(4)

RADC-TR-89-192
Final Technical Report
October 1989

DATABASE CONSISTENCY AND SECURITY

Sytek, Inc.

Sponsored by
Strategic Defense Initiative Office

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Services (NTIS) At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-89-192 has been reviewed and is approved for publication.

APPROVED: Emilie J. Siarkiewicz

EMILIE J. SIARKIEWICZ
Project Engineer

APPROVED: Raymond P. Urtz Jr.

RAYMOND P. URTZ, Jr.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER: Igor G. Plonisch

IGOR G. PLONISCH
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COTD) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

DATABASE CONSISTENCY AND SECURITY

B.T. Blaustein, A.P. Buchmann, U.S. Chakravarty - Computer Corp. of America
J.D. Halpern, S. Owre - Sytek, Inc.

Contractor: Sytek, Inc.
Contract Number: F30602-86-C-0263
Effective Date of Contract: 19 September 1986
Contract Expiration Date: 31 July 1988
Short Title of Work: Database Consistency and Security
Period of Work Covered: Sep 86 - Jul 87

Principal Investigator: Alejandro P. Buchmann, CCA
Phone: (617) 492-8860

RADC Project Engineer: Emilie J. Siarkiewicz
Phone: (315) 330-2158

This research was supported by the Strategic Defense Initiative Office of the Department of Defense and was monitored by Emilie J. Siarkiewicz, RADC (COTD), Griffiss AFB NY 13441-5700.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS: N/A
2a. SECURITY CLASSIFICATION AUTHORITY: N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): N/A	5. MONITORING ORGANIZATION REPORT NUMBER(S): RADC-TR-89-192
6a. NAME OF PERFORMING ORGANIZATION: Sytek, Inc. / 6b. OFFICE SYMBOL (If applicable):	7a. NAME OF MONITORING ORGANIZATION: Rome Air Development Center (COTD)
6c. ADDRESS (City, State, and ZIP Code): 1225 Charleston Road, Mountain View CA 94043	7b. ADDRESS (City, State, and ZIP Code): Griffiss AFB NY 13441-5700
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: Strategic Defense Initiative Office / 8b. OFFICE SYMBOL (If applicable):	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: F30602-86-C-0263
8c. ADDRESS (City, State, and ZIP Code): Office of the Secretary of Defense, Wash DC 20301-7100	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. 63223C; PROJECT NO. B413; TASK NO 02; WORK UNIT ACCESSION NO 18

11. TITLE (Include Security Classification): DATABASE CONSISTENCY AND SECURITY

12. PERSONAL AUTHOR(S): B.T. Blaustein, A.P. Buchmann, U.S. Chakravarty - Computer Corp. of America; J.D. Halpern, S. Owre - Sytek, Inc.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM Sep 86 TO Jul 87	October 1989	82

16. SUPPLEMENTARY NOTATION: This work was actually performed by one of the subcontractors; Computer Corp. of America (now Xerox Advanced Information Technology), Four Cambridge Center, Cambridge MA 02142.

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Distributed Databases; Specification/Verification; Multilevel Secure; Database Consistency
12	07		

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report addresses the issues of consistency in Secure Distributed Systems (SDS) and focuses on the special relationship between consistency and security in an SDS, their conflicts and possible trade-offs. It establishes a unified framework for treatment of consistency and security in a coherent and flexible manner. It identifies approaches to consistency from a variety of disciplines and proposes how these approaches may be useful in the realm of Secure Distributed Systems. Finally, attempts at formally specifying consistency conditions are reported. These specifications have highlighted strengths and weaknesses of some existing tools for formal specification and have led to the identification of new features that are required for the successful formal specification of consistent secure distributed systems.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: ☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: Emilie J. Siarkiewicz	22b. TELEPHONE (Include Area Code): (315) 330-2158 / 22c. OFFICE SYMBOL: RADC (COTD)

DD Form 1473, JUN 86 — Previous editions are obsolete. — SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

CONTENTS

ILLUSTRATIONS

PREFACE

In September 1986 Rome Air Development Center awarded a contract to Sytek, Inc., of Mountain View, California, to investigate techniques and tools for the specification and verification of secure distributed systems. The effort consisted of three phases. During the first phase distributed systems issues, such as temporal properties, database consistency, fault-tolerance, and adaptation, were studied with respect to the specification requirements for distributed systems. During phase two, existing specification/verification, database design, and other software engineering tools were to be analyzed in light of the results of the phase one studies and a near-term workbench designed and implemented. Phase three was to look at the requirements and design of the next generation of tools.

In mid-1987 Sytek decided to divest themselves of the their research and development contracts division. As a result, and despite many months of negotiation, it was decided to terminate the effort.

This report, written in July 1987, documents the database consistency study. The remaining studies and the work done on the workbench design will be documented elsewhere as a result of two very small contracts to two of the subcontractors to do some clean-up work.

1. Introduction

The design of complex Secure Distributed Systems requires the use of sophisticated software tools to ensure the reliability of the software for demanding applications such as SDI Reliability of software can be enhanced by careful specification and verification It is fair to say that no single specification and verification tool at present can meet the taxing demands of a complex Secure Distributed System Therefore an integrated workbench of software tools is needed and will be developed in the context of the present contract Understanding the requirements of Secure Distributed Systems is a prerequisite to the actual evaluation of existing software design tools and their selection for integration in the workbench Therefore the first phase of the present contract calls for three concurrent studies on temporal properties consistency and fault-tolerance of Secure Distributed Systems This report addresses the issues of consistency in Secure Distributed Systems and focuses on the special relationship between consistency and security in an SDS their conflicts and possible trade-offs It establishes a unified framework for treatment of consistency and security in a coherent and flexible manner it identifies approaches to consistency from a variety of disciplines and proposes how these approaches may be useful in the realm of Secure Distributed Systems Finally attempts at formally specifying consistency conditions are reported These specifications have highlighted strengths and weaknesses of some existing tools for formal specification and have led to the identification of new features that are required for the successful formal specification of consistent secure distributed systems

A basic condition for the reliable operation of a secure distributed system is the consistency of the data Equally important is the requirement that data be handled in accordance with security policies which determine who may access and modify data Since this report considers the interaction of data consistency and security and their formal specification it is essential to introduce first a common terminology Unfortunately many terms are used with different meanings in the security the formal specification and the data management communities Because data management issues are mostly confined to this report while formal specification and security issues are common themes throughout the contract we adapt conflicting terms from the data management realm and preserve the meaning a term has originally in the security or formal specification realms We only depart from this approach when we propose changes to the traditional meaning of some of these terms

Section 2 of this report introduces the notation for consistency and discusses the different types of consistency conditions their structure and evaluation Consistency is analyzed in the context of a database Therefore we introduce the model of a database establish the notions of consistent database states and the need for consistency conditions as an integral part of a database description We look at consistency under different points of view consistency under validation of database constraints or consistency conditions consistency under concurrent transactions consistency under replication and consistency under failure recovery

Section 3 expands on the conflicts that arise between consistency and security particularly integrity as used by the security community Many problems arise from the diffuse definition of integrity Therefore the notion of integrity is refined and building on the consistency conditions introduced before it is shown how security policies can be formulated in terms of constraints thereby establishing a common framework for consistency and security In such a framework security policies can be adapted to the requirements of a particular environment or can be exchanged as security requirements evolve It is beyond the scope of this report to attempt the formulation of a complete

new security policy. Instead. we present only the basic elements of a policy and cast them in the common framework for security and consistency.

Consistency is a dynamic property that has to be attained at the end of a given action or sequence of actions performed on the database. Therefore. in Section 4 we discuss transactions as a means of achieving atomicity and analyze their function in a database environment. For the sake of clarity. we analyze first a single transaction on a non-distributed database and introduce the basic notions. In subsequent portions we analyze the case of multiple transactions executed against a single database and. finally. the case of multiple transactions in a distributed environment.

We discuss the notions of commit and the effect of evaluating constraints at the beginning of a transaction (as required for security reasons) in contrast to doing it before committing the transaction (as required for database integrity). We review the notions of a log. recovery and rollback. We analyze the impact of external effects and consider necessary extensions to this notion caused by the security requirements. We also look at some recent proposals for flexible consistency handling. both through deferral of evaluation of consistency constraints . and through special actions that may be required in response to certain database constraint violations. These notions are just emerging in the database literature and are included in the hope that they may trigger some discussion of these issues in the context of secure systems.

In Section 5 we present examples of a database and some basic functions of a DBMS described in an experimental extended version of SYSPECIAL. This specification language is extended with the notions of a multilevel specification and a trace. With the help of these two constructs it is possible to model database transactions and serialization protocols. such as two-phase locking. SYSPECIAL proved to be a powerful tool for specifying consistency and security conditions in terms of SYSPECIAL's invariants and constraints. The specification of databases proved to be easy and the specifications readable.

The main contributions that we discuss in this report are:

A unified framework for consistency and security. This framework encourages adaptability in the sense that new security policies can be inserted by substituting sets of constraints. By using the same representation for security and consistency. we lay the groundwork for checking compatibility of security and consistency constraints. During an analysis of conflicts between consistency and security. we detected that the notion of integrity. as used in the security community. appears to be ambiguous Therefore. we divided the old notion of integrity into write-sensitivity and trustworthiness. This distinction avoids problems of downgrading write sensitivity when trustworthiness is degraded.

Security issues are analyzed in the context of transaction processing. both for the centralized case and the distributed case. The focus on the processing of the transactions and the unified view of consistency and security constraints is novel. This analysis is also carried out for the distributed case and a discussion of non-serializable protocols and their security implications is presented.

A close look at concurrency control helped identify extensions that are desirable for specification of consistent databases and the elements of a database management system. SYSPECIAL was experimentally extended to include the notions of a two-level specification and of a trace. The novelty consists in using the notion of a trace to support procedural constructs and the mappings between levels. Using the extended SYSPECIAL. it was possible to specify a database and the rudiments of a DBMS that uses two phase locking to provide serializability The initial success with the extensions warrants further development.

2. Consistency

Consistency is a correctness property that is attained whenever data comply with a set of constraints or correctness conditions that are defined for a given collection of data To avoid confusion a brief discussion on terminology is necessary.

The terms *consistency* and *integrity* are used in the database realm almost interchangeably A database possesses *integrity* (is consistent) if all the integrity constraints defined for the database are met and whenever individually correct transactions complete execution according to a consistency preserving schedule. In the security realm, *integrity* has been used as a dual to *security* and usually refers, in an ill-differentiated manner, both to write access rights and to trustworthiness We will not belabor the usage of the words at this point this section will deal with database aspects of integrity whereas the next section (Section 3) will use the term *integrity* as understood by the security community At this point we only intend to state the differences to justify our terminology and usage *Integrity* will be reserved throughout this report for use in a security context. *Consistency* will be used whenever referring to database integrity. At the same time, if the usage of a term is unambiguous for the context, then we prefer to use the term befitting the context without imposing any artificial restriction.

Similarly, the terms *constraint* and *invariant* are well-defined terms in the SYSPECIAL specification language, invariants being conditions defined on states, while constraints are defined on operations. Unfortunately, these terms are commonly used in the database realm both when a correctness condition is defined assertively on database states and when it is defined as a trigger associated with an operation. To avoid confusion, we shall use the term *consistency condition* instead of *constraint* whenever there is the possibility of confusion. Otherwise, we shall use the term *consistency constraint* since this term is best understood in a database context.

In this section we classify consistency according to four criteria validation of consistency constraints, consistency under execution of concurrent transactions, consistency under replication, and consistency after failure recovery. Before we proceed with the discussion of consistency, it is necessary to introduce first a model of a database on which the correctness conditions can be defined

2.1 Specification of a Database

A database is the representation of a portion of the world through a structured collection of data The valid structures are determined by the data model that is being used In the case of the relational data model the valid structures are flat, tables composed of n-tuples. Each table or relation has an intension and an extension. The intension of a relation is given by the attributes that comprise a relation. The extension of a relation is the set of ordered n-tuples in which each attribute is instantiated A relation is defined as a subset of the cartesian product of the domains corresponding to the attributes of the relation. That is

$$R \subseteq A1 \times A2 \times \ldots \times An$$

The intention of all the relations comprising a database is called the schema of the database.

For illustration purposes we will use the following database throughout this report:

```
Employee (SSN, Emp-name, Emp-address, department)
Employee-security (SSN, clearance)
Projects (Proj-id, Mgr-SSN, department, classif, location, travelfunds)
Trip (Trip-id, origin, dest, d-left, d-arrived, contact, charges)
Proj-Empl (SSN, Proj-id)
Proj-Trip (Proj-id, Trip-id)
```

This database schema can be formally specified using SYSPECIAL and is shown in Section 5, where all the Syspecial examples are consolidated and discussed.

For secure environments it is necessary that data in the database carry a classification tag. There are different forms in which tags can be assigned to data: the same classification tag for all the data in a relation, the same classification for all the instances of an attribute in a relation, the same classification for all the data values in a tuple, or individual classification tags for each atomic value (at the attribute level) in the database. We chose the last form for flexibility reasons. Each data value can be tagged. The values that a given tag can take are constrained through invariants defined on the schema. For reasons that are explained in the next section, each tag consists of a triplet of security attributes: a read-sensitivity label, a write-sensitivity label, and a trustworthiness label. The triplet is specified using the notation <r, w, t>, where r, w, t can be viewed as functions representing the read-sensitivity, write-sensitivity, and the trustworthiness respectively. The following is an example of the specification of a relation with its security tags:

```
Employee (SSN<st>, Emp-name<st>, Emp-address<st>,department<st>)<st>
```

where each attribute as well as the relation Employee itself is associated with security tags. It is also possible to group the attributes in an arbitrary way to associate a security tag with the set of attributes instead of individual attributes. A SYSPECIAL example of security tag representation for the database shown above is described in Section 5.

At any given time the database is in a state that can be consistent or inconsistent. A database state is consistent if all the data stored in the database conform to the consistency conditions defined on the database and the operations that map one database state into another preserve consistency.

2.2 The Structure of Consistency Conditions

When analyzing the consistency conditions that can be enforced on a database and the transitions from one database state to another we can identify essentially four groups of consistency conditions that correspond to four different aspects of consistency. Figure 2.1 summarizes the different aspects of consistency and the types of consistency conditions

Structure of Consistency Constraints

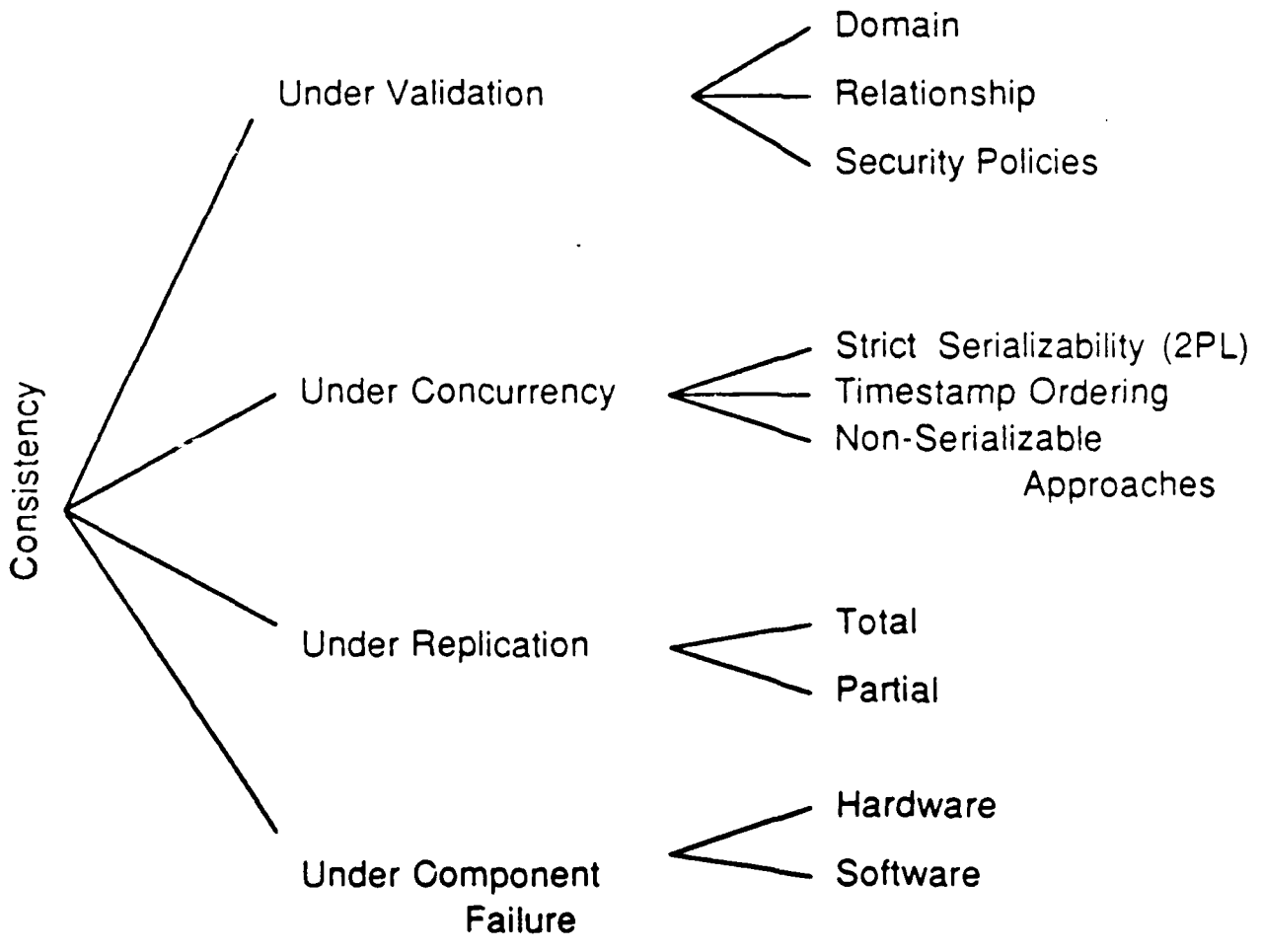

Figure 2.1

2.2.1 Consistency under Validation

Consistency under validation encompasses the validation of consistency conditions defined on the database, such as domains of attributes and relationship consistency conditions among two or more attributes in the database, and structural consistency conditions that are often data-model dependent.

Domain consistency conditions can be described either as *ranges* or by explicit *enumeration*.

Range consistency conditions determine acceptable values for an attribute and are expressed through lower and/or upper bounds. for example, the lower bound for wages is given by the minimum wage, or legal age values for employees may be between 18 and 65.

Enumeration consistency conditions determine acceptable values for an attribute by explicit enumeration of the members of a set of discrete values, for example the available classes on an airplane are coach, and first class. The values in the constraining set are constants like those used in defining ranges.

Relationship consistency conditions determine the relationship that exists among values of two or more attributes in the database. They are often extensions of the previous constraints, since those by themselves prove to be too restricted in their expressive power. For example, the available flight classes are a function of the origin and destination of the flight, since for national flights only coach and first class are available, on international flights however, the classes are economy, business and first class. Relationship consistency conditions are also used to express other validation criteria among attributes, such as those involving aggregates over an attribute or a direct relationship between attribute values in the database. An example of a consistency condition defined over an aggregate could be that no employee may receive a salary that is not within 5,000 dollars of the average for that job description. A consistency condition establishing the relationship among attributes can be as simple as saying that no employee may earn more than his/her manager or as complex as a simulation of an integrated circuit.

Certain *structural consistency conditions* are intimately related to the database model that is being used. For example, the relational data model requires that each tuple in the database be uniquely identified through a key. A key-uniqueness consistency condition is important in the relational model but is meaningless in other data models that do not depend on the notion of a key for identification of tuples or records. Models that do not enforce the key-uniqueness property have other consistency conditions, such as the number of component objects or subordinate records. Another example of structural consistency conditions that have to be validated is the referential integrity constraint, which specifies that an external key may not exist if the corresponding tuple with that same value as the primary key is not in the database.

Typically, consistency conditions are validated whenever a data value is written into the database. In this respect, any security validation concerned with writing values to the database falls into the same category. Security criteria ought to be expressed and validated in the same form as the other consistency conditions that we have listed above. In addition to the validation at the time of writing values into the database, security considerations also require a validation at the time of reading data. However, any security criterion can as well be expressed in the same format as the other consistency conditions. How this can be done is detailed in Section 3.

2.2.2 Consistency under Concurrent Transactions

While consistency under validation deals with consistency conditions that are defined on the database and have to be enforced for a new database state to be valid (i.e. a transition from one state to another has to be validated), consistency under concurrent transactions is concerned with ensuring consistency under simultaneously executing processes. Retrievals and updates to a database can be grouped into transactions. A correct transaction takes a database from one consistent state to another consistent state. However, if two or more correct transactions are executed against the same set of data, and they are interleaved at random, the final result may not be the same as if the transactions were executed sequentially one after the other. The database may not end up in a consistent state in spite of satisfying the individual consistency conditions.

As an example, let two transactions modifying a bank account try to access the balance of an account, one to add $100 and the other to subtract $200 from the same account. If the transactions do not interleave correctly, then one of the two updates may be lost and the account may end up either with $100 missing or with $200 in excess. This will happen regardless of whether a static consistency condition (e.g. final balance may not be negative) is met.

The solution is to impose dynamic ordering constraints that have to be satisfied during the execution of the concurrent transactions. The basic ordering is known as serializability and insures that the effect of two or more transactions executed in parallel on the same data is the same as the effect of executing the transactions sequentially. The problem one faces in formally specifying these constraints lies in the nature of database processing. Queries and update transactions are composed spontaneously by the user and the formal specification and verification of arbitrary transactions is beyond the current state-of-the-art in formal specification and verification. However, there is a large class of applications that require the data manipulation capabilities of a DBMS but have fairly stable data requirements that can be served by predefined queries and transactions. The problem is somewhat simplified if the transactions executed over the database are fixed in number enabling their individual verification as well as the mix using detailed static analysis.

The theory of serializability for concurrent execution of transactions is at a mature stage both for centralized and distributed model of database management systems. Many protocols that ensure serializability have been developed and proven by model-theoretic means. We shall use them without attempting to prove their correctness. However, most of these protocols rely on the notion of read- and write-locks which are potential information channels through which information can be tapped out by locking data in a given pattern. Alternate protocols based on eventcounts have been proposed to solve this problem. Given that the vast majority of concurrency control mechanisms are based on locking schemes and that additional safeguards are required in the architecture of any functional system, we will analyze concurrency control mechanisms based on locking in this report.

Other classes of non-serializable protocols that are based on timestamping or the notion of relaxation of consistency conditions are discussed in the next subsection, consistency under replication. They are mostly applicable to distributed database systems with total or partial data replication.

2.2.3 Consistency under Replication

The third type of consistency that needs to be considered is consistency of replicated data. This is a problem most common in distributed systems where multiple copies are maintained for reliability and performance and inconsistencies arise when one copy is modified asynchronously. To ensure consistency of multiple copies of data additional synchronization mechanisms have to be defined and enforced.

The basic synchronization mechanisms assume strong consistency conditions, i.e. the consistency conditions must be met by all copies of the replicated data or the transaction will be blocked. The underlying assumption is that communication channels are stable and that the network will not be partitioned. For environments in which this assumption is not met and where high availability is essential (for example in military command and control systems), alternative notions of consistency have to be defined and enforced. One such mechanism is to defer the enforcement of consistency conditions with the guarantee that eventually a consistent state will be established when communication is reestablished.

When serializability is not guaranteed, it becomes important to take compensating actions. If we assume that a transaction caused an effect on out-of-date data or because a consistency condition could not be evaluated a compensating action is required. For example, if a previous booking of an airline seat did not come through from a remote location a later booking may actually overbook the flight in a given class. The airline can take compensating actions by either rebooking the passenger on another flight, upgrading him to another class, or not doing anything hoping for a no-show.

2.2.4 Consistency under Failure Recovery

A fourth aspect of consistency is related to failures of hardware or software in the system. In this case the transactions against the data may be suspended before completion leaving the data in an inconsistent state. To restore consistency, an older but consistent copy of the database has to be activated and the recent changes have to be reexecuted satisfying the constraints as before. To be able to reconstruct the exact sequence of operations a log of all the transactions that were executed from the last available consistent state has to be kept.

The recovery problem entails a reexecution of the constrained actions that are discussed as consistency under validation, concurrency and synchronization. However, it is the maintenance of the log that causes potential security threats.

2.3 Flexible Schemes for Consistency Condition Evaluation

The higher the degree of consistency one wants to enforce on a database, the higher is the price in terms of performance and availability. In a secure environment, additional conflicts arise between consistency conditions and security requirements. These are described in more detail in Section 3. These conflicts further reduce the availability of the database and it appears convenient to explore ways of trading-off between consistency, availability, and security. This can be done through the use of flexible mechanisms for consistency condition evaluation.

The flexibility consists in modifying two aspects of the currently used mechanisms for consistency enforcement: the timing of evaluation and the actions that are taken in response to a violation.

Commercially available database management systems offer, usually, one mode for constraint evaluation constraints are precompiled with the schema definition and they are always enforced Enforcement occurs when a modification is tentatively performed in the buffer and before a transaction is ready to be committed to disk If a consistency condition is violated the DBMS automatically aborts the transaction This mechanism appears to be adequate for most conventional applications but not for the secure distributed systems we are concerned with

In Section 4 we discuss the different timing needs of secure distributed systems, and how deferred evaluation of consistency conditions can alleviate some conflicts between consistency and security and how that mechanism can increase availability even if this occurs at the expense of consistency These trade-offs may be necessary in crisis situations when it is more important to get to the data in time than protecting them or guaranteeing their consistency We also discuss what alternative actions are desirable, particularly in secure environments

3. A Framework for Security and Integrity Policies

3.1 Security and Integrity Policies

The essence of any secure computer system is the protection of sensitive data, and claims that a system protects its data from improper accesses must be demonstrated with as high a level of confidence as possible (see [DOD85] for the Department of Defense evaluation criteria.) Therefore, a number of formal models which attempt to capture the fundamental protection requirements have been proposed [LAND81 BELL73, BELL74 BELL75, BIBA77, FEIE77, FEIE79 FEIE80, KSOS78]. Most formal discussions of secure computer systems divide the problem of protecting sensitive data into two parts: *security* — protection from improper disclosure, and *integrity* — protection from improper modification (including creation and deletion)

Within the restricted notion of security, there is another distinction: mandatory *vs* discretionary controls. Mandatory controls are those that are enforced by law, stating that only individuals with appropriate clearances may have access to certain data. The restrictions on data are expressed by the sensitivity level (e.g., TOP SECRET), and possibly by compartment names (e.g., NATO) and caveats (e.g., NO FOREIGN). Discretionary controls are those that rest with the person releasing particular information, usually based on perception of the recipient's "need to know."

The issue of security is better understood than the more complicated one of integrity. Studies have concentrated more on formal security models than on integrity models. Bonyun [BONY86], Schell and Denning [SCHE86], and the working group on database integrity [NCCS86] have discussed at length various aspects of integrity which translate into formal properties that are quite different from those addressed by the traditional security policies. The thrust of our work is to provide a unified framework for expressing security and integrity properties and for integrating these properties into a database model that enforces the various aspects of consistency.

One of the stated requirements for the Strategic Defense System [DRC86a, DRC86b] is the need for security policies that are situation-dependent. It is assumed that for peace-time operation the distributed computing system would not be stressed to capacity and that delays derived from security enforcement are tolerable. On the other hand, any intruder has plenty of time to attempt to penetrate the secure system. In a crisis situation the opposite is true: the computing system is stressed to its capacity and an intruder has only a limited time to attempt penetration. Therefore, it may be desirable to use alternate, or adaptable, security policies. By modeling the security policies as constraints we provide a flexible framework that allows the specification of alternate policies, the verification of these policies, and the mapping of the policies into the underlying implementations.

In designing an automated system to enforce security controls, the key issue is the formal complexity of the rules to be enforced. Whichever aspects of integrity are considered important requirements for a particular application, and whichever types of controls are needed, the essential problem from the system designer's point of view is the way in which the rules are to be expressed and enforced. The analysis of the rules' complexities depends on the language used to express them, the data model, the particular data schema, and the power of the software that will enforce the rules. It is not our goal here to recommend specific security or integrity policies; indeed, it would be less useful to develop a system that could accommodate only a single set of security and integrity properties. Instead, our goal is to describe the problems of security and integrity from the viewpoint of database consistency and to investigate general techniques for enforcing whatever security and integrity policies

are chosen for a particular application. In addition to providing a general basis for a group of systems with different or changing requirements, this approach allows designers to identify and weigh the consequences of different security or integrity requirements.

3.2 Policies as Database Constraints

3.2.1 The Use of Database Constraints

Early information and database management systems offer no support for centralized consistency verification. Instead, each transaction has to include, interleaved in the application code, the necessary consistency verification procedures. This approach simplifies the database management system but places the burden of consistency enforcement on the applications programmer, is error-prone, and cannot be used for enforcement of data access restrictions.

A more sophisticated approach, and the one we develop here, views transactions as partial descriptions of the proposed operations. The rules and policies that may affect the data access and other operations on the data described by the transactions are expressed separately and are automatically enforced by the system. While this approach requires a language for expressing rules, software for enforcing the rules, and an architecture which accommodates system-invoked additions to the original transaction, it also results in many advantages. Rules that describe data accesses and data consistency criteria are consolidated in one place, and they are expressed uniformly. This allows for checking of mutual consistency among different rules that apply to the same data. Most important in the context of our current project is the fact that expressing both consistency constraints and security policies as sets of constraints defined on the database greatly increases verifiability. In general, this approach makes for both tighter control over data accesses and greater flexibility in the addition or modification of rules.

3.2.2 Policies and Types of Constraints

The security and integrity policies for a particular application can be expressed as database constraints. These constraints may be designated as *strong* or *weak* constraints. Strong constraints, which impose strict controls on data accesses, act as enforcers of policies that limit unauthorized or improper retrievals and updates. They filter out unauthorized transactions, letting only authorized transactions proceed through the system. Strong constraints are used to enforce critical security and integrity policies.

What, then, is the role of weak constraints in implementing security and integrity policies? Security and integrity policies are, by nature, strict controls. It doesn't make sense to allow "relaxations" of these rules -- as it may, for example, in traditional database consistency enforcement. However, weak constraints can play a very useful role within the context of security and integrity policies.

The traditional dichotomy inherent in security and integrity policies can be refined: instead of merely describing allowable and disallowed events, the policies can describe disallowed events, allowable routine or expected events, and allowable exceptional or unusual events. Events in this added last category, while meeting the strict security and integrity requirements of the application, may bear watching. For example, it might be allowable for certain *trusted* users, using certain transactions, to violate the strict "*no-write-down*" integrity policy. (Such a policy is mentioned in [FEIE77, KSOS78, LAND81] and discussed in more detail in a subsequent section). While these actions are explicitly allowed, it might be desirable to mark these actions for later review. Should an unexpected or suspect pattern arise, a security official may decide to revise the policy or the classifications of some users or

transactions

The ability to monitor certain events enhances the security policy, by allowing less restrictive policies where appropriate without relinquishing the opportunity of observing their use. Together with the use of a flexible framework for specifying policies, the ability to monitor events provides a good basis for adaptable and responsive policy designs. While strong constraints express policies which describe unallowable actions, weak constraints can be used to capture those policies which describe events to be monitored. Thus, weak constraints provide an automated means of handling legal, but nonetheless exceptional, events.

To summarize, security and integrity policies are defined through the statements of the policies, and also through the determination of which policies will be modeled as strong constraints -- requiring strict enforcement -- and which as weak constraints -- suggesting monitoring. An example of how strong and weak constraints can be used together follows in Section 3.4.5.

3.3 Conflicts Between Security and Consistency

While it is extremely useful to view security and integrity requirements as parallel to the more traditional database consistency requirements, there may be some conflicts between the two. Security requirements mandate strict restrictions on data flow and strong separation of data, while database consistency motivations tend toward increased analysis of complex relationships across broad ranges of data. Enforcing a database consistency constraint will typically involve reading data items that are not explicitly mentioned in the given update being tested. In the database consistency world, this is a good thing -- the user need not be aware of the many important relationships that constitute database consistency. In the security world, the same approach represents the potential for serious security breaches.

With some information about the constraints that the system enforces, a user may be able to infer much about specific data values, and the presence or absence of data with certain values, by submitting updates that cause a given constraint to be checked. Suppose, for example, that in the project database there is a constraint that Department HQ-87 has no more than $200,000 in total available travel funds. Suppose, furthermore, that Department HQ-87 sponsors a number of projects, some of which are classified SECRET and some of which are classified TOP SECRET. A user with a SECRET clearance may know that the total available travel funds shown for SECRET projects is $80,000. Suppose that this user submits an update (either innocently or maliciously) to add a new SECRET project to the database. This new project has a value of $5,000 for avail_travel. If the update is denied, with the user either being told directly or inferring that the denial is based on exceeding the department's total available travel funds, then the user knows -- despite the SECRET clearance -- that there must be between $115,000 and $120,000 in travel funds allocated to TOP SECRET projects in Department HQ-87.

Consistency constraints are metadata, i.e., data that describe the data. Whether these database consistency constraints are described by clauses or in a parameterized form in a constraint base, they have to be treated exactly like all the other data in the system with the added precaution that, because of their higher degree of abstraction, they may convey general information about whole segments of the database. Therefore, more information may be transmitted by reading a constraint than by a single, less abstract, data value. An immediate conclusion that can be drawn from this conflict is that constraints, since they carry information, have to be subjected to at least the same classification as the data to which they apply.

This solution highlights another problem, namely, as soon as a constraint spans more than a single level of classification it will enter into conflict either with the secrecy policy of "no read up" or with the integrity policy that will not allow a transaction of a higher level of integrity to commit because it used in the constraint validation process data of lower integrity. Thus, in the case of the user with SECRET clearance attempting to add a new SECRET project, the update would be barred because enforcing the constraint on total available travel funds may require reading TOP SECRET data.

The only safe solution to this problem appears to be the limitation of constraints to a single level of classification. This limitation may, however, impair the overall consistency of the database in two forms. First, constraints that are defined on aggregate values, as is the constraint on total available travel funds, can only be enforced if all the instances of a given attribute or data element have the same classification; and second, no constraints defined on types are legal if the types may contain instances of a lower classification.

If security is to be enforced through views, i e a mechanism that presents to the user only those data for which he or she has clearance and gives the illusion that those are all the data again the constraint may not span more than one classification level. If a constraint is defined globally, i e on the whole database, but is applied only to the data in a view, inconsistencies may arise.

The solution to the previous problem lies in the decomposition of constraints in such a way that it reflects the composition of the underlying data and can be aggregated upwards. For example, suppose that in addition to the constraint that total available travel funds be limited to $200,000, there is another constraint that limits total available travel funds for SECRET projects to $125,000. The need to restrict users with SECRET clearances from learning about TOP SECRET data implies, then, that there must also be a constraint that no more than $75,000 of travel funds can be at the TOP SECRET classification. Otherwise, users with SECRET clearances might see a situation in which an update is disapproved although the total available travel funds at the SECRET level would remain less than $125,000. The SECRET user could then still infer the travel funds at the TOP SECRET level. Even if the SECRET user does not know the global travel fund constraint, i.e., even if the restriction of all travel funds is classified at the TOP SECRET level, problems can still arise. If there is no separate restriction of TOP SECRET travel funds, the SECRET user can still infer partial information about TOP SECRET projects whenever updates that would leave SECRET available travel funds at less than $125,000 are denied. This simple example illustrates the need for testing of constraints for internal consistency.

The possibility to infer something from the denial of service raises the question as to what should be the appropriate action if a constraint is violated. Even seemingly innocuous constraints, such as those specifying the uniqueness of keys, may cause inferences by denial of service. For example, the constraints that every key has to be unique, allows a user with lower clearance to infer that another object with the same key but higher level of classification already exists, based on a rejection of an insertion because key uniqueness is violated. Thus, a malicious user could try to insert a number of new projects with low classifications, in an attempt to "guess" the project id of a highly-classified project.

A solution that is generally recommended for this kind of problem is to use poly-instantiation allowing the insertion of objects with the same visible identifier but modifying the identifier by concatenating the visible identifier with the classification. This solution, however, causes another consistency problem. The instances at the different levels may contain different values for a given attribute and additional constraints would be needed to make the copies compatible.

The main conclusions which can be drawn from this discussion of conflicts are.

1. Constraints have to be classified at the highest level of the data they touch.

2 Certain constraints. for example key uniqueness produce covert channels that have to be blocked by techniques. such as poly-instantiation [DENN86]

3 Denial of service can result in a covert channel. therefore it is necessary to specify actions that are adequate as response for a violation of a given constraint. The definition of a constraint at database design time should include the definition of the proper action to be taken upon violation of the constraint.

4 The design of a database has to include the design of the database consistency constraints defined on that database. Database design tools have to include support mechanisms that help the database designer to spot potential conflicts between consistency and security constraints

5 When security and other database consistency requirements conflict. the precedence of constraints has to be established. For sensitive applications. security should have precedence over other consistency considerations. Other applications may have different requirements. Note that when constraints are selectively enforced. i.e. when not all constraints are satisfied at all times. there may be serious problems raised by the presence of suspect or inconsistent data. If this data is read by later transactions or by the constraint enforcement module. there may be a cascading effect. Whole segments of the database may become questionable. In an application designed to selectively enforce constraints. it is imperative to specify the actions to be taken when suspect data is introduced.

6 Compatibility of consistency constraints and security policies should be tested as far as possible. at database design time.

7. Given the trade-offs between database consistency and security and the need for evaluation of compatibility it appears most convenient to have a common format for security and other consistency constraints. Constraints should be expressible in some language based on logic for conversion into theorems and automatic proof of correctness

While pointing out the advantages of expressing security policies as database constraints. we must also emphasize the critical differences between security classification data and other data. The security classification information. or "tag." that is associated with each controlled data item (be it a whole data type. a single record. or an individual field) has a special role. The values of these tags determine access to the associated data items. When the tags are changed. so are the access rights to data in the database. Access to the tags is. of course. subject to the same controls as other data accesses. However. updates to classification tags must be even more carefully limited and synchronized to prevent even momentary anomalies in the access rights.

It is too restrictive to require that all security policies employ completely static classification tags. For a secure DBMS to successfully meet the demands of a given application. changes in security classifications must be allowed. In addition to the strong requirement to handle sanitizations and classification downgradings. security classifications must be able to change as situations change. While it is possible to define a security policy that assumes that tags only change as the result of infrequent specialized transactions. such a policy must be very restrictive. A less restrictive policy that allows for special circumstances. on the other hand. may need to mandate changes in the tags to reflect the special actions taken.

A policy with static integrity classifications, for example, may not allow a subject to write data with an integrity tag value lower than that of the data read. Another policy may allow this "write-down" in special cases (e.g., if requested by a highly trusted user), but might increase the integrity tag value of the written data to match that of the read data. In this case, the policy itself would specify the action to be taken.

In the next section, we give some examples of static and dynamic security and integrity policies and show how they might be expressed as database constraints. We define a model that divides the standard notion of "integrity" as used in the security literature into two parts, using this extra precision to define a number of possible policies. By dividing the integrity classification tag into two tags, we can allow changes to a single tag while requiring the other to remain static. This type of policy can accommodate some useful special cases without relaxing important integrity requirements. In the next section, we discuss the significance of dividing "integrity" into two components.

3.4 An Example of Security and Integrity Policies

3.4.1 Basic and Restrictive Policies

To illustrate the use of weak and strong constraints as statements of security and integrity policies, we have formulated a sample policy that is similar to the models in [BIBA77, FEIE77, KSOS78] We have relied on the formulations in [LAND81]. These models ascribe two types of ratings, or sensitivity levels, to users, functions, and data: security and integrity. We believe that the notion of integrity in these systems is very broad, and so our model explicitly divides this notion into two parts. Therefore, our policy involves three types of ratings for all controlled users, functions, and data.

"Integrity," as it is used in the security literature, is a measure of the value and criticality of the data. Data with high integrity must be protected from compromising write operations. There are, however, two aspects to a data item's integrity. The first is its sensitivity, the amount of damage that could be caused by malicious (or erroneous) modifications. The second is the reliability of the data item as a correct reflection of the real world. The two aspects are often different, and they have been separated explicitly in the model below.

A discussion of the ways in which the separation of these aspects leads to more precise security policy statements follows the definitions of the three types of ratings used in our model.

Types of Ratings

1. Read-sensitivity (Security): a measure of the amount (type) of damage that could be caused by improper disclosure.
2. Write-sensitivity (part of the notion of "integrity" as it appears in the security literature): very similar to read- sensitivity -- a measure of the amount (type) of damage that could be caused by improper modification.
3. Trustworthiness (also usually included in standard literature as part of "integrity"): a measure of the reliability of the user, function, or object.

If a subject (a user or function) has a high read-sensitivity rating, the understanding is that the subject is not likely to disclose secure information improperly. Then, such a subject might be trusted to read highly secure data and write less secure data (i.e., if it wanted to disclose the highly secure data it could anyway.) Of course, the problem is not so much that the subject might deliberately

disclose data (even to a confederate), but that it might unknowingly use this data in such a way that some other malicious user with a (deservedly) lower security rating might get or infer the data. This, then, is where a separate trustworthiness rating comes in. Think of trustworthiness here as being an indicator of "smartness" or "subtlety." Although it seems highly impractical and imprudent to rate users' "smartness" as such, it might actually be that it is useful to assign users trustworthiness ratings based on their knowledge of the system. Thus, those with a very good idea of (and access to) large portions of the database would likely have high trustworthiness ratings. The trustworthiness ratings might be based in part on understanding of the workings of the DBMS and/or on depth of understanding of the application.

The key to this model is that a subject or object may well have different values for the three types of ratings. For example, an employee's salary might have a read-sensitivity rating of CONFIDENTIAL but a write-sensitivity rating of TOP SECRET. Missile targets could have very high read- and write-sensitivity ratings, meaning that improper disclosure or modification could be extremely detrimental to national security, but a particular entry could have a relatively low trustworthiness rating (meaning, perhaps, that the target was picked on the basis of incomplete intelligence reports). We show how different policies can exploit this distinction between write-sensitivity and trustworthiness.

The three types of ratings may be expressed using three different scales, or domains of values, or they may be expressed using the same scales. The choice depends largely on the policies and procedures already in place and on the system-enforced policies to be defined. In some applications read- and write-sensitivity and trustworthiness may be closely related; for example, information that is regarded as highly trustworthy may be more write-sensitive than its less trustworthy counterparts. In this case trustworthiness ratings may be some function of read- and write-sensitivity ratings. In other applications, complicated security policies may involve other combinations or cross-comparisons, suggesting a single scale for all ratings.

We use r-, w-, and t-rating to abbreviate read-sensitivity, write-sensitivity, and trustworthiness ratings, respectively.

To demonstrate the use of these types of ratings, we begin with the formal properties described and used by the KSOS project [KSOS78], as described in [LAND81], and reformulate them in terms of r-, w-, and t-ratings. The reformulated properties highlight the different roles of write-sensitivity and trustworthiness in what was originally the single "integrity" quality. We then discuss some useful expansions and modifications of these policies and show how they can be expressed through some simple additions to the basic constraints.

The policies described below ensure four basic, and fairly strict, restrictions.

1. READ AUTHORIZATION: A subject has read access to an object only if the subject's r-rating is greater than or equal to the r-rating of the object. (Essentially the simple security property of Bell and LaPadula [BELL74].)
2. WRITE AUTHORIZATION: No subject has write access to any object that has a w-rating greater than the w-rating of the subject. (The exact dual to the preceding property.)
3. NO WRITE-DOWN: A subject can modify an object O2 in a manner dependent on an object O1 only if the r-rating of O2 is at least that of O1.

4. TRUSTWORTHINESS: A subject can modify an object O2 in a manner dependent on an object O1 only if the t-rating of O1 is at least that of O2. (Information may not flow from less trustworthy data to more trustworthy data.)

3.4.2 Basic Security Constraints

We will use r(f), w(f), t(f) to refer to the read-sensitivity, write-sensitivity, and trustworthiness ratings (respectively) of a function reference f. Similarly, r(v), w(v), and t(v) refer to the ratings of a state variable v. In what follows, we will assume that the three types of ratings use the same domains. We use <= to refer to the partial ordering over this domain. In an application with different domains for the three ratings, specialized comparators would of course have to be used.

READ CONSTRAINT: If function reference f depends on state variable v, then

r(v) <= r(f), AND t(f) <= t(v).

Note that in the read constraint the KSOS use of integrity is replaced by our use of trustworthiness (*not* write-sensitivity).

WRITE CONSTRAINT: If function reference f may affect the value of state variable v, then

r(f) <= r(v) -- same as in KSOS
AND w(v) <= w(f) -- as in KSOS, buy divided into write authority
AND t(v) <= t(f) -- and trustworthiness here

READ/WRITE CONSTRAINT: If function reference f may cause the value of state variable v2 to change in a way dependent on state variable v1, then

r(v1) <= r(v2) -- same as in KSOS
AND t(v2) <= t(v1) -- trustworthiness for KSOS' "integrity" here

The necessary read and write authorizations are covered by applying the preceding two constraints to this situation. This constraint is very similar to the read/write constraint in the KSOS model, with the notion of integrity divided into write-sensitivity and trustworthiness. Some variations that lessen the restrictions of this constraint are discussed below.

3.4.3 Policies Involving Dynamic Ratings

One major way of changing the security and integrity policies of an application is to allow system-invoked changes to objects' ratings in some cases. In these cases, the system would allow selected operations that would have been blocked by the strict policies, but it would also ensure that relevant ratings are changed.

For example, rather than the strict controls on the flow of trustworthy data given above, it may be desirable in some applications to allow less trustworthy data to affect more trustworthy data. In these cases, it is probably reasonable to make trustworthiness a more dynamic property. The constraints used to enforce the policies might have associated actions that would automatically change an object's t-rating. The policies used to determine the new t-rating would be expressed as a mathematical formula. The trustworthiness of a changed variable may be the minimum of all relevant trustworthiness levels, for instance, or it may be higher if the function reference that changed it had a

high trustworthiness rating (The assumption is that such a trustworthy function would not use less trusted data unless it had other reasons to do so)

The framework we propose is well-suited to this approach In the sample constraints given above, each constraint had an (implicit) associated action to disallow an operation that violated the constraint This same mechanism can be used to define automatic changes to ratings by relaxing the definitions of the constraints somewhat, and adding new actions An illustration is given in the following version of the read/write constraint and its associated action.

READ/WRITE CONSTRAINT: If function reference f may cause the value of state variable v2 to change in a way dependent on state variable v1, then

```
r(v1) <= r(v2) -- same as in KSOS
r(v1) <= r(f) -- read authorization included explicitly
                 here for completeness of example
w(v2) <= w(f) -- write authorization included explicitly
                 here for completeness of example
ACTION  t(v2) := min(t(f),t(v1),t(v2))
```

3.4.4 Policies Involving Trusted Agents

The division of the "integrity" rating into w- and t-ratings allows for the definition of policies that are more responsive to special situations Here we discuss policies that include special privileges for trusted users, or agents.

A subject with a high r-rating, but a lesser t-rating, might not be allowed to "write-down". If the subject has a high t-rating, though, it may be allowed to write-down. The trustworthiness rating may be a way of capturing the different types of write functions -- one function might simply copy some data (this is the quintessential "write-down"), while another function might retrieve some information for a general, and record the general's changes to other information. The assumption here is that the general has sufficient knowledge of the database to ensure that the data written will not betray the higher level information retrieved earlier and the software correctly implements these changes. The t-rating of a function will typically depend on the t-rating of the user invoking the function and the degree of verification of the implementation of that function.

The version of the read/write constraint given here illustrates one policy that allows trusted functions to violate the trustworthiness policy. This formulation is meant only to suggest how such a policy might be expressed: other variations are of course possible That is the goal of describing a general framework for the enforcement of security and integrity policies.

READ/WRITE CONSTRAINT: If function reference f may cause the value of state variable v2 to change in a way dependent on state variable v1, then

```
      w(v2) <= w(f)
AND   r(v1) <= r(f)
AND   { [r(v1) <= r(v2) AND t(v2) <= t(v1)]
        OR max(t(v),t(v2)) <= t(f)  }
```

This constraint expresses the policy that the function f must *always* have a sufficiently high w-rating to write v2 and a sufficiently high r-rating to read v1. Furthermore, information should not flow from more r-sensitive to less r-sensitive, nor from less trustworthy to more trustworthy, unless the function is at least as trustworthy as the most trustworthy data involved.

3.4.5 Use of Weak Constraints

The policy just described involves a relaxation of the strict constraints, but only in the case of sufficiently trustworthy functions. This is exactly the type of situation mentioned earlier in the discussion of strong and weak constraints. Up to this point, all the constraints defined have been treated as strong constraints. This is natural, in that security and integrity policies must be strictly enforced if they are to be useful at all. However, when policies are defined to allow special cases (such as trusted functions), the role of weak constraints becomes very important.

Perhaps surprisingly, the inclusion of weak conditions in our framework serves to strengthen, rather than weaken, control over a system's security and integrity. The key point here is to view weak constraints not as unenforced constraints but as descriptions of events to be *monitored*. The interplay of strong and weak constraints is critical: all the necessary restrictions are expressed as strong constraints and therefore strictly enforced, and any special cases or questionable situations are expressed as weak constraints and therefore monitored. The way that these situations may be monitored depends on the application. When a weak constraint is violated, for example, the relevant information may be added to a log, the changed data may be marked, or a message may be immediately sent to a responsible authority.

In the example of the policy involving trusted functions, it may be desirable to keep a log of the function id (including user id), data involved, and timestamp of *all functions which were allowed to act only because of their trusted status*. This log could then be reviewed periodically to find suspicious patterns. If this log were itself part of the controlled data in the database, certain types of additions to the log could themselves be described as weak constraints, and these constraints in turn could be responsible for sending warning messages to authorities.

To illustrate the way in which weak constraints could be used to monitor specific situations, suppose we express a policy using the strong read/write constraint given in the previous section.

STRONG READ/WRITE CONSTRAINT: If function reference f may cause the value of state variable v2 to change in a way dependent on state variable v1, then

$$
\begin{aligned}
& w(v2) <= w(f) \\
\text{AND } & r(v1) <= r(f) \\
\text{AND } & \{\, [r(v1) <= r(v2) \text{ AND } t(v2) <= t(v1)] \\
& \text{OR } \max(t(v1), t(v2)) <= t(f) \;\}
\end{aligned}
$$

This strong constraint will ensure that no violations of this policy are allowed. However, we might want to augment our system by monitoring which functions would have violated the policy had they not been sufficiently trustworthy. Therefore, we would define the following weak constraint.

WEAK READ/WRITE CONSTRAINT: If function reference f may cause the value of state variable v2 to change in a way dependent on state variable v1, then

$$
\begin{aligned}
& \text{NOT } [r(v1) <= r(v2) \text{ AND } t(v2) <= t(v1)] \\
\text{AND } & \max(t(v1), t(v2)) <= t(f)
\end{aligned}
$$

Notice that there is no need to do any testing of read and write authorizations in the weak constraint: read and write authorizations are strict requirements and are enforced through the strong constraint. The only situations ever tested by the weak constraint are those that satisfy the strong constraint. To reiterate. the strong constraints are the barriers to improper reads and writes. while the weak constraints are observers of particular allowable events.

4. Security Issues and Transaction Processing

In this section we study the interaction of security constraints with the interleaved execution of transactions We highlight the new problems that surface when the interplay of security constraints and processing of transactions is investigated for a variety of contexts. centralized database systems and distributed database systems with and without data replication Our aim is to use the common framework that has been proposed in this report (for specification. verification and enforcement of database consistency constraints as well as security constraints) for modeling the synchronization mechanisms that take into account the security as well as data correctness aspects

We first introduce the notions of a transaction. atomicity. commit and then review several concurrency control mechanisms that are currently used for the interleaved execution of a set of transactions in the centralized database context. We then discuss the interplay of security issues with transaction processing in centralized databases. The discussion is extended to the distributed case. and in the last subsection we present flexible concurrency control mechanisms and discuss some of the implications of relaxing basic consistency notions

4.1 Transaction Model for Specifying Consistency under Concurrent Execution

A database system may be viewed as a triple <D. C. T>. where D is the set of database entities. C is the set of constraints over D. and T is the set of all programs that may access D A program is a sequence of actions. A program when executed alone is assumed to preserve consistency. that is. transform a consistent state of D to another consistent state In order to run programs with maximal concurrency (to increase performance). actions from several programs are interleaved and allowed to simultaneously operate on the entities of D Unless some control is exercised to restrict the way in which concurrently executing actions of several programs are interleaved. the actions may interfere. resulting in various anomalies (lost updates. inconsistent retrievals. inconsistent updates) and leave D in an inconsistent state

Hence. it is important. in the context of a database. to guarantee that an action or a sequence of actions (referred to as a program above) appears to be executed in isolation and in its entirety or not executed at all This concept is expressed using the notion of atomicity. A *transaction* is defined as a program which is executed atomically. that is

a) the transaction accesses shared data without interference from other transactions. and

b) if the transaction terminates normally. then all its effects are made permanent. otherwise it has no effect at all.

Note that a transaction is assumed to satisfy the consistency constraints at the beginning and after the transaction executes to completion (vacuously true for abnormal termination under atomicity) Note also that one purpose of grouping a sequence of actions into an atomic transaction is that the consistency constraints may be violated temporarily before the termination of the transaction For example. in an electronic fund transfer the account is debited before the other account is credited (or vice versa) and the condition that the sum of account balances before and after the transaction is the same is temporarily violated. Commit is an operation executed by the DBMS that indicates the

normal completion of a transaction and that all its effects should be made permanent. Similarly, abort signifies an abnormal termination of a transaction and that all of its effects should be obliterated.

In a multi-user system it is unreasonable to assume that the transactions are executed in isolation one after the other (serially) from the performance point of view. However, if transactions are executed concurrently interleaving actions from a set of transactions in an arbitrary manner, the actions are likely to interfere with each other resulting in a database state that does not belong to the set of valid states that result from executing the transactions serially. At the same time if a transaction were to terminate abnormally (program failure, system failure, intentional abort), the effect of that transaction needs to be undone in some way.

Hence, the notion of atomic execution of transactions gives rise to two problems, namely, the concurrency control problem and the recovery problem. Concurrency control ensures that each transaction submitted to the system executes atomically. This is achieved by controlling the interleaving of concurrent transactions, to give the illusion that transactions execute serially, one after the next, with no interleaving (and hence no interference) at all. On the other hand, recovery control monitors and controls the execution of each transaction so that the database includes only the results of transactions that run to normal completion. If a failure occurs while a transaction is executing, and the transaction is unable to finish executing, then the recovery control must wipe out the effects of the partially completed transaction. Moreover, it must ensure that the results of transactions that do execute to completion are never lost.

Security issues influence both the strategies chosen for concurrency control as well as for recovery. We concentrate mostly on concurrency issues in this section without going into the details of recovery control.

A variety of concurrency control mechanisms are currently used for the interleaved execution of a set of transactions. Proving the correctness of a given concurrency control mechanism (such as locking, timestamp ordering etc.) requires showing that the mechanism produces only serializable execution histories. An execution is *serializable* if it produces the same output and has the same effect on the database entities D as *some* serial execution of the same transactions. Since serial executions are correct (assuming that the individual transactions are correct) and serializable executions correspond to a serial execution, it follows that serializable executions are correct.

The theory of serializability is concerned with achieving this illusion without executing transactions in isolation. Serializability is the definition of correctness for concurrency control in database management systems. It gives precise rules and conditions for the correctness of concurrent execution of several transactions. A concurrency control algorithm is correct if all of its possible executions are correct. Since execution of transactions is modeled by histories (also known as logs or schedules) and serializability conditions are stated in terms of histories over transactions, we examine the characteristics of histories and transactions below.

4.1.1 Transactions

A transaction T_i is a partial order with ordering relation $<_i$ where

1. T_i is a subset of $\{r_i[x], w_i[x] \mid x \text{ is a database entity}\} \cup \{a_i, c_i\}$.

2. a_i is a member of T_i iff c_i is not a member of T_i.

3. if t is c_i or a_i (whichever is in T_i), for any other operation p in T_i, $p <_i t$.

4. if $r_i[x]$, $w_i[x]$ is a member of T_i, then $r_i[x] <_i w_i[x]$ or $w_i[x] <_i r_i[x]$

The above model of a transaction from [BERN87] captures the database operations (such as read (r), write (w), commit (c) and abort (a)) and not other details of a transaction such as initial values, assignments, etc. The notation o[x] stands for the operation o on the database entity x, where o can be any of the database operations mentioned above. We will later introduce additional database operations.

4.1.2 Histories

A *history* indicates the execution order along with the interleaving of the actions of a set of transactions. A history is again a partial order, as the operations can be executed in parallel. A history is also required to preserve the order of operations as specified by an individual transaction.

In addition, a history captures the order of all conflicting operations that appear in it. Two operations are said to *conflict* if they both operate on the same database entity and at least one is a write. Thus r[x] conflicts with w[x] whereas w[y] conflicts with both r[y] and w[y].

Formally, a history H corresponding to a set of transactions $T = \{T_1, T_2, \ldots, T_n\}$ is a partial order with ordering relation $<_h$ where:

1. H is the union of elements in T_i
2. $<_h$ is a superset of the union of $<_i$, and
3. for any two conflicting operations p, q belonging to H, either $p <_h q$ or $q <_h p$

A history represents a possibly incomplete execution of transactions.

4.1.3 Serializability Theorem [BERN87]

It is possible to determine whether a history is serializable by analyzing the graph derived from the history, called the *serialization graph*. Let H be a history over $T = \{T_1, T_2, \ldots, T_n\}$. The serialization graph (SG) for H, denoted by SG(H), is a directed graph whose nodes are the transactions in T that are committed in H and whose edges are all $T_i \rightarrow T_j$ ($T_i <> T_j$) such that one of T_i's operations precedes and conflicts with one of T_j's operations.

A history H is serializable iff SG(H) is acyclic.

4.1.4 The Effects of Serializability

There are basically two approaches to concurrency control, each addressing different sets of requirements. The first approach, serializability, was defined above. The advantage of this approach is that the database consistency can be guaranteed by requiring that each transaction individually preserves consistency. The disadvantage of serializability is that it requires analyzing each transaction as it is submitted, and backing-out (undoing the effect of the transaction) or blocking (disallowing) transactions which would conflict with other transactions. Serializability also often has an adverse effect on transaction throughput, because a transaction must wait until a conflicting transaction commits and releases locks before it can execute. Deadlock detection and resolution and deadlock avoidance are other problems that coexist with concurrency control mechanisms proposed in the literature

An alternative approach is to allow transactions to run non-serializably, providing high transaction throughput. An example of this is one developed by CCA for the RADC- and DARPA-sponsored SAC C3 project [SAR185]. Under this approach transactions are never blocked due to concurrency controls. Interdependent transactions may be run concurrently on different processors, and database inconsistencies may result. Should any inconsistencies arise, each processor analyzes the log of executed transactions and takes appropriate steps to restore consistency. These steps may involve undoing and redoing some transactions or running new transactions which compensate for the inappropriate actions of other transactions. When serializability is not enforced, although database consistency is not guaranteed at all times (unlike the previous approach), it is guaranteed eventually. The advantage of the approach is that it allows a large number of transactions to be processed without the immediate imposition of concurrency controls, and consequently it allows transactions to be processed in a distributed environment even when some processors cannot communicate with each other

4.2 Concurrency Control Mechanisms

Various mechanisms have been developed to enforce the conditions of the serializability theorem during the execution of a set of transactions. They include locking, timestamp ordering, serialization graph testing, optimistic strategy and synchronization with eventcounts and sequencers. In each case it has been shown that a scheduler, using a particular technique, generates only serializable schedules. Informally, a scheduler is a program or a collection of programs which controls the concurrent execution of transactions. It exercises control by restricting the order of execution of actions (read, write, commit, abort) associated with concurrently running transactions. The goal of the scheduler is to order and execute the actions of transactions in such a way that the resulting schedule is serializable (and recoverable).

Among the available techniques for concurrency control, locking has been studied extensively and has been widely used in extant database management systems. Below, we briefly examine some of the strategies.

4.2.1 Locking

Locking is a mechanism commonly used to solve the problem of synchronizing access to shared data. In this scheme every database entity has associated with it a unique lock. A transaction must lock an entity before it can access it. Basically by locking an entity, a transaction ensures that it is inaccessible to other transactions while it is being modified. If an entity is not already locked, then a transaction can lock it with a 'lock' action. If a transaction attempts to lock an entity which is already

locked. then it either waits for the entity to be unlocked. aborts itself. or preempts the transaction holding the lock. A transaction can relinquish the lock on a entity by an 'unlock' operation.

Transactions access database entities either for reading or for writing them. Hence two types of locks are assumed. namely. read locks and write locks. Let Readlock[x] and Writelock[x] denote the read lock and the write lock on database entity x. respectively. Similarly. Readunlock[x] and Writeunlock[x] are used for unlocking the database entity x. The definition of a transaction is extended to include the lock and unlock operations.

Two locks $pl_i[x]$ and $ql_j[y]$ conflict if x = y. i <> j. and the operation p and q are of conflicting type. That is. two locks conflict if they are on the same data item. they are issued by different transactions and one or both operations are write locks.

4.2.1.1 Two Phase Locking. In this scheme. restrictions are imposed on the acquisition and relinquishment of locks in order to produce only serializable histories by a scheduler which follows the restrictions. The rules for two phase locking are.

1. Before scheduling an operation the scheduler tests if $pl_i[x]$ conflicts with some $ql_j[x]$ that is already set. If so. it delays $p_i[x]$. forcing T_i to wait until it can set the lock it needs. If not. the scheduler sets $pl_i[x]$. and then schedules $pl_i[x]$.
2. Once the scheduler has released a lock for a transaction. it may not subsequently obtain any more locks for that transaction on any database entity.

The first rule prevents two transactions from concurrently accessing a database entity in conflicting modes. It is assumed that setting and releasing the locks themselves are guaranteed to be atomic. The second rule enforces the two phase property - a growing phase in which locks are acquired without releasing any lock and a shrinking phase which starts with the first unlock operation and during which a transaction releases locks without acquiring any more locks.

It can be shown that all histories generated by any scheduler which conforms to the rules of two phase locking stated above are serializable.

4.2.2 Non-Locking Strategies

Locking protocols involve the overhead of requesting a lock every time a database entity is accessed. Transactions may get blocked and have to wait for the locks to be released by other transactions. As an alternative to locking. a variety of synchronization protocols have been proposed which produce serializable schedules.

4.2.2.1 Timestamp Ordering. A timestamp is a unique system wide number which is assigned to a transaction and is chosen from a monotonically increasing sequence. Usually it is generated by a clock (with an appropriate least count) or a number that is incremented at the time of its generation.

A timestamp is essentially used in two ways. First. it is used to determine the currency or outdatedness of a request with respect to the data it is operating upon. Second. it is used to order events (requests) with respect to one another.

In timestamp ordering, a unique timestamp (ts) is assigned to each transaction as it enters the system (or when its first operation is scheduled). The timestamp associated with a transaction is attached to each operation of the transaction. Therefore the timestamp of an operation $o_i[x]$ is nothing but the timestamp of the transaction Ti. A scheduler following the timestamp ordering protocol simply orders conflicting operations according to their timestamps. More precisely, it enforces the following rule:

if $p_i[x]$ and $q_j[x]$ are conflicting operations then $p_i[x]$ is processed before $q_j[x]$ iff $ts(Ti) < ts(T_j)$

It can be shown that if H is the execution history produced by a scheduler obeying the above rule, then H is serializable

4.2.2 Optimistic Protocols. Optimistic approaches to synchronization offer maximum concurrency with the underlying assumption that the conflicts among transactions are rare and conflicts are exceptions rather than routine. A transaction always executes (albeit tentatively) concurrently with other transactions without any synchronization check. However, before a transaction s updates are made final (visible to other transactions), it is certified (or validated). It is the certification phase that determines whether there is a conflict and the transaction should be aborted and backed out or if it can be committed.

In most of the optimistic methods [KUNG81] the execution of a transaction is divided into three phases, read phase, validation phase and write phase. Read phase corresponds to the execution of a transaction making local changes. Validation phase precedes the write phase (making changes visible to other transactions). Improvements, in the form of transaction analysis, have been proposed to classify transactions into read-only and update categories.

4.2.3 Deadlock and Livelock

These situations occur potentially in systems where some form of synchronization is used in executing transactions (processes in general) concurrently. *Deadlock* is a situation in which two or more transactions are in simultaneous wait state, each waiting for one of the other to release locks before it can proceed. *Livelock* is a situation where a transaction T waits forever, waiting for a lock, even though there are unlimited number of times when T might have been given the lock it needed. Livelock is also equated with cycle restart where a transaction is aborted every time it tries to acquire a lock only to be restarted to encounter the same situation.

Two phase locking does not guarantee freedom from deadlocks. A deadlock condition is completely characterized by wait-for graphs [HOLT72, RYPK79]. Two philosophies are generally used in overcoming deadlocks: deadlock prevention and deadlock detection and resolution. Timestamp ordering and optimistic protocols do not have to deal with the problem of deadlock as deadlock is impossible in these cases.

Livelock can occur in all the concurrency control mechanisms discussed above and must be avoided using suitable techniques

The above succinct summary is provided for the sake of completeness. A parallel study on temporal properties is being conducted along with this study which will address deadlock and livelock issues in greater detail

4.2.4 Verification of Serializability

Verification of serializability even by model-theoretic means, has turned out to be a non-trivial task as illustrated by the proof of the SDD-1 protocols [BERN80] Formally specifying and verifying serializability appears even more difficult. As a first step, we make an attempt to formally specify a well-understood and widely used concurrency control mechanism, namely, two-phase locking using an experimental extension of the specification language SYSPECIAL This result is presented in Section 5 In that section we also discuss the experimental extensions that were introduced to SYSPECIAL to model the two-phase locking protocol, namely, a multilevel specification capability and the notion of a trace

4.3 Security Issues in Centralized Transaction Processing

As discussed in the above sections the major thrust of transaction management has been from the viewpoint of maintaining database consistency by avoiding interference caused by arbitrary interleaving of transactions Strategies for synchronization have been developed for maximizing transaction rate throughput and making the database highly available Security constraints and a need to enforce them in an environment of shared access add a novel perspective to the problem of simultaneous processing of transactions Since security considerations were not a factor that was considered in the design of most concurrency control mechanisms it is useful to analyze their behavior under the additional requirements of security We shall assume in this discussion that security policies can be expressed in the form of constraints as illustrated in Section 3

In contrast to the abundant work on protocols for synchronization and their correctness little work has been reported in analyzing security issues during concurrent execution of database transactions As demonstrated in the literature on security, e.g [DENN86 DENN79, DENN85], security breaches can be very subtle and their total avoidance may not be possible Special engineering solutions, such as encapsulation of certain functions, may be required. Even so, a complete elimination of security threats through such mechanisms as inference, may not be possible

Recent work [DENN86] on security issues pertaining to relational databases has concentrated on mandatory security policies Multilevel derived relations (defined using the notion of a view supported by a majority of systems based on the relational model) are presented as a means of separating the base object (which is a base relation) and accesses to it through a reference monitor which is protected Various other aspects of relational databases such as the need for poly-instantiation, and the need for the classification of consistency constraints themselves are also discussed

In this section we focus on the concurrency control mechanisms and try to identify their shortcomings from a security point of view We try to identify what assumptions that underly the traditional concurrency mechanisms are not valid in a secure environment and whether these mechanisms can be recast under security-related constraints

4.3.1 Correctness with Respect to Security and Consistency

Concurrent processing of transactions satisfying security conditions/protocols poses a range of problems that may void techniques traditionally used for enforcing correctness of database constraints

The problem of correctness for security can be expressed as two subproblems:

a) Correctness of an individual transaction from the security viewpoint. and

b) Correctness (with respect to security) when consistency and security preserving transactions are interleaved in an arbitrary manner (the interference problem)

4.3.1.1 Correctness of Individual Transactions. The first problem is that of showing that an individual transaction does not violate any security constraints/policies if it is executed individually as an isolated transaction. Before any effects of concurrent execution can be discussed. the correctness of individual transactions has to be guaranteed.

The notion of atomicity can be extended to include security correctness in a straightforward manner. This is the classical program verification problem and some of the techniques and automated tools developed for that purpose are applicable here. Verification of a transaction [GARD79. SHEA86] before its execution presupposes that the security requirements can be stated as pre- and post- conditions using a formal declarative language (such as first order logic). It is evident from the discussion of security policies (in section 3). that our approach enables us to capture security policies (mandatory as well as discretionary) as constraints on an appropriate model of the database thereby making it amenable to verification using the same tools for verification of consistency and security. Such a common framework is necessary if one intends to show compatibility between the security and the consistency constraints.

It should be pointed out that any data access including read (not just the update) is sensitive to security restrictions. Therefore. existing tools used for verification of individual transactions [SHEA86] have to be modified.

It is obvious that not all restrictions can be stated as pre- and post- conditions that can be verified statically at compilation time. As in database correctness. it is essential that one can express constraints that have condition(s) and compensatory action(s) associated with the conditions which need to be applied when the conditions are violated. Situation action rules and triggers belong to this category. These provide alternatives to the default action of aborting a transaction once the condition is violated. As an example. if a referential consistency constraint were to refer to two attributes (in different relations) with different security classifications. insertion of new tuples may have to be handled using poly-instantiation which can be specified as the action.

An approach to the verification of security constraints is to partition the constraints into two classes: those that are verifyable on the database state at compile time and others that have to be verified at run time. Techniques that have been developed for program verification can be employed for verifying compile time constraints. Run time verification of constraints (specified as situation action rules or triggers) is currently an active database research area. Run time verification is complicated by its interaction with synchronization mechanisms as well as recovery aspects Many of the proposed solutions are application-specific. Hence. one has to approach this problem with caution. but the techniques developed for database correctness will be extremely useful in the context of security At the end of this section we discuss some of the implications of triggers and the corresponding actions further.

A second class of problems is derived from intermediate feedback given during the execution of a transaction. Any response (output from the transaction, querying by the system, feedback produced by a transaction, or any external action) produced during the execution of a transaction before its normal completion may provide a channel which may be used to compromise security. This entails that the output of an active transaction (which has not committed yet) may have to be either seriously limited, delayed or undone in a manner consistent with security policies. This has ramifications in interactive environments if the transactions cannot be verified to be correct (with respect to security) prior to the start of the transaction.

There has been considerable interest in the area of informative answering [JANA81] wherein the user is guided by the system which provides answers that are meaningful (have more context and information) based on inferences the system draws about the intention of the user. The "extended" queries are obtained via query modification techniques from semantic information usually stored in the form of constraints. The opposite problem of "secure answering" is related to the inference problem that has been identified in secure environments and has received less attention. An interesting possibility would be to use the unified framework for consistency and security together with techniques employed in informative answering to identify better the threats derived from feedback to users and the pattern of queries made by a user.

In order to analyze the effect of outputs, it is useful to classify intermediate actions as internal actions (that are reversible) and external actions (in turn divided into those that can be compensated and those that are irreversible). Internal actions, those that did not produce any visible action, can always be undone or compensated. Some external actions can be compensated for, although the cost of doing so may be variable. For example, compensating for overbooking a flight in economy class by upgrading a passenger to first class is inexpensive. Having to blow up a missile that was fired by mistake is not. A case of irreversible external action would be the firing of a missile that has no self-destruction mechanism. Unfortunately, external actions that violate security fall into the class of irreversible external actions.

Internal actions of a transaction can be rolled back. External actions can be suppressed until their effect is clear. For example, it is possible to suppress the sensitive parts of the output. An alternative approach is to delay any feedback produced by a transaction until the transaction commits. Though this approach is suitable for "batch" transactions, for transactions requiring user inputs, it is essential to determine whether the output is security-sensitive or not.

The notion of transaction classes and their analysis may provide a means for determining whether to suppress or delay feedbacks produced by a transaction. If it is possible to classify transactions based on the outputs and feedbacks embedded in the transaction, then policies can be stipulated for a transaction rather than individual actions. This presupposes that a preanalysis of a transaction is viable, as discussed before under transaction-class analysis. This class concept can also be used to determine the mix of transactions that are being executed concurrently. The work of Stemple [SHEA86] is relevant in this context since it provides the means to verify a transaction with respect to a set of complex (but static) consistency constraints at compile time. If a common framework for consistency and security constraints is used, then it is conceivable to use the same tools for verification of consistency and security constraints.

4.3.1.2 Correctness of Interleaved Execution of Transactions.

The second subproblem is that of defining synchronization mechanisms that eliminate interference with respect to security when transactions are executed concurrently. Interference can arise both from dynamic changes in the security rating and through normal access (read and write) to common data Dynamic changes to classification can happen through a) sanitization and b) explicit change of classifications (reclassification). It may be unreasonable to assume that all sanitization and reclassifications take place in isolation in a shared database environment. Assumptions to the contrary will simplify the interference problem but will not eliminate it.

To guarantee database consistency. the order of execution of transactions is not critical as long as each transaction is consistent and interference among transactions is eliminated. Hence. serializability does not have to guarantee the order of the execution of transactions. However. this assumption may not be valid in secure environments in which on-line modification of any aspect of the security rating of an object (r.w. or t) is required. In this case. alternate concurrency control mechanisms will be needed that preserve the exact order of execution of transactions. for example. time-stamping mechanisms.

The very act of aborting a transaction may serve as a covert channel conveying some information about the database entities accessed by that transaction (a change in the classification of data during interleaved execution may abort a transaction that executed to completion earlier. or even the insertion of a new tuple that impacts a constraint that has to be satisfied by that transaction).

It has been pointed out [Reed79] that the synchronization protocols using read- and write-locks are potential information channels through which information can be tapped out by locking (or trying to lock) data in a given pattern. This observation has serious ramifications. as most current implementations of database management systems use locking as a synchronization mechanism and a large number of studies have analyzed variations of this generic method and their performance characteristics. An alternative protocol based on event counts and sequencers [Reed79] has been proposed to overcome this problem. Unfortunately. this method does not guarantee that a reader process will not be starved. To guarantee that a reader will not be starved. again an arbiter process would be required. It appears that specially engineered solutions are required.

The use of locks acting as an information channel is predicated on the assumption that the locking pattern is visible and somehow accessible to an unauthorized or malicious subject. One way to overcome this problem is to encapsulate the monitor that manages the locking protocol and the lock table itself in a secure module thereby making the locking pattern invisible. Also. it implies that the operations of locking and unlocking not be available as primitive operations at the user interface level. nor that any information about the cause for an unsuccessful termination of a transaction be transmitted to the user.

There is a parallel between optimistic protocols (described in Section 4.2.2.2) and the secure readers-writers problem using eventcounts and sequencers for synchronization. In both cases serialization is relaxed and transactions execute without getting blocked initially. If a conflict is detected. then one of the transactions is aborted (the reader in the case of event-counts). However. since this method also depends on an arbiter during the validation phase. an engineering solution that isolates the validator may be required. The advantage over locking protocols may be mostly a reduction of channel bandwidth. not the complete elimination of the channel.

Both sequencers and optimistic protocols can benefit from transaction-class analysis. During transaction-class analysis. similar transactions are studied as a class to identify other transaction classes with which a given transaction-class is compatible. Compatibility in this context means that the generic transactions touch only attributes which are non-conflicting with the attributes that are

touched by the transactions in another transaction-class. Since no instances are analyzed and the analysis can be performed off-line. the potential for tapping out information is greatly reduced. Early attempts of using transaction-class analysis are part of the SDD-1 system [BERN80. BERN81].

It appears that among existing alternatives. the safest approach is transaction-class analysis with concurrent execution of non-conflicting transactions and serial execution of any *potentially* conflicting transactions in time-stamp order. Any transaction that cannot be matched with a preanalyzed class has to be treated as potentially conflicting with all others. Unfortunately. such an approach may greatly degrade performance.

We have discussed above problems that arise during the concurrent execution of transactions in a centralized environment. In the next subsection we extend the discussion to the distributed case

4.4 Security Issues in Distributed Transaction Processing

4.4.1 Types of Distributed Systems

There are a number of reasons for implementing a particular application on a distributed system *one that incorporates and coordinates several distinct processor and storage sites. rather than on a* centralized system: high availability of data. quicker local responses. reliability and survivability. integration of separate existing systems. and accommodation of different local requirements. These *benefits can be realized by careful designs that respond to application requirements* A good distributed system design must take care to.

1. partition data correctly among the different sites.
2. maintain backup or redundant copies of critical data.
3. assign prime processing responsibility for particular requests to most efficiently use the system's resources
4. minimize communications overhead.
5. efficiently analyze requests. decompose them for distributed processing. and integrate the results.
6. make processing and data distribution as transparent to the users as possible. and
7. provide the option of a uniform user interface wherever possible.

There are different types of distributed systems. each suited to different requirements. Some systems are *homogeneous* -- all the local sites have identical (or very similar) configurations. while other systems are *heterogeneous* -- the system consists of sites with different configurations Homogeneous systems present far fewer problems of design and minimize the overhead in assigning processors. decomposing requests. and integrating results: heterogeneous systems allow the distributed system to *incorporate existing systems and to use specialized processors when needed.* Both homogeneous and heterogeneous systems raise many of the same issues with respect to database consistency and security. and in this section we will not distinguish between the types of systems unless specifically mentioned.

Another difference among distributed system designs is the presence or absence of *replicated data*. data which is stored redundantly at more than one site. Again. the choice of whether or not to include replicated data must be made on the basis of the application's requirements. Replicated data can promote higher data availability and survivability. When one site or communication link fails. backup copies of replicated data may be available at a different site. However. the presence of replicated data introduces opportunities for inconsistencies and increases the overhead needed to process updates. These problems are particularly critical for sensitive SDI BM/C3I applications. the very applications for which high availability and survivability are so necessary. This section will focus on the security ramifications of the database consistency problems caused by replicated data.

First we discuss the general database consistency issues raised by distributed systems. then the security issues raised by distributed systems. After a brief overview of existing standard database approaches to distributed concurrency control. we discuss the consequences of a non-serializable approach for a secure system

4.4.2 Database Consistency Issues

Database consistency constraints in a distributed system must be able to take into account data at all the local sites. The definition of consistency for a distributed database may involve complicated relationships among data stored at different sites. Distributed database concurrency control and replicated data make the task of preserving database consistency even more complex in distributed systems.

Concurrency control is one of the fundamental issues in designing and implementing a distributed system. While the key advantage of a distributed database system lies in the ability to simultaneously process queries over the database. the problems of concurrency control found in a centralized system are magnified in a distributed one. In addition to the general concurrency control problems described above for centralized systems. a distributed system introduces communications delays and encourages higher transaction volume

The mere presence of replicated data in a distributed database adds a new database consistency issue: mutual consistency of the replicated copies. The motivation for replicated data was to provide backup copies of data so that local site or communication failures would not halt processing: this goal can only be realized if the backup data are accurate copies. In essence. this means that for every replicated data item there is a new consistency constraint that mandates equality among all the copies.

The interplay of concurrency control and replicated data is important for maintaining database consistency. In weighing the relative importance of strict concurrency control against the need for fast response and simultaneous processing. one of the prime factors is an application's need for strict enforcement of database consistency constraints. Replicated data adds the constraint of mutual consistency. so concurrency controls must be sensitive to the need for propagating replicated updates. The increased processing and communications necessitated by replicated updates increases the overhead and delays introduced by the concurrency control procedures. On the other hand. if concurrency controls are relaxed. the enforcement of mutual consistency and other consistency constraints will also be relaxed. The consequences of these trade-offs are discussed in the next section.

4.4.3 Security Issues

In our framework, security policies are expressed as special database constraints. Therefore, most security issues raised by the use of distributed systems can be seen as special cases of the database consistency issues raised. Security constraints, however, have a special role in a secure distributed system, and so security issues merit special consideration. Unfortunately, the key role that security plays may often make database consistency problems even more critical and complex. Security requirements also pose some special problems of their own for distributed system design. We first discuss discuss security issues as special cases of consistency and then outline the special problems that they raise.

4.4.3.1 Special Cases of Consistency Security issues complicate the choice of a concurrency control strategy. In most systems that deal with security at all, security policies must be guaranteed to be enforced at all times. Partial or intermittent security enforcement is tantamount to no security enforcement at all. To enforce any consistency constraints, including security constraints, synchronization among sites and the concurrency controls that go along with it must be very tight. Security violations may result from any improper read/write interleavings, even those that occur only momentarily and are later repaired. Of course, tighter concurrency controls will adversely affect availability and response time.

Similarly, if requirements for mutual consistency are relaxed in order to allow for faster responses and increased availability, security policies that refer to replicated data cannot be enforced with a high level of confidence. If, for example, two copies of the same data item have different security classification tags, even if just for a short while, a query at one site may be granted improper access to the data item. Mutual consistency of critical replicated data, or limited blocking, is imperative for absolutely strict global security. Great care must be taken in determining which data should be replicated.

4.4.3.2 Special Security Considerations In contrast to standard database consistency, security enforcement is threatened by the use of covert channels and inferences based on system activities. Therefore, not all the security problems that a distributed system must address can be seen as special cases of standard database consistency.

4.4.3.2.1 Inference Problems When a database management system implements concurrency controls and enforces consistency constraints, some action must be taken to deal with improper transactions. If a transaction is blocked by conflicting transactions already in progress, or if one of its actions would violate a consistency constraint, the transaction is aborted (or possibly delayed). In the traditional database system, transaction abortions and delays are tolerated as long as they don't lead to a pattern of deadlock or starvation. In a secure database system, however, transaction abortions and delays may take on a new significance. The mere fact of a transaction's abortion or delay may convey information about other transactions and about data not referenced explicitly within the transaction. If any information is stored in a system log about the reason for the abortion or delay, then inferences may be made even more easily. A secure distributed system design, therefore, must make sure that all logs, traces, and return codes are suitably restricted.

A similar problem is presented when transactions invoke external actions i.e interactions with the external world. An automatic teller withdrawal transaction that causes money to be dispensed. or an update transaction that sends notification to a particular user or mailbox. gives information about its progress to the outside world directly. Just as clever sequences of queries to the database may yield information which can be used to infer more sensitive data. external effects of transactions can also be used as a basis for inferring information about other transactions and data.

4.4.3.2.2 Time Ordering. A user's access to certain data may change when the data (or other data referred to in a security constraint) changes. A user that may be allowed access to specific data during normal operation may be denied access if that data becomes more sensitive during a crisis. Therefore. implicit in the security policies is the stipulation that users only be granted access to certain data at certain times. This largely hidden dependence on time may pose special problems for the concurrency controls in a distributed database system.

Serializability guarantees a result that appears the same as *some* serial execution of transactions. but it does not guarantee a strict time ordering. There is no reason to assume that the system will not execute transactions out of the order in which they were submitted. In normal database applications. modifications of the actual time ordering may not present great problems. but sensitive applications may be affected.

The problem is particularly evident when looking at transactions which change security classification tags as well as other data. Suppose that a transaction changes a missile's status and raises its security classification from SECRET to TOP SECRET. Slightly later. someone with a SECRET clearance submits a query to read the missile's location. The system. in serializing the transactions. executes the read before the status/level change. Thus. the user with the SECRET clearance can find out the location of a missile which should be TOP SECRET at the moment. This type of scenario is probably not too likely with a single database system. but is more likely in a distributed system. Because of communications delays. the increased possibility of essentially simultaneous transaction submission. and clock synchronization problems. transactions may well be run out of "real-world order." During a communications delay. the malicious SECRET user could get a telephone call (or simply overhear a conversation) telling of the missile status upgrade and immediately submit a query to read the missile's location. hoping for a serialization in which the query is processed first.

The problem with time-ordering and the need for high availability may suggest a non-serializable concurrency control approach in the long term. The consequences of using non-serializable concurrency control in a secure system are explored in the next section.

4.4.3.3 Security Consequences of Non-Serializability. Some of the general advantages and disadvantages of non-serializable approaches to concurrency control have been mentioned above. but the particular implications for a secure distributed system are discussed here. This line of thought is a promising one for the medium and long term. and one that requires more work.

4.4.3.3.1 Advantages. The chief advantage of a non-serializable approach to concurrency control is high local availability of data. Local access to data is allowed even when global serializability cannot be guaranteed. Such high availability, including availability in the face of network partitions and remote site failures, can be of tremendous importance in SDI BM/C3I applications. If data is assigned to particular sites in accordance with real-world divisions such as security classifications and compartments, the lack of guaranteed coordination with other sites may not pose large security problems.

The non-serializable approach to concurrency control is also a good candidate for use in a flexible system. It is possible that a system may be designed so that it is able to switch concurrency control procedures when the situation warrants. For example, it may be desirable to choose local data availability rather than global consistency when the local system is operating in a crisis. When normal operation resumes, the system could revert to a serializable concurrency control strategy in which global consistency would be assured at the expense of small delays in availability. Thus, if a threat is perceived in the Pacific theater, the site containing relevant data may temporarily stop trying to synchronize transactions with the site containing data about the Atlantic. The high local availability for the Pacific data may be of the utmost importance, even if it means that some replicated data at the sites become mutually inconsistent.

Another advantage of the non-serializable concurrency control in SHARD is that it imposes a strict time-ordering on transactions. While it allows for temporary inconsistencies, due to the lack of serialization, these inconsistencies are repaired to reflect the strict order of transaction submissions.

4.4.3.3.2 Problems. Serializability theory was developed precisely to avoid the type of interleavings that can cause consistency constraint violations. Therefore, when serializability is not used, or when it is given up temporarily, consistency constraints may well be violated. In particular, mutual consistency of replicated data is likely to be compromised. As mentioned before, mutual consistency is very important for global security. In designing a secure distributed system, then, it is important to determine which parts of the schema the security constraints will need to read and to weigh the implications of replicating the relevant data. While it may be tempting to replicate such critical data to increase availability and performance, the price of such a decision may be either blocking (in the case of serializable concurrency control) or occasional inconsistencies (in the case of non-serializable concurrency control).

When a non-serializable concurrency control strategy is used, there must be some way of repairing or handling the database inconsistencies that may arise. This need leads to a number of possible security problems.

In order to repair inconsistencies, a database system may need to store a significant amount of information about which transactions have taken place and what their results were. Therefore, the possibility of covert channels and inference may be more serious when non-serializable concurrency control is used. Furthermore, to repair inconsistencies transactions may need to be undone and re-executed. If these actions are observed, whether through a log or through a clever sequence of reads, there are new possibilities of inference.

The methods used to repair inconsistencies may themselves create problems in a secure distributed system. Undoing and re-executing transactions significantly later than the initial execution may always present apparent anomalies that may be disconcerting to users. When sensitive data is involved, users may feel even less comfortable with and confident in the system.

Inconsistencies among data can have a cascading effect, making dependent data accessed by other transactions questionable. The algorithms used to reconcile inconsistencies may need to access many data items (especially logs) that were not directly read or written by the original conflicting transactions. If the system must restrict its analysis to certain subsets of the database, because of security considerations, the algorithms may be complicated further. The actions taken to analyze the transactions, and the intermediate results, must be protected from unauthorized observation. Furthermore, the additional overhead for analyzing, logging, undoing, and re-executing transactions will add even more delays to those already caused by the overhead for security checks.

The theory of serializability has provided the formal framework for proving correctness of schedulers in database management systems. To demonstrate the correctness of a scheduler which can generate schedules that are correct both from the consistency and the security points of view, a similar theory for security/consistency correctness is needed.

4.5 Flexible Evaluation of Consistency Conditions

In Section 2.3 we proposed flexible evaluation of consistency conditions as a way of avoiding conflicts between security and consistency and also as a mechanism that ensures higher availability. Possible conflicts between security and consistency are identified in Section 3. Here we discuss in more detail the issues of deferred evaluation of consistency conditions and of alternate actions.

4.5.1 Timing of Consistency Condition Evaluation

Traditional consistency condition evaluation calls for the evaluation of consistency conditions at update time and before commit of the transaction. In previous subsections we noted the need for additional specification of timing in the evaluation.

A more drastic departure from typical consistency condition enforcement is the notion of deferred evaluation of consistency conditions. In deferred evaluation, consistency conditions are not evaluated as data are updated. Instead, the update is performed but the data that were touched are marked as "unreliable", since no guarantee exists that they are in conformance with the consistency conditions. Unreliable data can be cleaned later by applying the consistency verification.

Several advantages are derived from deferred consistency condition evaluation:

Databases that are populated incrementally may have consistency conditions defined over data that have not yet been input. With traditional consistency condition enforcement methods, this would mean that data could either not be input, or that a long transaction would have to be defined, possibly spanning days, while the necessary data were all input. This is unsatisfactory, because it doesn't allow population of the database by several subjects that have to supply data independently. Every time the consistency condition is evaluated as partial data are input, the evaluation fails. A long transaction blocks the database for too long a time. Therefore, consistency validation can be deferred until the time when all the data have been supplied, without blocking other users from accessing the data that are already available. There is, however, a danger involved in inserting unverified data. In engineering design, the solution that was proposed [BUCH86], depends on marking unverified data. The database is slowly populated as data become available. Data that could not be validated are inserted but they are marked as "unreliable". Once all the elements necessary for consistency verification are available, the consistency conditions can be tested and the mark can be removed, thereby upgrading the quality status of the data. If a consistency condition evaluation fails, then all the data involved in the condition are reported as inconsistent and the intervention of an external agent may be required.

In a secure environment it is desirable to have consistency conditions defined over only one level of classification. If this is not possible, then population of a database may present the same problems as above, since some data are not accessible to the process or person inserting data. Deferring evaluation and marking data is a possible way of inserting data without compromising security. Consistency condition verification can be performed by a trusted subject at a later stage. The same trusted subject would be responsible for resolving conflicts. The user need not be aware of certain constraints for which he is not cleared, and by performing the transaction as if no violation had occurred, the user cannot infer any information through denial of service.

In a distributed environment in which high availability is critical, such as some command and control systems, deferred evaluation of consistency conditions can improve availability. For example, a consistency condition may require for its verification data that reside on another node. Again, the update may proceed but the data involved are marked as unreliable until the consistency condition is evaluated and found correct. If the consistency condition test fails, then the need for undoing the transaction arises, possibly with a ripple effect. Also, any transaction that uses unreliable data to obtain new data should propagate the mark to the data it generated. In this scheme, data must be cleaned at reasonably short intervals, to mitigate the risk that contaminated data could corrupt the whole database.

It has been suggested that for certain applications, such as SDI, different security policies may have to be enforced at peace time and during a battle [DRC86]. The rationale is that during peace time the timeframe for penetration is large but the load on the system is low; therefore, security mechanisms can be slower and consume more resources than during a battle when the timeframe for penetration is small and the load on the system is high. Deferred evaluation of consistency conditions is certainly an area in which performance during a crisis situation can be boosted without compromising security. Unfortunately, existing database systems do not allow for selective deactivation of the consistency enforcement mechanisms, and the danger, as described above, is a gradual degradation of the database's consistency.

Determining when consistency conditions have to be evaluated is both a policy and a database design issue. The design tools for such a system should be able to capture timing information associated with a consistency condition and whether enforcement can be deferred or not.

4.5.2 Alternate Responses to Consistency Violations

Existing database management systems provide only one standard response to violations of consistency conditions, namely the abort of a transaction. It is interesting to look at how alternate actions can improve the flexibility of a system, how this is especially useful in a secure environment, and what the implications are for specification tools.

In the previous subsection we saw how deferral of evaluation can boost performance and how it can provide a mechanism to circumvent conflicts between security and consistency. Another mechanism that is useful is the invocation of alternative actions in response to a consistency condition violation.

When a security condition is violated by a transaction and the system traps the attempted access, alternate actions are useful. Responses may range from the notification of the security officer, or the system may consult the subject's history of attempted violations to determine what further action to take. This action could be simple recording of the attempted violation or, if a pattern was detected, it could trigger more serious action.

Recently, proposals were made concerning the relaxation of consistency conditions in response to a violation [BORG85, BUCH86]. The motivation for defining exceptions is that consistency conditions are only useful if they are narrowly defined. For example, a consistency condition on salary that spans from 1.000 to 1.000.000 will most likely be useless to detect errors. Therefore, it has been proposed to define consistency conditions narrowly but allow for exceptions.

Another reason for allowing exceptions is the conflict between two consistency conditions. Since consistency conditions capture the semantics of the data, these conditions may conflict. For example, in a design environment, two design rules, expressed as consistency conditions on the database, may be generally valid. However, when they are both applied to the same case, they result in a conflict. In such a case, one of the two rules may have to be violated. It is important to note that not all consistency conditions are amenable to violation. It is necessary to distinguish between violatable consistency conditions and non-violatable consistency conditions.

In a secure environment it is feasible to relax a consistency constraint in favor of security constraints. This means that a constraint hierarchy may be needed, or at least some ordering by priority

The implication of the previous discussion for any specification tools is that it is highly desirable to have the possibility to specify for each consistency condition what actions ought to be taken in case the condition is violated, and also for the constraints whether it is acceptable that it be relaxed.

5. SYSPECIAL: Extensions and Examples

5.1 Formal Specification Examples

In this section we present the results of our attempts to formally specify database and DBMS constructs. We found that the database was easy to specify and that the resulting specification was natural and readable. DBMS properties such as serializability were much more difficult. In the end we did not specify serializability directly. Instead we were able to specify two phase locking and state appropriate invariants. In this approach we rely on proofs in the literature that two phase locking, if done correctly, insures serializability.

We chose to specify the examples in SYSPECIAL, extended as needed to express the necessary concepts. To accomplish the specification of two phase locking, we introduced the notion of a multilevel specification and of a trace. Both of these notions, in the context of SYSPECIAL, are as of now experimental. However our success with them indicates that additional development is warranted. The multilevel specification is based on the original concept of HDM. Our approach is novel in that it uses the notion of a trace to support procedural constructs in the mappings between levels.

The first example describes a database and security level assignments; it is specified using SYSPECIAL without extensions. The second describes a DBMS with a simplified database in order to illustrate how transactions can be specified by means of a two-level specification and how the notions of serializability and atomicity can be examined by means of a trace and two-phase locking.

Some explanation of SYSPECIAL is in order. SYSPECIAL is a specification language derived from HDM's SPECIAL [SILV79, SILV81]. It is a typed first-order language, with constructs to support integer arithmetic, sets, sequences and structures. A specification describes a state machine by describing the components of the state, called state-functions or VFUNs, and the state-changing operations also referred to as OFUNs (which don't return a value) and OVFUNs (which do). A SYSPECIAL specification consists of a number of (optional) sections: TYPES, PARAMETERS, DEFINITIONS, ASSUMPTIONS, LEMMAS, INVARIANTS, CONSTRAINTS, and FUNCTIONS.

The TYPES sections introduce the types. In these examples, there are only two type classes used - PENDING and STRUCTures. PENDING introduces an abstract data type. STRUCT_OF introduces a record type akin to PASCAL's records. The PARAMETERS section introduces functions and constants whose values don't depend on the state. Parameters are frequently used to give structure to PENDING types. DEFINITIONS provide a mechanism by which auxiliary (mathematical) functions may be defined.

ASSUMPTIONS, LEMMAS, INVARIANTS and CONSTRAINTS are all statements about the state machine (and its implementation). The ASSUMPTIONS section is used to specify requirements on the implementation which are then available as axioms when reasoning about the system. Lemmas are statements that follow from the rest of the specification and may be useful in verifying invariants and constraints. Invariants are statements that are valid for all possible states of the state machine. They are proved inductively starting from the initial state and considering all the operations for the inductive step. Constraints are statements that are valid for all the operations. Constraints typically describe a relation that holds between the old state and the next state of the state machine.

The FUNCTIONS section declares the state-functions (VFUNs) and operations of the specification. The VFUNs can be viewed as arrays indexed by their arguments. VFUNs may have an INITIALLY section which describes constraints on its initial value. Operations (OFUNs and OVFUNs) are the state-changing functions. Operations have arguments, a return value (for OVFUNs) and four optional subsections: DEFINITIONS, ASSERTIONS, EXCEPTIONS and EFFECTS. The definitions section is identical to the global definitions, but can refer to the input arguments. The assertions subsection specifies a list of conditions that must be guaranteed by any program calling the given operation; in reasoning about the operation, these conditions can be assumed. The exceptions section is a list of exception conditions; if any of the exception conditions is true when the operation is invoked, the operation returns immediately with a notification of the raised exception.

5.2 Modeling a Database in SYSPECIAL

This example illustrates one way of modeling a database in SYSPECIAL. The database schema we chose to model is:

```
Employee (SSN, Emp-name, Emp-address, department, clearance)
Projects (Proj-id, Mgr-SSN, department, classification,
                location, travelfunds)
Trip (Trip-id, origin, destination, date-left, date-arrived,
          contact, charges)
Proj-Empl (SSN, Proj-id)
Proj-Trip (proj-id, Trip-id)
```

In addition to modeling the above database, we model security level assignments to the database entries. These assignments are provided for each data item, not for records, attributes, databases or types. We chose this approach because it has the highest granularity; lower granularities can be specified by means of assumptions, as illustrated toward the end of this example.

The records of the database have STRUCTure types, and the database is modeled as five VFUNs corresponding to the five relations given above. Each of these takes as input a record of the appropriate type and returns a boolean value indicating whether the record is in the database. Note that there are no operations, so we are modeling only a database, not a DBMS.

Note that the database model is natural and direct. The record types correspond to the intention of the specified relation. The associated VFUNs correspond to the extension of the relation. The set of all the specified record types forms the database schema; the set of VFUNs constitute the actual database.

```
TYPES

  Ssn, Name, Address, Dept, ProjID, TripID, Date : PENDING;

  Employee: STRUCT_OF(ssn:Ssn; name:Name; address:Address;
                      dept:Dept; clearance:SL);
```

```
  Project: STRUCT_OF(proj:ProjID; mgr_ssn:Ssn; resp_dept:Dept;
                     classification:SL; location:Address;
                     avail_travel:INTEGER);

  Trip: STRUCT_OF(trip:TripID; origin, destination:Address;
                  date_left, date_arrived:Date;
                  contact:Name; cost:INTEGER);

  Project_Employee: STRUCT_OF(pe_ssn:Ssn; pe_proj:ProjID);

  Project_Trip: STRUCT_OF(pt_proj:ProjID; pt_trip:TripID);

FUNCTIONS

  VFUN employee_db(e:Employee) -> b:BOOLEAN;

  VFUN project_db(p:Project) -> b:BOOLEAN;

  VFUN trip_db(t:Trip) -> b:BOOLEAN;

  VFUN project_employee_db(pe:Project_Employee) -> b:BOOLEAN;

  VFUN project_trip_db(pt:Project_Trip) -> b:BOOLEAN;
```

For secure environments it is necessary that data in the database carry a classification tag. There are different ways of assigning security levels to data depending on the desired granularity: the same classification tag for all data in a relation, the same classification for all the instances of an attribute in a relation, the same classification for all the data values in a tuple, or individual classification tags for each atomic value (at the attribute level) in the database. We chose the last form for flexibility reasons. For reasons that are explained in earlier sections, each tag consists of a triplet of security attributes: a read-sensitivity label, a write-sensitivity label, and a trustworthiness label.

In order to specify security level assignments for each atomic value, we have declared an associated attribute type for each database record type. This type merely indicates the field names of the associated records. In the ASSUMPTIONS section we demonstrate how to specify security level assignments at lower granularities by asserting that the security level for a given item is the same as that of another related item. There are other kinds of assumptions specified; these are explained by comments preceding them.

The security level assignments are specified as VFUNs instead of PARAMETERS, this is to allow for upgrading and downgrading of data. Note that an employee's clearance is not necessarily equal to the security level assigned to the clearance field, for instance, if badge color is used to indicate the clearance of employees, then the security level assigned to the clearance information is unclassified.

The specification of a relation with its security tags is the following.

```
TYPES

  RS : PENDING;                          /* Read-sensitivity */

  WS : PENDING;                          /* Write-sensitivity */

  TW : PENDING;                          /* Trustworthiness */

  SL : STRUCT_OF(rs:RS; ws:WS; tw:TW);    /* Security tags */

  Employee_Attributes: {ssn, name, address, dept, clearance};

  Project_Attributes: {proj, mgr_ssn, resp_dept, classification,
                 location, avail_travel};

  Trip_Attributes: {trip, origin, destination, date_left,
                 date_arrived, contact, cost};

  Project_Employee_Attributes: {pe_ssn, pe_proj};

  Project_Trip_Attributes: {pt_proj, pt_trip};

PARAMETERS

  unclassified: SL

FUNCTIONS

  VFUN employee_sl(e:Employee; a:Employee_Attribute) -> sl:SL;

  VFUN project_sl(p:Project; a:Project_Attribute) -> sl:SL;

  VFUN trip_sl(t:Trip; a:Trip_Attribute) -> sl:SL;

  VFUN project_employee_sl(pe:Project_Employee,
                    a:Project_Employee_Attribute) -> sl:SL;

  VFUN project_trip_sl(pt:Project_Trip;
                  a:Project_Trip_Attribute) -> sl:SL;

ASSUMPTIONS

  /* If an employee works on a project,
      his clearance = project's classification */

  FORALL p:Project; e:Employee;
    project_employee_db(STRUCT(pe_ssn:e.ssn, pe_proj:p.paroj)) =>
       project_db(p, classification) = employee_db(e, clearance)
```

```
/* Uniqueness of key (ssn) for employees */

FORALL e1, e2:Employee;
    e1.ssn = e2.ssn  =>  e1 = e2;


/* Modeling security level assignments for employee_db
   at the database level */

FORALL e:Employee; a:Employee_Attributes;
    employee_sl(e, a) = unclassified;


/* Modeling security level assignments for trip,
   project_employee and project_trip records.
   These security levels are assigned
   at the record (rows) level   /

FORALL t:Trip; a:Trip_Attributes;
  trip_sl(t, a) = trip_sl(t, trip);

FORALL p:Project_Employee; a:Project_Employee_Attributes;
  project_employee_sl(p, a) = project_employee_sl(p, pe_ssn);

FORALL p:Project_Trip; a:Project_Trip_Attributes;
  Project_trip_sl(p, a) = Project_trip_sl(e, pt_proj);


/* Modeling security level assignments for the project_db
   at the attribute (columns) level. */

FORALL p1, p2:Project; a:Project_Attributes;
  project_sl(p1, a) = project_sl(p2, a);
```

5.3 Modeling a DBMS in an Experimental Extension of SYSPECIAL

In this example we show how to model a mechanism for ensuring serializability for transaction processing. The mechanism is two phase locking.

Treatment of serializability seems to require concepts that go beyond the basic ingredients of a state machine, i.e., state variables and state changing operations. It is concerned with a scheduling problem and therefore is at heart procedural. It seems that a natural way to model such problems in the context of a state machine is to enhance the state machine concept with the notion of an execution history or trace.

A trace captures the history of the machine. We will think of it as a sequence of operations with a first (dummy) operation which initializes the state. How are the "sequence" and "operations" i.e., elements of the sequence to be modeled? For purposes of specification, the concrete representation is unimportant. We will view the trace as an abstract data type with operations and relations.

Besides << for modeling the time ordering and NEXT for modeling the immediate successor in this ordering, we will have the concept of an identifier for an instance of an operation and we will have the concept of process or transaction identifier for grouping operations. Below we suggest constructs for such modeling in the context of SYSPECIAL. We have introduced some extractor operations on trace elements. Others may be introduced as we see the need for them.

TRACE_ELEMENT is a new built-in type for elements of the trace.

ID(te:TRACE_ELEMENT) models the unique identifier of te.

PID(te:TRACE_ELEMENT) -> pid:Pid
: models the process (transaction) identifier of te

OP(te:TRACE_ELEMENT)
: returns the name of the operation of te if the operation was successful. If not it returns information about the exception

ARG(te:TRACE_ELEMENT, arg:ARG)
: returns the value of the input argument named arg for the operation of te. (ARG will be a built-in type consisting of the formal arguments of OFUNs in the specification.)

IN_STATE(te:TRACE_ELEMENT, vf:VFUN)
: returns the value of vf in the oldstate component of te. (VFUN is a new built-in type consisting of the names of the VFUN's in the specification.)

: We believe an infix notation will be more convenient so we experiment with te"vf as an alternative for IN_STATE(te, vf) in the specification below.

OUT_STATE(te:TRACE_ELEMENT, vf:VFUN)
: returns the value of vf in the newstate component of te.

We also introduce two additional specification concepts: TRACE_ASSUMPTIONS and TRACE_INVARIANTS. TRACE_ASSUMPTIONS are axioms concerned with the trace. They are requirements on the implementation and can be used as axioms in reasoning about the specification. TRACE_INVARIANTS are properties of the trace that are provable from the specification. The following is an example of a two-level specification. The top level defines a state machine with a single state-changing operation, a database transaction called "move". We've chosen this transaction rather than one like "modify" because it involves changes to two records in the database and thus allows us to better exemplify two phase locking.

In this example, besides experimenting with the trace concept we are also experimenting with development of multilevel specifications as envisioned by HDM. HDM views each level of a specification as a state machine, a higher level is implemented by the levels below it. For purposes of this discussion we will assume that a level is implemented by the next lower level. The levels are

connected by mappings which describe the implementation. These mappings are to be written in a language with procedural constructs. The procedural constructs can be used to provide the implementation for operations at the top level in terms of operations at the lower level. In this example we experiment with only one procedural construct, SEQ, which models sequential invocation.

The intention of the top level specification is that the implementation has only one kind of state-changing operation and that any state change occurs as a result of a completed invocation of this operation. Another way of saying the latter is that the operation is atomic.

The second level includes locks and implements "move" in terms of more primitive operations. Serializability (implied by the locking mechanism) at the second level yields atomicity at the top level.

The requirement of atomicity on the higher level operations translates to a requirement of serializability at the lower level. More precisely, one has to show that for every sequence of state changing operations at the lower level there is a legal trace of the upper level machine such that the state changes induced on the upper level state through the mappings by the lower level sequence is equivalent to the state changes recorded in the legal trace. This description assumes that the mapping from an upper level state variable to a lower level state variable induces a function from values of the lower level variable to the values of the upper level variable. Thus a change in the lower level variable induces a unique, possibly trivial, change in the upper level variable.

5.3.1 The Top Level Module

Our specification is divided into two levels. The top level models a database as a sequence of data indexed by the type, Surrogate. This view of the database assumes that every surrogate has a slot in the database. The parameter, nonexistent, is used to indicate unused slots. The exact form of the data is unimportant for this example so we leave it unspecified. The top level has only one state variable, db, the database, and only one state operation, move. move(sur1,sur2) copies the contents of the sur1 to sur2 and deletes the contents of sur1.

```
MODULE top_level

TYPES
    Surrogate: PENDING;
    Data: PENDING;

PARAMETERS
    nonexistent: Data; /* indicates an unused surrogate in the DB.
                          Both the DB and buffers are modeled as
                          containing records for all surrogates */

FUNCTIONS

VFUN db(sur:Surrogate) -> data:Data;

OFUN move(sur1,sur2:Surrogate);
    EXCEPTIONS
      db(sur1) = nonexistent;
    EFFECTS
```

```
        'db(sur2) = db(sur1);
        'db(sur1) = nonexistent;

END_MODULE top_level;
```

5.3.2 The Second Level Specification.

This level specifies the rudiments of a DBMS. The DBMS uses two phase locking to achieve serializability for transactions which may run concurrently. The mapping from the top level to the second level is trivial in the sense that everything at the top level except the OFUN. move. is repeated at the second level. The OFUN move is implemented at the secondlevel using the experimental SEQ construct.

The only new type at this level is Pid. Pid will be used to identify transactions. It will be passed as a parameter to every state changing operation. This parameter passing is only one way of keeping track of operations done on behalf of a transaction. Another would involve setting a VFUN. but the approach we have adopted seems more likely to extend to the context of a distributed database.

Three additional state variables. readlock. writelock. and buffer are introduced at this level. Readlock is a two dimensional array modeled as a Boolean. Since many transactions can simultaneously read the same data. we need both a surrogate and pid index. Writelock is a unary function on surrogates which returns a pid. We use this representation since only one transaction at a time can have a writelock on any particular data item. Buffer is similar to db. but every transaction is entitled to its own buffer. so buffer is indexed by both Surrogate and Pid.

Two additional PARAMETERS. free and empty are introduced here. Empty is like nonexistent. It indicates that a data item corresponding to a surrogate has not been written into a buffer. Free is a dummy value of Pid to indicate the absence of a writelock.

Under ASSUMPTIONS we indicate some of our intention for the PARAMETER empty. namely that it is not used as a value for the db VFUN.

As for our state changing operations (OFUNs) set_readlock and set_writelock. clear_readlock and clear_writelock are self-explanatory.

> Read(sur. pid) reads a data item from the database to a buffer.
>
> Modify(sur. data. pid) sets buffer(sur. pid) = data
>
> Delete(sur. pid) sets buffer(sur. pid) = nonexistent.
>
> Commit(pid) updates the database with the appropriate contents of the buffer corresponding to pid.

The exceptions on these operations ensure that the locks have their intended effects.

```
MODULE second_level

TYPES

    Pid: PENDING;
    Surrogate: PENDING;
    Data: PENDING;

PARAMETERS

    free: Pid;   /* free is a dummy value used to indicate the
                 absence of a writelock */

    empty: Data; /* indicates an unused surrogate in the buffer.
                 The commit OFUN will leave db items unchanged
                 if their surrogates correspond to empty data
                 in the buffer.  /

    nonexistent: Data;

ASSUMPTIONS
    empty ~= nonexistent;
    FORALL s:Surrogate; db(s) ~= empty;


FUNCTIONS

VFUN readlock(sur:Surrogate, pid:Pid) -> set:BOOLEAN,
    INITIALLY set = FALSE;

VFUN writelock(sur:Surrogate) -> pid:Pid;
    INITIALLY pid = free;

VFUN buffer(sur:Surrogate; pid:Pid) -> data:Data;
    INITIALLY data = empty;

VFUN db(sur:Surrogate) -> data:Data;

OFUN set_readlock(sur:Surrogate; pid:Pid),
    ASSERTIONS
      pid ~= free;
    EXCEPTIONS
      writelock(sur) ~= free;
    EFFECTS
      'readlock(sur, pid) = TRUE;

OFUN set_writelock(sur:Surrogate; pid:Pid);
    ASSERTIONS
      pid ~= free;
```

```
    EXCEPTIONS
      writelock(sur) ~= free);
      EXISTS p:Pid (p ~= pid  AND  readlock(sur, p) = TRUE);
    EFFECTS
      'writelock(sur) = pid;

OFUN read(sur:Surrogate; pid:Pid);
    ASSERTIONS
      pid ~= free;
    EXCEPTIONS
      db(sur) = nonexistent;
      ~readlock(sur, pid);
    EFFECTS
      'buffer(sur,pid) = db(sur);

OFUN modify(sur:Surrogate; data:Data; pid:Pid);
    ASSERTIONS
      pid ~= free;
    EXCEPTIONS
      data = nonexistent OR data = empty;
      db(sur) = nonexistent;
    EFFECTS
      'buffer(sur,pid) = data;

OFUN delete(sur:Surrogate; pid:Pid);
    ASSERTIONS
      pid ~= free;
    EXCEPTIONS
      db(sur) = nonexistent;
    EFFECTS
      'buffer(sur,pid) = nonexistent;

OFUN commit(pid:Pid);
    ASSERTIONS
      pid ~= free;
    EXCEPTIONS
      EXISTS s:Surrogate;
          writelock(s) ~= pid  AND  buffer(s,pid) ~= empty;
    EFFECTS
      FORALL s:Surrogate;
          IF buffer(s,pid) ~= empty
              THEN 'db(s) = buffer(s,pid)
            ELSE 'db(s) = db(s);

OFUN clear_readlock(sur:Surrogate; pid:Pid);
    ASSERTIONS
      pid ~= free;
    EFFECTS
      'readlock(sur, pid) = FALSE;
```

```
OFUN clear_writelock(sur:Surrogate; pid:Pid);
    ASSERTIONS
      pid ~= free;
    EXCEPTIONS
      writelock(sur) ~= pid;
    EFFECTS
      'writelock(sur) = free;

TRACE_ASSUMPTIONS

    /* Pid arguments = PID of trace */
    FORALL x:TRACE_ELEMENT;
      ARG(x,pid) = PID(x);

INVARIANTS

    / NO read-write conflict /
    FORALL s:Surrogate; p1:Pid;
      ( (EXISTS p2:Pid;
          (p2 ~= p1 AND readlock(s, p2)) => writelock(s) ~= p1);

    /* NO write-write conflict */
    FORALL s:Surrogate; p1,p2:Pid;
      (p1 = writelock(s) AND p2 = writelock(s) => p1 = p2);

TRACE_INVARIANTS

    FORALL s:Surrogate; p:Pid; x:TRACE_ELEMENT
          ( (OP(x) = read  AND  PID(x) = p  AND  ARG(x,sur) = s)
            => x"readlock(s, p) );

    FORALL p:Pid; x:TRACE_ELEMENT;
      ( (OP(x) = commit  AND  PID(x) = p)
         => FORALL s:Surrogate;
             ( buffer(s,p) ~= empty  AND  buffer(s,p) ~= db(s) )
                              => x"writelock(s) = p) );

TRANSACTIONS

    move(sur1,sur2:Surrogate; pid:Pid) IS
      ASSERTIONS
          pid ~= free;
      EXCEPTIONS
          db(sur1) = nonexistent;

      SEQ
          set_readlock(sur1,pid);
          set_writelock(sur2,pid);
          read(sur1,pid);
```

```
        modify(sur2,buffer(sur1),pid);
        set_writelock(sur1,pid);
        delete(sur1,pid);
        commit(pid);
        clear_readlock(sur1,pid);
        clear_writelock(sur1,pid);
        clear_writelock(sur2,pid);
    ENDSEQ;
```

6. Summary

In this study we analyzed the problem of database consistency and its interactions with security and applied a powerful specification tool, SYSPECIAL, to obtain initial specifications of a database and the rudiments of a DBMS. To do this, SYSPECIAL was extended to include some additional general constructs.

Database consistency was studied in its full range, from declarative consistency constraints to transaction processing, both in a centralized and a distributed environment. The security implications of newer concurrency control techniques that do not rely on the well-studied principle of serializability have also been discussed. A framework for integrated handling of database consistency and security was proposed and notions, such as integrity, were refined for use in the context of secure databases

The major accomplishments are:

1 A unified framework for database consistency and security was established through declarative specifications of database and security constraints. This unified framework will allow easier testing for compatibility of consistency and security constraints and will make it possible to use the same tools for analyzing consistency and security policy specifications.

2 The unified declaration of security and consistency constraints encourages the use of adaptive policies, since policies can be substituted by exchanging sets of constraints

3 The notion of integrity was refined for use in secure databases by splitting it into its two components: write sensitivity and trustworthiness. This division allows more precision in specifying complex security policies.

4 The interaction of security and transaction management was studied, for centralized as well as distributed systems. Security implications were evaluated for serializable and some non-serializable protocols.

5 SYSPECIAL was extended with the notions of a multilevel specification and a trace. The multilevel specification is based on the original concept of HDM. The use of a trace is novel and supports the mapping of procedural constructs between levels

6. Using the extended SYSPECIAL, it was possible to specify databases with their security and consistency constraints in a natural way

7 The rudiments of a DBMS could be specified using the multilevel specification and the notion of a trace. It was possible to specify a two phase locking protocol to guarantee serializability. The extensions are also useful for definition of other DBMS components, such as the log

8 Additional specification tool features that would be desirable have been identified, mainly a technique for specifying alternate actions in response to database constraint violations, and mechanisms for specifying the exact timing of constraint evaluation within a transaction. As a support of the design process, tools that support the evaluation of compatibility among database constraints, both security and consistency, would be most useful

7. REFERENCES

[BELL73] D. E. Bell and L. J. LaPadula. "Secure computer systems: A mathematical model." MTR-2547. Vol.2. MITRE Corp.. Bedford. MA. Nov. 1973.

[BELL74] D. E. Bell and L. J. LaPadula. "Secure computer systems: Mathematical foundations and model." M74-244. MITRE Corp.. Bedford. MA. Oct. 1974.

[BELL75] D. E. Bell and L. J. LaPadula. "Unified Exposition and Multics Interpretation." Mitre Corporation. July 1975.

[BERN80] P.A. Bernstein and D.W. Shipman. "The Correctness of Concurrency Control Mechanisms in a System for Distributed Databases (SDD-1)." ACM Trans Database Systems 5. 1 (March 1980).

[BERN81] P A. Bernstein and N Goodman "Concurrency Control in Distributed Database Systems." ACM Computing Surveys 1. 2 (June 1981). 185-221.

[BERN87] P.A. Bernstein. V Hadzilacos and N Goodman. *Concurrency Control and Recovery in Database Systems*. Reading MA Adison Wesley. 1987

[BIBA77] K. J. Biba. "Integrity considerations for secure computer systems." MTR-3135. MITRE Corp.. Bedford. MA. Apr. 1977

[BONY86] D Bonyun. "A New Look at Integrity Policy for Database Management Systems." National Computer Security Center Workshop on Database Management System Security. Baltimore. MD. June 1986.

[BORG85] A Borgida. "Language Features for Flexible Handling of Exceptions in Information Systems." ACM TODS 10 (1985). 565-603

[BUCH86] A P Buchmann. R.S. Carrera. and M A. Vazquez-Galindo. "A Generalized Constraint and Exception handler for an Object-Oriented CAD-DBMS". *Proceedings of the International Workshop on Object-Oriented database Systems*. Pacific Grove. pp 38-49. 1986

[DENN79] D E. Denning and P. Denning. "Data Security". *ACM Computing Surveys*. Vol 11. No 3. pp 227-249. 1979.

[DENN85] D.E Denning. "Commutative Filters for Reducing Inference Threats in Multilevel Database Systems". Proceedings of the 1985 Symposium on Security and Privacy

[DENN86] D.E. Denning. T.F. Lunt. P.G. Neumann. R.R. Schell. M. Heckman. and W Shockley "Secure Distributed Data Views". Interim Report: A002. SRI Computer Science laboratory. 1986.

[DOD85] Department of Defense. *Department of Defense Trusted Computer System Evaluation Criteria* National Computer Security Center. 1985 DOD 5200.28-STD.

[DRC86a] "SDI and Distributed Systems." Dynamics Research Corporation. (15 April 1986).

[DRC86b] "Distributed Systems Technology Assessment for SDI" Dynamics Research Corporation. (30 September 1986)

[FEIE77] R. J. Feiertag. K. N. Levitt. and L. Robinson. "Proving multilevel security of a system design." in *Proc. 6th ACM Symposium on Operating Systems Principles. ACM SIGOPS Op Sys Review* 11.5. Nov. 1977.

[FEIE79] R. J. Feiertag and P. G. Neumann. "The foundations of a provably secure operating system (PSOS)." *Proc. AFIPS Natl. Computer Conf.*. Vol. 48. AFIPS Press. Arlington. VA. 1979.

[FEIE80] R.J. Feiertag. "A Technique for Proving Specifications are Multilevel Secure." SRI Computer Science Laboratory. January 1980.

[GARD79] G. Gardarin and M. Melkanff. "Proving Consistency of Database Transactions." Proc. Int Conf. Very Large Data Bases. October 1979 pp 291-298.

[HOLT72] R. C. Holt. "Some Deadlock Properties in Computer Systems". *ACM Computing Surveys*. Vol. 4. No. 3. pp 179-196. 1972.

[JANA81] J.M. Janas. "On the Feasibility of Informative Answers" Advances in Database Theory (eds. J. Minker and J. M. Nicolas). Plenum Press 1981.

[KSOS78] "KSOS Verification Plan." WDL-TR7809 Ford Aerospace and Communications Corp Western Development Lab Div. Palo Alto. CA. and SRI Intl. Menlo Park. CA. 1978

[KUNG81] H.T. Kung and J.T. Robinson. "On Optimistic Methods for Concurrency Control". *ACM Trans. on database Systems*. Vol. 6 No. 2. pp. 213-226. 1981

[LAND81] C.E. Landwehr. "Formal Models for Computer Security." *Computing Surveys* 13:3. Sept 1981.

[NCSC85] "Draft Trusted Network Evaluation Criteria". National Computer Security Center. 1985

[NCCS86] National Computer Security Center Workshop on Database Management System Security. Baltimore. MD. June 1986.

[REED79] D.P. Reed and R.K. Kanodia. "Synchronization with Eventcounts and Sequencers". *Communications of ACM*. Vol. 22. No. 2. pp. 115-123. 1979

[RYPK79] D.J. Rypka and A.P. Lucido. "Deadlock detection and Avoidance for Shared Logical resources". *Trans on Software Engineering*. pp. 465-471. 1979.

[SARI85] S.K. Sarin. B.T. Blaustein. and C.W. Kaufman. "System Architecture for Partition-Tolerant Distributed Databases." IEEE Transactions on Computers C-34. 12 (December 1985). pp. 1158-1163.

[SCHE86] R.R. Schell and D.E. Denning. "Integrity in Trusted Database Systems. National Computer Security Center Workshop on Database Management System Security. Baltimore. MD. June 1986.

[SHEA86] T. Sheard and D. Stemple. "Automatic Verification of Database Transaction Safety". Coins technical Report 86-30. University of Massachusetts. Amherst. 1986.

[SILV79] B.A. Silverberg. L. Robinson. and K.N. Levitt. "The HDM Handbook." Volumes I-III. SRI Computer Science Laboratory. June 1979.

[SILV81] B.A. Silverberg, W.D. Elliot, and D.F. Hare. "Revisions to HDM and its Tools." SRI Computer Science Laboratory, October 1981.

DISTRIBUTION LIST

addresses	number of copies
Emilie J. Siarkiewicz RADC/COTD	15
RADC/DOVL GRIFFISS AFB NY 13441	1
RADC/DAP GRIFFISS AFB NY 13441	2
ADMINISTRATOR DEF TECH INF CTR ATTN: DTIC-DDA CAMERON STA BG 5 ALEXANDRIA VA 22304-6145	5
RADC/COTD BLDG 3, ROOM 16 GRIFFISS AFB NY 13441-5700	1
HQ USAF/SCTT Pentagon Wash DC 20330-5190	1
DIRECTOR DMAHTC ATTN: SDSIM Wash DC 20315-0030	1
Director, Info Systems OASD (C3I) Rm 3E187 Pentagn Wash DC 20301-3040	1
Fleet Analysis Center Attn: GIDEP Operations Center Code 30G1 (E. Richards) Corona CA 91720	1

HQ AFSC/XRAE
ANDREWS AFB DC 20334-5000 1

HQ AFSC/XRK
ANDREWS AFB MD 20334-500 1

HQ SAC/SCPT
OFFUTT AFB NE 68113-5001 1

DTESA/RQEE
ATTN: LARRY G.MCMANUS
2501 YALE STREET SE
Airport Plaza, Suite 102
ALBUQUERQUE NM 87106 1

HQ TAC/DRIY
Attn: Mr. Westerman
Langley AFB VA 23665-5001 1

HQ TAC/DRCA
LANGLEY AFB VA 23665-5001 1

ASD/ENEMS
Wright-Patterscn AFB OH 45433-6503 2

ASD-AFALC/AXP
WRIGHT-PATTERSCN AFB OH 45433 1

ASD/AFALC/AXAE
Attn: W. H. Dungey
Wri ght-Patterson AFE OH 45433-6533 1

AAMRL/HE
WRIGHT-PATTERSCN AFB OH 45433-6573 1

AFIT/LDEE
BUILDING 640, AREA B
WRIGHT-PATTERSON AFB OH 45433-6583 1

AFWAL/MLPO
WRIGHT-PATTERSON AFB OH 45433-6533 1

Air Force Human Resources Laboratory
Technical Documents Center
AFHRL/LRS-TDC
Wright-Patterson AFB OH 45433 1

2750 ABW/SSLT
Bldg 262
Post 11S
Wright-Patterson AFB OH 454433 1

AUL/LSE
MAXWELL AFB AL 36112-5564 1

Defense Communications Engineering Ctr
Technical Library
1860 Wiehle Avenue
Reston VA 22090-5500 1

COMMAND CONTROL AND COMMUNICATIONS DIV
DEVELOPMENT CENTER
MARINE CORPS DEVELOPMENT & EDUCATION COMMAND
ATTN: CCDF DICA
QUANTICO VA 22134-5080 2

AFLMC/LGY
ATTN: CH, SYS ENGR DIV
GUNTER AFS AL 36114 1

U.S. Army Strategic Defense Command
Attn: DASD-H-MPL
P.O. Box 1500
Huntsville AL 35807-3801 1

COMMANDING OFFICER
NAVAL AVIONICS CENTER
LIBRARY - D/765
INDIANAPOLIS IN 46219-2189 1

COMMANDING OFFICER 1
NAVAL TRAINING SYSTEMS CENTER
TECHNICAL INFORMATION CENTER
BUILDING 2068
ORLANDO FL 32813-7100

COMMANDER 1
NAVAL OCEAN SYSTEMS CENTER
ATTN: TECHNICAL LIBRARY, CODE 9642B
SAN DIEGO CA 92152-5000

COMMANDER (CODE 3433) 1
ATTN: TECHNICAL LIBRARY
NAVAL WEAPONS CENTER
CHINA LAKE, CALIFORNIA 93555-6001

SUPERINTENDENT (CODE 1424) 1
NAVAL POST GRADUATE SCHOOL
MONTEREY CA 93943-5000

COMMANDING OFFICER 2
NAVAL RESEARCH LABORATORY
ATTN: CODE 2627
WASHINGTON DC 20375-5000

SPACE & NAVAL WARFARE SYSTEMS COMMAND 1
PMW 153-3DP
ATTN: R. SAVARESE
WASHINGTON DC 20363-5100

CDR, U.S. ARMY MISSILE COMMAND 2
REDSTONE SCIENTIFIC INFORMATION CENTER
ATTN: AMSMI-RD-CS-R (DOCUMENTS)
REDSTONE ARSENAL AL 35898-5241

Advisory Group on Electron Devices 2
Hammond John/Technical Info Coordinator
201 Varick Street, Suite 1140
New York NY 10014

UNIVERSITY OF CALIFORNIA/LOS ALAMOS 1
NATIONAL LABORATORY
ATTN: DAN BACA/REPORT LIBRARIAN
P.O. BOX 1663, MS-P364
LOS ALAMOS NM 87545

RAND CORPORATION THE/LIBRARY 1
HELFER DORIS S/HEAD TECH SVCS
P.O. BOX 2138
SANTA MONICA CA 90406-2138

AEDC LIBRARY (TECH REPORTS FILE) 1
MS-10C
ARNOLD AFS TN 37389-9998

USAG 1
Attn: ASH-PCA-CRT
Ft Huachuca AZ 85613-6000

JTFPMO 1
Attn: Director/Acvanced Technology
1500 Planring Research Drive
McLear VA 22102-5099

AFEWC/ESRI 4
SAN ANTONIO TX 78243-5000

485 EIG/EIER (DMC) 2
GRIFFISS AFB NY 13441-6348

ESD/AVS 1
ATTN: ADV SYS DEV
HANSCOM AFB MA 01731-5000

ESD/ICP 1
HANSCOM AFB MA 01731-5000

ESD/AVSE 2
BLDG 1704
HANSCOM AFB MA 01731-5000

HQ ESC SYS-2 1
HANSCOM AFB MA 01731-5000

The Software Engineering Institute 1
Attn: Major Dan Burton, USAF
Joint Program Office
Carnegie Mellor University
Pittsburgh PA 15213-3890

DIRECTOR 1
NSA/CSS
ATTN: T513/TDL (DAVID MARJARUM)
FORT GEORGE G MEADE MD 20755-6000

DIRECTOR 1
NSA/CSS
ATTN: R24
FORT GEORGE G MEADE MD 20755-6000

DIRECTOR 1
NSA/CSS
ATTN: R21
9800 SAVAGE ROAD
FORT GEORGE G MEASDE MD 20755-6000

DIRECTOR 1
NSA/CSS
ATTN: R5
FORT GEORGE G MEADE MD 20755-6000

DIRECTOR 1
NSA/CSS
ATTN: R8
FORT GEORGE G MEADE MD 20755-6000

DIRECTOR 1
NSA/CSS
ATTN: SC31
FORT GEORGE G MEADE MD 20755-6000

DIRECTOR 1
NSA/CSS
ATTN: S21
FORT GEORGE G MEADE MD 20755-6000

DIRECTOR 1
NSA/CSS
ATTN: V33 (S. Friedrich)
FORT GEORGE G MEADE MD 20755-6000

DIRECTOR 1
NSA/CSS
ATTN: W3
FORT GEORGE G MEADE MD 20755-6000

DoD COMPUTER SECURITY CENTER 1
ATTN: C4/TIC
9800 SAVAGE ROAD
FORT GEORGE G MEADE MD 20755-6000

Sytek, Inc. 1
1225 Charlestor Rd.
Mountain View CA 94043

Unisys Corp 1
Attn: Lorraire D. Martin
5151 Camino Ruiz
Camarillc CA 93011-6004

Harris Corp. 1
Government Information Systems Division
Attn: Rorca Henning
P.O. Box 98000
Melbourne, FL 32902

Ford Aerospace & Communications Corp. 1
Attn: Peter Baker (Mail Stop 29A)
10440 State Highway 83
Colorado Springs, CO 80908

MITRE Corp. 1
Attn: Dale M. Johnson (MS B330)
Burlington Rc.
Bedford, MA 01730

Honeywell Inc. 1
Secure Computing Technology Center MN55-7282
Attn: J. Thomas Haigh
2855 Anthony Lane South (Suite 130)
St. Anthony, MN 55418

SRI International 2
Computer Science Lab
Attn: Teresa Lunt
333 Raverswood Ave.
Menlo Park, CA 94025

MITRE Corp. 2
Attn: Joshua Guttman (MS A455)
Burlington Rc.
Bedford, MA 01730

Computational Logic, Inc. 1
Attn: Dr. Donald I. Good
614 N. 32nd St.
Austin, TX 78705

Odyssey Research Assoc. 1
Attn: Dr. Richard Platek
301A Harris B. Dates Dr.
Ithaca, NY 14850-1313

National Security Agency 1
Attn: Larry Hatch / R5
9800 Savage Road
Fort Meade, MD 20755

DIR NSA 1
Attn: Sylvan Pinsky / C33
9800 Savage Road
Fort Meade, MD 20755

Naval Research Laboratory 1
Attn: Carl E. Landwehr (Code 7593)
Washington, DC 20375

US Army CECOM/CENTACS 1
AMSEL-RD-COM-TC-2
Attn: John W. Freusse
Fort Monmouth, NJ 07703

Defense Intelligence Agency 1
Attn: Richard Myren (RSE-4)
Washington, DC 20301

Defense Communications Engineering Center 1
Attn: Code R820 (Peter Fonash)
1860 Wiehle Ave.
Reston, VA 22090-5500

Gemini Computers Inc. 1
Attn: Roger Schell
60 Garder Court (Suite 110)
Monterey, CA 93940

Boeing Aerospace Co. 1
Attn: Daniel Schrackenberg (8H-35)
P.O.Box 3999
Seattle, WA 98124

University of Delaware 1
Electrical Engineering Dept.
Attn: Peter C. von Glahn
140 Evans Hall
Newark, DE 19716

BBN Laboratories Inc. 1
Attn: Steve Vinter
10 Moulton Street
Cambridge, MA 02238

University of California 1
Computer Science Dept.
Attn: Prof. Richard A. Kemmerer
Santa Barbara, CA 93106

Institute for Defense Analyses 1
Computer & Software Engineering Division
Attn: William Mayfield
1801 N. Beauregard St.
Alexandria, VA 22311

Research Triangle Institute 1
Attn: John McHugh
CDSR Herbert Bldg., P.O. Box 12194
Research Triangle Park, NC 27709

Unisys Corp. 1
Attn: Deborah Cooper (MS 91-11)
2525 Colorado Ave.
Santa Monica, CA 90406-9988

Trusted Informations Systems 1
Attn: Stephen T. Walker
3060 Washington Rd.
Glenwood, MD 21738

SRI International 2
Computer Science Lab
Attn: John Rushby
333 Ravenswood Ave.
Menlo Park CA 94025

Unisys Corp. 1
Attn: LouAnna Notargiacomo
8201 Greensboro Dr, Suite 1000
McLean VA 22102

DIR NSA
Attn: Rob Johnson / C33
9800 Savage Rd.
Ft Meade MD 20755-6000

1

DIR NSA
Attn: Howard Stainer / C32
9800 Savage Rd.
Ft Meade MD 20755-6000

1

SPAWAR/Code 3242
Attn: LCDR Thomas Taylor
Washington DC 20363-5100

1

Odyssey Research Associates, Inc.
Attn: Ray Worg
525 Middlefield Rd.
Menlo Park CA 94025

1

Xerox Advanced Information Technology
Attn: Dr. Alejandro P. Buchmann
Four Cambridge Center
Cambridge MA 02142

5

Xerox Advanced Information Technology
Attn: Dr. Barbara T. Blaustein
1800 Diagonal Rd, Suite 300
Alexandria VA 22314

2

Strategic Defense Initiative Office
Office of the Secretary of Defense
Wash DC 20301-7100

1