Security Constraint Processing in a Multilevel Secure Distributed Database Management System

Bhavani Thuraisingham, Member, IEEE Computer Society, and William Ford, Senior Member, IEEE

Abstract—In a multilevel secure distributed database management system, users cleared at different security levels access and share a distributed database consisting of data at different sensitivity levels. An approach to assigning sensitivity levels, also called security levels, to data is one which utilizes constraints or classification rules. Security constraints provide an effective classification policy. They can be used to assign security levels to the data based on content, context, and time. In this paper we extend our previous work on security constraint processing in a centralized multilevel secure database management system by describing techniques for processing security constraints in a distributed environment during query, update, and database design operations.

Index Terms—Multilevel secure distributed database management system, security constraints, inference problem, security policy, distributed query processing, distributed update processing, multilevel distributed database design.

I. INTRODUCTION

In a multilevel secure database management system (MLS/DBMS), users cleared at different security levels access and share a database consisting of data at different sensitivity levels. An approach to assigning sensitivity levels, also called security levels, to data is one which utilizes security constraints or classification rules. Security constraints provide an effective classification policy. They can be used to assign security levels to the data based on content, context, and time. Such constraints are useful for describing multilevel applications.

Handling security constraints in multilevel database systems has received some attention during the past five years. An early attempt to identify various types of security constraints was made in [6] and [7]. Later, in an article published in this journal [17], we described the design of an MLS/DBMS in which security constraint processing was fundamental. The work reported in [19], [12], and [20] suggests ways of handling security constraints during query processing in such a way that certain security violations via inference do not occur. The work reported in [10] and [16] focuses on handling constraints during database design where suggestions for database design tools are given. In [17], issues on processing constraints during database updates are given. While the previous papers discuss how security constraints may be handled, they do not provide detailed approaches or algorithms for actually processing them. Unlike these previous papers published in security constraint processing, our recent work described in [8], [5], and [21] describes design and implementation techniques for actually processing security constraints in a centralized environment. We believe that processing security constraints is the first step towards controlling unauthorized inferences in an MLS/DBMS.

In this paper, we extend our previous work on constraint processing in a centralized environment by describing techniques for processing them in a multilevel secure distributed database management system (MLS/DDDBMS) during query, update, and database design operations. Since more and more C4I applications are becoming distributed, security constraints need to be enforced in a multilevel distributed environment. We first define various types of security constraints. We then describe our overall approach to processing security constraints. In this approach, security constraints are handled during query processing, during database updates, as well as during database design. We also describe an integrated architecture which shows the interactions between the query constraint processor, the update constraint processor and the database design tool. Finally we describe in detail the issues involved in distributed constraint processing. In particular, we describe the design of a 1) distributed constraint processor which functions during query processing, 2) distributed constraint processor which functions during update processing, and 3) distributed constraint processor which functions during database design.

The organization of this report is as follows. In Section II, we discuss issues on security constraint processing. In Section III, we discuss constraint processing during the query operation in a distributed environment. In Section IV, we discuss constraint processing during the update operation in a distributed environment. In Section V, we discuss constraint processing during distributed database design. The paper is concluded in Section VI.

II. ISSUES IN SECURITY CONSTRAINT PROCESSING

In Section II.A, we define various types of security con-

1. This paper, by an MLS/DBMS we mean a multilevel secure relational database management system.
2. Inference is the process of drawing conclusion from premises. It becomes a problem if the conclusions drawn are unauthorized.
3. The query constraint processor is the module that is responsible for processing constraints during the query operation. Similarly, the update constraint processor is responsible for processing constraints during the update operation.
A. Security Constraints

Security constraints are rules which assign security levels to the data. We have defined various types of security constraints. They include the following:

1) Constraints that classify a database, relation, or an attribute. These constraints are called simple constraints.

2) Constraints that classify any part of the database depending on the value of some data. These constraints are called content-based constraints.

3) Constraints that classify any part of the database depending on the occurrence of some real-world event. These constraints are called event-based constraints.

4) Constraints that classify associations between data (such as tuples, attributes, elements, etc.). These constraints are called association-based constraints.

5) Constraints that classify any part of the database depending on the information that has been previously released. These constraints are called release-based constraints. We have identified two types of release-based constraints. One is the general release constraint which classifies an entire attribute depending on whether any value of another attribute has been released. The other is the individual release constraint which classifies a value of an attribute depending on whether a value of another attribute has been released.

6) Constraints that classify collections of data. These constraints are called aggregate constraints.

7) Constraints that classify any part of the database depending on the security level of some data. These constraints are called level-based constraints.

8) Constraints which specify implications. These are called logical constraints.

9) Constraint which classify constraints and metadata. These are called meta-constraints.

We will give examples of constraints belonging to each category. In our examples, we assume that the database consists of two relations, TANKS and ASSIGNMENT. TANKS has attributes T# (Tank number), TYPE, COMMANDER, and A# (Assignment Number) with T# as the primary key. ASSIGNMENT has attributes A# (Assignment Number), MISSION, and DESTINATION with A# as the primary key. Note that TANKS.A# is a foreign key. The constraints may be expressed as some form of logical rules.

Simple constraints:

Format: Level(R.Aj) = L
Explanation: The values of the attribute Aj of the relation are classified at level L.
Example: Level(TANKS.COMMANDER) = Secret.

Content-based constraints:

Format: COND(Value(S1.B1, S2.B2, ... Sm.Bm)) -> Level(R.Aj) = L
Explanation: If the values of the attributes B1, B2, ... Bm of relation S1, S2,... Sm, respectively satisfy some boolean condition, COND, then the values of the attribute Aj of relation R are classified at level L.
Example: Value(TANKS.TYPE) = Sherman -> Level(TANKS.COMMANDER) = Secret.

Association-based constraints (also called context or together constraints):

Format: Level(Together(R1.A1, R2.A2, ... Rn.An)) = L
Explanation: The association between the values of the attributes A1, A2, ... An of relations R1, R2, ... Rn, respectively, is classified at level L. Note that such an association is possible only if the relations R1, R2, ... Rn can be joined via some common attributes. Our specification does not enable the join conditions to be stated explicitly. Any formal specification language should capture the join conditions.
Example: Level(Together(TANKS.TYPE, TANKS.COMMANDER)) = Secret.

Event-based constraints:

Format: Event(E) -> Level(R.Aj) = L
Explanation: Once an event E has occurred, the values of the attribute Aj of relation R are classified at level L.
Example: Event(Change of President) -> Level(TANKS.COMMANDER) = Secret.

General release-based constraints:

Format: Release(R.Ai, L1) -> Level(S.Aj) = L2
Explanation: If a value of the attribute Ai of relation R is released at level L1, then all of the values of the attribute Aj of relation S are classified at level L2.
Example: Release(TANKS.TYPE, Unclassified) -> Level(TANKS.COMMANDER) = Secret.

Individual release-based constraints:

Format: Individual-Release(R.Ai, L1) -> Level(S.Aj) = L2
Explanation: If a value of the attribute Ai of relation R is released at level L1, then the corresponding value of attribute Aj of relation S which is obtained via a join operation if S is different from R) is classified at level L2. (Note that our specification does not capture the join condition.)

Aggregate constraints:

Format: Set(S, R) AND Satisfy(S, P) -> Level(S) = L
Explanation: This means that if R is a relation, S is a set containing tuples of R, and S satisfies some property P, then S is classified at level L. Note that P could be any property such as "number of elements is greater than 10." In other words, aggregate constraints classify collections of tuples taken together at a level higher than the individual levels of the tuples in the collection. There could be some semantic association between the tuples.

Level-based constraints:

Format: Level(R.Ai) = L1 -> Level(S.Aj) = L2
Explanation: If the values of an attribute Ai of relation R are...
classified at level $L_1$ via a simple constraint, then the values of the attribute $A_j$ of relation $S$ must be classified at level $L_2$.

Example: \[ \text{Level(TANKS.TYPE)} = \text{Secret} \rightarrow \text{Level(TANKS.COMMANDER)} = \text{Top Secret}. \]

**Logical constraints:**

Format: $R.A_i \rightarrow S.A_j$

Explanation: The value of an attribute $A_i$ of relation $R$ implies the value of attribute $A_j$ of relation $S$. That is, logical constraints are rules which are used to derive new data from existing data. The derived data could be classified using one of the other constraints. Note that logical constraints are not really security constraints. That is, they do not assign security levels to the data. They are in fact integrity constraints.

Example: \[ \text{ASSIGNMENT.MISSION} \rightarrow \text{ASSIGNMENT.DESTINATION}. \]

**Meta-constraints:**

These are constraints which classify constraints and metadata. We have utilized only one such constraint which is stated below. "If all of the relations and attributes specified in a constraint are visible at level $L$, then the constraint itself must be classified at a level $L^* \leq L$."

For example, consider the constraint \( TANKS.TYPE = \text{Secret} \). If the names TANKS and TYPE are visible at the Unclassified level (such as the specification of the relation TANKS is Unclassified), then the constraint must also be Unclassified.

**B. Integrated Architecture**

As stated in Section I, security constraints enforce a classification policy. Therefore, it is essential that constraints are manipulated only by an authorized individual. In our approach, constraints are maintained by the system security officer (SSO). That is, constraints are protected from ordinary users. We assume that constraints themselves could be classified at different security levels. However they are stored at system-high. The constraint manager, which is trusted, will ensure that a user can read the constraints classified only at or below his level.4

From an analysis of the various types of security constraints, we believe that they are a form of integrity constraints enforced in an MLS/DBMS. This is because in a multilevel database, one can regard the security level of an entity to be part of the value of that entity. Therefore, security constraints specify permissible values that an entity can take. Since security constraints can be regarded as a form of integrity constraints, many of the techniques developed for handling integrity constraints in nonmultilevel relational database systems by the logic programming researchers (see, for example, [9]) could be used for handling security constraints in an MLS/DBMS. In these techniques, some integrity constraints, which are called derivation rules, are handled during query processing; some integrity constraints, known as integrity rules, are handled during database updates; and some integrity constraints, known as schema rules, are handled during database design. Our approach to handling security constraints has been influenced by (1) the design of Lock Data Views [17], a high assurance MLS/DBMS, in which constraint processing is fundamental and (2) the approach taken to process integrity constraints by the logic programming researchers [13].

We have designed an integrated approach to constraint processing. In this approach, certain constraints are handled during query processing, certain constraints during database updates, and certain constraints during database design. The first step was to decide whether a particular constraint should be processed during the query, update, or database design operation. After some consideration, we felt that it was important for the query constraint processor to have the ability to handle all of the security constraints. Our thesis is that inferences can be most effectively handled and thus prevented during query processing. This is because most users usually build their reservoir of knowledge from responses that they receive by querying the database. It is from this reservoir of knowledge that they infer unauthorized information. Moreover, no matter how securely the database has been designed, users could eventually violate security by inference because they are continuously updating their reservoir of knowledge as the world evolves. It is not feasible to have to redesign the database simultaneously.

The next step was to decide which of the security constraints should be handled during database updates. After some consideration, we felt that except for some types of constraints, such as the release and aggregate constraints, the others could be processed during the update operation. However, techniques for handling constraints during database updates could be quite complex, as the security levels of the data already in the database could be affected by the data being updated. Therefore, initially our algorithms handle only the simple and content-based constraints during database updates.

The constraints that seemed appropriate to handle during the database design operation were those that classified an attribute or collections of attributes taken together. These include the simple and association-based constraints. For example, association-based constraints classify the relationships between attributes. Such relationships are specified by the schema and therefore such constraints could be handled when the schema is specified. Since a logical constraint is a rule which specifies the implication of an attribute from a set of attributes, it can also be handled during database design.

Note that some constraints can be handled in more than one way. For example, we have the facility to handle the content-based constraints during query processing as well as during database updates. However, it may not be necessary to handle a constraint in more than one place. For example, if the content-based constraints are satisfied during the database update operation, then it may not be necessary to examine them during query processing also. Furthermore, the query operation is performed more frequently than the update operation. Therefore, it is important to minimize the operations performed by the query constraint processor as much as possible to improve

---

4. We assume that a trusted process is a process whose functions are security critical (that is, enforces part of the security policy). Therefore it must be ensured that it operates correctly. The techniques used to ensure its correctness depend on the level of assurance that is expected of the system.
performance. However, there must be a way to handle all of the constraints during query processing. This is because if the real-world is dynamic, then the database data may not satisfy all of the constraints that are enforced as integrity rules, or the schema may not satisfy the constraints that are processed during database design. This means that there must be a trigger which informs the query constraint processor that the multilevel database or the schema is not consistent with the real-world, in which case the query constraint processor can examine the additional constraints.

Below, we briefly illustrate the architectures for processing constraints during the query, update, and database design operations. The architecture for query processing is shown in Fig. 1. This architecture can be regarded as a loose coupling between a multilevel relational database management system and a deductive manager. The deductive manager is what we have called the query constraint processor. It has to operate on-line.

The architecture for update processing is shown in Fig. 2. This architecture can be regarded as a loose coupling between a multilevel relational database management system and a deductive manager. The deductive manager is what we have called the update constraint processor. It could be used on-line where the security levels of the data are determined during database inserts and updates, or it could be used offline as a tool which ensures that data entered via bulk data loads and bulk data updates are accurately labeled. If the tool is used offline, however, it may be difficult to recompute the levels of the data already in the database if these levels are affected by the new data that are being inserted.

It should also be noted that there are two aspects to constraint handling. One is constraint generation and the other is constraint enforcement. Constraint generation is involved with analyzing the specification of a multilevel application and generating an initial set of schema and constraints. This is typically performed by an Applications Specialist, possibly together with the SSO. Constraint enforcement, on the other hand, is involved with techniques for enforcing the constraints that are produced by the constraint generation task. While our research has focussed on constraint enforcement, the relationship between the two tasks is illustrated in Fig. 4. That is, the constraint generator takes the specification of the multilevel application and outputs the initial schema and the constraints that must be enforced. The database design tool takes this output as its input and designs the database. The constraints and schema produced by the database design tool are used by the update constraint processor and the query constraint processor.

The tool which handles security constraints during database design, illustrated in Fig. 3, can be used by the SSO to design the schema. The input to the tool is the set of security constraints that should be handled during database design and the schema. The output of the tool is the modified schema and the constraints. We envisage such a tool to be operated off-line.

Although the query constraint processor, update constraint processor, and database design tool are separate modules, they all constitute the solution to constraint processing in MLS/DBMSs. That is, these three approaches provide an integrated solution to security constraint processing in a multilevel environment. Fig. 5 illustrates the integrated architecture. In this architecture, the constraints and schema which are pro-
duced by the constraint generator are processed further by the database design tool. The modified constraints are given to the Constraint Updater in order to update the constraint database. The schema is given to the MLSDBMS to be stored in the metadatabase. The constraints in the constraint database are used by the query and update constraint processors. We assume that there is a trusted constraint manager process which manages the constraints. In a dynamic environment where the data and the constraints are changing, the query constraint processor will examine all the relevant constraints and ensure that users do not obtain unauthorized data.

C. Extending to a Distributed Environment

We have extended the integrated architecture described in Section II.B to a multilevel distributed environment. The details of constraints processing in an MLS/DDDBMS will be discussed in Sections III, IV, and V. In this section, we describe the architecture for an MLS/DDDBMS. In an MLS/DDDBMS, users cleared at different security levels access and share a distributed database consisting of data at different security levels without violating security. The system architecture for an MLS/DDDBMS that we considered in our investigation is shown in Fig. 6. In our architecture, the MLS/DDDBMS consists of several nodes that are interconnected by a trusted network. We assume that the nodes are homogeneous. That is, all of the nodes are designed identically. Each node is capable of handling multilevel data. Each node has an MLS/DDBMS which manages the local multilevel database. Each node also has a distributed processing component called the Secure Distributed Processor (SDP). The components of SDP, shown in Fig. 7, are the Distributed Query/Update Processor (DQUP), Distributed Transaction Manager (DTM), Distributed Metadata Manager (DMM), and the Distributed Constraint Processor (DCP). The DQUP consists of two components: the Distributed Query Processor (DQP), which is responsible for distributed query processing, and the Distributed Update Processor (DUP), which is responsible for handling distributed updates. The DTM is responsible for distributed transaction management. The DMM manages the global metadata. The global metadata include information on the schemas which describe the relations in the distributed database, the way the relations are fragmented, the locations of the fragments, and the constraints enforced. The DCP, whose detailed design is the subject of this paper, is responsible for handling security constraints during query and update operations.

Fig. 4. Constraint generation and enforcement.

Fig. 5. Integrated architecture.

There are some outstanding issues on constraint processing. These include consistency and completeness of constraints. Some issues on the consistency and completeness of security constraints are described in [1]. Since our constraints could be specified as horn clauses, various techniques for ensuring the completeness and consistency of horn clause programs that have been developed (see, for example, [13]) could possibly be utilized for security constraints specified as horn clauses also. While research needs to be done, the work of logic programmers shows some promise on ensuring the completeness as well as verifying the consistency of security constraints expressed as horn clauses.

Fig. 6. System architecture for an MLS/DDDBMS.

5. For a discussion on trusted networks, we refer the reader to [23]. A trusted network is also called a multilevel secure network.
6. A transaction is a program unit that must be executed in its entirety or not executed at all. This paper does not address transaction processing.
7. DCP could also be extended to handle integrity constraints. Also, since a transaction consists of a sequence of queries and update requests, security constraints need to be processed during transaction execution also. These issues will be the subject of our future research.
The DQ/UP, DTM, and DCP communicate with the DMM for the metadata required to carry out their functions. The DCP also communicates with the DQ/UP and the DTM as it processes security constraints during query, update, and transaction execution. Since a transaction is a series of query and update requests, we assume that the DQ/UP is invoked by the DTM in order to carry out the individual requests. SDP may be implemented as a set of processes separate from the local MLSDBMS. Two DMMs (DQ/UPs, DTM, DCPs) at different nodes communicate in accordance with the security policy enforced.

The security policy of an MLS/DDBMS depends on the security policy enforced by the local operating system at each node, the local MLSDBMS, the SDP, and the network. In this paper we have focussed mainly on mandatory security policy issues. We assume that a user at level L can read any data in the distributed database classified at or below the level L. A user at level L can update the data in the distributed database classified at level L. Furthermore, the system must ensure that two DMMs (DQ/UPs, DTM, DCPs) at different nodes can communicate with each other only if they both operate at the same level. More details of the issues in MLS/DDBMSs are given in [22].

![Secure distributed processor](image)

**III. DISTRIBUTED CONSTRAINT PROCESSING DURING QUERY OPERATION**

In this section, we describe distributed constraint processing during the query operation. We discuss the architecture in Section III.A. Details are described in Section III.B. Operation of an MLS/DDBMS during query processing is given in Section III.C.

**A. Architecture**

Fig. 8 shows an architecture for distributed constraint processing during the query operation. At each site, we have augmented an MLS/DBMS with a Distributed Query Processor (DQP). The DQP at each site is augmented with a Distributed Query Constraint Processor (DQCP). As stated earlier, DQCP is part of the DCP. The security constraints, relevant release information, and real-world knowledge are stored in the knowledge base. The knowledge as well as the data are distributed across the nodes. For example, each node may store only the constraints that are enforced on relations that are stored at that node. Information on knowledge and data distribution is stored in the global metadatabase.

The DQCP processes the security constraints in such a way during query processing that certain security violations cannot occur. Two techniques implemented by the DQCP are query modification and response processing. In the query modification technique, the security constraints are examined by the DQCP and the query is modified in such a way that the response to the modified query will not result in any security violation in general. The query that is modified is passed to the DQP for execution. The response to the modified query is assembled by the DQD and passed to the DQCP. In certain cases, the DQCP does some response processing. That is, it examines the constraints and some of the responses that have been released previously and sanitizes the response to the current query if necessary.

Note that the MLS/DBMS provides the basic mandatory protection. That is, it ensures that a subject can read only the data (e.g., relation, tuple, element, etc.) stored at a level which it dominates. Also, the operating system will ensure that a subject can read files classified at or below its level. Each piece of data is assigned a security level (via the DUCP or otherwise) and is stored at that level. The security constraints are examined by the DQCP and the query is modified in such a way that the response to the modified query will not result in any security violation in general. The query that is modified is passed to the DQP for execution. The response to the modified query is assembled by the DQD and passed to the DQCP. In certain cases, the DQCP does some response processing. That is, it examines the constraints and some of the responses that have been released previously and sanitizes the response to the current query if necessary.

The DQCP makes use of the DQP to provide distributed query processing services. We assume that there is a DQP process per security level. A DQP process executing at level L services requests of users at level L. The system must ensure that two DQPs at different nodes communicate only if they both operate at the same level. A user’s request at level L will be processed entirely at level L. As a result, information which
is not dominated by level \( L \) will not be used during the processing. Whether parts of the DQP should or should not be trusted depends on the overall security policy enforced. For example, if trusted labels are not a requirement, then we envisage the DQP to be untrusted. If not, we envisage parts of the DQP which process response received from an MLS/DBMS to be trusted. Since the focus of this paper is on constraint processing, we do not give the details of the DQP. For a discussion, we refer to [15].

Fig. 8. An architecture for distributed constraint processing during query operation.

B. Details of Constraint Processing

In Section III.B.1, we describe a security policy for the distributed constraint processing during the query operation. In Section III.B.2, we describe the functionality of the DQCP. In Section III.B.3 we describe the modules of the DQCP. Some examples are given in Section III.B.4.

B.1. Security Policy

A security policy for the distributed constraint processing that we propose extends the simple security property in [2] to handle inference violations. This policy is stated below:

1) Given a security level \( L, E(L) \) is the knowledge base associated with \( L \). That is, \( E(L) \) will consist of all responses that have been released at security level \( L \) over a certain time period and the real world information at security level \( L \).

2) Let a user \( U \) at security level \( L \) pose a query. Then the response \( R \) to the query will be released to this user if the following condition is satisfied:

   \[
   \text{For all security levels } L^* \text{ where } L^* \text{ dominates } L, \text{ if } (E(L^*) \cup R) = X \text{ (for any } X) \text{ then } L^* \text{ dominates Level}(X). 
   \]

   Where \( A \Rightarrow B \) means \( B \) can be inferred from \( A \) using any inference strategy and Level(\( X \)) is the security level of \( X \).

We assume that any response that is released into a knowledge

8. Such a policy was first proposed in the LDV design [11].

9. In this paper, we have considered only the logical inference strategy.

The policy states that whenever a response is released to a user at level \( L \), it must be ensured that any user at level \( L^* \geq L \) cannot infer information classified at a level \( L^* > L \) from the response together with the knowledge that he has already acquired. This is in general not desirable, because activities of higher level users would then interfere with the response that must be released to a lower level user. The only solution to this problem is to ensure that such a situation does not occur (i.e., constraints are enforced in such a way that higher level constraints or information released to higher level users do not impact the processing of a lower level query) or to ignore the actions of higher level users completely. Note that while we consider only hierarchical levels in specifying the policy, it can be extended to include non-hierarchical levels also.

B.2. Functionality of the DQCP

In this section, we discuss the techniques that we have used to implement the security policy. They are query modification and response processing. Each technique is described below. As stated earlier, we have considered only logical inferences.

Query Modification: Query modification technique has been used in the past to handle discretionary security and views [18]. This technique has been extended to include mandatory security in [7]. In our design of DQCP, this technique is to modify the query depending on the security constraints, the relevant previous responses released, and real world information. The modified query is then evaluated. The security constraints, the relevant previous responses released, and any real-world information are stored in the knowledge base.

Consider a database which consists of relations TANKS and ASSIGNMENT where the attributes of TANKS are \( \text{T\#} \), TYPE, COMMANDER and \( \text{A\#} \) with \( \text{T\#} \) as the key; and the attributes of ASSIGNMENT are \( \text{A\#, MISSION} \) and DESTINATION with \( \text{A\#} \) as the key. Let the knowledge base consist of the following rules:

1) \( \text{TANKS(X, Y, Z, D) and Release(Z, Unclassified)} \rightarrow \text{Level}(Y, Secret) \)

2) \( \text{TANKS(X, Y, Z, A) and Release(Z, Unclassified)} \rightarrow \text{Level}(Y, Secret) \)

3) \( \text{TANKS(X, Y, Z, A) and Release(Z, Unclassified)} \rightarrow \text{Level}(Y, Secret) \)

4) \( \text{TANKS(X, Y, Z, A)} \rightarrow \text{Level}(Y, Secret) \)

5) \( \text{TANKS(X, Y, Z, A)} \rightarrow \text{Level}(Y, Secret) \)

6) \( \text{NOT(Release(X, Secret)} \rightarrow \text{Level}(X, Top Secret)) \rightarrow \text{Level}(X, Unclassified) \)

The first rule is a content-based constraint which classifies a tank type whose commander is Smith at the Secret level. Similarly, the second rule is also a content-based constraint which classifies a tank type whose assignment number is 10 at the Top Secret level. The third rule is an association-based constraint which classifies tank types and commanders taken together at the Secret level. The fourth and fifth rules are additional restrictions that are enforced as a result of the context-based constraint specified in rule 3. The sixth rule states that the default classification level of a data item is Unclassified.
Suppose an Unclassified user requests the tank types in TANKS. This query is represented as follows:

\[ \text{TANKS}(X, Y, Z, A) \]

Since a tank commander is classified at the Secret level if either the captain is "Smith" or the captain Commander is already released at the Unclassified level, and it is classified at the Top Secret level if the assignment is "10," assuming that the captain Commanders are not yet released to an Unclassified user, the query is modified to the following:

\[ \text{TANKS}(X, Y, Z, D) \text{ and } Z \neq \text{Smith and } A \neq 10. \]

Note that since query modification is performed in real-time, it will have some impact on the performance of the query processing algorithm. However, several techniques for semantic query optimization have been proposed recently for intelligent query processing in a nonsecure environment (see, for example, [4]). These techniques could be adapted for query processing in a multilevel environment in order to improve the performance.

**Response Processing:** For many applications, in addition to query modification, some further processing of the response such as response sanitization, may need to be performed. We will illustrate this point with examples.

**Example:** Consider the following release constraints discussed earlier. That is,

1) all tank types whose corresponding commander names are already released to Unclassified users are Secret and

2) all commander names whose corresponding tank types are already released to Unclassified users are Secret.

Suppose an Unclassified user requests the tank types first. Depending on the other constraints enforced, let us assume that only certain tank types are released to the user. Then the tank types released have to be recorded into the knowledge base. Later, suppose an Unclassified user (does not necessarily have to be the same one) asks for commander names. The commander name values (some or all) are then assembled in the response. Before the response is released, the tank types that are already released to the Unclassified user need to be examined. Then the commander name value which corresponds to a tank type value that is already released is suppressed from the response. Note that there has to be a way of correlating the tank types with the commanders. This means the primary key values (which is the T#) should also be retrieved with the commander names. In addition, the tank numbers must be stored with the tank types in the knowledge base.

**Example:** Consider the following aggregate constraint:

A collection of 10 or more tuples in the relation TANKS is Secret.

Suppose an Unclassified user requests the tuples in TANKS. The response is assembled and then examined to see if it has more than 10 tuples. If so, it is suppressed. There are some problems associated with maintaining the release information. As more and more relevant release information gets inserted, the knowledge base could grow at a rapid rate. Therefore, efficient techniques for processing the knowledge base need to be developed. This would also have an impact on the performance of the query processing algorithms. Therefore, one solution would be to include only certain crucial release information in the knowledge base. The rest of the information can be stored with the audit data which can then be used by the SSO for analysis.

**B.3. Modules of the DQCP**

Fig. 9 illustrates the components of the DQCP and the knowledge base. The modules of DQCP are the User Interface Manager (UIM), the Query Modifier (QM), the Response Processor (RP), the Release Database Manager (RDM), and the Constraint Manager (CM). Knowledge base consists of the constraint database and the release database. The constraint database consists of the security constraints and real-world constraints. The release database consists of relevant information previously released.

Note that the MLS/DBMS provides the basic mandatory protection. That is, it ensures that a subject can read only the data (e.g., relation, tuple, element, etc.) stored at a level which they dominate. Also, the operating system will ensure that a subject can read files classified at or below its level. Each piece of data is assigned a security level (via the DUCP or otherwise) and is stored at that level. The DQCP provides protection when the constraints have changed in the real world. The major issue is whether DQCP should be trusted since it enforces security policy extensions. In our approach, we trusted those components of the DQCP which access data and knowledge stored at a level higher than the user's level. The trust placed on the other components of the DQCP depends on the security policy enforced. For example, any query that is passed to the MLS/DBMS must be via the DQP. Furthermore, any response generated by the MLS/DBMS is passed to the DQCP via the DQP. Therefore, if the DQP is not trusted, then even if the DQCP is trusted, a Trojan horse in the DQP could release sensitive data. However, if the DQCP is untrusted, then constraint processing during the query operation will be of little use. What we must ensure is that not only are there no security vulnerabilities by introducing the DQCP, its function is also useful. Next we discuss the functions of each module of the DQCP and the trust that needs to be placed.

**User Interface Manager:** This module accepts a user's query and checks for syntax. It parses the query and gives it to QM. If we rely on the operating system's authentication mechanism, then the query interface component of the UIM need not be trusted. UIM functions at the user's level. UIM is also responsible for releasing the response to the user. If trusted labels is a requirement, then parts of the response interface component of the UIM may have to be trusted. It should, however, be noted that many of the MLS/DBMS designs do not provide trusted labels.

**Query Modifier:** This module takes the parsed request from the UIM. It then sends a message to CM to retrieve the relevant constraints for the query. If we assume that only the ap-

10. As stated earlier, if we are not concerned with the knowledge acquired by higher level users, then the DQCP does not have to access information stored at a higher level.
plicable constraints classified at or below the user’s level are returned, then not trusting the QM will not introduce security vulnerabilities. However, if the QM is untrusted, then it cannot be guaranteed to apply the constraints correctly. If QM does not function correctly, then its existence will be of little use. Therefore, we envisage QM to be a trusted process if it is to be useful. Note that if higher level constraints are applied to the query, then it may be possible for users to get some idea about these higher level constraints from the responses received. Although our current design assumes that higher level constraints may be returned to the QM for a particular query, we envisage that constraints could be designed in such a way that it may not be necessary to have applicable constraints classified at a level higher than the user’s level. The QM modifies the query according to the constraints. Also, if there are any general release constraints applicable, then QM sends a message to the RDM to 1) check whether any of the relevant attributes have been released previously and/or 2) record that certain attributes will be released. The second action (i.e., recording that certain attributes will be released) is necessary only because DQCP is functioning in a distributed environment, and it is necessary to prevent different users from posing queries from different sites and assembling the responses and deducing unauthorized information. Once the query is modified, the modified query is given to the DQP operating at the user’s level.

Constraint Manager: Constraint Manager manages the security constraints. Since constraints enforce the classification policy, the constraints can only be updated by someone authorized, such as the SSO. We assume CM acts on behalf of the SSO and therefore it is trusted and operates at system high. Note that the constraints are stored at system high but could have security labels attached to them to reflect their sensitivity. Since the only process that manipulates the constraints is the CM, the security labels cannot be tampered with. CM communicates both with QM and RP. It also communicates with remote CMs to retrieve relevant constraints. For our initial design we have assumed that the constraints are replicated at each node.

Response Processor: The response assembled by the DQP is given to RP. RP sends a message to CM to retrieve the relevant release constraints. It then sends a message to RDM to retrieve the relevant information that has been released. It then sanitizes the response accordingly. The sanitized response is then given to the user via UIM. If certain release constraints apply, then RP must also send the information to be released to RDM to update the release database. Since the response was given to the RP by the DQP, if the DQP is not trusted, then even if RP is trusted, a Trojan horse in the DQP could release sensitive information to the user. If RP is not trusted, then the correct application of security constraints cannot be guaranteed. In any case, if the function of RP is to be useful, it must be trusted. Although in our present design it is possible for higher level release constraints to be applicable for a particular query, as stated earlier, we envisage that constraints could be designed in such a way that it is not necessary to have applicable constraints classified at a level higher than the user’s level.

Release Database Manager: When a request arrives from QM or RP, RDM examines the release database to determine which information is to be passed. The RDMs at different nodes have to communicate in order to obtain release data. Since QM is trusted, if RDM does not pass correct information to the QM, then the function of the QM will be of little use. In certain cases, RDM will have to examine the release data stored at higher levels. For example, if the following constraint “once the name of a mission is released to an Unclassified or Confidential user then its location cannot be released to an Unclassified or Confidential user” is enforced, and an Unclassified user requests locations, it is checked both at the Confidential and Unclassified levels to see whether the corresponding names have been released. This means that the RDM must be trusted. It should be noted that if the mission names have been released at the Confidential level and if the Unclassified users have knowledge of the constraints, then they will know that the mission names have been released at the Confidential level. We feel that it will be difficult to design constraints in such a way that a problem can be avoided. When RP gives release information to RDM, it updates the release database at the appropriate level.

B.4. Operation of the DQCP

We will describe constraint processing with some examples involving release constraints. We consider release constraints as they are more difficult to handle.

Example 1: Suppose there is a general release constraint which classifies all tank types at the Secret level if commander names have been released. An Unclassified user at site 1 could pose a query to retrieve T# and COMMANDER pairs from TANKS and at the same time, an Unclassified user at site 2 could pose a query to retrieve T# and TYPE pairs. If the user at site 1 gets the commander names and the user at site 2 gets the tank types, the system has been compromised. Next we will describe how the DQCP will handle such a situation. Suppose an Unclassified user requests for commander names at site 1. UIM parses the query and sends it to QM. QM sends a mes-
sage to CM to retrieve relevant constraints. CM will examine the constraint database, communicating with remote CMs if necessary, and return the general release constraint to QM. QM sends a message RDM to first record that commander names have been released and to check whether tank types have been released. RDM first records in temporary storage that commander names have been released and then it may communicate with remote RDMs, if necessary, to get the release information on tank types. If tank types have been released previously, then the query is not processed further. RDM will delete the recording that commander names have been released. Suppose tank types have not been released previously to an Unclassified user. Then the query is given by CM to DQP. The response assembled by DQP is sent to RP. In this case, RP does not have to perform any sanitization. It gives the response to UIM. RDM then deletes the recording in temporary storage and records in permanent storage that commander names are released.

Suppose an Unclassified user at site 2 queries for tank types at the same time an Unclassified user at site 1 queries for commander names. At site 2, the first step will be to record in temporary storage that tank types have been released. Then the DQCP at site 2 will send a message to the DQCP at each site querying whether commander names have been released. Note that if it has already been recorded in either temporary or permanent storage at site 1 that names have been released to an Unclassified user, then the tank types will not be released at site 2. However, at site 1, if it is not yet recorded that names are released, then tank types are released at site 2. Names can now never be released at site 1. This is because eventually at site 1, it will be recorded in temporary storage that names have been released. Then, a message will be sent to the DQCP at site 2 requesting whether tanks have been released. Either in temporary or permanent storage, it will already be recorded at site 2 that types have been released. Therefore names cannot be released. Note however that if it is recorded in temporary storage at site 1 that names have been released and it is recorded about the same time in temporary storage at site 2 that types have been released, then there could be a possibility that neither the names nor the types are released. Although this places an unnecessary restriction, there is no security violation.

**Example 2:** Suppose there is an individual release constraint which classifies a tank type at the Secret level if the corresponding commander name has been released. Let us assume that an Unclassified user has posed a query at site 1 to retrieve certain commander names and that the names have been released to him. The release database at site 1, if it is recorded in temporary storage at site 1 that names have been released and it is recorded about the same time in temporary storage at site 2 that types have been released, then there could be a possibility that neither the names nor the types are released. Although this places an unnecessary restriction, there is no security violation.

**C. Operation of the MLS/DDBMS**

In this section, we illustrate the entire processing of the system during the query operation with an example. Let the database consist of two relations, TANKS and GROUPS. The attributes of TANKS are Number, Commander, Type, Date, and Assignment. Its primary key is Number. The attributes of GROUPS are Number, Location, Mission, and Mission Code. Its primary key is Number. We assume that TANKS.Assignment and GROUPS.Number take values from the same domain. Also, TANKS.Assignment is a foreign key. The database is populated as shown below. We assume that the number 1 denotes the Unclassified level, the number 10 denotes the Secret level, and the number 16 denotes the Top Secret level. We assume that both TANKS and GROUPS are assigned level 1. Furthermore, all of the tuples are also stored at level 1.

**Relation TANKS at Site 1**

<table>
<thead>
<tr>
<th>No.</th>
<th>Commander</th>
<th>Type</th>
<th>Date</th>
<th>Assign</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>A10001</td>
<td>Capt. Johnson</td>
<td>Abrams</td>
<td>May 75</td>
<td>003</td>
<td>1</td>
</tr>
<tr>
<td>A10002</td>
<td>Maj. Hanson</td>
<td>Abrams</td>
<td>Sep 68</td>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>A10004</td>
<td>2nd. Lt. Smith</td>
<td>Leopard</td>
<td>Feb 43</td>
<td>003</td>
<td>1</td>
</tr>
<tr>
<td>B10001</td>
<td>Capt. Julian</td>
<td>Centurion</td>
<td>Jan 83</td>
<td>005</td>
<td>1</td>
</tr>
<tr>
<td>C20002</td>
<td>Capt. Brown</td>
<td>Centurion</td>
<td>Sep 75</td>
<td>006</td>
<td>1</td>
</tr>
</tbody>
</table>

**Relation TANKS at Site 2**

<table>
<thead>
<tr>
<th>No.</th>
<th>Commander</th>
<th>Type</th>
<th>Date</th>
<th>Assign</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>F10004</td>
<td>Sgt. Heavy</td>
<td>Patton</td>
<td>May 76</td>
<td>003</td>
<td>1</td>
</tr>
<tr>
<td>G10001</td>
<td>Capt. Avenger</td>
<td>Aventer</td>
<td>Sep 87</td>
<td>003</td>
<td>1</td>
</tr>
<tr>
<td>G10002</td>
<td>Sgt. Stone</td>
<td>Leopard</td>
<td>Jun 69</td>
<td>003</td>
<td>1</td>
</tr>
<tr>
<td>G10003</td>
<td>Sgt. Rock</td>
<td>Patton</td>
<td>Dec 63</td>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>G10004</td>
<td>Capt. Suribachi</td>
<td>Abrams</td>
<td>Nov 59</td>
<td>009</td>
<td>1</td>
</tr>
<tr>
<td>A10006</td>
<td>Lt. NiMo</td>
<td>Abrams</td>
<td>May 59</td>
<td>005</td>
<td>1</td>
</tr>
<tr>
<td>A10007</td>
<td>Maj. Muller</td>
<td>Centurion</td>
<td>Jan 81</td>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>B10002</td>
<td>Sgt. Wilson</td>
<td>Leopard</td>
<td>May 83</td>
<td>006</td>
<td>1</td>
</tr>
<tr>
<td>B10003</td>
<td>Lt. Swanson</td>
<td>Sherman</td>
<td>Nov 61</td>
<td>009</td>
<td>1</td>
</tr>
<tr>
<td>D10007</td>
<td>2nd. Lt. Connor</td>
<td>Sherman</td>
<td>Sep 34</td>
<td>001</td>
<td>1</td>
</tr>
</tbody>
</table>

**Relation GROUPS at Site 1**

<table>
<thead>
<tr>
<th>Number Location</th>
<th>Mission</th>
<th>Code</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>North Germany</td>
<td>Covering Action</td>
<td>001</td>
</tr>
<tr>
<td>002</td>
<td>Central Germany</td>
<td>Soviet Patrol</td>
<td>002</td>
</tr>
<tr>
<td>003</td>
<td>N. Saudi Arabia</td>
<td>Iraq Crisis</td>
<td>006</td>
</tr>
<tr>
<td>004</td>
<td>Philippines</td>
<td>Stabilize Government</td>
<td>005</td>
</tr>
</tbody>
</table>

**Relation GROUPS at Site 2**

<table>
<thead>
<tr>
<th>Number Location</th>
<th>Mission</th>
<th>Code</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>005</td>
<td>Oman</td>
<td>Iraq Crisis</td>
<td>004</td>
</tr>
<tr>
<td>006</td>
<td>Oman</td>
<td>Training Exercises</td>
<td>004</td>
</tr>
<tr>
<td>007</td>
<td>Panama</td>
<td>Soviet Reconnaissance</td>
<td>003</td>
</tr>
<tr>
<td>008</td>
<td>Panama</td>
<td>Training Exercises</td>
<td>003</td>
</tr>
<tr>
<td>009</td>
<td>Japan</td>
<td>Korean Reinforcement</td>
<td>001</td>
</tr>
</tbody>
</table>
Constraints Enforced:

Logical Constraint:
GROUPS.MISSION => GROUPS.LOCATION

Content Constraints:
GROUPS.LOCATION = Japan =>
Level(GROUPS.LOCATION) = 16
TANKS.TYPE = Sherman => TANKS.TYPE = 10
TANKS.TYPE = Centurion => TANKS.TYPE = 16

Each constraint is classified at level 1
All of the constraints are replicated both at
site 1 and site 2

Query:
Select TANKS.TYPE, GROUPS.MISSION from TANKS,
GROUPS where TANKS.ASSIGNMENT = GROUPS.NUMBER

Query Site: 1; User’s Level: 1

Query Processing: The query is given to the DQCP
at site 1. DQCP will examine the constraints
(which we assume are stored at site 1) and
will modify the query to the following:
Select TANKS.Type, GROUPS.Mission from TANKS,
GROUPS where TANKS.Assignment = GROUPS.Number and
TANKS.Type ≠ Sherman and
TANKS.Type ≠ Centurion and
GROUPS.Location ≠ Japan

DQCP gives the query to DQP at site 1. DQP at
site 1 decomposes the query into the follow-
ing subqueries:
1) Select TANKS.Assignment, TANKS.Type from
TANKS where TANKS.Type ≠ Centurion and
TANKS.Type ≠ Sherman
2) Select GROUPS.Number, GROUPS.Mission from
GROUPS where GROUPS.Location ≠ Japan

DQP at site 1 sends both subqueries to the
MLS/DBMS at site 1 and the DQP at site 2. DQP
at site 2 sends both subqueries received to
the MLS/DBMS at site 2. The MLS/DBMSs at both
sites will evaluate the two subqueries.

Let TANKS-1 and TANKS-2 be the result of execut-
ing the first subquery at sites 1 and 2 re-
spectively. Let GROUP-1 and GROUPS-2 be the
result of executing the second subquery at
sites 1 and 2 respectively. TANKS-1, TANKS-2,
GROUPS-1, GROUPS-2 are as follows (note that
duplicate values are not eliminated):

Fragment TANKS-1

<table>
<thead>
<tr>
<th>Type</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abrams</td>
<td>003</td>
</tr>
<tr>
<td>Abrams</td>
<td>001</td>
</tr>
<tr>
<td>Leopard</td>
<td>003</td>
</tr>
<tr>
<td>Abrams</td>
<td>003</td>
</tr>
<tr>
<td>Abrams</td>
<td>003</td>
</tr>
<tr>
<td>Leopard</td>
<td>009</td>
</tr>
</tbody>
</table>

Fragment TANKS-2

<table>
<thead>
<tr>
<th>Type</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Patton</td>
<td>003</td>
</tr>
<tr>
<td>Avenger</td>
<td>003</td>
</tr>
<tr>
<td>Leopard</td>
<td>003</td>
</tr>
<tr>
<td>Patton</td>
<td>001</td>
</tr>
<tr>
<td>Abrams</td>
<td>009</td>
</tr>
<tr>
<td>Abrams</td>
<td>005</td>
</tr>
<tr>
<td>Leopard</td>
<td>006</td>
</tr>
</tbody>
</table>

Fragment GROUPS-1

<table>
<thead>
<tr>
<th>Number</th>
<th>Mission</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>Covering Action</td>
</tr>
<tr>
<td>002</td>
<td>Scout Patrol</td>
</tr>
<tr>
<td>003</td>
<td>Iraq Crisis</td>
</tr>
<tr>
<td>004</td>
<td>Stabilize Govern</td>
</tr>
</tbody>
</table>

Fragment GROUPS-2

<table>
<thead>
<tr>
<th>Number</th>
<th>Mission</th>
</tr>
</thead>
<tbody>
<tr>
<td>005</td>
<td>Iraq Crisis</td>
</tr>
<tr>
<td>006</td>
<td>Training Exercises</td>
</tr>
<tr>
<td>007</td>
<td>Soviet Reconnaissance</td>
</tr>
<tr>
<td>008</td>
<td>Training Exercises</td>
</tr>
</tbody>
</table>

The next step will be for the DQP at site 2 to send the frag-
ments TANKS-2 and GROUPS-2 to the DQP at site 1. The
DQP at site 1 will merge the fragments TANKS-1 and
TANKS-2 to produce TANKS*. The fragments GROUPS-1
and GROUPS-2 will be merged to obtain the fragment
GROUPS*. The fragments TANKS* and GROUPS* are as
follows:

Fragment TANKS*

<table>
<thead>
<tr>
<th>Type</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abrams</td>
<td>003</td>
</tr>
<tr>
<td>Abrams</td>
<td>001</td>
</tr>
<tr>
<td>Leopard</td>
<td>003</td>
</tr>
<tr>
<td>Abrams</td>
<td>003</td>
</tr>
<tr>
<td>Abrams</td>
<td>003</td>
</tr>
<tr>
<td>Leopard</td>
<td>009</td>
</tr>
</tbody>
</table>

Fragment GROUPS*

<table>
<thead>
<tr>
<th>Number</th>
<th>Mission</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>Covering Action</td>
</tr>
<tr>
<td>002</td>
<td>Scout Patrol</td>
</tr>
<tr>
<td>003</td>
<td>Iraq Crisis</td>
</tr>
<tr>
<td>004</td>
<td>Stabilize Govern</td>
</tr>
<tr>
<td>005</td>
<td>Iraq Crisis</td>
</tr>
<tr>
<td>006</td>
<td>Training Exercises</td>
</tr>
</tbody>
</table>
007 Soviet Reconnaissance
008 Training Exercises

The final step will be to join TANKS* and GROUPS* where TANKS*.Assignment = GROUPS*.Number. The resulting fragment is then projected on the attributes, Type and Mission. The response, which we will call RESULT, is as follows:

RESULT

<table>
<thead>
<tr>
<th>Type</th>
<th>Mission</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abrams</td>
<td>Iraq Crisis</td>
</tr>
<tr>
<td>Abrams</td>
<td>Covering Action</td>
</tr>
<tr>
<td>Leopard</td>
<td>Iraq Crisis</td>
</tr>
<tr>
<td>Abrams</td>
<td>Iraq Crisis</td>
</tr>
<tr>
<td>Patton</td>
<td>Iraq Crisis</td>
</tr>
<tr>
<td>Avenger</td>
<td>Iraq Crisis</td>
</tr>
<tr>
<td>Leopard</td>
<td>Iraq Crisis</td>
</tr>
<tr>
<td>Patton</td>
<td>Covering Action</td>
</tr>
<tr>
<td>Abrams</td>
<td>Iraq Crisis</td>
</tr>
<tr>
<td>Leopard</td>
<td>Training Exercises</td>
</tr>
</tbody>
</table>

Note that several tuples are duplicated in the result. For example, the pair (Abrams, Iraq Crisis) appear several times in the response. This shows that the tank type Abrams was used in the mission Iraq Crisis and furthermore Iraq Crisis has occurred several times. If necessary, the duplication could be eliminated. Therefore, the final response RESULT* will be as follows:

RESULT*

<table>
<thead>
<tr>
<th>Type</th>
<th>Mission</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abrams</td>
<td>Iraq Crisis</td>
</tr>
<tr>
<td>Abrams</td>
<td>Covering Action</td>
</tr>
<tr>
<td>Leopard</td>
<td>Iraq Crisis</td>
</tr>
<tr>
<td>Patton</td>
<td>Iraq Crisis</td>
</tr>
<tr>
<td>Avenger</td>
<td>Iraq Crisis</td>
</tr>
<tr>
<td>Patton</td>
<td>Covering Action</td>
</tr>
<tr>
<td>Abrams</td>
<td>Iraq Crisis</td>
</tr>
<tr>
<td>Leopard</td>
<td>Training Exercises</td>
</tr>
</tbody>
</table>

IV. DISTRIBUTED CONSTRAINT PROCESSING DURING THE UPDATE OPERATION

In Section IV.A, we describe the architecture for constraint processing during the update operation. The details are discussed in Section IV.B. The operation of the MLS/DDBMS during update processing is discussed in Section IV.C.

A. Architecture

Fig. 10 shows an architecture for distributed constraint processing during the update operation. At each site, we have augmented an MLS/DDBMS with a Distributed Update Processor (DUP). The DUP at each site is augmented with a Distributed Update Constraint Processor (DUCP). That is, the DUCP is the user of the DUP. Note that DUCP is part of the DCP discussed in Section II.C. The security constraints are stored in the knowledge base. The knowledge as well as the data are distributed across the nodes. For example, each node may store only the constraints that are enforced on relations that are stored at that node. Information on knowledge and data distribution is stored in the global metadatabase. The DUCP processes the security constraints in such a way during update processing so that security levels are computed not only for the data being updated, but also for any other data affected by the update. The DUP makes use of the DUP to update the tuples at the appropriate levels. For example, if the DUCP determines that a tuple should be inserted at level L, then it gives that tuple to the DUP operating at level L. The DUP is a user to the local MLS/DDBMS. We assume that the update security policy of each local MLS/DDBMS is write-at-your-level. Therefore, the DUP operating at level L updates tuples at level L.

The DUP is also responsible for handling database updates in a distributed environment. In particular, it checks the locations of the various fragments and determines the best update execution strategy. For example, if a relation is fragmented across several sites, and if a user requests to insert a tuple, the tuple is usually inserted into a fragment which has the least number of tuples provided the transmission cost is not high. That is, the site selected for the update is determined by the number of tuples of the fragments stored at that site and the cost to transmit the insert request to that site.

B. Details of Constraint Processing

In Section IV.B.1, we describe the security policy implemented by the DUCP. The functionality and utilization of the DUCP is described in Section IV.B.2. The modules of the DCUP is described in Section IV.B.3. The operation of the DCUP is described in Section IV.B.4. The utility of the DUCP is discussed in Section IV.B.5. We assume that the execution is serial. That is, concurrent updates are not considered in this paper.

B.1. Security Policy

The security policy of the DUCP is the following:

1) All users are granted a maximum clearance level. A user...
may log in at any level that is dominated by his maximum clearance level. Subjects act on behalf of users at the user’s login security level.

2) Objects are the rows, tables, and databases, and every object is assigned a security level upon creation.

3) A subject has read access to an object if the security level of the object dominates the security level of the subject.

4) A subject has write access to an object if the security level of the object dominates the security level of the subject.

B.2. Functionality of the DUCP

Theoretically, a DUCP can handle all of the constraints that we have specified. However, the techniques used to implement such a DUCP would be extremely complex. This is because the security level to be assigned to data could depend not only on the data values but also on the data already in the database, data to be entered in the future, data already released to the user, and real-world data. Furthermore, data inserted could very well change the security level of data already in the database. While the techniques developed for integrity constraint checking show much promise, more research needs to be done before techniques for processing all of the constraints during database updates can be developed.

In our initial design, the DUCP utilizes simple and content-dependent security constraints as guidance in determining the security level of the data being updated. The use of security constraints can thereby protect against users incorrectly labeling data as a result of logging in at the wrong level, against data being incorrectly labeled when imported from systems of different modes of operation such as a system high, and against database inconsistencies as a consequence of the security label of data in the database being affected by data being entered into the database.

The security level of an update request is determined by the DUCP as follows. The simple and content-dependent security constraints associated with the relation being updated and with a security label greater than the user login security level are retrieved and examined for applicability. If multiple constraints apply, the security level is determined by the constraint that specifies the highest classification level. If no constraints apply, the update level is the login security level of the user. The Update Processor, therefore, does not determine the security level of the data solely from the security constraints, but utilizes the constraints as guidance in determining the level of the input data. The following examples illustrate the functionality of the Update Processor.

Consider a database that consists of a relation, TANKS, whose attributes are number, commander, type, date, and assignment, with number as its primary key. The content-based constraint which classifies all tanks with type Sherman as Secret is expressed as: TANKS.type = "Sherman" -> Level(TANKS.all) = Secret. A user at login security level Confidential enters the following data to insert a tuple into the TANKS relation: Insert TANKS values ("A10001"; "Johnson"; "Abrams"; "Feb 84"; "008").

The Update Processor will receive this insert and retrieve the constraints associated with the TANKS relation which specify a level greater than the user level, which is Confidential, and whose level is less than or equal to the user level. The content-based constraint stated above is retrieved. Since the data entered for the type field are not "Sherman," the security constraint associated with the TANKS relation will not affect the classification level of the insert, and the Update Processor will determine the insert level to be the user level, which is Confidential.

Suppose a user at login security level Confidential then enters the following: Insert TANKS values ("B10002"; "Smith"; "Sherman"; "Mar 89"; "009"). The Update Processor will again retrieve the content-based constraint associated with the TANKS relation, which specifies a level greater than the user level and whose level is less than or equal to the user level. Since the data for the type field are "Sherman" the Update Processor will determine the insert level to be Secret. If, however, the user entered this insert at login security level Top Secret, the Update Processor would perform the insert at the user level since the user level is higher than the level specified by the security constraint.

The update operation of the DUCP functions similarly to the insert operation. As an example, suppose a user at the Confidential level enters the following: Update TANKS set type = "Sherman" where commander = "Johnson." The DUCP will retrieve the security constraints associated with the TANKS relation which specify a level greater than the user level and whose level is less than or equal to the user level. The content-dependent constraint stated above will be retrieved, and the DUCP will determine the update level to be Secret since the type field is being modified to "Sherman." The tuple with a primary key of "A10001" as defined above will then be updated at the Secret level, and the original tuple will be deleted.

B.3. Architecture

Fig. 11 illustrates the components of the DUCP and the knowledge base. The modules of the DUCP are the User Interface Manager (UIM), the Security Level Computer (SLC), the Level Upgrader (LU), and the Constraint Manager (CM). Knowledge base is the constraint database. The constraint database consists of the security constraints and real-world constraints. We discuss the functions of each module and the trust that needs to be placed.

User Interface Manager: This module accepts a user’s update request and checks for syntax. It parses the request and gives it to the Security Level Computer. If we rely on the operating system’s authentication mechanism, the UIM need not be trusted. The UIM functions at the user’s level.

Security Level Computer: This module takes the parsed request from the UIM. It then communicates with the Constraint Manager to obtain the relevant constraints and computes the security level of the update request. In order to do this, it may have to query the database (via the distributed query processor) to obtain data already in the database. This is because 1) the security level of the data to be updated may depend on the data already in the database and/or 2) the security level of the data already in the database may be affected by the data to be
updated. Once the SCL determines the correct security levels of the data, it passes this information to the Level Upgrader. Since SLC performs a security critical function, it must be trusted. (Note that if the computed security level is not dominated by the user's level, then the data are simply entered at the user's level or the request is ignored.)

**Level Upgrader:** For each (old-data, old-level, new-data, new-level) tuple that LU receives from the SLC, it creates processes (which are versions of the DUP) at the old-level and new-level (could be the same process if the levels are the same) and gives the corresponding data to these processes. The DUP process which receives old-data deletes these data from the database, and the DUP process which receives the new data inserts the data into the database. (We assume that the security policy of each local MLS/DBMS is read-at-or-below-your-level and write-at-your-level.) LU must be a trusted process.

![Diagram](image)

**Fig. 11. Modules of DUCP.**

**Constraint Manager:** This module manages the security constraints (note that CM could be the same process for the DQCP and the DUCP). Since constraints enforce the classification policy, the constraint can only be updated by someone authorized such as the SSO. CM could act on behalf of the SSO, in which case CM is trusted and operates at system high.

**B.4. Operation of the DUCP**

In this section we describe the operation of the DUCP with an example. Let the database consist of two relations, TANKS and GROUPS. The attributes of TANKS are Number, Commander, Type, Date, and Assignment. Its primary key is Number. The attributes of GROUPS are Number, Location, Mission, and Mission-Code. Its primary key is Number. We assume that TANKS.Assignment and GROUPS.Number take values from the same domain. Also, TANKS.Assignment is a foreign key. Let the following content-based constraint be enforced: “TANKS.Type = Secret if the corresponding GROUPS.Location = N. Saudi Arabia.” Suppose an Unclassified user requests to insert the tuple (A10001, Johnson, Sherman, Feb ’90, 003).

The first step would be for the SLC modules at the request site to compute the level of the request. It will communicate with CM to retrieve the relevant constraints. If the constraints are not replicated, then CM communicates with the remote CM to retrieve the relevant constraints. In this example, the security level of the tuple depends on the tuple which may or may not be present in the relation GROUPS. If a tuple with GROUPS.Number = 003 is not in the relation GROUPS, then it is not possible to compute the level of the tuple to be inserted. In this case, one of the following actions could be taken:

1) the tuple is inserted at the user's level,
2) the tuple is not inserted at all,
3) the tuple is stored in temporary storage until the tuple with GROUPS.Number = 003 is inserted into GROUPS, or
4) the user is requested to enter a tuple in GROUPS with number = 003.

If there is already a tuple in GROUPS with number 003, then actions to be taken will be determined by the level at which the tuple in GROUPS is stored. For example, if the tuple in GROUPS is stored at a level higher than the Unclassified level, then such a tuple has no impact on the tuple to be inserted. That is, as far as the Unclassified user is concerned, that tuple does not exist. If the tuple in GROUPS is stored at the Unclassified level, then it is checked to see whether the location is N. Saudi Arabia. If so, the tuple (A10001, Johnson, Sherman, Feb ’90, 003) must be inserted at the Secret level. If the location is N. Saudi Arabia, then the tuple is inserted at the Unclassified level. Note that if the relation TANKS is not stored at the request site, then the tuple is sent to the appropriate remote site via the DUPS. On the other hand, if the relation TANKS is replicated at more than one site, then the request is sent to the DUPS at the appropriate sites by the DUP at the request site. Even if the relation TANKS is stored at the request site, if TANKS is fragmented and stored at multiple sites, then DUP may perform load balancing and insert the tuple at a site which is not congested. Also, to retrieve data from GROUPS, the DUCP communicates with the DQP. If the relation GROUPS is not stored at the request site, or if the tuple in GROUPS with number 003 is not at the request site, the data have to be retrieved from a remote site via the DQPs. Note that whenever the DUP enters a tuple at a particular level L, if there is already a tuple at that level, then the tuple is not entered. That is, we do not permit two different tuples with the same primary key to exist at the same level.

**B.5. Utility of the DUCP**

An MLS/DBMS provides assurance that all objects in a database have a security level associated with them and that users are allowed to access only the data to which they are cleared. Additionally, it provides a mechanism for entering
multilevel data but relies on the user to login at the level at which the data are to be entered. The DUCP will provide a mechanism that can operate as a stand-alone tool with an MLS/DBMS to provide assurance that data are accurately labeled as they are entered into the database. This could significantly enhance and simplify the ability of an MLS/DBMS to assure that data entered via bulk data loads and bulk data updates are accurately labeled.

Another significant use for a DUCP is in operation with a DQCP which functions during query processing. The DQCP protects against certain security violations via inference that can occur when users issue multiple requests and consequently infer unauthorized knowledge. The DQCP also utilizes security constraints as its mechanism for determining the security level of data. The security constraints are used as derivation rules as they are applied to the data during query processing. Addressing all of the security constraint types mentioned above could add a significant burden to the DQCP particularly if the number of constraints is high. To enhance the performance of the DQCP, the DUCP can be utilized to add certain constraint types as data are entered into the database, in particular, simple and content-based constraints, alleviating the need for the DQCP to handle these constraint types. We assume that the security constraints remain relatively static, as reliance on the DUCP to ensure that data in the database remain consistent would be difficult, particularly in a volatile environment where the constraints change dynamically. An additional concern is that database updates could leave the database in an inconsistent state. The DUCP, however, is designed to reject updates that cause a rippling effect and thus leave the database in an inconsistent state.

C. Operation of the MLS/DBMS

In this section we illustrate the entire processing of the system during the update operation with an example. Let the database consist of two relations, TANKS and GROUPS. The attributes of TANKS are Number, Commander, Type, Date, and Assignment. Its primary key is Number. The attributes of GROUPS are Number, Location, Mission, and Code. Its primary key is Number. We assume that TANKS.Assignment and GROUPS.Number take values from the same domain. Also, TANKS.Assignment is a foreign key.

Constraint Enforced:
Content Constraint: GROUPS.LOCATION = N. Saudi Arabia

The database is populated as shown below. We assume that the number 1 denotes the Unclassified level, the number 10 denotes the Secret level, and the number 16 denotes the Top Secret level. We assume that both TANKS and GROUPS are assigned level 1. Note the content-based constraint that is enforced is satisfied by the data in the distributed database.

<table>
<thead>
<tr>
<th>Relation TANKS at Site 1</th>
<th>No.</th>
<th>Commander</th>
<th>Type</th>
<th>Date</th>
<th>Assign.</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Di0005</td>
<td>Lt. Col. Hamilton</td>
<td>Abrams</td>
<td>Feb 67</td>
<td>003</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>A10005</td>
<td>2nd Lt. Swanson</td>
<td>Sherman</td>
<td>Apr 69</td>
<td>001</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Relation TANKS at Site 2</th>
<th>No.</th>
<th>Commander</th>
<th>Type</th>
<th>Date</th>
<th>Assign.</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Di0001</td>
<td>Cpt. Avenger</td>
<td>Avenger</td>
<td>Sep 87</td>
<td>003</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>A10003</td>
<td>Sgt. Rock</td>
<td>Patton</td>
<td>Dec 63</td>
<td>001</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>B10002</td>
<td>Sgt. Wilson</td>
<td>Leopard</td>
<td>May 83</td>
<td>006</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Relation GROUPS at Site 1</th>
<th>No.</th>
<th>Location</th>
<th>Mission</th>
<th>Code</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>North Germany</td>
<td>Covering Action</td>
<td>001</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>002</td>
<td>Central Germany</td>
<td>Scout Patrol</td>
<td>002</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>003</td>
<td>N. Saudi Arabia</td>
<td>Iraq Crisis</td>
<td>006</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Relation GROUPS at Site 2</th>
<th>No.</th>
<th>Location</th>
<th>Mission</th>
<th>Code</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>005</td>
<td>Oman</td>
<td>Iraq Crisis</td>
<td>004</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>006</td>
<td>Oman</td>
<td>Training Exercises</td>
<td>004</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>007</td>
<td>Panama</td>
<td>Soviet Reconnaissance</td>
<td>003</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>008</td>
<td>Panama</td>
<td>Training Exercises</td>
<td>003</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Update Request 1: Insert (F10005, Cpt. Johnson, Sherman, Feb 90, 006)
User's Level: 1; Request Site: 1
The DUCP communicates with the DQP at site 1 to retrieve the tuple in GROUPS with number = 008. Since the tuple is at site 2, the DQP at site 1 retrieves this tuple via the DQP at site 2. Since the location field of this tuple is not N. Saudi Arabia, the level of the insert is computed to be 1. The DUCP passes the request to DUP at site 1 operating at level 1. Although the DUP at site 1 could pass this request to the local MLS/DBMS, since TANKS at site 2 has fewer tuples, it will send the request to the DUP at site 2. The DUP at site 2 will insert the tuple at level 1. The only relation updated is TANKS at site 2. This relation is shown below.

<table>
<thead>
<tr>
<th>Relation TANKS at Site 2 After Update Request 1 is Processed</th>
<th>No.</th>
<th>Commander</th>
<th>Type</th>
<th>Date</th>
<th>Assign.</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Di0001</td>
<td>Cpt. Avenger</td>
<td>Avenger</td>
<td>Sep 87</td>
<td>003</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>A10003</td>
<td>Sgt. Rock</td>
<td>Patton</td>
<td>Dec 63</td>
<td>001</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>B10002</td>
<td>Sgt. Wilson</td>
<td>Leopard</td>
<td>May 83</td>
<td>006</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>F10005</td>
<td>Cpt. Johnson</td>
<td>Sherman</td>
<td>Feb 90</td>
<td>006</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Update Request 2: Insert (G10007, Cpt. Smith, Abrams, Feb 91, 003)
User's Level: 10; Request Site: 1
The DUCP communicates with the DQP at site 1 to retrieve the tuple in GROUPS with number = 008. Since the tuple is at site 1, the DQP at site 1 retrieves this tuple via the DQP at site 2. Since the location field of this tuple is N. Saudi Arabia, the level of the insert is computed to be 16. The DUCP passes the request to DUP at site 1 operating at level 16. Although the DUP at site 1 could pass this request to the local MLS/DBMS, since TANKS at site 2 has fewer tuples, it will send the request to the DUP at site 2. The DUP at site 2 will insert the tuple at level 16. The only relation updated is
TANKS at site 2. This relation is shown below.

Relation TANKS at Site 2 After Update Request 2 is Processed

<table>
<thead>
<tr>
<th>No.</th>
<th>Commander</th>
<th>Type</th>
<th>Date</th>
<th>Attr.</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>GI0001</td>
<td>Cpt. Avenger</td>
<td>Average</td>
<td>Sep 87</td>
<td>003</td>
<td>16</td>
</tr>
<tr>
<td>GI0003</td>
<td>Sgt. Rock</td>
<td>Patton</td>
<td>Dec 63</td>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>B1002</td>
<td>Spc. Wilson</td>
<td>Leopard</td>
<td>May 93</td>
<td>006</td>
<td>1</td>
</tr>
<tr>
<td>F1008</td>
<td>Cpt. Johnson</td>
<td>Sherman</td>
<td>Feb 90</td>
<td>006</td>
<td>1</td>
</tr>
<tr>
<td>G10007</td>
<td>Cpt. Smith</td>
<td>Abrams</td>
<td>Feb 91</td>
<td>003</td>
<td>16</td>
</tr>
</tbody>
</table>

Update Request 3:
Insert (H10001, Capt. James, Abrams, Mar 89, 009)

User's Level: 1; Request Site: 1

The DUQP communicates with the DQP at site 1 to retrieve the tuple in GROUPS with number = 009. Since the tuple is not at site 1, the DQP at site 1 sends a message to the DQP at site 2 to retrieve the tuple. The tuple is not at site 2 either. Therefore, the level of the tuple to be inserted cannot be computed. As mentioned earlier,

1) the tuple could be inserted at the user's level, in which case referential integrity is violated;
2) the tuple is stored in temporary storage until a tuple in GROUPS is entered at level 1 with number = 009;
3) the tuple is rejected;
4) the user is requested to enter a tuple in GROUPS where number = 009.

We recommend either option 3 or 4 for the following reasons: option 1 would result in referential integrity violation, and with option 2, the tuple can never be inserted if the corresponding tuple in GROUPS is not inserted. Assuming that we take option 3, in this example, the database is not modified after request 3 is processed.

V. DISTRIBUTED CONSTRAINT PROCESSING DURING DATABASE DESIGN OPERATION

In our approach, we first design the multilevel database as if it were a centralized system and then allocate the fragments to the various sites. In Section V.A we discuss how constraints may be processed during the design of the global schema, and in Section V.B we discuss how the schema gets fragmented and allocated to the various nodes.

A. Constraint Processing During Database Design

A.1. Overview

In our approach we first consider the distributed database as a single data store and generate the global schema by processing certain security constraints. Once the global schema is generated, we use techniques that have been developed for nonmultilevel database systems to design the allocation and fragmentation schema.

As described in Section II, after examining the various types of constraints, we feel that those that classify attributes or collections of attributes without any dependency on the data values could be handled during database design. Therefore among the constraints that we have considered, the association-based constraints, the simple constraints, and logical constraints could be handled during database design.

An association-based constraint classifies a collection of attributes taken together at a particular security level. What is interesting about the association-based constraint is that it can generate several relationships between the various attributes. For example, if there is a relation TANKS whose attributes are T#, TYPE, and MISSION, and if an association-based constraint classifies the TYPE and MISSION taken together at the Secret level, then one of the pairs (T#, TYPE), (T#, MISSION) should also be classified at the Secret level. Otherwise, an Unclassified user can obtain the (T#, TYPE) and the (T#, MISSION) pairs and infer the Secret association (TYPE, MISSION). There has been much discussion in the literature as to the appropriate place to handle these association-based constraints. Some argue that they should be handled during database design [14] while others argue that they should be handled during query and update processing [17]. However, none of the work reported so far studied the properties of these association-based constraints, nor has it provided any technique to generate the additional association-based constraints that can be deduced from an initial set of association-based constraints.

We first describe an algorithm which processes a given set of association-based constraints and outputs the schema for the multilevel database. Given a set of association-based constraints and an initial schema, the algorithm will output clusters of attributes and the security level of each cluster. We then prove that the attributes within a cluster can be stored securely at the corresponding level. A tool based on this algorithm can help the SSO design the multilevel database. The algorithm that we have designed does not necessarily have to be executed during database design only. It can also be executed during query processing. That is, the query processor can examine the attributes in the various clusters generated by the algorithm and then determine which information has to be released to the users. For example, if the algorithm places the attributes A1, A2 in cluster 1 at level L, and the attributes A3, A4 in cluster 2 at level L, then after an attribute in cluster 1 has been released to a user at level L, none of the attributes in cluster 2 can be released to users at level L.

Since simple constraints can be regarded as a special form of association-based constraints, where only one attribute is classified, we feel that such constraints could also be handled during database design. Another constraint that could be handled during database design is the logical constraint. For example, if attribute A implies an attribute B, and if attribute B is classified at the Secret level, then attribute A must be classified at least at the Secret level. It should be noted that if any of the constraints have conditions attached to them, then handling them during database design time would be difficult. For example, consider the following constraint: "Type and Mission taken together are Secret if location is a Middle East country". Such a constraint depends on data values. Therefore, they are best handled during either query or update processing.

In Section V.A.2, we describe an algorithm which determines the security levels of the attributes given a set of asso-
A.2. Handling Association-based Constraints

In this section we describe an algorithm for handling association-based constraints. The input to this algorithm is a set of association-based constraints and a set of attributes. The output of this algorithm is a set of clusters for each security level. Each cluster for a security level L will have a collection of attributes that can be safely classified at the level L. That is, if A1, A2, and A4 are attributes in a cluster C at level Secret, then the attributes A1, A2, and A3 can be classified together safely at the security level Secret without violating security. The clusters are formed depending on the association-based constraints which are input to the program. Once the clusters are formed, then the database can be defined according to the functional and multivalued dependencies that are enforced.

Algorithm HABC (Handling Association-Based Constraints)

Begin

Let C be the set of security constraints and W1, W2, ... Wm be the set of attributes which are input to the program.

For each security level L, do the following:

Begin

Let C[L] be the largest subset of C and A = (A1, A2, ... An) be the largest subset of (W1, W2, ... Wm), such that the elements of subset of C and A are all visible at level L. Since n is the number of attributes which are visible at level L, clusters C1, C2, ... Cn will be formed as follows:

Set C1 = C2 = C3 = ... = Cn = Empty-set.

For each i (1 ≤ i ≤ n) do the following:

Begin

Find the first cluster Cj (1 ≤ j ≤ n) such that Aj, together with any of the attributes already in Cj, is classified at a level dominated by L by the set of constraints C[L].

Place Aj in the cluster Cj. (Note that since we have defined n clusters, there will definitely be one such Cj.)

End (for each i).

Output all the non-empty clusters along with the security level L.

End (for each security level L).

End (HABC)

We now trace the algorithm with a simple example. Let the attributes be A1, A2, A3, A4, and A5. Let the constraints be the following:

CON1: A1 • A2 = Secret
CON2: A1 • A5 = Secret
CON3: A1 • A4 • A5 = Secret
CON4: A2 • A4 = Secret
CON5: A3 • A4 = Secret

Note that some of the constraints are redundant. For example, CON2 implies CON3. In this paper we are not concerned with the redundancy of the constraints. Since the maximum classification level assigned is Secret, all the attributes can be stored in a file at the level Secret or Higher. At the Unclassified level, the following clusters are created:

C1 = {A1, A3}; C2 = {A2, A5}; C3 = {A4}.

It should be noted that, although the algorithm guarantees that the constraints are processed securely, it does not provide any guarantee that the attributes are not overclassified. More research needs to be done in order to develop an algorithm which does not overclassify an attribute more than is necessary.

A.3. A Note on Simple Constraints

Since simple constraints classify individual attributes at a certain security level, they could also be handled during database design. Note that when an attribute A in relation R is classified at level L, then all elements which belong to A are also classified at level L. Therefore, we can store A itself at level L.

The algorithm which handles simple constraint is straightforward. Each attribute that is classified by a simple constraint is stored at the level specified in the constraint. Once the algorithm for processing simple constraints is applied and the corresponding schema is obtained, then this schema is given as input to the algorithm handling association-based constraints. The association-based constraints are then applied and the final schema is obtained.

We illustrate the combined algorithm with an example. Let relation R have attributes A1, A2, A3, and A4. Let the constraints enforced be the following:

Simple constraint: A4 is Secret
Association-based constraint: A2 and A3 together are Top Secret.

Applying the algorithm for handling simple constraints we obtain the following:

A1, A2, and A3 are Unclassified; A1, A2, A3, and A4 are Secret.

Next we apply the algorithm for handling association-based constraints. The final output is:

A1 and A2 are Unclassified; A1, A2, A4 are Secret; A1, A2, A3, and A4 are Top Secret.

A.4. Handling Logical Constraints

Logical constraints are rules that can be used to deduce new data from existing data. If a security constraint classifies the
new data at a level that is higher than that of the existing data, then the existing data must be reclassified. Logical constraints could be straightforward such as \( Ai \implies Aj \) or they could be more complex such as \( A1 & A2 & A3 & \ldots & An \implies Am \). If \( Aj \) is classified at the Secret level, then \( Aj \) must be classified at least at the Secret level. If \( Am \) is classified at the Secret level, then at least one of \( A1, A2, \ldots, An \) must be classified at least at the Secret level.

In Section III we showed how the logical constraints may be handled during query processing. For example consider the constraint \( Ai \implies Aj \). If \( Aj \) is classified at the Secret level, and an Unclassified user requests for \( Ai \) values, the query processor will ensure that the \( Ai \) values are not released. That is, although \( Ai \) may be explicitly assigned the Unclassified level, since the logical constraint is treated as a derivation rule, it does not cause any inconsistency. That is, during query processing, the security level of \( Ai \) will be computed to be Secret.

For logical constraints which do not have any conditions attached, it appears that they could be handled during database design. That is during design time the logical constraints are examined, and the security levels of the attributes specified in the premise of a constraint could be computed. For example, if \( Aj \) is classified at the Secret level then it must be ensured during design time that \( Aj \) is classified at least at the Secret level also. The following algorithm will ensure that the security levels are computed correctly.

1) Do the following for each logical constraint. (Note that we have assumed that the constraints are expressed as horn clauses. That is, there is only one atom in the head of a clause.)

2) Check whether there are any simple constraints which classify the attribute appearing in the head of the logical constraint at any level. If not, ignore the constraint.

3) If so, find the highest security level \( L \) that is specified for this attribute.

4) Check whether any of the attributes appearing as premises of the logical constraint are classified at least at level \( L \). If so, ignore the constraint.

5) If not, classify one of the attributes (say, the first one) at the level \( L \).

The algorithm given above does not ensure that the attributes are not overclassified. In order to avoid the overclassification problem, modifications must be made to step 5. That is, once an attribute is assigned a security level, it is possible for the level to be reassigned based on other logical constraints that are handled. Our current research includes investigating techniques for successfully assigning security levels to the attributes and at the same time avoiding overclassification.

When logical, simple, and association-based constraints are combined, then the first step would be to handle the simple constraints. The next step would be to apply the algorithm given above for the logical constraints. Finally the algorithm given in Section V.A.2 is applied for the association-based constraints.

B. Schema Distribution

In this section we discuss briefly the generation of fragmentation and allocation schemas. Note that the techniques that we use for multilevel distributed database are those that are already developed for nonmultilevel distributed databases. (see, for example [3]). There are three types of fragmentation. They are:

1) horizontal—where the relation is partitioned horizontally,
2) vertical—where the relation is fragmented vertically, and
3) mixed—where a relation is first fragmented horizontally (or vertically) and then fragmented further vertically (horizontally).

There are two types of horizontal fragmentation. One is primary fragmentation where a relation is fragmented depending on the values of a collection of attributes. For example, the TANKS relation could be fragmented horizontally depending on the assignment date if most application request information form TANKS based on the assignment date (such as retrieve all tuples from TANKS which were in action before January 1991). The second type of horizontal fragmentation is derived horizontal fragmentation. In this type of fragmentation, a relation is not fragmented based on the values of its attributes, but is derived from the horizontal fragmentation of another relation. Vertical fragmentation is useful if many applications execute using just one fragment. For example, the TANKS relation could be partitioned into two fragments: one consisting of attributes Number, Command, and Type and the other consisting of the attributes Number, Assignment, and Code. If the applications can be grouped into two categories where one category of applications will require information mostly from the first fragment and the other category of applications will require information mostly from the second fragment, then such a fragmentation is useful. Mixed fragmentation utilizes both horizontal and vertical fragmentation. Such a fragmentation is useful only if there are applications which require information about a certain part of a relation depending on the values of some specified attributes.

Generating the allocation schema will depend on whether the allocation is nonredundant (that is, without replication) or redundant (with replication). Determining nonredundant allocation is easier. A popular approach in determining nonredundant allocation is the best-fit method. Given a fragment, a measure is associated for each possible allocation. This measure depends on various parameters such as number of possible retrievals and updates of the fragment and the locations of the applications which access the fragment. The site with the best measure is selected to store the fragments.

Replication makes the algorithm more complex because the degree of replication should then be considered as a parameter in computing the measures. Furthermore, an application can select one of the several replicated fragments that are available. An approach that has been proposed for generating the allocation schema with replication is the following. Determine the set of all sites where the benefit of allocating a copy at that site exceeds the cost of allocating a copy at that site. Allocate a copy at each of the sites of the set.
VI. FUTURE CONSIDERATIONS

In this paper we have described the issues involved in processing security constraints in a multilevel secure distributed database management system. We first provided some background information on constraint processing and discussed an integrated approach for a centralized environment. Then we discussed the architecture for an MLS/DDBMS for constraint processing. Finally we described in detail constraint processing during query, update, and database design operations.

We believe that constraint processing is the first step toward controlling unauthorized inferences in an MLS/DDBMS. We have provided an integrated solution to constraint processing in a centralized and distributed environment. The SSO can decide whether to adopt this integrated solution or whether to process constraints during the individual operations such as query, update, or database design. For example, for an application that is static, it is sufficient to process constraints during database design and update operations. Furthermore, if the constraints are fairly straightforward, where the security level of the data to be inserted is not affected by the data already in the database, the update processor could be used off-line.

While much has been accomplished, there are still several issues that need to be resolved. First, the implementation of the integrated approach to constraint processing needs to be carried out. Second, constraint processing techniques need to be incorporated into transaction processing algorithms. Third, the consistency and completeness of the constraints need to be ensured. Finally, a tool must be developed to generate the initial set of constraints from the specification of the multilevel application. We believe that many of the issues identified can be resolved by adapting the techniques that relational database and logic programming researchers have developed to handle integrity constraints in nonmultilevel relational and deductive databases. This is because in our work we have shown that security constraints are a special form of integrity constraints in multilevel databases. Effective processing of security constraints is, therefore, the first step toward developing multilevel secure intelligent database systems.

The discussion in this paper has assumed that the environment is a homogeneous one. That is, the MLS/DDBMSs are designed and operate identically. A more realistic environment would have some form of heterogeneity. Therefore, the techniques described in this paper have to be extended to accommodate heterogeneity. As a first step, we have begun an investigation on constraint processing in a multilevel distributed environment where not all of the nodes handle the same accreditation ranges. That is, one node may handle the range from Unclassified to Secret while another node may handle the range from Secret to Top Secret. Our research and implementation results will be described in forthcoming papers.

12. We first completed a preliminary proof-of-concept implementation of a prototype which handles constraints during the query operation in a distributed environment. Since then we have completed the implementation of the integrated tool.

ACKNOWLEDGMENTS

We gratefully acknowledge the U.S. ARMY CECOM for sponsoring the work reported in this paper under contract DAAB07-91-C-N751. It was carried out at the MITRE Corporation during FY91. We thank John Preusse and Andres Abreu of CECOM for their support and encouragement. We thank Andres Abreu, Leonard Binns (of the Office of INFOSEC Computer Science, DoD), Rae Burns, Gary Gagnon, and Jonathan Millen (of MITRE) for their comments on this paper.

We are grateful to Marie Collins (of MITRE) for her contributions to constraint processing during update operations which we have extended to a distributed environment.

Disclaimer: All of the examples used in this paper are hypothetical.

REFERENCES

Bhavani Thuraisingham is a principal engineer with MITRE Corporation's Network and Distributed Information Systems Center and heads the Corporate Initiative on Data Management Research. She is currently working on real-time database systems, massive multimedia database management, and database security. Her interests also include heterogeneous database integration and object-oriented design and analysis techniques for developing various information systems applications. Her previous work at MITRE includes the design and implementation of a secure distributed query processor, database inference controller, and secure multimedia/object-oriented database system. She provides directions in database management research and development for the Department of Defense and the Intelligence Community. She is a codirector of MITRE's Database Specialty Group and serves on the Corporate Technology Area Council in Database Systems.

Prior to joining MITRE, she conducted research and development activities at Honeywell Inc., where her work included the design of the secure database system Look Data Views, the design of a network operating system for space station applications, and also the application of object-oriented technology for developing next-generation process control systems and for integrating heterogeneous data dictionaries. Before that she was at Control Data Corporation, where she worked on the product development of CDCNET. She was also an adjunct professor of computer science and member of the graduate faculty at the University of Minnesota.

Her work has been published in over 40 journal papers and several other conference/workshop papers. She is also an inventor of a U.S. patent on database inference control. Dr. Thuraisingham gives tutorials in object-oriented database systems, heterogeneous database systems, secure database systems, and real-time database systems to various government organizations.

Mr. Ford was at AOG Corporation, where he was responsible for the design and implementation of data dictionary systems and real-time control software systems. He has also designed and implemented software systems for Decision Data Corporation and the University of Delaware.

Ford received the BS degree in computer science at the University of Delaware in 1975 and the BS degree in electrical engineering at the University of Delaware in 1976. He has published papers in various IEEE and ACM conferences. Mr. Ford is a senior member of the IEEE.

William Ford is an independent consultant in computer security and database systems and has more than 17 years of software development experience. Previously he was a member of the technical staff at MITRE Corporation, where he conducted design and implementation activities in security constraint processing, secure distributed database systems, and the inference problem. He also worked on other computer security projects, including security guards and privacy of electronic mail. Prior to joining MITRE,