieve this, we plan to introduce an XConstraint Optimizer (Constraint Optimizer) module.

Transactions and Fault Tolerance

We also would like to consider the issue of transactions, concurrency control, and fault tolerance for Constraint Checker, XConstraint Checker, and Metadatabase modules. We plan to introduce a concurrency control manager module along with the constraint checker, which would handle concurrent requests for updates. We also plan to pursue research on indicating a tolerance level for each constraint. This is especially true for Bioinformatics databases, as sometimes the biologists would like to ignore the issue of satisfying constraints.

7. BIBLIOGRAPHY

[1] R. Ahmed, P. De Smedt,W. Du,W. Kent,M. Ketabchi,A. Litwin, W. A., Rafii, and M. C. Shan. The Pegasus heterogeneous multidatabase system. *IEEE Computer*, 1991, pp. 19-27.

[2] Y. Arens, C. A. Knoblock and W.Shen. Query Reformulation for Dynamic Information Integration. *Journal of Intelligent Information Systems*, 6(2/3), 1996, pp. 99-130.

[3] M.Benedikt, C.Y. Chan,W. Fan,J.Freire and R.Rastogi.Capturing both Types and Constraints in Data Integration. *ACM SIGMOD*, 2003.

[4] S. Bergamaschi, G. Cabri, F. Guerra, L. Leonardi, M. Vincini, F. Zambonelli. Supporting Information Integration with Autonomous Agents. *CIA* 2001: 88-99.

[5] A.R. Bobak. Distributed and Multi-Database Systems. *Artech House Publishers*, San Francisco,California, USA, 1996.

[6] B. Bouchou, M. Halfeld-Ferrari-Alves, and M. Musicante.Tree Automata to Verify XML Key Constraints. *WebDb* 2003.

[7] Y.Breitbart, Hector Garcia-Molina, and Abraham Silberschatz. Overview of multidatabase transaction management. *VLDB Journal: Very Large Data Bases*, 1(2):181-293, 1992.

[8] B. Brewington, R. Gray, K. Moizumi, D. Kotz et al. Mobile Agents in Distributed Information Retrieval, *Intelligent Information Agents*, Springer-Verlag, 1999.

[9] M. W. Bright, A. R. Hurson, and S. H. Pakzad. A taxonomy and current issues in multidatabase systems. *IEEE Computer*, pages 50--59, Mar.1992.

[10] O. Bukhres and A. Elmagarmid. Object-Oriented Multidatabase Systems: A Solution for Advanced Applications. Prentice Hall, New Jersey, 1996.

[11] P. Buneman, S. Davidson, W.Fan, C.Hara, and W.Tan. Keys for XML. In *WWW10*, 2001, pp.201-210.

[12] S. Ceri, and J. Widom. Managing Semantic Heterogeneity with Production Rules and Persistent Queues. Proceedings of the *Nineteenth International Conference on Very Large Data Bases*, pages 108-119, Dublin, Ireland, August 1993.

[13] S. Chawathe, H. Garcia-Molina, J. Hammer, K.Ireland,Y. Papakonstantinou,J. Ullman and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. Proceedings of *IPSJ Conference*, 1994.

[14] Y. Chen, S.B. Davidson, C.S. Hara, and Y. Zheng . RRXF: Redundancy Reducing XML Storage in Relations. Proceedings of the *International Conference on Very Large Databases*, 2003.

[15] Y.Chen, S.B. Davidson, and Y.Zheng. Constraint Preserving XML Storage in Relations. In *WebDB*, 2002.

[16] Y.Chen, S. Davidson,Y. Zheng. XKvalidator:A Constraint Validator For XML. Proceedings of *ACM CIKM*, 2002.

[17] S. K. Das, K. Shuster, C. Wu. ACQUIRE: agent-based complex query and information retrieval engine. *AAMAS* 2002: 631-638.

[18] S.K. Das, and M.H. Williams. Extending integrity maintenance capability in deductive databases. In the proceedings of the *UK ALP-90 Conference* (Bristol, England, January), Oxford, England: Intellect, pp.75-111, 1990.

[19] A. Elmagarmid, M. Rusinkiewicz, and A. Sheth. Management of heterogeneous and autonomous database systems. *Morgan Kaufmann*, Boston, London, 1999.

[20] R.A. Elmasri and S.B. Navathe, S.B. Fundamentals of Database Systems. *Addison-Wesley*, 2003, 4th edition.

[21] M. Fernandez and J. Siméon.Growing XQuery. *European Conference on Object Oriented Programming (ECOOP)*, 2003.

[22] G. Gardarin, A. Mensch, T. Tuyet, and D.L. Smit. Integrating Heterogeneous Data Sources with XML and XQuery. Proceedings of the 13th *International Workshop on Database and Expert Systems Applications*, 2002.

[23] G.Glass. Overview of Voyager: ObjectSpace's Product Family for State-of-the-Art Distributed Computing. Technical report, ObjectSpace, 1999.

[24] P. Grefen, and P. Apers. Integrity Control in Relational Database Systems - An Overview, *Journal of Data and Knowledge Engineering*, 10 (2), 187-223, 1993

[25] P. Grefen, and J, Widom. Protocols for integrity Constraint Checking in Federated Databases. *International Journal of Distributed and Parallel Databases*, 5(4): 327-355, October 1997

[26] S. Grufman, F. Samson, S.M. Embury, P.M.D. Gray, and T. Risch. Distributing Semantic Constraints between Heterogeneous Databases. Proceedings of the

Thirteenth *International Conference on Data Engineering, ICDE* 1997, April 7-11, pages 33-42, Birmingham U.K., April 1997

[27] A. Gupta, Y. Sagiv, J.D. Ullman, and J. Widom. Constraint Checking with Partial Information. Proceedings of the *Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems,* pages 45-55, Minneapolis, Minnesota, May 1994.

[28] A. Gupta and J. Widom. Local Verification of Global Integrity Constraints in Distributed Databases. Proceedings of the *ACM SIGMOD International Conference on Management of Data*, pages 49-58, Washington, D.C., May 1993

[29] C. G. Harrison, D.M. Chessm, A.kershenbaum. Mobile Agents: Are they a good idea? *Research Report, IBM Research Division*, 1994.

[30] H. Ibrahim. A Strategy for Semantic Integrity Checking in Distributed Databases. Proceedings of the ninth *International Conference on Parallel and Distributed Systems, ICPADS* 2002, pages 139-144

[31] B. Kane, H. Su, and E. A. Rundensteiner, Consistently Updating XML Documents using Incremental Constraint Check Queries. *Workshop on Web Information and Data Management (WIDM'02)*, Nov. 2002. page 1-8,2002.

[32] G. Karjoth, D. Lange, and M. Oshima A Security Model for Aglets. *IEEE Internet Computing*, Vol. 1, No. 4, July-August 1997.

[33] R. Koblick. Concordia. *Communications of the ACM*, 42(3): 96-99, March 1999.

[34] C. T. Kwok and D. S. Weld . Planning to gather information. *Technical Report UW-CSE-96-01-04.Department of Computer Science, University of Washington*,Seattle, WA, 1996.

[35] D.B. Lange and M. Oshima. Mobile Agents with Java: The Aglet API. *World Wide Web* 1(3): 111-121 (1998).

[36] A. Laux and L. Martin. XUpdate Working Draft, 2000, last accessed on August 20, 2004 from http://xmldb-org.sourceforge.net/xupdate/xupdate-wd.html

[38] P. Madiraju, S. K. Prasad, R. Sunderraman et al. An Agent Module for a System of Mobile Devices, Proceedings of the 3rd *International Workshop on Agents and Peer-to-Peer Computing(AP2PC 2004)* in conjucntion with Third International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS 2004),New York, July 2004.

[39] P. Madiraju and R. Sunderraman. Mobile Agent Approach for Global Database Constraint Checking. Proceedings of *ACM Symposium on Applied Computing* (SAC'04), Nicosia, Cyprus, 2004, pp. 679-683.

[40] P.Madiraju and R. Sunderraman. An Efficient Constraint Planning Algorithm for Multidatabases. Accepted in 2005 *ACS/IEEE International Conference on Computer Systems and Applications* (AICCSA-05)

[41] P. Madiraju, R. Sunderraman, and S.B. Navathe. Semantic Integrity Constraint Checking for Multiple XML Databases. Proceedings of 14th *Workshop on Information Technology and Systems* (WITS 2004), Washington D.C., December, 2004

[42] Praveen Madiraju, Rajshekhar Sunderraman, Shamkant B. Navathe and Haibin Wang. Semantic Integrity Constraint Checking for Multiple XML Databases, *Journal of Database Management* (under second revision)

[43] F. Ozcan, S. Nural, P. Koksal, C. Evrendilek, A. Dogac: Dynamic Query Optimization in Multidatabases. Bulletin of the *IEEE Computer Society Technical Committee on Data Engineering*, 1997.

[44] S. K. Prasad, A. G. Bourgeois, E. Dogdu, R. Sunderraman, Y. Pan, and S. Navathe. 2003. Implementation of a Calendar Application Based on SyD Co-ordination Links, Proceedings of The Third *International Workshop on Internet Computing and E-Commerce* in conjunction with the 17th Annual International Par-allel & Distributed Processing Symposium (IPDPS 2003), 22-26 April, Nice, France

[45] S. K. Prasad, A. G. Bourgeois, E. Dogdu, R. Sunderraman, Y. Pan, S. Navathe and V. Madisetti. 2003. Enforcing Interdependencies and Executing Transactions Atomically Over Autonomous Mobile Data Stores Using SyD Link Technology, Proceedings of *Mobile Wireless Network Workshop* held in conjunction with The 23rd International Conference on Distributed Computing Systems (ICDCS'03), May 19-22, Providence, Rhode Island.

[46] S. K. Prasad, V. Madisetti, S. B. Navathe, R. Sunderraman, E. Dogdu, A. Bourgeois, M. Weeks, B. Liu, J.Balasooriya, A.Hariharan, W. Xie, P. Madiraju, S. Malladi, R. Sivakumar, A. Zelikovsky, Y. Zhang, Y. Pan, and S. Belkasim. SyD: A Middleware Testbed for Collaborative Applications over Small

Heterogeneous Devices and Data Stores, Procs. ACM/IFIP/USENIX, 5th *International Middleware Conference*, Toronto, Ontario, Canada, October 18th - 22nd, 2004.

[47] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational Databases for Querying XML documents: Limitations and Opportunities. Proceedings of the *International Conference on Very Large Databases*, 1999.

[48] A.P. Sheth and J. A. Larson. Federated database systems for managing distributed, heterogeneous,and autonomous databases. *ACM Computing Surveys*, 22(3):183--236, Sept. 1990.

[49] R. Sunderraman, E. Dogdu, P. Madiraju, L. Malladi. "A Java API for Global Querying and Updates for a System of Databases", Proceedings of 43rd *ACM South East Conference*, Georgia, March, 2005.

[50] I. Tatarinov, Z. G. Ives, A.Y. Halevy and S. Daniel. Updating XML. Proceedings of the *ACM SIGMOD Conference on Management of Data*, 2001.

[51] C. Türker, and M. Gertz. Semantic integrity support in SQL: 1999 and commercial (object) relational database management systems. *VLDB Journal* 10(4): 241-269 (2001).

[52] J. Widom, and S. Ceri. Active Database Systems: Triggers and Rules for Advanced Database Processing. *Morgan Kaufmann*, San Francisco, California, 1996.