Knowledge-/rule-based Semantics For Large Database Systems.

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road Ann Arbor MI 48106-1346 USA
313 761-4700 800 521-0600

Order Number 9513145

Knowledge-/rule-based semantics for large database systems

Yurchak, Kirk Elliott, Ph.D.

Lehigh University, 1994

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

Knowledge-/Rule-based Semantics for Large Database Systems

by

Kirk E. Yurchak

Presented to the Graduate and Research Committee

of Lehigh University

in Candidacy for the Degree of

Doctor of Philosophy

in

Computer Science

Lehigh University

October, 1994

Approved and recommended for acceptance as a dissertation in partial fulfillment of the requirements for the degree of Doctor of Philosophy:

August 10, 1994
Date

Dissertation Director

August 10, 1994
Accepted Date

Committee Members:

Dr. Donald J. Hillman

Dr. Glenn D. Blank

Dr. Laura I. Burke

Dr. Edwin J. Kay

For my Mother, Joleita, my Father, Russell,
and especially my wife, Patricia.

Thank you for your love and support throughout this endeavor.

Acknowledgments

Foremost, I wish to thank Dr. Donald J. Hillman for his invaluable advisement in both my academic and commercial pursuits throughout my graduate years. The various meetings and seminars with him greatly helped to inspire and focus the work herein, and for that I am truly grateful. I thank Dr. Glenn D. Blank, Dr. Laura I. Burke, and Dr. Edwin J. Kay for their interest, advisement, and support in this effort. Gratitude is extended to the Ben Franklin Technology Center and Valley Foundation Consultants Group, Inc., for my initial graduate funding and setting the stage for the slow, primitive budding of these ideas. I wish to thank Rx Returns, Inc., and Origination Alternatives, Inc., not only for their financial assistance throughout the latter part of my graduate career, but also for providing a practical environment in which much of these concepts were put to the test of cold, hard implementation. I thank these companies for opening their commercial doors to a young, eager, and ambitious computer scientist and trusting him to help guide their paramount technical paths.

Appreciation is also extended to Robert S. Voros and William C. Voros for introducing me into many of the academic and commercial pursuits with which I have been engaged throughout my graduate career. Special thanks is noted to Robert Voros for co-developing the *Signature C* generic database interface to which the SDBMS semantic engine prototype was linked.

Finally, I thank my family without whose grateful patience and undying support would surely have proven to make this endeavor impossible.

Table of Contents

List of Figures

(Figures may be found directly succeeding the straight text of this dissertation beginning on page 112.)

Abstract

Since their dawn, database management systems have saturated virtually every area of academic and commercial domains. They have proven to be as indispensable as the paper and pencil. Yet while relational database management systems (RDBMSs) have been made so readily available, most are still quite primitive with regard to *semantic abstraction*. It may be argued that these RDBMSs are not much more than mere vessels of information having little knowledge of the *semantics*--the underlying, *meaningful* aspects--which apply to the data they represent. Few systems allow tables to "know" about other tables around them, and almost no systems allow knowledge of tables governed by different types of RDBMSs. The notion of data semantics spans a single record's relation to itself (cross-field semantics), a record's relation to other records in the same table (cross-record semantics), a record's relation to records of other tables (cross-table semantics), and even a record's relation to records of other tables governed by other RDBMSs (cross-platform semantics). It suggests the concept of inferring certain data-manipulative actions based on other committal actions performed on a given database.

This dissertation proposes a knowledge-/rule-based approach to inject high-level semantics into today's various RDBMSs. The system has been dubbed the Semantic Database Management System (SDBMS) and uses rule-bases associated with each database under its control to represent semantic repercussions relative to data-manipulations (inserts, updates, deletes, and the like). The databases governed by the SDBMS may be of potentially differing RDBMSs. A semantic engine (SE) is used to control inferences within the rule-bases and translate manipulative consequents to respective RDBMS engines. Users, developers, and programs alike access data governed by the SDBMS through a semantic interface (SI).

The goal of this work is to provide centralized access of many forms of data through a universal medium, making primitive database engines more powerful and powerful database engines more flexible. It provides a means for communication between RDBMSs and promotes more "intelligent" systems. Most importantly it lessens the burden previously posed on producing a myriad of ad hoc customized programs subsequently requiring linkage to RDBMSs to inject the otherwise lacking semantic knowledge.

1.0 INTRODUCTION AND BACKGROUND

The use of database systems has become as common as the paper and pencil in today's business and scientific communities. Database technology has literally worked its way into every facet of commercial development. Yet while database systems, especially large ones, have proven themselves extremely useful, their creation remains somewhat tedious. To combat the problem of designing these large systems researchers have derived several *semantic modeling* paradigms. These paradigms allow the database creator to produce a detailed model of the domain which the databases will represent.

Often, however, the power of these models does not extend much beyond the design phase of large database systems. As the systems pass from design to implementation much of the semantic information is lost due to the implementation platform's inability to integrate the information. Figure 1 depicts the standard evolution of a large database system.

In phase one of this cycle a requirements analysis is conducted which sketches the *domain* which the database system is meant to represent. This involves analyzing the system as a whole and determining what parameters dictate the formation of the system. Phase two traditionally involves some form of modeling--usually a semantic model--which is generated to define the global scope of such a system. This phase solidifies work done in the requirements analysis, outlining the *entities*, *objects*, *classes*, and *relationships* required to represent the system's domain. In phase three the entities, objects, classes, and relationships outlined in phase two are translated (mapped) to implementation platform representations. This is most often necessary since the implementation platform is not entirely capable of representing *all* information within the semantic model of phase two and hence a one-to-one mapping

of representations is not possible. It is at this time that much of the useful semantic information is lost. Phase four occurs mainly in relational database system implementations and involves "cleaning up" the relations derived in phase three. This encompasses such aspects as the removal of redundant relations, reduction to more efficient relations, etc. The design/implementation cycle is now complete. Often, however, this resulting implementation is not powerful enough to fully administrate its use among multitudes of non-expert end users. Therefore, a fifth phase is commonly required to combat this problem. The fifth phase involves the creation of a customized user-interface which replaces semantic information lost during the mapping of phase three. This is an arduous process requiring low-level codification of the lost semantic information; information which was implicitly included in semantic design, but must now be explicitly integrated into the final implementation. Hence, integration of semantic database information is performed twice with today's technology--once during design and once again in implementation.

The thrust of this dissertation is, therefore, to develop a formal system for integrating high-level semantics into the design *and* implementation of large database systems. Such a system shall be deemed the *semantic engine* (SE) and shall be accessed by users and programs alike via a *semantic interface* (SI). Semantic information associated with databases shall be incorporated immediately during the design *and* implementation phases and continue to evolve throughout the system's life span. Figure 2 depicts how such a semantic system would be integrated as a front end to today's database management systems. Users, designers, database administrators, and programs communicate with the SE through the SI. The SE accepts special *semantic commands* from the SI which are translated into the appropriate database engine commands which, in turn, physically access the data. In this manner universality of representation and access is achieved as the SE may be generically

programmed with knowledge of how to communicate with multiple database engines. The balance of this dissertation details the power required by such a semantic system and how it can be integrated into today's database technologies.

2.0 SEMANTIC MODELING AND THE ENTITY/RELATIONSHIP MODEL

> "The motivation for [semantic modeling] research... [is] as follows: Database systems generally--relational or otherwise--really have only a very limited understanding of what the data in the database *means*; they typically "understand" certain simple atomic data values, and perhaps certain many-to-one relationships among those values, *but very little else*..."
>
> -- Date, 1990

It may be said that existing relational database management systems (RDBMSs) are merely an amalgamation of data types, data values, fields, and records, and that these systems do not generally "understand" what meaningful constraints exist between them, nor do they "understand" *how* they function together to form a cohesive representation of a particular domain. It may be unfair, however, to state that existing DBMSs are totally lacking in semantic aspects. The use of domains, primary keys, and foreign keys indeed brush the edge of "semantics," yet they represent a mere fraction of the true semantics which may apply to the DBMS as a whole.

There has been a continuing effort to integrate more semantics into database paradigms. To date, however, semantic integration has mainly profited the design phase of database systems, leaving much of the implementation aspects to customized program interfaces. Researchers have proposed several different "semantic modeling" techniques which may be adhered to during the design phase of a given database

system. These modeling approaches inject a certain degree of data semantics by splitting representations into *entities*, *types*, *properties*, *identities*, and *relationships*. Using these generic semantic concepts one may maintain that the world is made up of entities which represent conceptual and/or physical objects (e.g., Daniel, container-25, equation-45, etc.). Each entity has associated with it one or more properties describing the characteristics, attributes, etc., commonly associated with the given object (e.g., hair color, volume, number of parameters, etc.). Types are used to define the various classifications of entities (e.g., a person, a storage container, a mathematical equation, etc.). A type defines which properties an entity has when belonging to that type. Every entity has an identity which uniquely differentiates that entity from other entities of the same type (e.g., Dan's full name: "Daniel John Smith," etc.). The semantic notion of identity is similar to the relational concept of the primary key--as an identity uniquely distinguishes an entity, a key uniquely identifies a record. Finally, relationships describe the interactions between entities--how one entity relates to another.

In 1976, Chen formalized these generic terms into the Entity/Relationship (E/R) Model. In this model entities are split into two categories: weak entities and regular entities. Weak entities are those whose existence is dependent on the existence of another entity. Regular entities are those which are not weak (i.e., those entities which exist independent of any other entities). Properties were divided into several categories: simple, composite (formed by the concatenation of two or more properties), key (a uniquely-identifying property), single-/multi-valued, missing (properties representing "unknown" or "not applicable" aspects), and base or derived (those which *derive* their values from some calculation of other property values). Two categories represent possible relationships: type relationships (those which form relationships between types/classifications--e.g., a human "is a" mammal) and token

relationships (those relationships which simply link one entity to another entity--e.g., Dan "loves" Sue). Types, in turn, are made up of two classifications: subtypes and supertypes. Subtypes are said to *inherit* the properties from their supertypes. *Generalizations* are formed by working one's way from subtype to supertype, while *specializations* are derived from supertype to subtype.

Using these terms one may depict E/R Diagrams which map the various features of the E/R Model onto a particular domain representation. Using this diagram, one may then translate the design to the implementation platform by using some of the following rules of thumb:

A. Transformation of entity types
   1. each entity type is a base-relationship
   2. the key of this base-relationship is the key property of the entity type
   3. all other properties are mapped to simple fields in the base relationship

B. Transformation of binary relationships
   1. mandatory membership classes
      a. if an entity type E_2 is a *mandatory* member of a many-to-one relationship with entity E_1, then the relation scheme for E_2 contains the prime attributes of E_1
      b. a key posted to another relation is called a *foreign key*
   2. optional membership classes
      a. if entity type E_2 is an *optional* member of a many-to-one relationship with entity type E_1, then the relationship is usually represented by a separate relation scheme containing the prime attributes of E_1 and E_2, along with any attributes of the relation
   3. many-to-many binary relationships
      a. always represented by a separate relation consisting of prime attributes of each of the participating entities, along with any attributes of the relation itself

C. Transformation of n-ary relationships

1. represented by a separate relation consisting of prime attributes of each of the n participating entities, along with any attributes of the relation

D. Transformation of subtypes
   1. represented by a separate relation containing the prime attributes of the supertype, along with any additional attributes of the relation

It is during this mapping stage from the design to the implementation where much of the useful semantic aspects are lost. The diagram is in essence broken into its constituent parts, losing the global cohesion of the system as a whole (i.e., one is left with many independent tables--each of which having a limited "understanding" of the tables around them). Thus, in the mapping to an implementation database management system one loses *semantic abstraction*. Beyond this most notable problem it can be stated that while the E/R Model promotes a solid ground for the semantics behind the *structure* of the database system--superficial semantics--it does not, however, prove to be as useful in describing *what* the data actually represents. The modeling scheme seems to be deficient in several areas. Among these deficiencies: a lack of context-sensitive restriction of values (e.g., a $T_1.F_1$--the value of field F_1 of table T_1--may only contain values α, β, or γ if $T_1.F_2$ is equal to δ), constrained field-value acquisitions, inferable field values (e.g., the value of $T_1.F_2$ may be inferable from the combined values of $T_1.F_1$ and $T_2.F_4$), semantic "key" violations (e.g., the key for table T_1 may not be valid given other field values of the same record or its relationship with records from other tables--i.e., the key may be *syntactically* valid, but *semantically* invalid), etc. A more detailed discussion of these semantic requirements missing from existing modeling schemes is left to the subsequent chapters of this dissertation.

Given the drawbacks of existing semantic modeling approaches--profitability only in the design phase, degradation of semantics in implementation, requirement for reintegration of semantics through customized program front-ends, the inherent lack of complex semantic representations--it is clear that a viable solution to the integration of complex data semantics must inject new, more powerful semantic aspects, while at the same time allowing a seamless translation to existing database platforms *without* the loss of semantic information.

Other semantic modeling approaches have been presented over the years, but most bear much resemblance to the E/R Model and shall, therefore, be dispensed with. The advent of object-oriented database management systems (ODBMSs) has re-injected much of the semantic aspects which were traditionally lost by their relational counterparts. However, the codification of object-oriented databases remains somewhat tedious, often relying on heavily trained "object engineers" to write the complex methods required to embed semantic aspects. Further, although the use of ODBMSs is growing in industry, it still has yet to prove itself as a domineering force in data management. RDBMSs currently dominate the global information pool and it is this dissertation's focus to present a means for complex semantic support for these systems, while at the same time centralizing many different types of RDBMSs with a universal data interface.

2.1 WHAT EXACTLY IS MEANT BY "DATABASE SEMANTICS?"

To more clearly understand the terminology of "database semantics" let us examine the model posed in figure 7. This design represents a very simplistic production plant environment were raw materials are used to create products.

Chemicals (the raw materials) are brought into a plant site and placed in a holding area until they are processed in a reactor to produce a particular product. In this example the state of the production plant may be represented by a handful of relational databases. A *chemical* database is used to store the dictionary of chemicals utilized by the plant. A *holding* database is used to keep track of which of those chemicals are currently available at the plant site and what their quantities are. A *products* database is required to hold the dictionary of products which may be produced by the plant's reactors. Since the production of a given product may require particular quantities of more than one chemical, a *recipe* database is integrated to describe which chemicals and quantities thereof are required to produce a given product. Finally, an *in-process* database is required to keep track of which products will be in-process at what times and in which reactors (an *in-process chemical* database similarly keeps track of which chemicals will be "in-process" to produce a given "in-process" product).

Having created these databases, we are confronted with the reality that the databases by themselves are not much more than mere vessels of information. But what of the underlying semantic issues of this representation scheme? Indeed, what does one mean by "semantics" in this context? Consider the following "semantic" assertions about the fictitious production plant:

a. "The ID of a *new* chemical record should be automatically assigned by the system--never by the user."

b. "Once a chemical ID is assigned it may never change."

c. "The ID field of a new chemical record should not be assigned until the chemical name is known."

d. "Explosive-type chemicals are volatile by nature."

e. "A product which is assigned to the in-process table must exist in the products database."

f. "An in-process record which references a product that requires an authorization number must itself identify a valid authorization number."

g. "Product IDs referenced in the in-process table must exist in the products table."

h. "The value of the special handling field of a chemical record must be either 'Y' or 'N'."

i. "Volatile chemicals require special handling."

j. "Volatile products require authorization numbers."

k. "The inclusion of one or more volatile chemicals in a product's recipe makes that product volatile."

l. "Should insufficient quantity of a given chemical be present in the holding area, any product which requires that chemical based on its recipe may not be produced (i.e., may not be assigned to the in-process list)."

m. "..."

Admittedly a few of the assertions above may be implicitly handled by the database management system chosen for implementation. However, most of the above items cannot be *implicitly* handled by existing RDBMSs. Further, one should plainly see that most of these semantic assertions do not lend themselves to the E/R Model as described above. These types of semantics proceed far beyond the notion of entities, properties, and types. As a result semantic issues such as these must at present be handled by customized front-end applications which control the semantics of the data *before* they reach their respective database destinations. Figure 8 depicts this scheme of front-end semantic integration. Procedural semantics must be codified in a particular programming language for every application. This code must then be cloned and included into each application which need make use of it. Changing one set of semantics may require changes to program code in multiple applications. As one can

readily see semantic integration in this manner is highly decentralized, unconnected, and unorganized. Indeed, a full overview of semantics encompassing the entire system may not be possible due to the hodgepodge sprinkling of semantics between multiple applications. One must take all program code from all applications into account when referencing system-wide semantics. For every new application developed to be integrated with the databases, program code representing semantic aspects must first be identified in other applications and reproduced in the new application--a nightmare for both designer and developer alike. These problems are further compounded when a heterogeneous system is developed as depicted in figure 9. Multiple database engines and multiple programming environments can cause severe organizational problems. Imagine not only reproducing procedural code intended to control semantics, but having to then translate it from one programming language to another. A simple, problematic pattern results: the larger such a system becomes, the more confusing and unwieldy.

3.0 LOGICAL RULE-BASED SEMANTICS

Research into several semantic modeling methodologies has led this researcher to settle on a logic/rule-based representation. By incorporating rule-based technology into its representation scheme database systems can be closely coupled with the power of artificial intelligence/expert systems. Instead of the traditional database management system with an expert system front-end, this research strives to merge the two into one autonomous unit, eliminating the often tedious task of codifying complex linkages between the two. The expert system portion of this marriage would inject semantic control into an otherwise lacking database management system.

This representation must not be confused with other existing forms of database representation schemes such as relational algebra, relational calculus, or SQL. Each of these existing representations boast pros (and cons) of their own, yet all seem to share a mutual absence of higher level abstraction as was mentioned in the preceding chapter--that of *semantic abstraction*.

Research into the merging of rule-base and database technologies is currently dominated by the object-oriented community. Systems such as POSTGRES [Stonebraker, et al, 1991] and STARBURST [Lohman, et al, 1991] have attempted to integrate rule-bases with various object-oriented data management techniques to promote the concept of "active" databases--databases which allow the invocation of rules to perform automated processing in response to specific changes made to the data, regardless of what entity made those changes. Take for example the following POSTGRES and STARBURST rule examples:

POSTGRES:

```
ON event (TO) object WHERE
POSTQUEL-qualification
THEN DO [instead]
POSTQUEL-command(s)
```

STARBURST:

```
CREATE RULE non_empty_dept ON Departments
WHEN DELETED
IF      'SELECT *
        FROM Employees
        WHERE deptno IN
           (SELECT dno
            FROM dd AS (DELETED()))',
THEN 'SELECT d.dno, 'non-empty'
        FROM d as (DELETED())
        WHERE d.dno IN
           (SELECT deptno
```

```
        FROM Employees)',
'ROLLBACK WORK');
```

Rule systems such as these use trigger mechanisms to control rule processing. The POSTGRES syntax listed above allows users to define rules which trigger on specific *events* (e.g., insert, update, retrieve, etc.). The majority of these systems, however, embed the inference engines required for rule processing deep within the database management system itself. As a result, only databases explicitly dedicated to those systems may reap the benefits of rule-based semantics (i.e., semantic integration is homogeneous with respect to POSTGRES-compatible implementations). Little research has thus far been dedicated to relational systems--those systems which continue to dominate academic and commercial domains. STARBURST does share a fundamental link with the research proposed herein as it attempts to extend the existing relational model to include objects and rules. However, STARBURST's rules are beneficial only to SQL-compatible RDBMSs (i.e., STARBURST may be considered heterogeneous only among SQL-based implementations). Single record-manipulative database systems which are not SQL-compatible may not benefit from this set-based, query extension approach. Further, the declaration of rules in this manner may be very difficult for those who understand the "semantics" of the rules they wish to employ, but who may not be fluent in the pragmatics of complex, nested SQL representations. The rule-based approach proposed herein dwells on a logic-based representation scheme which is not dependent on SQL compatibilities and is intended to provide a heterogeneous linkage to any relational system--both single-record manipulative and multi-record manipulative systems alike.

Indeed, there does exist a commonality of purpose between the research presented in this dissertation and existing object-oriented rule-based semantics. The focus of this research, however, is to provide a means for the integration of high-level

rule-based semantics with *existing* relational database management systems, and that this approach should act in a *front-end* capacity to allow a universal bridge to many different types of RDBMSs. Instead of modifying each individual database engine to embed rule-based control deep within, this research proposes to create a single *semantic engine* which then interacts universally with existing database engines to control rule-based semantics without explicit modifications to the database engines themselves (figure 10). By controlling rule processing at the front-end one is able to unify many types of database systems, which would not have been possible through currently proposed embedded approaches.

It is important to note here that the bulk of this research centers itself around the semantics of data *definition* and *manipulation*. Semantic issues described herein concern themselves primarily with the semantic constraints of a database's existence--row/column constraints--and how its existence relates to other databases--table/table constraints. This dissertation is therefore concerned with the creation (*data definition*) of large database systems and their functionality--inserts, deletes, updates, and the like (*data manipulation*). The investigation into semantic repercussions of queries (*data utilization)* is somewhat beyond the scope of this paper. Nevertheless, semantic details of data definition and manipulation would definitely play a role in such an investigation.

Definition and manipulation aspects of database semantics play a crucial role in large database systems--especially systems which are designed to function in a processing environment. These types of systems do not concern themselves so much with large amounts of querying responsibilities (although they certainly can), but are more concerned with keeping track of the *state* of some process (e.g., a warehouse, a plant-site, a tracking system, etc.). Often these types of large databases are tied-in with some form of automation processing system with little (sometimes even no)

human intervention. Thus, it becomes important for such systems to understand the semantics of the structures and global interactions of the databases they employ.

The succeeding chapters detail a logical rule-based approach for representing *and* implementing higher-level database semantics. Later it will be shown how this approach may be integrated into both single-record manipulative RDBMSs and multi-record manipulative RDBMSs alike, providing a *universal* medium for centralized access of many different types of database systems.

3.1 Semantic Rule Syntax

Before proceeding with a formal declaration of the logical rule-based semantics an overview of the rule's syntactical conventions is necessary. Rules within the Semantic Database Management System (SDBMS) abide by the syntactical restrictions of the following extended Backus-Naur form (BNF) grammar (*note: BNF operators are distinguished in* ***bold-face****, and should not be confused with valid SDBMS operators which are not in bold-face)*:

<rule> ::= {<category>} <lhs> ⇒ <rhs>
<lhs> ::= <test-comp> **[** <operator> <lhs> **]** | ¬ (<lhs>)
<rhs> ::= <rhs> ∧ <rhs> | <set-comp> | <function>
<operator> ::= ∧ | ∨
<test-comp> ::= <table> **[** [<index>] **]** .<field> **[** [Δ] **]** (<t-binding>)
<set-comp> ::= <table> **[** [<index>] **]** .<field>(<s-binding>)
<t-binding> ::= <t-oper> <t-binding> | <variable> | <constant>
<s-binding> ::= <s-binding> <s-oper> <s-binding> | <variable> | <constant>
<t-oper> ::= > | < | ...
<s-oper> ::= + | - | ÷ | ×
<category> ::= A | C | R
<function> ::= <any valid SDBMS function/operation>
<index> ::= <a number greater than 1 (i.e., 2, 3, 4, ...)>

<variable> ::= <an upper case letter (i.e., A, B, C, ..., Z)>
<constant> ::= <a constant expression (e.g., 12, 32.4, 'Dark', etc.)>
<table> ::= <any database name within the semantic domain>
<field> ::= <any valid field name for the given table>

Some valid syntactic examples of SDBMS rules are as follows:

1. {C} plant2.reactor_id(X) ∧ reactor_schedule.id(X)
   ∧ reactor_schedule.status('to-be-cleaned')
   ⇒ plant2.override('shut down')

2. {C} reactors.chemical_id(X) ∧ ¬ (chemicals.id(X))
   ⇒ Abort(reactors)

3. {C} chemicals.chemical_id[Δ](X)
   ∧ reactors.chemical_id(X) ∧ chemicals.chemical_id(Y)
   ⇒ reactors.chemical_id(Y) ∧ Update(reactors)

We can see in the first example--a committal rule, as distinguished with the "{C}" prefix--that "plant2" and "reactor_schedule" denote table names, while "reactor_id," "id," "status," and "override" reference field names. The binding "X" indicates a variable while the binding 'shut down' is a constant. In the second rule example--also a committal rule--we see the use of the negation operator (¬, or NOT) and the use of an SDBMS function, namely "Abort." The convention has been adopted to note <table>s and <field>s in all lower case (e.g., plant1.reactor_id). String constants are always shown within single quotes (e.g., 'Yes', 'No', 'Y', 'Fred', etc.), while numeric constants simply appear as numbers (e.g., 1, 15, 32.43, etc.). Variables are distinguished as a single upper case letter (e.g., X, Y, etc.). Finally, SDBMS functions are customarily denoted with the first letter of the function name capitalized (e.g., Abort, Delete, Update, etc.).

Having detailed the syntactic constraints of such a language it is important to distinguish between two types of *semantics* which will be discussed in the succeeding

chapters of this dissertation. The first type shall be deemed *rule-semantics* and is intended to indicate the semantics of the SDBMS's rule-based *language* itself (i.e., the meaning behind the rule-language represented by the BNF grammar above as understood by the semantic engine). The second type shall be deemed *data-semantics* and is meant to reference the semantic knowledge represented *by the rules* (i.e., the actual semantics which pertain to the databases as represented by the SDBMS rules). Data-semantics can be further broken down into two subtypes: *extensional semantics* and *intensional semantics*. Extensional semantics are defined herein to refer to the semantics of the tuples of the base relations or *inter*-table relations. With extensional semantics one may define the relations between tables on a global scale. Intensional semantics are defined herein to refer to the meaning *within* a given table or *intra*-table relations. Where extensional semantics concerns itself with table to table contentions, intensional semantics concerns itself with row-to-row and column-to-column constraints and other repercussions within a single table on a more local scale. Figure 3 depicts the differences between the extensional and intensional paradigms.

The next section illustrates the rule-semantics of the SDBMS logical language. The various types of semantic database information which are capable of being stored within SDBMS rules--extensional and intensional data-semantics--are detailed in succeeding chapter(s).

3.2 Semantic Rule Categories and Rule-semantics

Within this proposed semantic representation each database system has associated with it a semantic knowledge base, a set of logical rules, which govern its integrity constraints, consistency, redundancy verification/elimination, inferable field

values, type checking, security, etc. Rules described within each semantic knowledge base are divided into four categories: *acquisition* rules, *committal* rules, and *removal* rules. Acquisition rules are referenced during *buffering* of a new database record prior to insertion of the record into the database. This represents a slightly new approach to database technology as the SDBMS has access to new data before it is actually committed to the database itself. Committal rules govern semantic functionality after a new record has been sufficiently buffered *just prior* to insertion and committal to the database. Committal rules also govern semantic functionality *just prior* to modification of an existing record (i.e., record updates). Removal rules are referenced *just prior* to a queued record's deletion from the database.

3.2.1 *Acquisition* Rules

The main premise for the *acquisition* rule category was to increase user-interface performance. By adding a built-in buffering capacity for a new record's field values the SDBMS becomes a powerful tool since real-time semantics may be enforced as *each* field value is independently acquired--before the actual insertion of the entire record into its respective database. Field values for a new record are acquired through the SDBMS Semantic Interface (SI). As each new field value is acquired by the SI it is immediately conveyed to the SDBMS semantic engine (SE) where any semantic rules pertaining to that new field value are considered. Results/consequences of those rules are immediately returned to the semantic interface. Thus, semantic constraints may be enforced on data *before* it is actually committed to the storage medium.

Take for example the following acquisition rule:

$$\{A\}\ plant1.product\_id(X) \wedge \neg(production1.product\_id(X)) \Rightarrow RejectValue(plant1.product\_id)$$

Here, the "{A}" prefix denotes an acquisition rule. The atoms "plant1" and "production1" refer to tables within the semantic domain. The atom "product_id" denotes a field belonging to those tables. "X" represents a variable which is to be bound with a field value. Semantically, this rule maintains that any newly acquired product identification number for a record which is to be inserted into the "plant1" database *must* currently exist in the "production1" database. This particular example pertains directly to the *referential integrity rule*:

"The database must not contain any unmatched foreign key values."

-- Date, 1990

In this example we may think of "plant1.product_id" as a *foreign key* to the "production1" table. Since some forms of database implementation platforms allow for referential integrity it may seem unclear why such a semantic system should redundantly accomplish the same task. The SDBMS's ability to handle referential integrity provides several benefits over simple foreign key verification. First, should a database implementation platform be incapable of handling referential integrity the SDBMS may enforce this integrity itself. Hence, all database implementation platforms governed by the SDBMS are now capable of handling referential integrity. Second, should the database platform be capable of enforcing referential integrity the SDBMS offers an alternate approach (i.e., instead of traditionally declaring foreign keys one may simply represent referential integrity constraints by way of logical rules). Third, since acquisition rules are processed immediately as field values are acquired

the user/program can be made aware of a referential integrity breach immediately *before* other field values are declared and the record is inserted (or updated) in the table. Fourth, by defining referential integrity constraints with logical rules we are not limited with foreign key constraints. The referential integrity rule insists that foreign keys must match, very specifically, *primary keys*, not alternate keys of other tables. By using logical rules the SDBMS is capable of handling *extended referential integrity*. This is similar to referential integrity (as was seen in the above example), however, we may define referential integrity constraints which *do not* insist upon specifically matching foreign keys. Take for example the following rule:

$$\{A\}\ \text{plant1.type('toxic')} \wedge \neg(\ \text{production1.type('toxic')}\ ) \Rightarrow \text{RejectValue(plant1.type)}$$

In this example, assuming that the "product_id" field is the primary key in both the "plant1" and "production1" tables, and the "type" field is a simple field in those tables, the SDBMS is able to extend the notion of referential integrity without the use of foreign keys. Here a "plant1" record may not contain the value 'toxic' in the simple field "type" unless there exists *at least one record* in the "production1" table whose simple field "type" contains the value 'toxic.'

Acquisition rules always fire in a forward-chaining fashion matching the *first component* of the antecedent of the rule to the newly acquired field value. The first component of the antecedent of an acquisition rule *always* pertains to a newly acquired field value. In the original sample rule shown above the variable X is bound to the new value of the "product_id" field from the "plant1" database. This bound variable is then used in the second component of the rule's antecedent to test (search) for its existence in the "production1" database. Since the second component is surrounded by the negation operator ($\neg$), should there not exist a "production1" record where the

"product_id" field value equals X (i.e., a failed search for a record where production1.product_id = X) the rule's consequent is executed. Here the only component of the consequent states that the newly acquired field value must be rejected. Exactly how the SE goes about considering a given SDBMS rule will be discussed in subsequent sections of this chapter.

The interested reader will note that this semantic rule exhibits what is known as the *closed world assumption*.

> "[The closed world assumption] states that omission of a certain tuple from a given relation implies that the assertion corresponding to that tuple is false."
>
> -- Date, 1990

Thus, if we think of each row (tuple) in a given table as a logical assertion about the existence of something in the world, the absence of such a row (and hence the absence of the assertion) indicates that that 'thing' does not exist.

3.2.1.1 Buffering New Database Records

In order to make use of the SDBMS acquisition rules a *dialogue* must ensue between the user and the SE by way of the SI. A sample dialogue might look as follows: (For now SI communication will take the form of a simple procedure-like command language. Later it will be described how the SI can be integrated into both single-record manipulative and multi-record manipulative relational database frameworks and how basic natural language techniques may be incorporated to create a powerful interface.)

```
NewRecord( "plant1" );
```

```
SetField( "plant1", "product_id", "A123B920" );
...
```

The first command issued to the SE, *NewRecord*, functions in two capacities. First, it creates a *context* for the semantic dialogue--namely, that the "plant1" database will be used in the succeeding dialogue and hence its semantic knowledge base must now be loaded into working memory if it has not been previously. Second, a buffer must be created for the new record's data values to be stored before the record is committed to the database by an insertion command. As each field value is independently acquired via the *SetField* command the value is incorporated into the buffered data structure and any acquisition rules which contain the field name in the first component of its antecedent are considered. Once an antecedent is proven true the resulting consequent of the acquisition rule is then executed. Should the SetField command fail (because of a breach in semantic integrity as dictated by the acquisition rules), the user/program is made aware of this breach immediately and the field's value is not set.

3.2.2 *Committal* Rules

The second category of semantic rules is the *committal* rule. These rules like the acquisition rules are forward-chaining. However, unlike semantic acquisition rules, committal rules are considered in bulk. Given an acquisition rule, it is only considered if and *when* the field value of the first component in its antecedent is acquired--during this time all other non-conforming acquisition rules are ignored. Alternately, committal rules may be considered en mass by the SE as each rule must be verified before the actual insertion or update of the buffered record into its respective database occurs. Take the following committal rules for example:

1. {C} plant1.product_id(*null*) ⇒ Abort(plant1)

2. {C} plant1.product_id(X) ∧ production1.product_id(X) ∧
   production1.serial_required('Y') ∧ plant1.serial(*null*)
   ⇒ AcquireValue(plant1.serial)

3. {C} plant1.chemical_id(X) ∧ chemicals.id(X) ∧
   chemicals.volatile('Y')
   ⇒ plant1.special_handling('Y')

The prefix "{C}" denotes the committal rule category. The first rule maintains that the "product_id" must be non-null for insertion/update to succeed. This form of semantic representation may not be required in most database management implementations as they implicitly test for non-null values--usually reserved for key fields. However, some lower-level database platforms do not allow implicit non-null checking and hence can be made more powerful by the SDBMS. The interested reader will note that this particular rule could not be classified as an acquisition rule since the component in its antecedent binds to *null*. The reason for this is obvious since a field value is *null* until it is acquired. Further, since acquisition rules are considered only as a field-value is acquired, this type of rule would never be considered if the "product_id" field value was never acquired and hence was *null*.

The second committal rule states that if a product is to be inserted/updated in the "plant1" database and the product is listed as requiring a serial number in the "production1" database, then the field "serial" for "plant1" must be non-null. Here we begin to see the power of the SDBMS as a field can be *semantically* defined as non-null within a specific context. In one context the "plant1" record's "serial" field is defined as non-null (i.e., when the "production1" database dictates that the product requires a serial number). In another context the "plant1" record's "serial" field is

allowed to be null (i.e., when the "production1" database dictates that the product does not require a serial number).

In the third example the rule states that if the "plant1" record to be inserted/updated is listed as a volatile chemical, then the "special_handling" field of the "plant1" record should be automatically set to 'Y'. This type of rule implies an *inferable field value* since the value of the "special_handling" field of a "plant1" record *may* be (indirectly) inferred from the "chemical_id" field value.

Semantic committal rules conform to several rule-semantic constraints. The first component of the antecedent is *always* a "<table>.<field>(<binding>)" designation, where the <table> denotes the type of record which is currently under consideration for insertion or update and the <field> denotes a valid field name for a record in that table. The <table> portion of this component references the record which has been previously buffered in semantic context (as described above)--either a new record which awaits insertion or an existing record which has been modified and now awaits update and committal to its respective database. Any reference to this <table> in the remainder of the rule will access the buffered data for that record in semantic context. The <binding> of the <table>.<field> pair represents the currently buffered field *value* for that record in semantic context. The <binding> itself may denote either a variable or a constant. In the case of a variable binding the current value of the designated field is immediately pulled from the buffered record currently in semantic context and bound to the variable. Once the variable is bound consideration of the rule's antecedent continues--that variable may *not* be re-bound in the remainder of the rule's consideration. In the case of a constant binding the field value which has been buffered within the semantic context is *tested* against the constant. If the test succeeds the remainder of the rule's antecedent is then

considered. Should the test fail, consideration of the rule ends and the next applicable committal rule is considered.

It is important to note that all committal rules begin with a "<table>.<field>(<binding>)" component and that the <table> atom of this component defines which *type* of record the rule is to be applied. By this convention the SE can immediately discern that the three sample rules listed above apply to a "plant1" record to be committed to the "plant1" database, as all rules begin with a component which references the "plant1" table. Hence, the <table> atom of the first component in an committal rule references the record which has just been *buffered* in semantic context. Any further references to that <table> in the remainder of the committal rule will access the buffered field values of that record.

Two types of binding are applied during rule consideration: *record binding* and *field-value binding*. Within field-value binding there exist two subtypes of binding: *variable binding* and *constant binding* (described above). To illustrate the functionality of these two types of bindings--namely, record binding and field-value binding--let us examine exactly *how* the SE considers the third rule in the example shown above.

As previously declared the first component defines the type of record which is under consideration for committal: "plant1." The "plant1" reference is immediately *bound* to the "plant1" record which has been buffered in semantic context. Field-binding occurs as the variable "X" is bound to the buffered "chemical_id" field value of the "plant1" record. The <table> atom of the second component of the antecedent references the "chemicals" database. At this time the record which is referenced by the "chemicals" atom of the component is *unbound*. Implicit in the interpretation of this second component is the SE's knowledge about the "chemicals" table's *structure*--how many fields make up the table's primary key, what fields belong to the composite

key, etc. In this example let us say that the "chemicals" database consists of a single-field primary key: "id." Since (1) the "chemicals" record is unbound, (2) the variable "X" is bound, and (3) the "id" field uniquely identifies a "chemicals" record (since it is the only field composing the primary key), the SE interprets the "chemicals.id(X)" component as follows: "search for the record in the 'chemicals' database where the value of field 'id' is equal to 'X.'"

Should this search fail, the "chemicals" record cannot be bound and hence consideration of the rule ceases since its antecedent cannot be proven true. Upon a successful search the "chemicals" record is *bound* to the record resulting from the search. The bound record now becomes part of semantic context and any further references to fields of the "chemicals" database will access the bound record's field values. Since the "chemicals" record is now bound, the "'Y'" in the third component instructs the SE to *test* whether or not the "volatile" field value of the "chemicals" record, which was bound in the previous step, is equal to 'Y.' Should the third component hold true, the consequent of the rule is carried out.

As noted above, "plant1" was bound to the new-record which was previously buffered in semantic context. Thus, the only component of rule three's consequent--"plant1.special_handling('Y')"--instructs the SE to act on the buffered record for "plant1." The component is fully translated by the SE to mean: *set* the value of the field "special_handling" of the buffered record for the "plant1" table to 'Y' (note the single quotes around the 'Y' making it a constant value and not the variable Y).

In general, bindings of components found within the antecedent of an SDBMS rule instruct the engine to *variably bind* or *test* the existing field value of a record within semantic context. Bindings found within the consequent of an SDBMS rule always instruct the SE to *set* the value of a field. This set operation will override any existing value for that field in favor of the value inferred by the rule. Special semantic

functions are also available for use within the consequent of an SDBMS rule. The first rule in the above example uses the "Abort" function which instructs the SE to cease all remaining committal rule considerations and abort the insertion/update operation of the buffered record into its respective database. Additional SDBMS functions will be discussed in subsequent chapters.

3.2.3 *Removal* Rules

Removal rules are similar in format to committal rules. However, removal rules are considered just prior to deletion of a record from a database. Take the following removal rule for example:

$$\{R\}\ storage.chemical\_id(\ X\ ) \wedge chemical\_removals.chemical\_id(\ X\ ) \wedge chemical\_removals.instances(\ Y\ ) \Rightarrow chemical\_removals.instances(\ Y + 1\ ) \wedge Update(\ chemical\_removals\ )$$

The prefix "{R}" denotes a removal rule. As in the case of the SDBMS committal rules, the first component of the antecedent in a removal rule identifies what type of record the rule is to be applied before a deletion occurs. Hence, the <table> atom of the first component in the rule's antecedent is always bound to the record which is to be deleted. More accurately, this <table> atom is bound to the <table>'s current record within the semantic context. The sample removal rule above, therefore, pertains to a "storage" record which is to be deleted. Upon consideration of the rule by the SE the reference "storage" is immediately bound to the "storage" *record* which is about to be deleted. By the same token the variable X is bound to the *value* of the "chemical_id" field of the bound "storage" record. The <table> atom of the second

component in the antecedent "chemical_removals" is currently unbound and, therefore, requires a search to bind its record. The SE carries out the search for the "chemical_removals" record where the field "chemical_id" is equal to the variable X. Should this record be found it is bound to the "chemical_removals" reference and consideration of the rule's antecedent continues. Since "chemical_removals" record is now bound, the third component of the antecedent binds the variable Y to the "instances" field value of that record.

The first component of the consequent makes use of a semantic <operation> (as syntactically outlined in the BNF grammar above)--namely, the "+" operation. This component causes the SE to *set* the "instances" field value of the previously bound "chemical_removals" record to Y + 1 (e.g., if Y=1, then Y+1=2). The SE's translation of the "+" operation makes use of the *operator overloading* paradigm, controlled by the SE. If the operands of the "+" operation are character strings, the result of the operation is the concatenation of those strings. If the operands of the "+" operation are numbers, the result is the numerical addition of those numbers.

The second and final component in the rule's consequent makes use of another SDBMS function "Update." This function instructs the SE to update the current record in semantic context for the "chemical_removals" table--the record bound by the search which had taken place during consideration of the second component of the antecedent.

3.3 Use of the Index Constraint in Rule-semantics

In the preceding section many rule-semantic repercussions of the SDBMS BNF grammar have been touched on. One aspect, however, has been left unattended: the

use of the <index> constraint on record bindings. Consider the following portions of the BNF grammar listed above:

<test-comp> ::= <table> [[<index>]] .<field>(<t-binding>)
<set-comp> ::= <table> [[<index>]] .<field>(<s-binding>)
<index> ::= <a number greater than 1 (i.e., 2, 3, 4, ...)> | Δ

To fully understand how the <index> constraint is utilized within the SDBMS, let us examine the following removal rule:

{R} storage.product_id(X) ∧ ¬(storage[2].product_id(X))
∧ storage.type(Y) ∧ type_log.type(Y) ⇒ Delete(type_log)

This rule maintains that should a certain product record be removed from the "storage" database and no other records of that product exist in the "storage" database, and there exists a "type_log" record for that product's type, then remove that "type_log" record from the "type_log" database. The first component of the antecedent binds the reference "storage" to the record which is to be deleted. The variable "X" is then bound to that record's "product_id" field value. The second component makes use of the negation operator (¬) which indicates that the result of the second component should be negated. The <index> constraint is used in the second component ("storage[2]") to indicate that, although it is to be bound to a "storage" record, it *must* be a *different* record than the "storage" record bound in the first component of the antecedent. The "storage[2]" reference at this point is, therefore, unbound--implying a search, since it lies within the rules antecedent (as noted in the previous sections). Based on the remaining atoms of the second component in the antecedent the SE executes a search of the "storage" database where the value of the "product_id" field is equal to "X." This is a somewhat complex

search since the SE must ensure that any record found during this search is *not* identical to the record bound to the original "storage" reference--recall that the originally bound "storage" record has not yet been deleted from the database and therefore still resides within it. The SE accomplishes this task by performing a basic search and then, in the case of a keyed database, comparing each (possibly composite) key field value of the record resulting from the search with the respective field values of the previously bound "storage" record. If the similarity check fails, the record resulting from the search is bound to the "storage[2]" reference. If the similarity check succeeds, the search continues until a differing record is found or further searching is not possible. Should the "storage" database be non-keyed (i.e., more than one identical record may exist) the SE must employ some other form of similarity check (perhaps checking the literal record number as identified by the database file). Continuing with consideration of the rule, should the search prove successful, the negation operator halts consideration of the antecedent and the next applicable removal rule is then considered. However, should the search fail, the negation operator causes continued consideration of the remainder of the rule's antecedent. The third antecedent binds the variable Y to the value of the "type" field for the buffered "storage" record bound by the first component. The fourth component of the antecedent searches out the "type_log" database for a record whose "type" field is equal to "Y" and, if found, binds this record to the "type_log" reference. The rule's consequent makes use of yet another SDBMS function "Delete" which performs a removal action on the "type_log" record which was bound during consideration of the fourth component in the antecedent. Note that this removal action, in turn, spawns consideration of all removal rules which apply to a "type_log" record.

It may be stated that any reference to a <table> atom of a component which does not make use of an <index> atom is implied to be of index, 1. For example, the above rule may be thought of as follows:

$$\{R\}\ \text{storage}[1].\text{product\_id}(X) \wedge \neg(\text{storage}[2].\text{product\_id}(X)) \wedge \text{storage}[1].\text{type}(Y) \wedge \text{type\_log}[1].\text{type}(Y) \Rightarrow \text{Delete}(\text{type\_log}[1])$$

Further, any use of a $\langle table_1 \rangle[\ \langle index_1 \rangle\ ]$ component implies uniqueness of that bound record to all other $\langle table_1 \rangle[\ \langle index_i \rangle\ ]$ references within that SDBMS rule where $i \neq 1$. The use of the <index> atom gives great power to the SDBMS as rules may be written which define semantic repercussions of one record within the context of other record(s) of the same type.

3.3.1 Use of the Δ Index

Careful examination of the committal rule category might lead one to the question: "What if a particular system requires a certain committal rule to fire for an update of a record, *but not* for an insert of that record--how can this be accomplished when both update and insert manipulations are governed by the same rule category?" For example, let us say we had a database system which required the tracking of various user updates. In particular let us say there exists a "daily_log" database, and we wish to keep track of how many times the value of the field "location" is modified within this record. More specifically, we are not interested in how many times the record (as a whole) was updated, but rather how many times a specific field value was modified. This type of rule may be represented as follows:

{C} daily_log.location[Δ](X) ∧ daily_log.id(Y)
∧ tracking.id(Y) ∧ tracking.displacements(Z)
⇒ tracking.displacements(Z + 1)
∧ Update(tracking)

The Δ index is used to reference *differing* field values from the time an existing record is queued to the time the record is to be re-committed to its respective database. In this example, should a particular "daily_log" record be queued with an original "location" value of L_1, and during the time of buffering this value changes to L_2 (via the SDBMS command SetField), the binding of "daily_log.location" would equal L_2, and the binding of "daily_log.location[Δ]" would equal L_1.

A <table>.<field>[Δ] binding is *only* valid (i.e., can be bound) when an *existing* record is queued and the SDBMS SetField command is used on that <table>.<field> designation. Thus, if a "daily_log" record is queued, but the "location" field is not acted upon by a SetField command, and the record is updated, the above rule would not be applicable, since there would be no binding for "daily_log.location[Δ]" (i.e., since the component cannot be bound, its truth value is FALSE, and consideration of the rule terminates). Further, any buffered field-values for a new record (i.e., using the SDBMS SetField function after use of the SDBMS NewRecord command) *never* associate with a Δ index reference, since the record is brand new, there can be no a priori field values.

Thus, the above rule functions wonderfully for tracking modifications to the "location" field value, since "daily_log.location[Δ]" may only be bound when the "location" field value actually changes. Further, we see that *only* modifications/updates are tracked (i.e., not inserts) since by definition of the Δ index reference, no Δ index bindings are possible when a new "daily_log" record is born (i.e., NewRecord(daily_log)...Insert(daily_log)). However, upon close inspection of the rule one may discover a fallacy in that the rule always assumes the existence of a

"tracking" record for the "id" in question. Indeed, for this type of tracking technique to operate successfully we must add the following committal rule to the semantic knowledge base for the "daily_log" database:

$$\{C\}\ \text{daily\_log.location}[\Delta](\ X\ ) \wedge \text{daily\_log.id}(\ Y\ ) \wedge \neg(\ \text{tracking.id}(\ Y\ )\ )$$
$$\Rightarrow \text{NewRecord}(\ \text{tracking}\ ) \wedge \text{tracking.id}(\ Y\ )$$
$$\wedge\ \text{tracking.displacements}(\ 1\ ) \wedge \text{Insert}(\ \text{tracking}\ )$$

This rule maintains that a "tracking" record should be inserted upon the first modification of the "location" field value in an existing "daily_log" record. In addition this new "tracking" record should contain a "displacements" field value of 1, since insertion of this record indicates the first time the "location" field was updated in an existing "daily_log" record. With the addition of this committal rule, our displacement tracking technique is sound.

It is important to note that the Δ index adheres to the closed world assumption. Take the following component reference for example, assuming the variable X has been *previously* bound to a value:

$$\neg(\ \text{plant1.product\_id}[\Delta](\ X\ )\ )$$

This expression is true in two cases: (1) if "plant1.product_id[Δ]" cannot be bound (i.e., the value for field "product_id" remains unchanged); or (2) if "plant1[Δ].product_id" can be bound, but its value is not equal to X. Thus, we see evidence of the closed world assumption since the assertion is FALSE if a Δ index reference does not exist (i.e., cannot be bound).

4.0 IMPLEMENTATION OVERVIEW OF THE SDBMS SEMANTIC ENGINE

Having described the syntactic and rule-semantic aspects of the SDBMS logical rule-based language, it must be illustrated how such a system would function. To accomplish this illustration, we must examine the basic functionality which is required by the SDBMS to facilitate its usage as a universal medium between many different database management system platforms (as depicted in figures 2 and 10). Although an explicit implementation of the semantic engine is beyond the scope of this dissertation, it is nonetheless important to describe the basis upon which such an implementation must conform.

4.1 SEARCH-TEST-ACT Chain Reductions

Taking the rule-semantic constraints described in chapter 3 into consideration, one may assert the basic conclusion that all rules, regardless of classification, may be reduced to a *SEARCH-TEST-SET chain*. To explain this assertion let us consider the following committal rule:

$$\{C\}\ \text{plant1.chemical\_id}(\ X\ ) \wedge \text{chemicals.id}(\ X\ ) \wedge \text{chemicals.volatile}(\ \text{'Y'}\ ) \Rightarrow \text{plant1.special\_handling}(\ \text{'Y'}\ )$$

The astute reader will recognize this example rule as one presented earlier in chapter 3. Summarizing the SE's consideration of this rule we have the following: (1) the reference "plant1" is bound to the "plant1" record buffered in semantic context; (2) variable "X" is bound to the value of the buffered "plant1" record's "chemical_id" field; (3) since the value of field "id" uniquely identifies a record in the "chemicals"

database (recall, "id" is the only field composing the primary key), the value bound to "X" is used to SEARCH the "chemicals" database for a record with that "id" value; (4) should this record exist, (5) the "volatile" field value of the bound "chemicals" record is then TESTED for its equivalence to 'Y;' (6) if all components of the antecedent prove true the only component in the consequent SETs the value of the "special_handling" field of the bound "plant1" record to 'Y.' Thus, the rule may be reduced as follows:

SEARCH: chemicals.id = plant1.chemical_id
TEST: ε
TEST: chemicals.volatile = 'Y'
SET: plant1.special_handling = 'Y'

In this particular reduction we see four components in the SEARCH-TEST-SET chain: one SEARCH component, two TEST components, and one SET component. Looking back at the above summary for the SE's consideration of this rule, binding occurs in steps (1) and (2). Step (3) is carried out by the SEARCH component in the chain. The SEARCH component is read as follows: the <table>.<field> operand to the left of the equal sign (=) identifies the <table> to be searched and the first <field> to be constrained in the search. The <table>.<field> operand to the right of the equal sign (=) identifies the *bound* record (<table>) and field-value (<field>) in semantic context. This second operand provides the constant required to complete the search. The first TEST component accomplishes step (4). The ε parameter is a Boolean variable, global to the SE, which is always set when a SEARCH occurs. This variable is set to true if the SEARCH was successful, false if unsuccessful. Step (5) occurs during evaluation of the second TEST component. Finally, if all TEST components prove true, the SET component executes step (6).

The SET portion of the SEARCH-TEST-SET chain reduction should more explicitly be referred to as an ACT--i.e., SEARCH-TEST-ACT. We say ACT because we may not only wish to have the ability to simply SET the field values of records queued in semantic context, but rather to perform powerful ACTIONS on them. Several rules found in the preceding chapter have, within their consequents, special SDBMS function assertions which instruct the SE to perform various actions on the context record(s) queued within semantic context.

Thus, we should re-evaluate the above reduction chain to the following form:

SEARCH: chemicals.id = plant1.chemical_id
TEST: ε
TEST: chemicals.volatile = 'Y'
ACT: Set(plant1.special_handling = 'Y')

One should also note that the SEARCH-TEST-ACT chain may be reduced further *if and only if* the database engine for the table(s) referenced within the antecedent are SQL-compatible. This further reduction would result in a slightly different chain as the second TEST portion of the above example could be incorporated into the SEARCH portion. Given the rule listed above the reduction could be as follows:

SEARCH: chemicals.id = plant1.chemical_id
∧ chemicals.volatile = 'Y'
TEST: ε
ACT: Set(plant1.special_handling = 'Y')

And, the SQL interpretation of the SEARCH portion would be:

```
EXISTS
    ( SELECT    *
      FROM      chemicals
      WHERE     id = X AND volatile = 'Y' );
```

Note: X = plant1.chemical_id (a buffered field value which would be constant at time of execution)

Some database engines, however, do not boast such powerful query capabilities. Borland International's PARADOX™ ENGINE, for example, only allows record searches on subsequent composite key fields or a single non-key field. In this case, since "id" is the only field in the primary key of the "chemicals" table, and "volatile" is *not* a key-field, the SE must make use of the SEARCH-TEST-TEST-ACT chain reduction, listed above, instead of the latter SEARCH-TEST-ACT chain reduction.

Thus, it is the semantic engine's duty to determine which database engine applies to the referenced database(s) and hence which form of reduction is required. For less powerful database engines, e.g., those which *only* allow searching on keyed fields or any other *single* field value, the SEARCH-TEST-TEST-ACT reduction is necessary. However, as we have seen for more powerful database engines which allow detailed searching, the SEARCH-TEST-ACT chain reduction may be a better strategy--note, however, that the SEARCH-TEST-TEST-ACT chain reduction is *still* possible with more powerful engines, but may not represent the most efficient accessing technique for those database implementations.

Some rules may simply require a TEST-ACT chain reduction as in the following inferable field-value, committal rule:

{C} production1.intensity_level(5) ⇒ production1.reaction_time(6.3)

Reduction:

TEST: production1.intensity_level = 5
ACT: Set(production1.reaction_time = 6.3)

One final piece of information which is paramount in rule reduction is the *type* of rule (i.e., acquisition, committal, or removal) and the *table/database* to which it applies. By incorporating this information into the reduction construct the SE knows *when* to consider a rule and *what* buffered record within semantic context is under consideration.

The following lists the final reductions for the two sample rules in this section, respectively:

RULE: C/plant1
SEARCH: chemicals.id = plant1.chemical_id
TEST: ε
TEST: chemicals.volatile = 'Y'
ACT: Set(plant1.special_handling = 'Y')

RULE: C/production1
TEST: production1.intensity_level = 5
ACT: Set(production1.reaction_time = 6.3)

Hence, in the first reduction the RULE component dictates to the SE that this rule should be considered *before* a buffered "plant1" record is committed to the "plant1" database and that any "plant1" references within the reduction should be bound to the field-values buffered in semantic context--thus, record and field-value binding may be accomplished. Appendix A lists some sample rules and their SEARCH-TEST-ACT chain reductions.

Once again it must be stressed that SEARCH-TEST-ACT chain reductions require the SE's knowledge of the *structures* of the databases referenced within the

rule. The above examples have dealt with single-key field databases and were reduced on that basis. However, not all databases conform to single-field keys and require a bit more work in reduction. Take the following rule, for example:

{A} order.type('toxic') ∧ order.customer_id(X) ∧ order.locale(Y)
∧ customer.id(X) ∧ customer.locale(Y)
∧ customer.handling_level(2)
⇒ order.shipping('rail car')

Let us say in this particular example that the table "order" has a composite key made up of fields "customer_id" and "locale" respectively. Similarly, let us say that the "customer" table has a similar composite key made up of the fields "id" and "locale" respectively. Given this information the SE can reduce the rule as follows:

RULE:	A/order
TEST:	order.type = 'toxic'
SEARCH:	customer.id = order.customer_id
	∧ customer.locale = order.locale
TEST:	ε
TEST:	customer.handling_level = 2
ACT:	Set(order.shipping = 'rail car')

Should the SE only have searched on "id," we would not be guaranteed that the correct record was queued. Thus, we see the use of two search constraints instead of simply one, as the table in question requires two field-values (i.e., "id" and "locale," respectively) to uniquely identify a record.

Not only does the structure of a database play an important role in rule reduction, but also the power of the database engine governing that database which will ultimately be used to consider (and, if necessary, fire) the rule. Once again the preceding example assumes a low-level database engine which only allows searching on keyed fields. Should a more powerful engine be available for the referenced

database(s) (e.g., a SQL-compatible engine) the following reduction would most likely prove more efficient or at least somewhat more elegant:

```
RULE:     N/order
TEST:     order.type = 'toxic'
SEARCH:   customer.id = order.customer_id
          ∧ customer.locale = order.locale
          ∧ customer.handling_level = 2
TEST:     ε
ACT:      Set( order.shipping = 'rail car' )
```

It should be noted that rule-reduction is based not only on the format of the rule itself, but also on the structures of the database(s) referenced within the rule *and* the power of the database engine(s) applicable to those database(s).

4.2 Semantic Context

The term, *semantic context*, has been used somewhat loosely in the preceding chapters. To understand exactly how the SE would function, this term must be clearly defined. Perhaps the easiest analogy to semantic context is a working memory which maintains a dictionary of record and field-value bindings. Two types of data references are held within this dictionary: (1) *explicit* references--the buffered field-values of records to be inserted, updated, or deleted in their respective databases, or the buffered field-values of records simply queued for reference via the SI; and (2) *implicit* references--buffered field-values of any other records queued during consideration of a particular rule. The first type of references exist within the system (i.e., are non-volatile) until explicitly dumped by the system (as will be described in detail below). The second type of references exist only during consideration of the

rule in which they were queued (i.e., are volatile immediately upon completed consideration and potential firing of a rule).

For example, let us examine the following database structures, a sample rule, and its SEARCH-TEST-ACT chain reduction:

TABLE: **plant2**

reactor_id : *alpha-numeric field (only KEY)*
...
override : *alpha-numeric field*

TABLE: **reactor_schedule**

id : *alpha-numeric field (only KEY)*
...
status : *alpha-numeric field*

{C} plant2.reactor_id(X) ∧ reactor_schedule.id(X)
∧ reactor_schedule.status('to-be-cleaned')
⇒ plant2.override('shut down')

RULE: C/plant2
SEARCH: reactor_schedule.id = plant2.reactor_id
TEST: ε
TEST: reactor_schedule.status = 'to-be-cleaned'
ACT: Set(plant2.override = 'shut down')

Figure 4 depicts a visual account of this example. Prior to consideration of this rule the field-values of a new "plant2" record have been buffered within semantic context (part A of figure 4). Let us say 'SAM1' has been buffered as the value of "plant2.reactor_id." This indicates that semantic context has the value 'SAM1' associated with the data reference "plant2.reactor_id." Substituting this value pulled from semantic context, the SEARCH portion of the reduction chain then reads: "reactor_schedule.id = 'SAM1,'" and this record is searched-out. Let us say the

record does exist (i.e., ε = TRUE). At this time, since a "reactor_schedule" record has just been queued, its field values are loaded into semantic context (part B of figure 4). In particular let us say its "status" field does indeed equal 'to-be-cleaned,' causing the second TEST component to prove true. Since all TESTs prove true, the ACT is executed, associating the data reference "plant2.override" with the value 'shut down.'

The diagram visualizes SDBMS semantic context as a series of hash tables[1]. The first hash table is used to reference <table> entries (i.e., the names of *records* which currently reside within semantic context). These <table> entries shall be deemed *table cells* and are depicted in the diagram as rectangles. Note that appended to each table cell identity is its index. In the diagram all table identities are appended with "[1]"--recall that any <table> reference which does not explicitly display an index is assumed to be of index, 1. Each table cell points to a secondary hash table which is used to store the <field> references (and in turn the respective values) for that record. <Field> entries shall be deemed *field cells* and are depicted as diamonds in the diagram. Each field cell subsequently points to a *value cell* (each depicted as an ellipse in the diagram) which holds the data-value currently associated with that <table>.<field> designation. One should note that their exists two pointer references within each field cell. The unlabeled pointer is used to identify the *current* value of that <table>.<field> reference. The pointer reference labeled "Δ" is used to identify the last value of that <table>.<field> reference before modification (note that this example does not require knowledge of past field values and therefore the Δ-pointers contain null values.

[1] Note that the use of hash tables is certainly *not* a requirement for representations of this nature. The working memory model for semantic context could have as easily been described by way of linked lists or sorted arrays. The hash table scheme was simply selected for its undying efficiency and elegance in storing large amounts of referenced data.

Each table cell within semantic context is tagged with either an 'E' or an 'I.' Table cells exhibiting an 'E' tag, indicate *explicit* references--cells which must remain non-volatile until such time as they are explicitly dumped by the system. Those cells which exhibit an 'I' flag indicate *implicit* references--cells which become volatile as soon as the current rule has been fully considered and fired (if applicable). Upon completion of a rule's consideration and firing (should the antecedent prove true), all 'I'-designated table cells, including all cascading field/value cells which are linked to them, are purged from semantic context. This purging accomplishes two tasks: first, working memory is free of "garbage" at all times; second, the SE cannot confuse like-references in differing rules.

Explicit ('E'-designated) table cells and their cascading field/value cells may remain within semantic context indefinitely. There are only three ways explicit references may be purged from semantic context: (1) if the record referenced by the table cell is deleted, (2) if the SDBMS NewRecord command is called on the same reference, or (3) if a different record of the same type is searched-out in the database. One should note that the last two procedures do not actually purge the reference per say, rather they change the field/value references associated with that record (table cell). Figure 5 gives a visual account of these three ways of purging semantic context. Portion A of figure 5 shows the deletion of a "plant1" record. Portion B shows what happens when there exists a "plant1[1]" record in semantic context, but the SDBMS command "NewRecord(plant1[1])" is called. Note that no field/value cells exist after the NewRecord command-call. If a particular field cell does not exist in semantic context, the SE deduces that value to be *null* (i.e., not yet acquired). Part C of figure 5 demonstrates what occurs when an existing reference is re-used (i.e., the old reference is purged in favor of the new reference) via searching/queuing an existing record of the same type. Note that when an existing record is queued via searching,

all field values which are not equal to *null* are loaded into semantic context (hence, the use of the ellipsis in the field cell hash table).

Why keep records in semantic context after inserts or updates? As *new* data cells are acquired for the buffered record (via the SDBMS SetField command), acquisition rules are considered and potentially fired, changing the state of the record buffered in semantic context. Upon execution of the SDBMS Insert or Update command, all committal rules pertaining to that record are considered and fired (if applicable)--again changing the state of the buffered record. Finally, when all rules have been fully considered and potentially fired, the record is physically inserted into its respective database via the database engine which governs it. The record continues to remain in semantic context for two important reasons: first, should the insert/update operation succeed, the user may wish to access certain field values which may have been modified by the semantic constraints represented within the committal rules (e.g., inferable field-values)--or the user/program may simply wish to re-access the data within that record at a later time; second, should the insert/update operation be aborted for some reason, it is imperative that the record *remain* buffered within semantic context to allow the user/program, which communicated the new record via the SI, to salvage data values which were accepted by the committal rules and perhaps attempt to re-insert/re-update the record. This is important since semantic rules are capable of aborting insertion/modification of a record if, for example, a certain field value breaches semantic integrity. Should the system purge the buffered data references at the point of abortion, the user/program would lose all modifications made to that record even if insertion failed because of a single bad value. Thus, failure of a routine of this nature should retain buffered references and allow the user/program to correct the problem at which point an attempt may be made to re-insert the record (if desired).

4.3 Semantic Engine *AGENDA*

Given these potentially enormous rule-bases which manage the semantic integrity of particular databases, in what order (if any) should such a system consider each rule? Obviously, overall consideration of a particular rule is directly associated with the rule category to which it belongs (i.e., a removal rule for a certain type of record would not be considered during an insert action for that type of record). The acquisition rule category constrains consideration of rules even further to only those rules whose first component of the antecedent matches the field being acquired. However, once a set of rules is identified for consideration by the semantic engine, is the order of consideration relevant? To answer this question consider the following generalized committal rules:

1. $\{C\}\ \alpha_1.\beta_1(X) \wedge \alpha_2.\beta_1(X) \wedge \alpha_2.\beta_2(\gamma_2) \Rightarrow \alpha_2.\beta_3(\gamma_3) \wedge \text{Update}(\alpha_2)$

2. $\{C\}\ \alpha_1.\beta_4(\gamma_4) \Rightarrow \text{Abort}(\alpha_1)$

Let us assume in this case that an α_1 record is about to be inserted. Both committal rules in the example apply to an α_1-type record (i.e., records belonging to the table α_1). Let us assume that the semantic engine considers each of these rules in the order in which they are listed above. Further, let us assume that upon consideration of each rule the antecedent of that rule is found to be true and its consequent is duly carried out. Immediately one can see a serious flaw as rule 1 implies the setting of an inferred field-value of another record (α_2) and its subsequent update. When rule 2 is fired, the insert operation of the α_1 record is *aborted* indicating breach of semantic integrity and refusal to commit the record. However, we

have already updated another record in rule 1 with respect to the α_1 record, and now we find that the α_1 record is invalid. This flaw would result in inconsistent data in a very short period of time. Thus, in this case it is important for the semantic engine to consider rule 2 *before* rule 1.

To further illustrate rule consideration anomalies let us examine an additional committal rule:

3. {C} $\alpha_1.\beta_5(\gamma_5) \wedge \alpha_1.\beta_6(\gamma_6) \Rightarrow \alpha_1.\beta_4(\gamma_4)$

Once again let us say that an α_1 record is awaiting insertion and rules 2 and 3 (ignoring rule 1 for the moment) are considered respectively. Further, let us say that upon consideration of rule 2, $\neg\alpha_1.\beta_4(\gamma_4)$ is true given the state of the α_1 record in question. Thus, rule 2 fails to fire. Rule 3 is then considered and fires setting the value of $\alpha_1.\beta_4$ to γ_4 (i.e., $\alpha_1.\beta_4(\gamma_4)$). At this time rule 2 would imply abortion of committal of this record, but consideration of rule 2 has come and gone. Again we see the potential for inconsistent data due to erroneous ordering of consideration.

To solve this problem the semantic engine incorporates an *agenda scheme*. The agenda scheme may be thought of as follows:

Semantic Engine *Primary* Rule Consideration *Agenda Scheme*:

Given a set of rules which are relevant to the current context:
(1) consider all *non-committal-/non-abort*-consequent rules,
(2) consider all *abort*-consequent rules, and finally,
(3) consider all *committal*-consequent rules.

A committal-consequent rule is one which contains a committal action (e.g., Insert, Update, Delete, etc.) within its consequent. An abort-consequent rule is one which contains an Abort action within its consequent. Hence, non-committal-/non-

abort-consequent rules identify all those rules which remain (i.e., those rules which neither contain a committal action nor an abort action within their consequents). With (1) the semantic engine ensures that any inference to extend the state of a record is carried out immediately so that (2) and (3) may act upon the *maximally extended* state of the record. By maximally extended we mean that there exists no rule which may modify or add to the information state of a record. The ordering of (1) and (2) ensure that anomalies such as the one presented in the above rule consideration example of rules 2 and 3 cannot occur. The ordering of (2) and (3) is necessary to abolish the problem caused by the rule consideration example of rules 1 and 2 above.

Within each step in the primary agenda, a secondary agenda must be maintained which manages the forward chaining process itself. For instance, consider the following two non-committal/non-abortion rules:

4. $\{C\}\ \alpha_1.\beta_1(\gamma_1) \wedge \alpha_1.\beta_2(\gamma_2) \Rightarrow \alpha_1.\beta_3(\gamma_3)$
5. $\{C\}\ \alpha_1.\beta_1(\gamma_1) \wedge \alpha_1.\beta_4(\gamma_4) \Rightarrow \alpha_1.\beta_2(\gamma_2)$

At time of consideration within step (1) of the primary agenda (for committal of an α_1 record) the secondary agenda would contain rules 4 and 5. Let us assume that in a particular context $\neg\alpha_1.\beta_2(\gamma_2)$ holds, causing rule 4 not to fire. However, if in the same context $\alpha_1.\beta_4(\gamma_4)$ holds true, rule 5 would fire (assuming, of course $\alpha_1.\beta_1(\gamma_1)$ held true as well), bringing $\alpha_1.\beta_2(\gamma_2)$ into context. Thus, we must have a way of bringing rule 4 back into the secondary agenda so it may be reconsidered. AI literature often refers to forward-chaining rules as *if-added* rules, indicating that actions described in those rules should be taken *when* a value found within the antecedent of that rule becomes available. Within the semantic context of the SE the only way a value can "become available" is by the execution of the SDBMS SetField

command. Thus, the secondary agenda is *driven* by calls to the SetField command. Each time the SetField command is called within a given primary agenda step the available rules within that primary agenda are tested for any occurrences of that particular field within the antecedent of the rule. Should a component within the antecedent of one of these rules correspond to the field being set, that rule is placed on the secondary agenda. When all rules within the secondary agenda have been considered, inference continues with the next step on the primary agenda.

4.4 SDBMS Symbol Dictionary

Given that interpretation of the semantic rule base is based heavily on the knowledge about the structures of the tables referenced therein and the ability to identify which database engines govern which tables, the SE must be provided with a *symbol dictionary* (as depicted in figure 2). This dictionary links the various symbols used in the semantic rule-base which reference specific databases with information about the database engine, structure of the table (e.g., key fields, non-key fields, field data types, etc.), and (if applicable) its logical location on some (potentially networked) storage device. This symbol dictionary would be referred to continually by the SE when converting an SDBMS semantic rule to its corresponding SEARCH-TEST-ACT chain, where information about the database's structure and governing database engine are paramount issues (as detailed previously). Beyond the *conversion* of a given semantic rule to its SEARCH-TEST-ACT chain, identification of a particular table's database engine is obviously important when *executing* the given components of SEARCH-TEST-ACT chains, as translation must occur from any of

the SDBMS manipulative commands (e.g., Search, Insert, Delete, Update, etc.) to there implementation platform equivalent.

Many database engines provide the capability of querying the structure of their databases. In this environment when the SE is presented with a semantic rule which references a *new* table (i.e., one which has not already been incorporated into the symbol dictionary), it need only ask the user for its governing database engine (i.e., the *type* of database) and logical location (if applicable). The SDBMS would then be able to fill in the required information itself by querying that table's structure--given the database engines structural-query commands. Some database management systems, however, do not boast such powerful structural-query capabilities. In such an environment when presented with a new table--one unknown to the SDBMS--the SDBMS would require the user to enter information about the database's structure and engine directly into the symbol dictionary. As an alternative to this cumbersome need to enter structural information twice (i.e., once when creating the database and once when linking it to the SDBMS), one might build additional SDBMS commands designed to control database creation. For example, if a particular database engine was capable of allowing creation/structural-modification of databases, but did not allow querying about existing databases' structures, the SDBMS might first acquire the information required for building the database via the SI, incorporate the necessary information about the new database's structure into the symbol dictionary, and use the respective database engine commands to physically create the database. In this manner structural description of new databases is only required once. One must take care, however, that any creations of new databases or structural modifications to existing databases which are to be referenced by the SDBMS *must* be done so through the SI to avoid describing the structural information twice.

With knowledge about the databases--their structures and the database engines which regulate them--the SDBMS may oversee manipulative aspects in a seamless manner, accepting manipulative database commands from the user/program via the SI and interpreting any semantically defined repercussions of those actions through the SE. Having discussed implementation aspects of the SDBMS which are global to all database engines, let us investigate specific implementation constraints posed by different types of database engines.

4.5 Semantic Engine Interaction with Single-record Manipulative Database Engines

SE control of primitive, single-record manipulative database engines follows quite naturally from the way in which the SDBMS has been defined. These types of database engines only allow users/programs to reference data one record at a time. Often what must occur is first queuing a particular record and then acting upon it. Multiple-record manipulations rely upon the user/program. To accomplish a multiple-record manipulation the user/program must initiate a loop, external of the database engine, where each single record is queued and manipulated.

4.5.1 SDBMS *Inserts* with Single-record Manipulative Database Engines

The insertion of records into databases which support only single-record manipulations abides by the following template (again, we shall assume that SI communication takes the form of a simple procedure-like command language):

NewRecord(T_1);
SetField(T_1, F_i, V_i);
...
SetField(T_1, F_k, V_k);
Insert(T_1);

In this format, T_1 represents some table name which is under SDBMS control. F_i and F_k represent valid field names for table T_1. The reference V_i represents the value to be set for field F_i whereas V_k represents the same for field F_k. The SDBMS NewRecord command sets up semantic context for the declaration of a new T_1 record. The reference for this record within semantic context is defined as $T_1[1]$ (as was described in the preceding sections). If there was a previous $T_1[1]$ record queued in semantic context its field and value cells are purged in favor of the new record which is assumed to initially have all *null*-value fields. The SDBMS command SetField is then used in a sequential manner to set the desired values of particular fields. Upon execution of a SetField(T_i, F_j, V_j) command any acquisition rules of the form "{A} $T_i.F_j(?)$ ..." are considered and, should their antecedents prove true, their consequents are executed. Should any of these consequents contain the SDBMS command RejectValue(T_i) the value V_j for the field F_j of table T_i is not incorporated into semantic context and the user/program which initiated the SetField command is made aware of the rejection. A simple message may be compiled by interrogating the rule which caused the value rejection. For example, if we had a referential integrity acquisition rule of the form:

$$\{A\}\ T_1.F_1(X) \wedge \neg(T_2.F_2(X)) \Rightarrow \text{RejectValue}(T_1.F_1)$$

A rejection message could be compiled as follows: "Value rejected because 'X' does not exist in table 'T_2'." ('X', of course, would be displayed as the value bound to the variable X). Use of the SetField changes the field-value state of the new record

both by the field which is being set directly, and by any potential consequents inferred by the acquisition rules spawned from setting that particular field. The new record continues to be defined in this manner until the Insert command is issued.

Upon issue of an Insert command all committal rules are interrogated, once again changing the state of the buffered record. Once consideration of all committal rules is concluded and no rules brought about the SDBMS Abort command, the record is inserted into its respective database using the technique appropriate to the database engine which governs it. At this time field cells which do not appear within semantic context for the record to be inserted are assumed to be of value *null*.

4.5.2 SDBMS *Queuing* of Records with Single-record Manipulative Database Engines

In order to accomplish one of the other two SDBMS manipulative commands (i.e., Delete and/or Update), a record must first be queued. Queuing usually would take the form of an SDBMS Search command. However, other SDBMS queuing commands could be available to the user (assuming the database engine in question can handle such queuing commands): FirstRecord(<table>), NextRecord(<table>), PreviousRecord(<table>), and LastRecord(<table>). These additional commands are relatively straight forward. The SDBMS Search command, however, does require some discussion as it must be universally capable of covering a wide variety of search techniques (since many database engines have their own, usually different, ways of representing search criteria).

The SDBMS Search command is of the following format:

Search(T_1, $F_i = V_i \wedge F_j = V_j \wedge \ldots F_k = V_k$, SCOPE);

In this format T_i identifies some table *i* under SDBMS control which is to be searched. F_x is some field *x* belonging to table *i*. V_x is some valid value *v* for field *x* of table *i*. The conjunctive symbol '$\wedge$' is used to specify additional search constraints. The reference SCOPE is used to indicate the scope of the search to be performed. The SCOPE parameter may not be applicable to some primitive forms of database engines. However, Borland International's PARADOX™ ENGINE, for example, allows the scope of a search to be constrained in three different ways: (1) search begins at the first record in the database, (2) search begins at the *currently queued* record, or (3) search begins at the first record in the database and continues to the record which matches the criteria the closest. As will be seen in the succeeding section SQL-compatible systems *always* search from the first record in the database and therefore the SCOPE parameter is irrelevant.

The second parameter of the Search command identifies what criteria the search should constrain itself to. This second parameter may have several different interpretations depending upon the complexity of the engine. Once again let us use Borland International's PARADOX™ ENGINE to illustrate differing interpretations. The PARADOX™ ENGINE allows two techniques for searching. The first technique--PXSrchKey--allows searching on one or more *consecutive* fields which belong to the primary key, starting with the first field of the primary key. For example, consider the table A which has fields a_1, a_2, a_3, a_4, a_5, a_6, and a_7. Further, let us say that fields a_1 through a_3 compose the primary key of table A. We may then use the PXSrchKey command to search on *just* the field a_1 or we may use the same command to search on fields a_1 *and* a_2 or on all fields of the primary key, a_1, a_2, and a_3. A second technique which the PARADOX™ ENGINE supports is PXSrchFld, which allows a search on a *single* field value. Thus, we could use the PXSrchFld command to search on field a_4 or a_5 or a_6 or a_7, but no combination there of. It is the SE's

obligation to have knowledge of the various search techniques supported by the various database engines under its control and to know how and when to use them.

Some simple translations from the SDBMS Search command to pseudo code for the PARADOX™ ENGINE might look as follows:

SDBMS: Search(A, $a_1 = v_1$, FIRST);
PARADOX: PXSrchKey on table A with field $a_1 = v_1$, SEARCHFIRST.

SDBMS: Search(A, $a_1 = v_1 \wedge a_2 = v_2$, NEXT);
PARADOX: PXSrchKey on table A with fields $a_1 = v_1$, $a_2 = v_2$, SEARCHNEXT.

SDBMS: Search(A, $a_6 = v_6$, FIRST);
PARADOX: PXSrchFld on table A with field $a_6 = v_6$, SEARCHFIRST.

The SE is capable of determining which type of PARADOX™ command to use based on its knowledge of the structure of the database in question. In the second example the SE can infer the use of the PXSrchKey command since fields a_1 and a_2 are consecutive fields in the primary key and field a_1 is the first field in the primary key. Thus, the constraints for using the PXSrchKey command have been fulfilled and therefore may be used. In the third example the SE can infer the use of the PXSrchFld command since field a_6 is a non-key field of the table A.

4.5.2.1 SDBMS *Enhanced Queuing* of Records with Single-record Manipulative Database Engines

Indeed, the SE may use its knowledge of the workings of a particular database engine to provide enhanced querying capabilities, as the following PARADOX™ ENGINE interpretation demonstrates:

SDBMS: Search(A, $a_1 = v_1 \wedge a_2 = v_2 \wedge a_6 = v_6$, FIRST);
PARADOX: if (PXSrchKey on table A with fields $a_1 = v_1$, $a_2 = v_2$,
SEARCHFIRST) is successful,
AND if ((field $a_6 = v_6$) *OR*
[if (PXSrchFld on table A with field $a_6 = v_6$,
SEARCHNEXT) is successful,
AND if the current rec's field value for a_1 is v_1,
AND if the current rec's field value for a_2 is v_2],
THEN the search was successful;
ELSE the search was *not* successful.

In this example we see that since fields a_1 and a_2 are consecutive fields belonging to the primary key of table A and since field a_1 is the first field of the primary key, the SE can make use of the PXSrchKey command to accomplish the initial portion of the search. Further, since field a_6 is a simple field of table A the SE can use the PXSrchFld command to search from the record found with the PXSrchKey command, to the next record where $a_6 = v_6$ (if the record found in the initial search has $a_6 \neq v_6$). Note, however, that should this next record be found, its values for fields a_1 and a_2 must be verified (since the next record where $a_6 = v_6$ may not have $a_1 = v_1$ and $a_2 = v_2$). Note also that this technique assumes that the table's records are sorted in an ascending manner relative to the primary key.

Generalizing this technique one arrives at the following algorithm which the SE can use to enhance the querying capabilities of PARADOX-like database engines:

Algorithm for Enhanced Single-record Querying:

1. Given the command
   Search(A, $a_i = v_i \wedge a_j = v_j \wedge \ldots \wedge a_x = v_x$, SCOPE);
2. Re-organize the criteria parameter to the form $a_i = v_i \wedge a_j = v_j \wedge \ldots \wedge a_x = v_x$, where given the structure of the table A the logical ordering of field a_i precedes field a_j precedes ... precedes field a_x.

3. If a_j is *not* the first field (given the structure) of table A or if table A has no primary key, then set the variable NEXTSCOPE = SCOPE and GOTO step 7.
4. Starting with field a_j, strip off each *consecutive* field which belongs to the primary key of table A--call this the *key criteria*.
5. Use PXSrchKey to search on table A with key criteria acquired in step 4, starting the search from the record identified by SCOPE.
6. If a record was found in step 5, then set the variable NEXTSCOPE = SEARCHNEXT and continue with algorithm,
   If a record was *not* found in step 5, then exit algorithm with *failure* (i.e., no record found).
7. If the current record matches the *remaining* criteria, then exit the algorithm with *success*!
8. Use PXSrchFld to search on table A with single-field criteria equal to the first field remaining in the criteria, starting the search from the record identified by the variable NEXTSCOPE.
9. If a record was found in step 8, then set the variable NEXTSCOPE = SEARCHNEXT.
   If a record was *not* found in step 8, then exit algorithm with *failure*.
10. If *key criteria* exists (i.e., step 4 was executed above) and the current record does *not* match that criteria, then exit algorithm with *failure*.
11. GOTO step 7.

Thus, one can see that the SDBMS Search command may be interpreted in several distinct ways depending upon (1) the abilities of the database engine which governs the table to be searched, and (2) the structure of the table. With this comprehensive queuing strategy the SDBMS enhances existing technology, as was seen in the case of PARADOX™ ENGINE interpretations, while at the same time providing a universal medium for accomplishing the queuing of records over different types of single-record manipulative database engines.

4.5.3 SDBMS *Deletes* with Single-record Manipulative Database Engines

Having discussed the queuing process for single-record manipulative database engines, the idea of record deletions becomes trivial. Once a record belonging to a certain table T is queued (as described in the preceding section) a simple SDBMS command of the form Delete(T) deletes the record from the table. One must keep in mind, however, that before the physical deletion of the record from table T occurs, all SDBMS removal rules of the form "{R} $T.F_i$(?) ..." must be considered and fired should their antecedents prove true. Should an applicable removal rule fire which, within its consequent, contains the SDBMS command Abort(T) the record is *not* deleted from the table and the initiator of the Delete command is made aware of the infraction which caused the abortion by way of the SI.

Upon successful completion of an SDBMS Delete command the record which was just deleted is completely removed from semantic context (i.e., removal spans not only all field and value cells which related to that record, but also the table cell which referenced the record at its highest level). Part A of figure 5 depicts this type of total removal from semantic context.

4.5.4 SDBMS *Updates* with Single-record Manipulative Database Engines

The use of the SDBMS Update command to modify existing records follows the format detailed above for declaration of new records prior to insertion, except that the desired record would be queued instead of cleared via the NewRecord command. The SDBMS SetField command would then be used to modify field values of the queued record. As the SetField command is invoked any SDBMS acquisition rules

which relate to that field are considered and fired (if applicable) as outlined above. Note, however, that once an existing record has been queued, the SDBMS SetField command's function is slightly different in that for each field *a* belonging to the queued record *A[1]* a Δ index reference of the form *A[1].a[Δ]* is incorporated into semantic context. The value cell associated with this Δ index field cell contains the value of *A[1].a* just prior to modification by the SetField command. As was explained above Δ index references may be used in rules to semantically distinguish between inserts and updates.

Once the desired field values have been modified for the queued record via the SetField command(s), the SDBMS Update command is used to commit the modifications to the database. As with inserts all committal rules applicable to the type of record being updated are considered and potentially fired. Should an Abort command be encountered during this interrogation, the update is aborted and the initiator of the Update command is informed. When all committal rules have been examined, and no Abort command was encountered, the modifications for the record are committed to the database.

4.6 Semantic Engine Interaction with Multiple-record Manipulative Database Engines

Multiple-record manipulative database engines (e.g., SQL-compatible systems) do not allow single-record accesses per say and hence the SE must take a slightly different route to interact with such systems. Whereas a command issued to a single-record manipulative engine effects a specific, unique record, a single multiple-record engine command may result in the manipulation of several records. However, any multiple-record manipulative system must ultimately access one record at a time.

Given this important factor SE interaction with such database engines follows naturally from the SE implementation as described in the previous section with few additions to the grand scheme.

SE interaction with multiple-record database engines involves (1) interrogation of the command, (2) determination of which record(s) will be affected by the command, (3) determination of which SDBMS rules associate with the command, (4) application of those rules on each record affected by the command, and finally (5) execution of the original command itself on each record which did not result in a semantic infraction. SQL will be used in the remainder of this section to outline SE interaction with multiple-record manipulative systems. One should note, however, that the strategy described herein would apply to virtually any multiple-record manipulative environment.

4.6.1 Internal and External Multiples

Given a multiple-record command the SE must break the command into sequential record accesses so it can be sure each record is acted upon appropriately by the semantic rule-base. This is accomplished by way of record *cursors*--common to embedded SQL. Two types of cursors which reference multiple-records (henceforth referred to simply as "multiples") are utilized within the SE: *external multiples* and *internal multiples*. To understand the nuances of the first type of multiple let us consider the following generalized format of a multiple-record SQL command:

[action...] WHERE [criteria...]

Here, [action...] denotes some multiple-record action such as update, insert, delete, etc. The parameter [criteria...] references the constraints which records must meet in order to be included in the multiple-record operation. If we take the [action...] portion of this command and replace it with a SELECT ... FROM operation we can precisely identify those records which will be affected by the [action...] operation (as dictated by the [criteria...] parameter). To clarify let us consider the following SQL UPDATE command:

```
UPDATE production1
SET production1.serial_required = 'YES'
WHERE production1.type = 'TOXIC';
```

Taking the WHERE clause into account we may produce the following SELECT statement which will identify all those records which would be effected by the UPDATE command:

```
SELECT *
FROM production1
WHERE production1.type = 'TOXIC';
```

Using embedded SQL (our link from the SDBMS to the RDBMS) we may then acquire a cursor on this statement as follows:

```
EXEC SQL DECLARE X1 CURSOR FOR
        SELECT *
        FROM production1
        WHERE production1.type = 'TOXIC';
```

The cursor X_1 identifies the multiple-records external to the semantic rule-base (i.e., external multiples). Using the embedded SQL command FETCH the SE may sequentially isolate each record and independently act upon that record (in this

example considering all committal rules relevant to a production1 record), processing any semantic repercussions as represented by the rule-base. As each record is fetched from the cursor its field values are loaded into semantic context (as in the single-record queuing strategy defined above). New field values are then set for that record using the SDBMS SetField command (if required as dictated by the [action...] portion of the original command). The SetField command spawns interrogation of relevant acquisition rules for that type of record. Should all new values pass interrogation of the acquisition rule-base (i.e., no RejectValue-consequents arise), the rule-base relevant to the [action] is then considered (i.e., committal rules or removal rules) and should no Abort-consequents arise, the [action] is finally performed *on that unique record*. Exactly how each [action] is performed by the SDBMS is detailed in subsequent sections of this chapter. Should a semantic infraction occur, a log is created outlining the infraction (this may be in the form of an immediate interface with the user, allowing him/her to correct the infraction in the middle of processing the original command, or it may be written out to a log file where the user/program could return to after all valid records have been acted upon). After each record is acted upon or rejected the SE then fetches the next external multiple and begins the process again. Processing continues in this fashion until all records identified by the cursor have been acted upon.

Internal Multiples are multiple records which arise from consideration of a single rule within the context of a single subject record. To illustrate this type of multiple consider the following semantic removal rule:

```
{R}  supplier.s#( X ) ∧ shipments.s#( X )
        ∧ AcquireExistingValue( "Enter the supplier # for the
              supplier who will be taking over supplier |X|'s
              shipments:", supplier.s#, Y )
        ⇒ shipments.s#( Y ) ∧ Update( shipments )
```

This rule states that should a supplier be removed from the database any shipments which were assigned to him/her should now be taken over by another (remaining) supplier. In this case the record "supplier" would be bound to a single record in semantic context which is awaiting removal from the database. The reference "shipments" would then bind to several records (i.e., a single supplier may have many shipments assigned to him/her). Thus, "shipments" identifies multiple records internal to the consideration of this particular removal rule (i.e., internal multiples)--specifically, all shipment records assigned to supplier number, X. The special SDBMS function AquireExistingValue is used to obtain a valid supplier number from the supplier database, which, in this example, identifies the supplier who will take over supplier number X's shipments.

Hence, for each external multiple this particular rule may have to be applied to many internal multiples. Note that internal multiples not only apply to multiple-record manipulative database systems but also may apply to single-record manipulative systems as well. For example, the above rule would be just as valid if the databases "supplier" and "shipments" were governed by a single-record manipulative database engine. Because of this important point the SE must handle internal multiples in such a way as to provide generality among the different types of database engines which are overseen by the SDBMS.

4.6.1.1 SDBMS Implementation Scheme for Internal Multiples

Recalling that each semantic rule which governs the manipulation of a given database may be reduced to a SEARCH-TEST-ACT chain, it is easy to see how such a chain may be invoked to manage internal multiples. Take for example the following

rule (as detailed in the previous section) and its SEARCH-TEST-ACT chain reduction:

```
{R} supplier.s#( X ) ∧ shipments.s#( X )
      ∧ AcquireExistingValue( "Enter the supplier # for the
            supplier who will be taking over supplier |X|'s
            shipments:", supplier.s#, Y )
      ⇒ shipments.s#( Y ) ∧ Update( shipments )

RULE:       R/supplier
SEARCH1:    shipments.s# = supplier.s#
TEST1:      ε
ACT1:       AcquireExistingValue( "Enter the supplier # for the
                  supplier who will be taking over supplier
                  |supplier.s#|'s shipments:", supplier[2].s# )
ACT2:       Set( shipments.s# = supplier[2].s# )
ACT3:       Update( shipments )
```

In this particular example the SEARCH portion of the chain may potentially bind several different "shipments" records--each of which must be independently considered by the rule. Thus, by thinking of the SEARCH portion as an iteration over one or more records found by the search we arrive the following flow:

```
Loop while SEARCH1
      Begin loop
            TEST1
            ACT1
            ACT2
            ACT3
      End loop
```

For single-record manipulative systems the loop would be carried out by searching out the first record identified by $SEARCH_1$, iterating the loop once, searching for the next record under $SEARCH_1$, iterating once again, etc. Multiple-record manipulative systems (e.g., SQL) would require obtaining a cursor on

$SEARCH_1$ and fetching the next record identified by the cursor for each iteration. In this manner universality is achieved by simply providing an SDBMS Search procedure for each database engine which is controlled by the SDBMS. The syntax for the Search procedure call would remain the same (as detailed in previous sections) while the functionality of the procedure would be dependent upon the database engine which governs the table in question.

4.6.1.2 SDBMS Implementation Scheme for External Multiples

Having implemented internal multiples by way of the SDBMS Search command, external multiples are handled quite easily in the same fashion. Indeed, since external multiples are acquired (as in SQL) by replacing the [action...] portion of an "[action...] WHERE [criteria...]" command with a "SELECT ... FROM" query, the SE, in effect, generates an SDBMS Search query which will identify those records which meet the [criteria...]. This "external" Search is handled in precisely the same manner as Searches within a given rule, except that during iteration the entire (relevant) rule-base is considered for each of those records queued by the Search and the [action] is independently performed for each of those records as well (assuming no semantic infractions occur, of course). To clarify this proposed flow let us consider the following multiple-record delete command and how the SE would go about implementing such a command:

Initial Multiple-record Command:

```
DELETE
FROM supplier
WHERE supplier.orders < 100;
```

External Multiples Identified By Command:

EXEC SQL DECLARE X_0 CURSOR FOR
SELECT *
FROM supplier
WHERE supplier.orders < 100

(which implies the following SDBMS Search command:)

$Search_0$(supplier, orders < 100, N/A)

(Note: the "N/A" indicates scope is irrelevant to this database engine.)

Recalling the sample removal rule in the previous section, relevant to a "supplier" record, we would have the following SE flow:

Loop while **SEARCH$_0$** *(external multiples)*
Begin loop
... <consideration of initial removal rules> ...
Loop while **SEARCH$_1$** *(internal multiples)*
Begin loop
TEST$_1$
ACT$_1$
ACT$_2$
ACT$_3$
End loop
... <consideration of remaining removal rules> ...
IF <NO SEMANTIC INFRACTIONS>,
THEN Delete(supplier)
End loop

Let us say further that in the above flow $SEARCH_0$ is governed by cursor X_0 and that $SEARCH_1$ is governed by cursor X_1. Given these stipulations, ACT_3 (i.e., Update(shipments)) may be carried out by the following SQL translation (assuming that supplier[2].s# was acquired in ACT_1 as "Y22") :

```
EXEC SQL UPDATE shipments
        SET shipments.s# = 'Y22'
        WHERE CURRENT OF X1;
```

Finally, should no semantic infractions occur in consideration of the removal rules for a "supplier" record, the Delete(supplier) command is carried-out as follows:

```
EXEC SQL DELETE
        FROM supplier
        WHERE CURRENT OF X0;
```

Thus, by way of cursors we are able to control the iterative processes, both external and internal, of rule-base consideration. Once again cursors would not be required in single-record manipulative systems as actions are performed on the *currently queued* record (i.e., that record which was iteratively queued by the most recent SDBMS Search command for that record type).

4.6.2 SDBMS *Inserts* with Multiple-record Manipulative Database Engines

SQL allows two modes of insertion: single-record inserts and multiple-record inserts. Should the user/program wish to execute a single-record insert, two avenues are available. The first route would allow the user/program to initiate the insert as was described for single-record manipulative systems (section 4.3.1) via dialogue through the SI--ideal for systems where user-interface is of primary importance. This scheme provides two important advantages: first, the application controlling the interface need not concern itself with providing a buffering structure for the information as this is accomplished by the SDBMS semantic context. Second, as each value is set (via the SDBMS SetField command) acquisition rules can be applied immediately causing

feedback to the user of any inferred field values or value rejections resulting from those rules. When the SDBMS Insert function is finally initiated, the committal rule-base would be consulted and, should no abort consequent present itself, the SE would compile the necessary SQL command to insert the given record into its respective database. This insert command can be easily generated by taking into account all non-null field-values within semantic context for the record being inserted.

The second route of single-record insertion would be to send the (SQL) command directly to the SE. This command would be of the following generalized format:

INSERT
INTO α (α_1, α_2, ..., α_i)
VALUES (v_1, v_2, ..., v_i) ;

Given this command the SE would simply generate the following SDBMS function calls which would ensure semantic integrity and (assuming no breach of integrity) insert the record into its respective database (as described for single-record manipulative database engines in 4.3.1):

NewRecord(α);
SetField(α, α_1, v_1);
SetField(α, α_2, v_2);
...
SetField(α, α_i, v_i);
Insert(α);

Should, however, a multiple-record insertion be required, the corresponding SQL command would have to be sent to the SE, where it would be interrogated, identifying each record within the command. Each record is processed as an external multiple (see 4.4.1), involving (1) consultation of the acquisition rule-base for each

new field-value specified, (2) consultation of the committal rule-base (prior to insertion), and (3), should no semantic infractions occur, insertion of the single record into its respective database.

4.6.3 SDBMS *Deletes* with Multiple-record Manipulative Database Engines

Given the description of the SDBMS handling of external multiples, multiple-record deletions become quite trivial. To summarize, the DELETE operation is transformed into a SELECT...FROM operation, identifying those external multiples which will be deleted. Iteration continues for each external multiple, consulting the removal rule-base respective to the database from which the record is to be deleted, and should no semantic infractions occur the record is then physically removed from the database.

4.6.4 SDBMS *Updates* with Multiple-record Manipulative Database Engines

Multiple-record updates are performed much in the same way as multiple-record inserts. Take the following generalized update command format:

UPDATE α
SET $\alpha_i = v_i, \alpha_j = v_j, \ldots$
WHERE $\alpha_k = v_k, \ldots$

The SE would then obtain a cursor on the following query:

```
EXEC SQL DECLARE X0 CURSOR FOR
        SELECT *
        FROM α
        WHERE αk = vk, ...
```

Iteration would then take place over the records identified by X_0 (as described for external multiples in 4.4.1) and for each record the following SDBMS function calls would be invoked:

```
SetField( α, αi, vi );
SetField( α, αj, vj );
...
Update( α );
```

Each call to SetField moves the old field-value into the field's Δ slot, places the new field-value into the field's current field-value slot, and considers any relevant acquisition rules (as described in 4.3.4). Upon execution of the SDBMS Update command the SE generates the following SQL command:

```
EXEC SQL UPDATE α
        SET αx = vx, αy = vy, αz = vz, ...
        WHERE CURRENT OF X0;
```

In this command, α_x, α_y, α_z, ..., refer to all field cells which contain values both in their current field-value slots *and* in their Δ slots; whereas v_x, v_y, v_z, ..., refer to the current field-values of field cells α_x, α_y, α_z, ..., respectively.

5.0 SDBMS REPRESENTATION OF SEMANTIC INFORMATION

Given a detailed account of how the SDBMS functions the following question presents itself: "What *kinds* of semantic information can be represented in such a

system?" To demonstrate the types of representations possible with the SDBMS let us examine figure 6. This figure depicts the logical design and subsequent E/R model of a database system which is to keep track of the daily activities of a generic production plant. Raw materials at the plant site consist of various chemicals which are produced by manufacturers and stored in a holding area prior to their consumption. Products are then produced by merging the chemicals in a reactor. Products are then stored in various types of storage containers awaiting transportation to customers. The logical design in figure 6 is based on this description, providing a general overview of the plant's daily throughput. Below the logical design is found a simple Entity/Relationship (E/R) model. The E/R model expands upon the logical design, displaying all relations required to implement the logical design. Bold lines represent a "many" relationship, whereas non-bold lines represent a "one" relationship. For example, many chemicals may be found in the holding area at any given time, many products may be produced in a single reactor, etc. While this type of model details how the information will be stored, it does not go very far in promoting a semantic awareness of the information itself. By this it is meant that although the E/R model yields a good definition of the *structure* of the tables required to represent the plant's throughput, it does not provide any means for injecting complex semantic constraints/stipulations about *what* may be validly stored within those tables.

The succeeding chapters discuss various types of "semantic" information which pertain to the production plant system of figure 6, and how that information may be represented within the SDBMS.

5.1 Semantic Restriction of Context-sensitive Values

Once a product has been processed in one of the plant's reactors, it must be stored in one or more containers while awaiting shipment to the customers. It would be ideal, for example, if the system "knew" that liquid products should only be stored in liquid-holding containers (e.g., one would not store a liquid product in a cardboard box). The following SDBMS committal rule takes care of this:

```
{C} storage.product_id( X ) ∧ product.id( X ) ∧ product.type( 'liquid' )
        ∧ ¬( storage.container( 'drum' ))
        ∧ ¬( storage.container( 'tank' ))
        ∧ ¬( storage.container( 'tank truck' ) )
              ⇒ Abort( storage )
```

This rule ensures that all liquid products must be stored within liquid-compatible containers such as a drum, tank, or tank truck. This may seem to be a trivial type of semantic constraint and some database management systems do allow field value-constraining. However, examining this rule more closely one finds that the value-constraint posed herein is *context-sensitive* (i.e., the "container" field value is only constrained to "drum," "tank," or "tank truck," when the value "product.type" is equal to "liquid"). Some database management systems allow one to constrain field-values, but these constraints are *always* applied (i.e., global to all records within that table). As has been shown here the SDBMS promotes a semantic approach to value-constraining, while at the same time centralizing the semantic information in the rule-base. Certain values may be semantically constrained based upon the *context* of other field values within that record. This notion of context need not be local to a single record of a single table. For instance, one could semantically define a context based on field value(s) of one or more records within a single table; one or more records

within a different table; one or more records within multiple tables; or even one or more records within multiple tables governed by different relational database engines.

5.2 Cross-table Indexes (Cross-reference Tables)

When building reports it is often necessary to employ indexes to promote an efficient means of sortation. Further, should the desired sortation scheme bridge two or more tables, a standard database index is not possible (as indexes pertain to one or more fields within a *single* table). Using SQL's GROUP BY function, for example, would allow one to arrive at the desired multi-table sortation. However, if access time is paramount this approach may be undesirable as large tables may cause execution of the GROUP BY command to take a considerable amount of time. In this scenario one would ideally like to make use of a multi-table indexing scheme which was maintained in real-time. In this manner an auxiliary table could be implemented, which would contain the multi-table items required by the sortation in its primary key. This cross-reference table would then have to be maintained in real-time in order to be synchronous with the modifications/additions of the multi-table items contained therein.

This scheme is particularly useful when two tables are required to represent a single entity. For example, let us say that for any given product there exists approximately two thousand field-values which are to be associated with that product. Further, let us say that the database system which we have chosen to implement the representation of this product information allows a maximum of one thousand fields per table. Thus, to accomplish this representation scheme at least two tables are required, each of which containing an identical primary key structure (e.g., a single

field "product_id"), and splitting the two thousand or so fields between each of the tables (e.g., table "product1" and table "product2"). Let us then say that a real-time maintainable cross-reference is required between one field of the "product1" table and another field of the "product2" table. The following semantic rules accomplish this:

{C} product1.product_id(X) ∧ product2.product_id(X)
∧ ¬(cross_reference1.product_id(X))
∧ product1.field1(Y) ∧ product2.field2(Z)
⇒ NewRecord(cross_reference1)
∧ cross_reference1.product_id(X)
∧ cross_reference1.field1(Y)
∧ cross_reference1.field2(Z)
∧ Insert(cross_reference1)

{C} product1.field1[Δ](X) ∧ product1.field1(Y)
∧ product1.product_id(Z) ∧ cross_reference1.product_id(Z)
⇒ cross_reference1.field1(Y)
∧ Update(cross_reference1)

{C} product2.field2[Δ](X) ∧ product2.field2(Y)
∧ product2.product_id(Z) ∧ cross_reference1.product_id(Z)
⇒ cross_reference1.field2(Y)
∧ Update(cross_reference1)

{R} production1.product_id(X) ∧ production2.product_id(X)
∧ cross_reference.product_id(X)
⇒ Delete(production2) ∧ Delete(cross_reference)

The first rule listed above handles maintenance of the cross-reference table due to insertion of a new product. The next two rules handle maintenance of the cross-reference table should one of the two field values which comprise the cross-reference be changed. The last rule applies to deletion of a product and the subsequent deletion of the secondary table extension and cross-reference table. The interested reader may note that two additional rules have been left out, yet are nonetheless paramount to the maintenance of the cross-reference table. These two rules would update the cross-

reference table should the "product_id" field of either the "product1" table or the "product2" table would change. The rules themselves are left to the reader.

5.3 Maintained Pool of Logged Values

Certain systems may require a memory feature for values entered by the user (i.e., an encyclopedia of previously used values). This is especially important with today's graphical user-interfaces (GUIs) where programmers wish to maximize the "point-and-click" feature for their applications. A system utilizing this type of environment might display a list of previously entered values for a given field and allow the user to select one of those values (if applicable). It should also allow the user to enter new values on-the-fly if the value required did not occur in the past.

Let us say that in our production plant system it would be useful to have a pool of product types (e.g., a given product type might be "GASOUS10-3," "GASOUS09-1," "LIQUID21-1", etc.). Further, let us say that product types are often reused when new product information is acquired and that the list of product types grows at a slow rate. This would be a prime candidate for a maintained pool of values, allowing the programmer to pull existing values from the pool to form a point-and-click list for the user. The following simple rules maintain this pool quite well:

$$\{C\}\ product.type(X) \wedge \neg(product\_types.type(X)) \Rightarrow NewRecord(product\_types) \wedge product\_types(X) \wedge Insert(product\_types)$$

$$\{R\}\ product.type(X) \wedge \neg(product[2].type(X)) \wedge product\_types.type(X) \Rightarrow Delete(product\_types)$$

{C} product.type[Δ](X) ∧ ¬(product[2].type(X))
∧ product_types.type(X)
⇒ Delete(product_types)

The first rule handles the instance where a new product type is incorporated into the pool. The second two rules remove a product type from the pool when (as in the second rule) a "product" record is removed and there exist no other "product" records which share its "type" field value, and when (as in the last rule) a product's type is changed and there exist no other "product" records which share that products old "type" field value.

5.4 Constrained Field Value Acquisition

Conceivably one might want to constrain acquisition of certain field values upon preceding acquisition of other field values. For instance, given the "schedule" database which is to keep track of when, where, and how certain raw materials are to be processed in what reactors to produce which products, it would not be practical to assign a load time for a batch unless a reactor was first chosen to load the materials into. One might wish to capture this semantic notion in the form of an SDBMS rule and force the system to reject the acquisition of a schedule's "load_time" field value before its "reactor" field value has been assigned. Although this example may seem nonsensical, its analogy can be applied to the extraction of complex information in which the precedent value of X is absolutely required before value Y can be accepted and verified by the system. The following type of semantic rule would suffice:

{A} schedule.load_time(X) ∧ schedule.reactor(*null*)
⇒ RejectValue(schedule.load_time)

5.5 System Maintained *Meta-Tables*

Sometimes it is important to track the day-to-day user actions in regard to possible debugging of applications, report generations, or simply to maintain an audit trail for later reference. In such a scenario one might wish to make use of a table whose records relate which items have been acted upon at which times. It would be nice if one could then explain, semantically, to the system that it is to keep track of those actions and hence maintain the table without having to introduce or modify existing program code to effectuate this task. For example, let us say that for whatever reason it is important for our plant managers to keep track of the number of chemicals deleted from the holding area on a daily basis. Consider the following rules:

```
{R} holding.chemical_id( X )
        ∧ ¬( chemical_removals.chemical_id( X ) )
                ⇒ NewRecord( chemical_removals )
                        ∧ chemical_removals.chemical_id( X )
                        ∧ chemical_removals.instances( 1 )
                        ∧ Insert( chemical_removals )

{R} holding.chemical_id( X ) ∧ chemical_removals.chemical_id( X )
        ∧ chemical_removals.instances( Y )
                ⇒ chemical_removals.instances( Y+1 )
                        ∧ Update( chemical_removals )
```

Both rules are sufficient in maintaining a log of *how many* chemicals were removed from the "holding" database.

5.6 Inferable Field Values

Many systems often require the use of default values or inferred information. Often this semantic knowledge must be hard coded into an application program or

expert system which interfaces with the storage medium (database management system). It would be nice if this semantic knowledge could be directly linked with the information itself without the cumbersome addition of customized interfaces.

Inferable field values can span simple context-dependent value acquisitions to maintenance of redundant data. The incorporation of this semantic knowledge into the SDBMS gives it expert system-like capabilities. It closely couples artificial intelligence techniques with the storage medium, centralizing both knowledge and data alike.

5.6.1 Default Field Values

Often database systems require default information to minimize the time required by data entry and to add to the basic inferable information about a new item. By "default" it is meant that those values are initially inferable, but may be overridden at some point in the future. SDBMS default values are implemented using the format of the following rules:

{C} manufacturer.country(*null*) ⇒ manufacturer.country('USA')

{C} container.type('steel') ∧ container.thickness(>5.0)
∧ container.toxic_compatible(*null*)
⇒ container.toxic_compatible('yes')

The first rule represents a globally defaulted value (i.e., in all contexts of a new "manufacturer" record one can assume the value of field "country" to be "USA"). The second rule makes use of context-dependence, defaulting the field "toxic_compatible" to "yes" only if the container's type is "steel" and its thickness is greater than 5.0. Hence, by utilizing the *null* parameter one may describe default values for any fields under SDBMS control.

5.6.2 Standard Inferable Field Values

Standard inferable field value rules are those which relate the assignment of one field value with the acquisition of another field value. Take, for example, the following rule:

$$\{C\}\ \text{holding.chemical\_id}(X) \wedge \text{chemicals.id}(X) \wedge \text{chemicals.volatile}(\text{'yes'}) \Rightarrow \text{holding.special\_handling}(\text{'yes'})$$

This rule ensures that any volatile chemical which is incorporated into the holding area should be flagged for "special handling." The structure of these rules may become quite in-depth, allowing for a complex array of integrated semantic knowledge. One should note that the above rule will set the "special_handling" field only within a certain context (i.e., when the given chemical is volatile). This does not prohibit the user from directly setting the "special_handling" field to "yes" if the chemical is non-volatile. Should this be the only constraint for special handling of chemicals, one might wish to incorporate the following rule, which would bullet-proof the inference of the "special_handling" field:

$$\{C\}\ \text{holding.chemical\_id}(X) \wedge \text{chemicals.id}(X) \wedge \neg(\text{chemicals.volatile}(\text{'yes'})) \Rightarrow \text{holding.special\_handling}(\text{'no'})$$

5.6.3 Maintenance of Redundant Data Via Inferable Field Values

Many implementations to date which require immensely large databases have abandoned much of the database "normal form" conventions in an attempt to maximize performance. This breach of convention often involves the integration of *redundant data*--long since considered a "dirty word" in database design and development. However, redundant data can be extremely useful albeit cumbersome in large systems which attempt to minimize the total number of access operations. For instance, if our "chemical" database was absolutely immense and our "manufacturers" database was equally large, the designer might opt to eliminate the "chemicals" database which merges the unique keys of both tables to keep track of which manufacturers supply which chemicals. Instead the designer might wish to tack on the manufacturer ID to the "chemical" database which describes each chemical used by the plant. Along these lines let us say that when a chemical is accessed it is important to know AS SOON AS POSSIBLE what the manufacturer's *name* is and what *country* they are located in. Traditionally, the manufacturer's name and country should be held within the "manufacturer" database, and would hence require a secondary access to move from the "chemical" database to the "manufacturer" database. This same technique would actually involve three look-ups if we made use of a "chemicals" database, as we would (1) access the "chemical" database, then (2) access the "chemicals" database to determine the manufacturer ID, and finally (3) access the "manufacturer" database to determine the manufacturer's name and country. By adding the manufacturer's ID, name, and country to the "chemical" database we reduce the number of required accesses from two or three down to a single access. However, by doing so we introduce redundant data into the system. Since our

production plant has plenty of computer information space available, the vote is to increase performance at the expense of space, so our database designers must concern themselves with the maintenance of this redundant data, since failure to maintain redundant information would no doubt result in inconsistent and erroneous data. The following rules allow maintenance to flow simply from the semantic model:

{A} chemical.mfg_id(X) ∧ manufacturer.id(X)
∧ manufacturer.name(Y) ∧ manufacturer.country(Z)
⇒ chemical.mfg_name(Y) ∧ chemical.mfg_country(Z)

{A} chemical.mfg_id(X) ∧ ¬(manufacturer.id(X))
⇒ RejectValue(chemical.mfg_id)

{C} chemical.mfg_id(*null*)
⇒ chemical.mfg_name(*null*) ∧ chemical.mfg_country(*null*)

Thus, we ensure that redundant data is maintained properly. However, examining this problem more closely one will see that the above rules handle the redundancies arising from the "chemical" database to the "manufacturer" database, but not vice versa. Taking note of the following SDBMS rules one will see why the above rules are necessary but not sufficient to handle redundancy maintenance:

{C} manufacturer.id[Δ](X) ⇒ Abort(manufacturer)

{C} manufacturer.name[Δ](X) ∧ manufacturer.name(Y)
∧ manufacturer.id(Z) ∧ chemical.mfg_id(Z)
⇒ chemical.mfg_name(Y) ∧ Update(chemical)

{C} manufacturer.country[Δ](X) ∧ manufacturer.country(Y)
∧ manufacturer.id(Z) ∧ chemical.mfg_id(Z)
⇒ chemical.mfg_country(Y) ∧ Update(chemical)

The first rule ensures that once a manufacturer has been assigned an ID it can never be reassigned. The second rule will modify all relevant "chemical" records in the event that the manufacturer's "name" designation changes. The third rule functions in a similar capacity, maintaining "country" modifications.

With the integration of these simple SDBMS rules the database management system may boast an increase in performance brought forth by data redundancy while at the same time ensuring that no inconsistencies will arise because of such redundancy.

5.6.4 Automatic Manipulative-assignment of Inferred Field Values

Certain applications may require the system to assign identification numbers to new entries, facilitating a unique identification scheme and tracking over the life time of those items. Often for systems which boast a large amount of new item acquisitions and which operate under a multi-user environment, user-assignment of unique identification is not possible. Rather it is left to the system to carry-out some form of automated ID assignment. Given the production plant scenario let us say that new customers are continually being associated with the plant, and that a unique ID is required for each customer to facilitate orders, shipping, billing, etc. Further, since many different users may be entering new customers into the system it is not possible to rely on user-assignment of those unique IDs. The following rules handle automatic assignment of customer IDs as they are acquired:

$$\{C\}\ \text{customer.name}(X) \wedge \text{numbers.table}(\text{'customer'}) \wedge \text{numbers.field}(\text{'id'}) \wedge \text{numbers.number}(Y) \Rightarrow \text{customer.id}(Y+1) \wedge \text{numbers.number}(Y+1) \wedge \text{Update}(\text{numbers})$$

{C} customer.id[Δ](X) ⇒ Abort(customer)

In this example the production plant database management system makes use of a "numbers" database which associates the last used ID number for a given "table"/"field" pair. Thus, as a new customer is entered the "number" field value of the "numbers" record identified by "customer"/"id" will yield the last used ID number. Adding one (1) to this number results (simply) in the next useable unique customer ID. The first rule listed above accomplishes this quite well, assigning the new ID to the new "customer" record and updating the "numbers" database with the newly used value. The second rule ensures that once an "id" has been assigned to a "customer" record it may never be altered--as we have given the system complete control over the numbering scheme.

5.6.5 Indirect Automatic Manipulative-Assignment of Inferred Field Values Through Redundant Data

Given redundant data the system may allow users to define a new record in table A, as a repercussion of inserting a new record into table B. Proceeding along the lines of required redundant data with the chemical/manufacturer relationship consider the following semantic rule:

```
{C} chemical.mfg_name( X ) ∧ ¬( manufacturer.name( X ) )
        ∧ numbers.table( 'manufacturer' ) ∧ numbers.field( 'id' )
        ∧ numbers.number( Y )
                ⇒ chemicals.mfg_id( Y+1 )
                        ∧ NewRecord( manufacturer )
                        ∧ manufacturer.id( Y+1 )
                        ∧ manufacturer.name( X )
                        ∧ ...
                        ∧ Insert( manufacturer )
```

∧ numbers.number(Y+1)
∧ Update(numbers)

In this rule the system allows the user to define new manufacturers on-the-fly while inputting new chemical information. The rule tests the entry of the "chemical.mfg_name" field with its possible existence in the "manufacturer" table, and if it is not found there, inserts a new "manufacturer" record with automated ID assignment. One should be careful, however, with such rules since the manufacturer's name *may not* be unique among manufacturers. Indeed, should the user miss-type the name of an existing manufacturer, the system would (by this rule) insert the miss-entry as a new manufacturer with a new unique ID--definitely *not* a desired side effect. To combat this problem one could incorporate a new SDBMS function *ConfirmUnique* which would take a <table>.<field> pair as one parameter and a value as a second parameter and display all close references to that value within the <table>. The function would then request confirmation of the *value* versus the existing close values. Confirmation of the value's uniqueness (i.e., a TRUE result returning from the *ConfirmUnique* function) should be sufficient to indicate firing of the above rule. Syntactically the rule might be rewritten as follows (taking into account the requirement for user-confirmation):

{C} chemical.mfg_name(X) ∧ ¬(manufacturer.name(X))
∧ ConfirmUnique(manufacturer.name, X)
∧ numbers.table('manufacturer') ∧ numbers.field('id')
∧ numbers.number(Y)
⇒ chemicals.mfg_id(Y+1)
∧ NewRecord(manufacturer)
∧ manufacturer.id(Y+1)
∧ manufacturer.name(X)
∧ ... ∧ Insert(manufacturer)
∧ numbers.number(Y+1)
∧ Update(numbers)

5.7 Semantic "Key" Violations

Designers of SDBMS database systems need not be content with the simple key violations of typical relational systems, but may rather describe semantic context-dependent constraints which constitute rejection of committed records. Perhaps certain combinations of field values for a given record would be impossible or undesired. By representing this knowledge in the form of semantic rules one gives the system the capability of rejecting unmeaningful, inconsistent, or impossible information. The following semantic rule disallows the event of producing a toxic product directly after the production of a non-toxic product in the same reactor:

$$\{C\}\ \text{schedule.type('toxic')} \wedge \text{schedule.prior\_serial\_no(X)} \wedge \text{schedule[2].serial\_no(X)} \wedge \neg(\ \text{schedule[2].type('toxic')}\ ) \Rightarrow \text{Abort(schedule)}$$

By using semantic rules the SDBMS may extend the capabilities of more primitive database management systems. The following rule allows a database management system to disallow a *null* key entry, even though the DBMS would be incapable of such restrictions:

$$\{C\}\ \text{in\_process.product\_id(} \textit{null} \text{)} \Rightarrow \text{Abort(in\_process)}$$

Using the SDBMS Abort command allows the system designer to describe semantic or *meaningful* reasons for why certain combinations of values would be inappropriate. It gives the designer the ability to define these constraints in a context-dependent manner and centralizes this knowledge into a unified rule-base.

5.8 Built-in Referential Integrity

Many new database management systems boast the ability to control referential integrity from within the database system itself. However, older or more primitive systems still rely on external means with which to manage referential integrity (see chapter 3 for a discussion of referential integrity). By defining referential integrity in the form of semantic rules one accomplishes two feats: (1) all database management systems under SDBMS control may now support referential integrity, (2) referential integrity is centralized into a readily accessible knowledge base and hence all database systems share the same representation scheme (i.e., one need not concern oneself with the differences of how two databases engines would handle referential integrity). The following example depicts a referential integrity scenario and the semantic rules which would maintain such integrity:

Referential integrity:

schedule.product_id $\rightarrow$ product.id

Insertion into the "schedule" database:

{A} schedule.product_id(X) $\wedge$ $\neg$(product.id(X))
$\Rightarrow$ RejectValue(schedule.product_id)

Deletion of "schedule" record--OK since deletes do not cascade upward to "product" database.

Insertion/modification of "product" record:

{C} product.id[Δ](X) $\wedge$ schedule.product_id(X)
$\wedge$ product.id(Y)
$\Rightarrow$ schedule.product_id(Y) $\wedge$ Update(schedule)

Deletion of "product" record:

{R} product.id(X) ∧ schedule.product_id(X)
⇒ Delete(schedule)

Thus, with these few simple semantic rules we are able to maintain the referential integrity of schedule.product_id → product.id, providing any database management system under SDBMS control with the means to manage referential integrity.

5.9 Semantic Rejection of Values

Much in the same way one can define the semantic constraints under which *records* may be rejected, one may also define semantic constraints under which *field values* may be rejected. Again field value rejection need not be defined globally (e.g., in the case of "field F_1 may only contain the values A, B, or C, and no others"), but rather may be described in a context-dependent manner (e.g., "field F_1 may normally contain the values A, B, or C, but may contain the field value D if the value of field F_2 is E," etc.). Take for instance the following:

{A} ¬(storage.container('drum')) ∧ ¬(storage.container('tank'))
∧ ¬(storage.container('tank truck'))
∧ ¬(storage.container('cardboard box'))
∧ ¬(storage.container('plastic bag'))
∧ ¬(storage.container('paper bag'))
∧ ...
⇒ RejectValue(storage.container)

{A} storage.product_id(X) ∧ product.id(X) ∧ product.type('liquid')
∧ ¬(storage.container('drum'))
∧ ¬(storage.container('tank'))
∧ ¬(storage.container('tank truck'))
⇒ RejectValue(storage.container)

In this example the first rule globally defines the valid values of field "storage.container," while the second rule constrains those values to a specific subset should the type of product to be stored be "liquid." This technique may also be used to define certain aspects of referential integrity (as described above) and to control redundant data (also mentioned above) as in the following:

{A} chemical.mfg_name(X) ∧ chemical.mfg_id(Y)
∧ manufacturer.id(Y) ∧ ¬(manufacturer.name(X))
⇒ RejectValue(chemical.mfg_name)

Thus, we ensure that any change to the redundancy of the manufacturer's name in the "chemical" table must jive with the manufacturer's ID to be accepted by the SDBMS.

5.10 Extended Referential Integrity

By using semantic rules the SDBMS is capable of handling *extended referential integrity* (as was described in chapter 3). To reiterate, extended referential integrity is similar to standard referential integrity except that constraints may be defined which *do not* insist upon specifically matching foreign keys, but rather may be dependent on the simple field values of other records, or even the existence of several other records. The following rule insists that an "in_process" record may not contain the value 'toxic' in the simple field "type" unless there exists *at least one record* in the "product" table whose simple field "type" contains the value 'toxic.':

{A} in_process.type('toxic') ∧ ¬(product.type('toxic'))
⇒ RejectValue(in_process.type)

5.11 Context-dependent Forced Value-acquisition

Certain contexts may insist that specific field values be acquired from the user (or program interface). Take for example the following SDBMS rule:

{C} in_process.product_id(X) ∧ product.id(X)
∧ product.type('toxic') ∧ in_process.handling_auth(*null*)
⇒ AcquireValue(in_process.handling_auth)

This semantic rule ensures that any production of a toxic product must be accompanied by a handling authorization number before proceeding. Hence, with this type of rule we force the interfacing entity to obtain a handling authorization number before the record may be committed to the "in_process" database.

5.12 Concluding Remarks On SDBMS Representation of Database Semantics

One could continue to identify a potentially endless roster of semantic information representable by the SDBMS rule-based language. The types of semantic information which regulate the manipulative aspects of a given database system may range anywhere from simple inferable field values to complex integrity maintenance. Rules need not only reference single types of database management systems, but may rather reference a host of differing databases, all of which requiring unique database engine interaction. Thus, semantic knowledge may span many different database implementations, providing a means for universal data exchange and platform independence.

We have seen that complex semantic information can be easily represented with this rule-based language, providing a close coupling of knowledge and data. This close coupling is important for both developer and user in that semantic information may be readily accessible to both. The developer need no longer be concerned with producing complex customized programs to implement semantic knowledge as has been required with most of today's database management systems. By adding a few simple semantic rules to the system the developer may accomplish complex tasks which in the past would have required integration of complex interfaces, customized programs, or even loosely coupled expert systems. Semantic rules which are added to the system may be done so by a host of developers, centralizing all semantic knowledge and making semantic changes readily available for all developers. Users may find the rule-base indispensable not only in the sense that the system itself would automate much unnecessary data entry and maintenance, but also in the sense that the rule-base itself may be used to inform them of semantic repercussions due to certain actions on the data. By associating a documentation paragraph with each semantic rule an interested user would be able to browse through the semantic information embedded in the rule-base. This could be carried out by using backward-/forward-chaining methods to determine applicable rules to a <table>.<field> pair requested by the user and displaying appropriate documentation associated with those rules.

At best the SDBMS minimizes the often haphazard integration of independent customized applications required to implement semantic aspects of database manipulations, eliminating the problem of modification migration from customized program to customized program. One gains a powerful means to semantically and universally centralize enormous amounts of shared data.

6.0 THE SDBMS SEMANTIC INTERFACE (SI)

We finally concern ourselves with the basic requirements for *how* a user, developer, or even a program might interact with the SDBMS. The SI's primary function is to provide a means for a user, developer, or program (henceforth referred to as the "operating entity") to manipulate a universal array of relational data through a single interface. The SI shuttles the operating entity's SDBMS command to the SE where it is interrogated by applicable rule-base(s), processing any semantic repercussions brought forth by the command. The SI itself may take different forms depending on the type of operating entity.

6.1 Semantic Interface to the User

The most notable database manipulative aspects concerning human operating entities (users) would lie in the areas of (1) finding the intended data to be modified, (2) acquiring a basic understanding of the intended data (if necessary), and finally (3) modifying said data. Given the first point the user must have the capability of *precisely* identifying the database which he/she intends to modify (i.e., the user must specify the correct database by matching it with an item in the SDBMS symbol dictionary). Once the database symbol is acquired from the symbol dictionary the SDBMS has an immediate understanding of the database's type (i.e., which database engine is required to manipulate it), its structure, and its location on some (potentially networked) storage device. With this information the SDBMS is able to "open" the specified database (i.e., acquire the database as a resource), refer to the database's semantic rule-base, and "open" any databases which may be affected by the rules

regulating that database's semantic integrity. However, identifying the correct database symbol could pose a substantial problem for the user as the SDBMS symbol dictionary may be quite large. To combat this problem the SI could make use of a simplistic form of natural language processing (NLP) to assist the user in his/her navigation through the symbol dictionary.

Having the database administrator attach one or more descriptive sentences (documentation) to a database symbol during its creation would provide at least some form of natural language assistance during database navigation. Applying the same scheme to a database's structure (i.e., attaching descriptions to field names) would provide an even more detailed synopsis on a given database's purpose. The envisioned result would be a form of *information browser* which would take an initial list of keywords from the user, query those keywords against the descriptions of entries in the SDBMS symbol dictionary, and generate a list of descriptions to potential tables matching the user's criteria. The user could then browse through this list, perhaps issuing further keyword constraints, until the desired database is identified.

The second aspect of user interaction concerns itself with the user acquiring a "basic understanding of the database" which he/she intends to manipulate. By this it is meant that the user may be curious about repercussions of certain database manipulations. For example, if a user asks the question "if I update field α_1 of table α, what tables (if any) will be affected by the action?" By consulting the given database's semantic rule-base, an answer to this question may be automatically compiled by forward-chaining on the <table>.<field> pair in question. This ability to navigate through semantic repercussions constitutes a form of *semantic browser* in that the user may be made readily aware of semantic aspects concerning database manipulations.

The information and semantic browsers would prove especially indispensable with regard to database systems intended to train individuals in the workings of the particular domains which are represented by the database systems. The same methodology could be applied to a user asking *why* a particular inferable field value came to be as the result of a certain database manipulation. In short, the same descriptive capabilities, previously boasted only by expert systems, are now possible with regard to database manipulations through use of the SDBMS.

Once the symbol for the desired database is made aware to the user he/she may manipulate the data as desired, issuing SDBMS commands to the SI, which then hands-off to the SE for processing. Certainly the use of today's graphical user interfaces (GUIs) along with advanced NLP integration would be most useful in this type of interface. Whatever the interface the most important duties of the SI is to assist the user in navigating through the vast array of available information and provide a single interface capable of accessing multiple database engines and hence universally centralizing information.

6.2 Semantic Interface to the Designer/Developer/Database Administrator

SI aspects which apply to the general user would certainly assist a designer/ developer/database administrator. Beyond the ability to browse through existing information (via the symbol dictionary) and the semantic links between them (via the rule-base) it would be nice to include some computer assisted software engineering (CASE) tools with respect to the acquisition of rules. Some developers may find the logical language of the SDBMS complex and confusing. CASE tools could be developed to ease the generation of complex rules. Referential integrity rules, for

example, could be generated quite easily by acquiring from the developer the two <table>.<field> pairs to be linked and having the system prompt the developer for specific actions pertaining to cascade effects. Take the following scenario for example:

Developer's requested referential integrity:

schedule.product_id $\rightarrow$ product.id

SDBMS automatic generation of rule for insertion into the "schedule" database:

{A} schedule.product_id(X) $\wedge$ $\neg$(product.id(X))
$\Rightarrow$ RejectValue(schedule.product_id)

SDBMS automatic generation of rule for insertion/modification of "product" record:

{C} product.id[Δ](X) $\wedge$ schedule.product_id(X)
$\wedge$ product.id(Y)
$\Rightarrow$ schedule.product_id(Y) $\wedge$ Update(schedule)

SDBMS:

"Should deletion of schedule record(s) remove the applicable product record(s)?"

Developer:

"No."

SDBMS:

(Deletion of "schedule" record--OK since deletes do not cascade upward to "product" database.)

SDBMS:

"Should deletion of product record(s) remove the applicable schedule record(s)?"

Developer:

"Yes."

SDBMS automatic generation of rule for deletion of "product" record:

{R} product.id(X) $\wedge$ schedule.product_id(X)
$\Rightarrow$ Delete(schedule)

These automations can be accomplished by making use of the following pseudo-code template for referential integrity rule acquisition.

REFERENTIAL INTEGRITY($\alpha.\alpha_1 \rightarrow \beta.\beta_1$):

GENERATE:
{A} $\alpha.\alpha_1$(X) $\wedge \neg$($\beta.\beta_1$(X))
$\Rightarrow$ RejectValue($\alpha.\alpha_1$)

GENERATE:
{C} $\beta.\beta_1[\Delta]$(X) $\wedge$ $\alpha.\alpha_1$(X) $\wedge$ $\beta.\beta_1$(Y)
$\Rightarrow$ $\alpha.\alpha_1$(Y) $\wedge$ Update(α)

IF (*"Should deletion of α record(s) remove the applicable β record(s)?"* **== YES) THEN GENERATE:**
{R} $\alpha.\alpha_1$(X) $\wedge$ $\beta.\beta_1$(X) $\Rightarrow$ Delete(β)

IF (*"Should deletion of β record(s) remove the applicable α record(s)?"* **== YES) THEN GENERATE:**
{R} $\beta.\beta_1$(X) $\wedge$ $\alpha.\alpha_1$(X) $\Rightarrow$ Delete(α)

Conceivably, additional templates could be configured for other types of semantic rules, thus easing the rule acquisition process. With this approach developers can quickly embed complex semantics directly into the databases themselves without the need to integrate tedious customized programs.

Attaching one or more sentences of documentation to each rule (much in the same way as described for the symbol dictionary) would provide the means for a *semantic-rule browser*. With the use of such a browser developers could quickly and intelligibly navigate through the vast array of semantic rules. Given a database symbol acquired from the developer the SI could produce a visual map which would provide an overview of tables *used* by the given database (i.e., those tables appearing within the *antecedent* of rules applicable to the database) and tables *affected* by the given database (i.e., those tables appearing within the *consequent* of rules applicable to the database). The visual map could then be extended by applying the same scheme to each of the tables linked to the originally queried database. Merging this type of browser with descriptions of each table and its fields can produce a highly detailed diagram of how the information is intended to interact--even cross-platform interaction (i.e., interaction between differing database engines). Once again the SDBMS has achieved centralization of information and a universal interface for both user and developer alike.

6.3 Semantic Interface to Programs

Although the need for customized programs which interact with databases has been greatly minimized by the SDBMS (as has been previously noted), it may still be necessary from time to time to link database information to specific applications. This would most likely be the case for automated information acquisition, automated reporting, etc. The programmer would no doubt find the browsing capabilities of the SDBMS indispensable in developing such programs. The browsers would be used during development of the applications to acquire the necessary database symbols

required for data manipulation through the SDBMS. Once the database symbols are known the programmer would embed the respective SDBMS commands directly into the application. The SDBMS would, in effect, function much in the same way as a typical database engine, requiring the program to first procure a linkage between itself and the SDBMS, and then send the desired SDBMS commands directly to the SE for processing. Thus, use of the SDBMS would provide application designers with a seamless linkage to many different types of databases, while maintaining a single type of programming interface for all engines.

7.0 SDBMS PROTOTYPE IMPLEMENTATION

An SDBMS prototype was developed for use under Microsoft™ Windows 3.1x. The system was written in C using the Borland™ C++ 4.0 compiler and utilized Borland's Paradox™ Engine as its database platform. The prototype was intended to demonstrate the basic functionality of the three categories of semantic rules--(A)quisition, (C)ommittal, and (R)emoval--and provide proof of concept for the methodologies presented in chapter 4.

The prototype consists of a main window which allows the user to view one of several Paradox databases by selecting the desired database from a drop-down combobox located in the upper left corner of the main window. Once a database is selected the main window will display the valid fields of the chosen database and the field values of the first record in the database. Figure 11 depicts the main window when displaying a record in the "INPROC" database.

The button-bar located at the top of the main window allows the user to define a *new* record to be inserted into the current database; *search* for a particular record in the current database (see figure 12a); *delete* the currently displayed record; queue the

next or *prior* record in the database (as sorted by the database's primary key); *quit* the prototype; ask *why* the last semantic repercussion(s) occurred; or *undo* any changes made to the currently displayed record. To change a particular field's value one simply moves the cursor to the appropriate edit box within the main window and enters the respective information.

7.1 Implementation Model of the SDBMS Prototype

The chosen model for implementation is that which was described in section 2.1 above. Figure 7 depicts the model and some of the databases used to represent the information of a fictitious production plant. Appendix B gives a listing of the semantic rules which were included in the prototype.

The prototype was designed to demonstrate the basic functionality of SDBMS semantic rules. As was detailed in previous chapters, any semantic rule may be reduced to a SEARCH-TEST-ACT chain, and it is the processing of these chains which has been directly implemented within the prototype. The rules themselves are stored within two databases. The first database, "SE_RULE," references the rule's *type* (acquisition, committal, or removal); the <table> or <table>.<field> to which the rule applies--facilitating forward-chaining; the rule's *id*; and a natural language *reason* for why the rule would potentially fire. The second database, "SE_CHAIN," references the SEARCH-TEST-ACT components of the rule. For any given rule there exists *x* number of records found within this table--one for each component in the rule's SEARCH-TEST-ACT chain. Each record contains the following: the rule's *id*; the component's sequence in the chain beginning with 1; the chain component's *command* (e.g., test, search, set, abort, rejectValue, update, insert, etc.); and three sets

of fields defining the two operands and the operator of the command (i.e., the command's parameters). Take for instance the following rule, its SEARCH-TEST-ACT chain, and its representation within the two tables utilized by the prototype:

3300. {A} holding.chemical_id(X) ∧ chemical.id(X) ∧
chemical.name(Y)
⇒ holding.chemical_name(Y)

RULE: A/holding.chemical_id
SEARCH: chemical.id = holding.chemical_id
TEST: ε
ACT: Set(holding.chemical_name = chemical.name)

"A chemical ID implies a specific chemical name."

SE_RULE:

RULE TYPE:	"A"
TABLE:	"HOLDING.CHEMICAL ID"
RULE ID:	"3300"
REASON:	"A chemical ID implies a specific chemical name."

SE_CHAIN:

RULE ID:	"3300"
SEQUENCE:	"1"
COMMAND:	"SEARCH"
LOP1:	"CHEMICAL.ID"
OP1:	"="
ROP1:	"HOLDING.CHEMICAL ID"

RULE ID:	"3300"
SEQUENCE:	"2"
COMMAND:	"TEST"
LOP1:	"<EXIST>"
OP1:	"="
ROP1:	"<TRUE>"

```
RULE ID:    "3300"
SEQUENCE:   "3"
COMMAND:    "SET"
LOP1:       "HOLDING.CHEMICAL NAME"
OP1:        "="
ROP1:       "CHEMICAL.NAME"
```

7.2 Core Semantic Engine Functions

The SE itself was written in C and uses several generic database functions to control semantic context and manipulate the Paradox databases. The core SE functions are as follows:

```
SE_SetField( <tableName>, <LGIndex>, <fieldName>, <fieldValue> );
SE_InsertRecord( <tableName>, <LGIndex> );
SE_UpdateRecord( <tableName>, <LGIndex> );
SE_DeleteRecord( <tableName>, <LGIndex> );
```

The SE_SetField function is used to access semantic context. Setting a field for a given <tableName> and <LGIndex> will associate that field value with the record buffer for <tableName>/<LGIndex>. The <LGIndex> parameter is used to keep track of two or more records of the same type--i.e., one may wish to access the same table in two or more record locations, maintaining record buffers for each. A call to the SE_SetField function initiates a search of the "SE_RULE" database for any acquisition-type rules applicable to the <tableName>.<fieldName> which is to be set. Should an applicable acquisition rule be found, its SEARCH-TEST-ACT chain is processed--each applicable record of the "SE_CHAIN" database. This command is called from the Windows interface each time a user enters a field value. Any consequents which arise from a rule's firing are stored within the SE's *reason chain*. The current reason chain may be accessed at any time by pressing the "Why?" button

located in the button bar on the main window. Figure 12c depicts a portion of a reason chain which would be presented to the user upon pressing the "WHY?" button. Should a user enter a value which ultimately results in a RejectValue consequent, an audio alert sounds and the old value (if any) is reset. Pressing the "Why?" button at this time allows the user to determine the reason for that particular value's rejection.

The SE_InsertRecord command is called whenever a new record is entered from the interface. To enter a new record the user first presses the "New" button, enters the subsequent field values relative to the new record (causing the SE to consider any/all relevant acquisition rules), and commits (inserts) the record into the database. Committal automatically occurs when the user presses any button in the button-bar or selects a new table to view. When the SE_InsertRecord command is called, the SE searches the "SE_RULE" database for any relevant committal rules associated with the <tableName>. Any consequents which arise from fired committal rules are stored within the reason chain and may be accessed via the "Why?" button as described above. Should a committal rule result in an Abort consequent, a message box appears (figure 12b), and the record continues to be displayed until the information is verified by a subsequent committal or the "Undo" button is pressed. Once all rules have been considered and no Abort-consequent is inferred, the record is inserted into the database.

The SE_UpdateRecord command is called when a user commits new information for a pre-existing record. The functionality is for the most part identical to SE_InsertRecord. However, when all rules have been considered and no Abort-consequent arises, the record is updated as opposed to inserted. Similarly the SE_DeleteRecord command processes any removal rules associated with the <tableName>, and, should no Abort-consequent occur, removes the record from the database.

7.3 SEARCH-TEST-ACT Chain Processing

The SE incorporates a recursive strategy when processing rules. Whenever a SEARCH occurs, an internal multiple loop is initiated--as described in chapter 4. At this point, should the initial search succeed, the rest of that SEARCH-TEST-ACT chain is processed with respect to the binding of that searched-out record. Upon completed processing of the chain, control returns recursively to search for the *next* valid record meeting the given criteria, and, should a "next" record be found, the remainder of the rule is again processed with respect to the new binding. This internal multiple loop continues until no more records may be found. This strategy is analogous to the notion of unification and exhaustive search mechanisms of Prolog.

The recursive nature is further employed during forward chaining. Take for example rule 2500 of Appendix B. One of the consequents of this rule initiates an SE_SetField for the "volatile" field of the currently queued "products" record. Thus, by calling SE_SetField any acquisition rules applicable to "products.volatile" will be considered and potentially fire (in fact, rule 1700 of Appendix B would fire)--the firing of those consequents possibly chaining further in the rule-base. Similarly the Update consequent of rule 2500 would result in forward-chaining on any committal rules associated with "products."

7.3.2 De-aliasing Within SEARCH-TEST-ACT Chain Processing

A de-aliasing strategy is used within SEARCH-TEST-ACT processing to yield specific bindings within the rule. For example, the operand "<DELTA> products.id" would be de-aliased to yield the delta (last) value of the "id" field for the currently

queued "products" record. The operand "products.id" would be de-aliased to yield the current value of the "id" field of the currently queued "products" record. The "<EXIST>" operand is de-aliased to be either "<TRUE>" or "<FALSE>" depending on whether the last SEARCH was successful. Once a particular parameter is de-aliased it may be acted upon by the command. The SE's de-aliasing of parameters is somewhat equivalent to the binding/unification which occurs in Prolog when moving from a variable-designation to a bound value or calculation.

7.4 Signature C™

Although the semantic engine currently links to only Paradox databases (i.e., the prototype is homogeneous with respect to a single database engine), the system was developed using a powerful database engine front-end--*Signature C™*. This front-end engine was developed over a two-year period by this author and co-developed by Robert S. Voros. Signature C is a database engine CASE (computer-aided software engineering) tool which acts as a generic interface to the Paradox engine and eases program coding. The SDBMS semantic engine was built as a front-end to Signature C. Take for example the following lines of code (Paradox engine vs. Signature C):

PARADOX ENGINE...

```
char buffer[40];
TABLEHANDLE tblHandle;
RECORDHANDLE recHandle;

PXTblOpen( "C:\KEY\DATABASE\se_rule", &tblHandle, 0, 1 );
PXRecBufOpen( tblHandle, &recHandle );
PXRecBufEmpty( recHandle );
```

```
PXPutAlpha( recHandle, 1, "A" );
PXPutAlpha( recHandle, 2, "CHEMICAL" );
if ( PXSrchKey( tblHandle, recHandle, 2, SEARCHFIRST ) ==
      PXSUCCESS )
   {
   PXRecGet( tblHandle, recHandle );
   PXGetAlpha( recHandle, 3, 40, buffer );
   ...
   }
```

SIGNATURE C:

```
LG_SetField( "SE_RULE", 1, "RULE TYPE", "A" );
LG_SetField( "SE_RULE", 1, "TABLE", "CHEMICAL" );
if ( LG_Search( "SE_RULE", 1, "KEY", 2, SEARCHFIRST ) )
   {
   LG_GetField( "SE_RULE", 1, "RULE ID", buffer );
   ...
   }
```

As one can easily see signature C considerably reduces the codification required for database integration. Signature C was designed on the premise that fundamental database operations are shared by every relational database system (e.g., setting field values, retrieving field values, searching, inserting, deleting, etc.). Although the SDBMS prototype is admittedly homogeneous with regard to Paradox databases, it can be stated that it is heterogeneous-ready. By modifying Signature C routines to access other database engines one may gain a heterogeneous system. Hence, this heterogeneous system would be obtained by enhancing the database engine front-end while leaving domain-specific application code untouched (i.e., one need only enhance Signature C not the semantic engine to gain a heterogeneous system).

7.5 Execution of the Prototype

Let us examine a particularly complex operation (i.e., complex for the SDBMS--not the user) to understand how the prototype functions. Let us say the user enters a new record for the "INPROC" table as depicted in figure 11. Upon attempted committal of this new record rule 2700 is first considered. In essence this rule attempts to verify that all chemicals which are listed within the product's recipe exist within a record of the "holding" database (recall the "recipe" records define which chemicals/quantities are required to produce a given product, while the "holding" database lists which chemicals and quantities thereof are currently available at the plant). Should there exist a particular chemical which is referenced in the product's recipe which does *not* exist within the "holding" database, the committal of the new record is aborted (i.e., if no "holding" record exists for a given chemical, the system may infer zero quantity of that chemical, and if there exists zero quantity of a chemical which is required to make a product, then that product cannot be produced). If this particular rule does not fire, then one can be certain that all required chemicals are currently inventoried at the plant. Note that this rule does not necessarily maintain that there is *sufficient* quantity of required chemicals at the plant, but simply that all required chemicals are present at the plant.

Rule 2800 would be considered next. This rule is similar to rule 2700, except that it concerns itself with the actual quantities of chemicals currently inventoried at the plant. For each chemical listed in the product's recipe there must be sufficient quantity of that chemical in holding. Hence, should x quantity of chemical y be required to make product z and there exists $<x$ quantity of chemical y currently at the plant, then product z cannot be produced and the committal of the new in-process record must be aborted.

Rule 2900 would next be considered (provided an Abort has not already occurred). This rule performs two important tasks: first, it updates the quantities of all chemicals in holding which are required to make the new product based on its recipe; second, it inserts a new record into the "INPRCHEM" (in-process chemical) table, which keeps track of which chemicals are in-process making which in-process products. If there exists x z quantity of chemical y in holding and z quantity is required to produce product p, then the holding quantity of chemical y is modified to equal x-z. Note that setting the "quantity" field of the "holding" record for chemical y forces the consideration of any relevant acquisition rules. Similarly, the update of the "holding" record forces the consideration of any relevant committal rules. Subsequently setting the fields of a new "IMPRCHEM" record and inserting that record results in consideration of any respective acquisition and committal rules respectively. In particular, should the "INPRCHEM" chemical ID be set to one which references a volatile chemical, rule 900 ensures the proper "special handling" setting for that new chemical in-process.

Thus one sees the power of the SDBMS as a simple insert of a record into a particular database can not only test the *semantic validity* of such an action (as depicted in the verification of proper chemical quantities based on a product's recipe), but may also cause a *semantic repercussion* which may affect one or more records of one or more differing tables--normally thought of as simple reservoirs of information, but now semantically linked by the SDBMS.

8.0 CONCLUSION

Clearly, since the use of database management systems has saturated virtually every facet of commercial, scientific, and educational domains, it has become a

necessity to make this information more accessible to more users. By directly incorporating high-level semantics into the basic functionality of today's database systems through the use of the SDBMS one accomplishes many feats. The most important contribution of this work is perhaps the centralization of vast amounts of information stored within differing database platforms through a universal semantic interface. By themselves relational database systems have proven to be not much more than mere vessels of information, each having limited knowledge (if any) of the databases around them--and no knowledge of databases governed by different database management platforms. Integration of the SDBMS allows these "blind" databases the essential ability to *communicate* with one another. Semantic knowledge may be coded into SDBMS rule-bases to connect two or more tables of potentially differing platforms, thus centralizing a vast amount of information. For example, an employee database governed by database engine *A* may now be semantically linked to a payroll database governed by a different engine *B* which otherwise would have not been possible. With the ability to unify many different database systems the global information exchange is increased considerably. Companies need no longer create redundant databases to capture like-information in differing platforms as the information may flow seemlessly from system to system through the SDBMS.

Another important contribution of the SDBMS is the notion of linking a semantic knowledge-base with *each* database. These knowledge-bases embody the complex semantics associated with the various databases--semantics capable, for example, of answering such questions as: "What is the underlying *meaning* of changing a product type from solid to liquid?" A transition of this type does not merely effect the superficial modification of a single field-value, but may in fact result in a causal chain of events required to maintain the *semantic integrity* of such a change; a change which could possibly effect multiple records in multiple tables

spanning multiple platforms (e.g., storage container types for a liquid versus a solid may require modification as well--a bottle versus a cardboard box; reactor privileges required to make such a product might be dictated by its type--solid or liquid--and thus would require a change in scheduling the production of that product; etc.). Hence, the SDBMS rule-base is able to both represent and implement these data-manipulative semantics.

Throughout this dissertation it has become clear that the semantic aspects--the *meaning*--of a database extends far beyond the simplistic notion of a data structure (i.e., keys, field data types, referential integrity, etc.). Data semantics span from the internal dependencies of a single record's field-values to cross-table/cross-platform dependencies of other records. It has been shown that the context or *state* of a record can effect its causal relationship(s) with other field values, other tables, or even other platforms. Depending upon its field values simple generic rules may not always apply to records, but may rather require many rules describing the various states which may occur within that record which would cause data-manipulative repercussions elsewhere.

It has been shown in the preceding chapters of this dissertation that the SDBMS's rule-based language is capable of overseeing such data-manipulative semantic aspects as integrity management, data consistency, forced-redundancy verification, field-value inferences, context-dependencies, data type checking, security, etc. In a sense the SDBMS acts as an automated database administrator, overseeing much (if not all) of the operations which were previously only possible through human intervention or the tedious integration of customized programs. With the fusion of rule-base technology with existing database technologies the SDBMS makes primitive database management systems far more powerful and makes powerful database management systems more flexible. Database systems may be developed and

implemented on-the-fly, embedding complex semantic aspects which were previously only boasted by semantic modeling schemes, but which are now directly implementable through the SDBMS rule-based language. Database systems themselves may be more readily understood by both developers and end-users alike given the extensive documentation-embedding techniques available to rule-bases associated with databases governed by the SDBMS. The browsing capabilities and proposed integration of simplistic natural language processing techniques boasts a more powerful, less-confusing interface for both users and developers alike.

Another crucial contribution of this work lies in substantially lessening the burden posed to programmers; those of whom in the past have had to expend a great deal of time and effort dedicated to coding complex applications that would be subsequently linked to a particular RDBMS to implement the otherwise lacking data-manipulative semantics. Semantics can be directly built-in to databases, minimizing (if not eliminating) the need for the introduction of ad hoc customized programs. The fundamental result is that databases may evolve at a greatly accelerated rate since complex, independent codification (external to the database) is no longer necessary. Finally, database systems seem more "intelligent" as the database itself would "know" that a single command might infer the execution of several other commands--transparent to the user--based on the semantic knowledge represented within the SDBMS rule-bases.

8.1 Extending the SDBMS; Future Investigations

Over the past decade considerable attention has been paid to the research and development of object-oriented database management systems (ODBMSs). These

systems have integrated the object-oriented paradigms, which resulted from past research in artificial intelligence, to provide a modular or encapsulated approach to data management. The interested reader will note a commonality of purpose between existing ODBMSs and the SDBMS described herein. Both systems attempt to inject some degree of high-level semantics into database systems. Where the ODBMS uses an object-oriented approach to represent data semantics, the SDBMS adopts a knowledge-based approach. With the SDBMS, although semantic rule-bases are directly linked to a particular database, the rules are not encapsulated within the database and therefore promote a more shared approach to semantic representations. By not encapsulating the semantics the SDBMS is able to act in a front-end capacity. This front-end aspect is what allows the SDBMS to interface with a wide variety of database engines.

Indeed, one could certainly expand the SDBMS to interface not only with single-record manipulative and multiple-record manipulative RDBMSs, but also with ODBMSs. Such an integration would not be as difficult as may be initially conceived. Rule-base translation would require little modification as the root commands *Insert*, *Update*, and *Delete* would still apply to ODBMSs. The notion of inheritance could be taken care of through linkages of symbols in the semantic symbol dictionary in that the rules applicable to a parent class would also be applicable to the child class. For example, if A is a child of (inherits from) B, then all semantic rules applicable to B would also apply to A (i.e., any reference to B in those rules would be translated to reference A). Perhaps the most challenging enhancement would lie in modifying the working memory of semantic context to facilitate the unlimited array of user-defined types possible with ODBMSs. Where most relational database management systems have only a handful of data types, ODBMSs allow users to define new data types by arranging core types in different configurations. Thus, the working memory of

semantic context would have to be capable of generically handling a wide variety of data structures.

Another important enhancement might include extending the procedural nature of the SDBMS rule-language. Some very complex semantic issues may require the integration of functions/procedures not directly available through the SDBMS. It would be nice if developers could compile their own functions/procedures and dynamically link them to the SDBMS, calling them from the rule-base itself.

Further extensions to the semantic interface would be most useful. The areas of advanced natural language processing (NLP) and complex graphical user interfaces (GUIs) would increase the usefulness of such a universal interface. Merging advanced NLP with the extensive documentation included in the rule-bases would yield extremely powerful and user-friendly browsers for the full gambit of users and developers.

Another significant extension would include expanding the semantic rule-base to handle not only data-manipulative aspects of database management systems, but also data-utilization aspects (i.e., querying). One could easily conceive additional rule categories beyond *acquisition*, *committal*, and *removal*, to handle such aspects as query optimization, rejection of meaningless queries, correction of unintentional queries, etc. The current composition of the SDBMS--its symbol dictionary, semantic context, universal linkage to a wide variety of database engines, etc.--would no doubt prove quite useful in such an investigation.

Whatever the extension, the SDBMS should prove an invaluable asset in unifying many different forms of database management systems and promote more "intelligent" and user-friendly systems accessible to users, developers, and programs alike. The paramount result of such a semantic system would increase the access to shared information, while at the same time allowing more information to be acquired

at a much faster rate than was previously possible with more primitive systems. We may no longer be content with the vast array of differing database platforms and the ad hoc approaches to integrate them here and there. By directly merging the aspects of both rule-bases and databases into a single, unified force, we grow ever closer to the notion that information is knowledge. As a result we promote more "intelligent" systems and thus make our own lives that much easier.

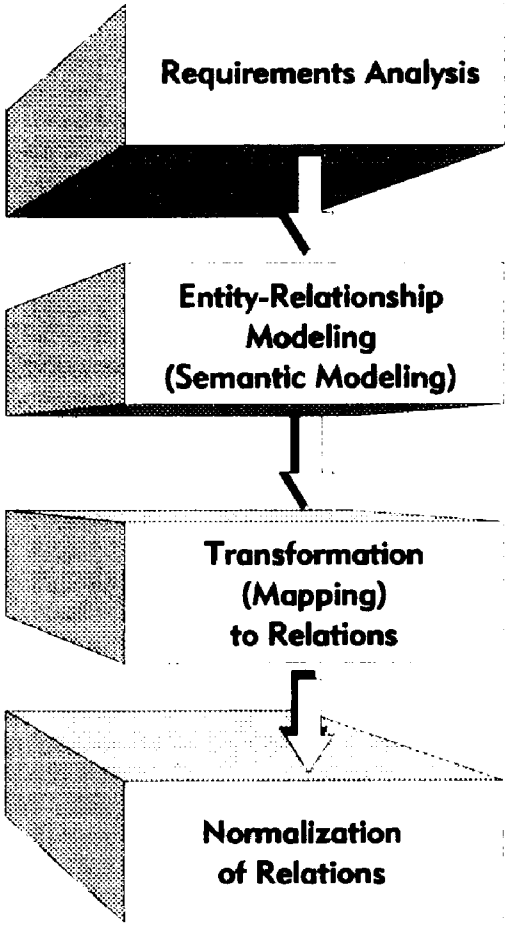


Figure 1. Design-implementation-normalization-customization Cycle.

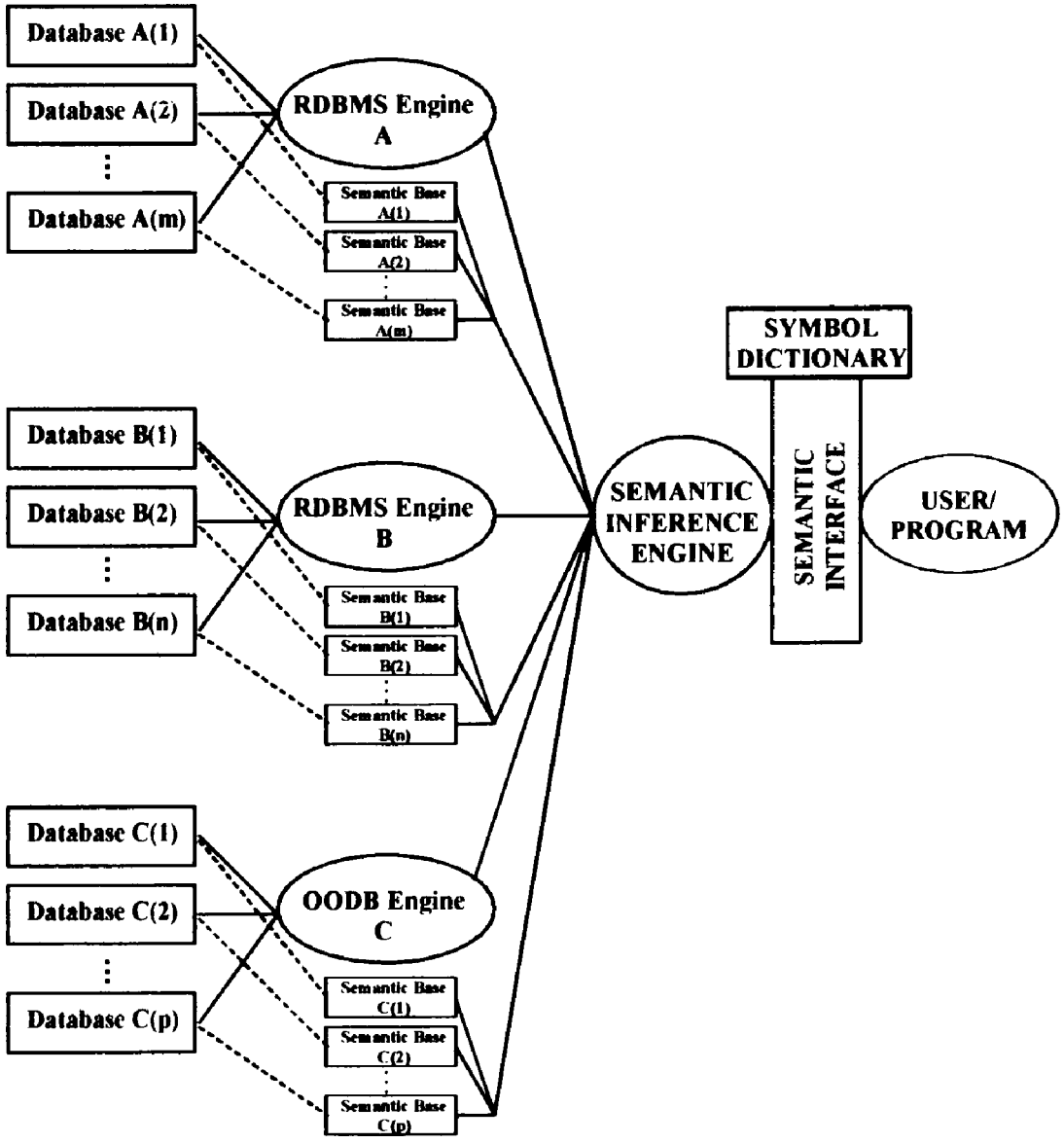


Figure 2. Front-end Universal Medium Diagram.

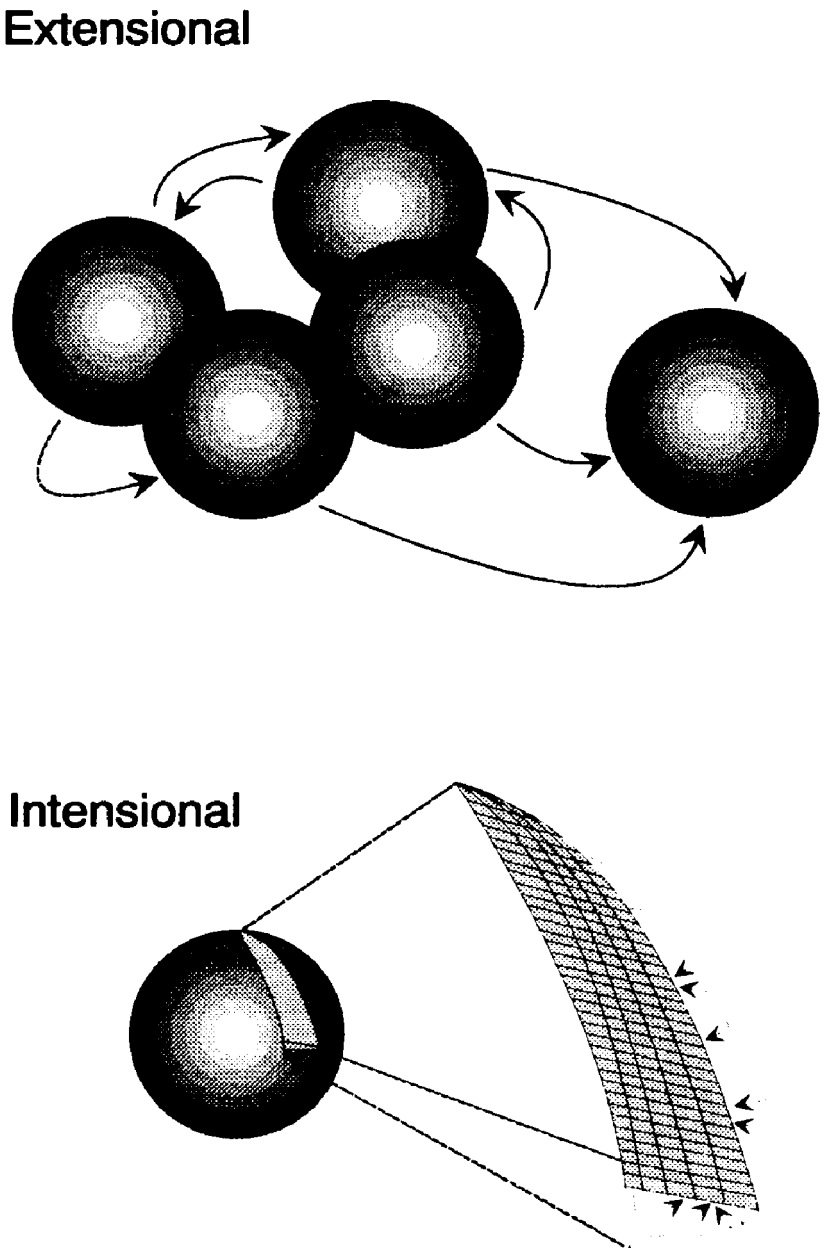


Figure 3. Extensional and Intensional Paradigms.

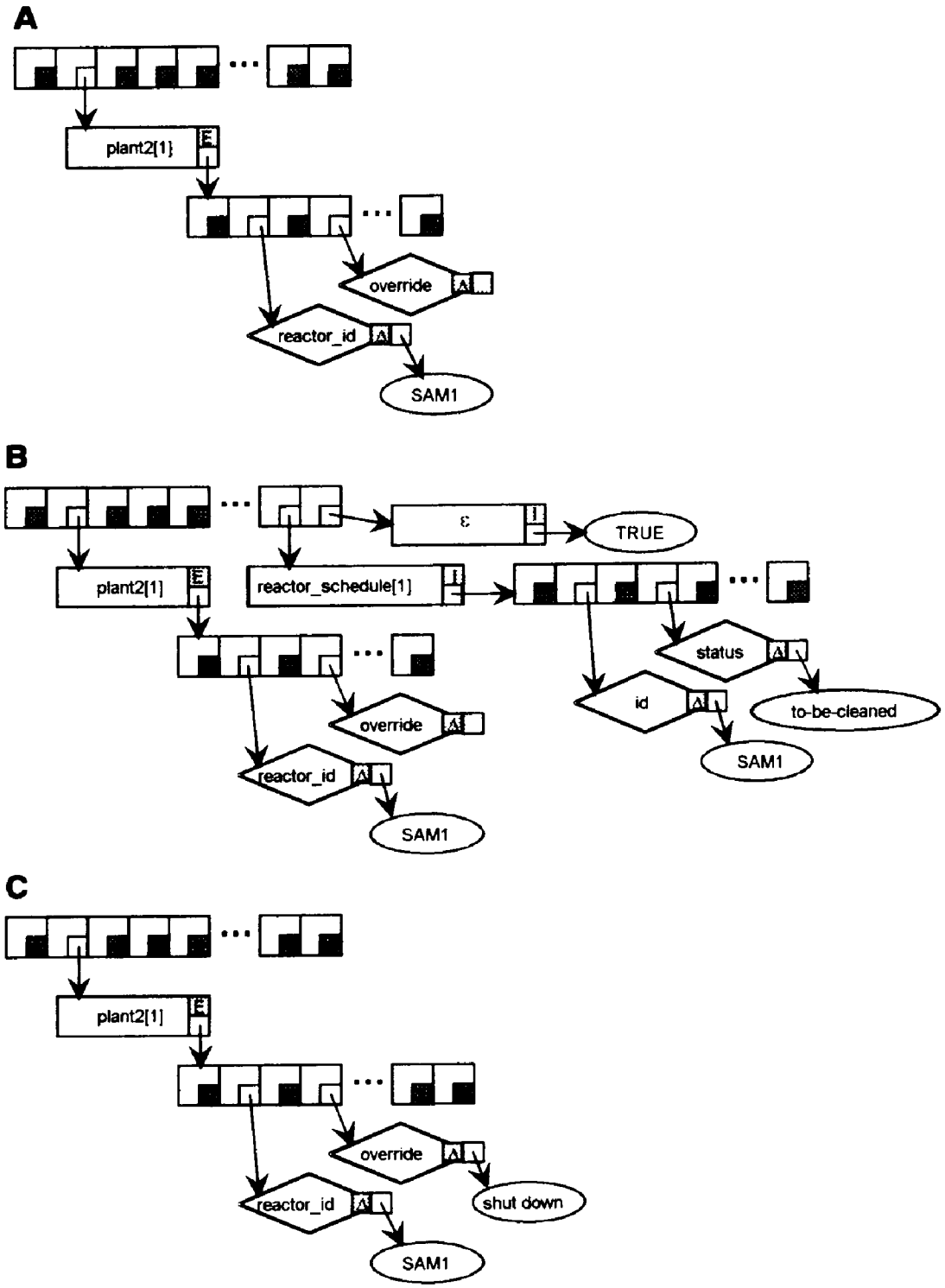


Figure 4. Example of Semantic Context Interaction.

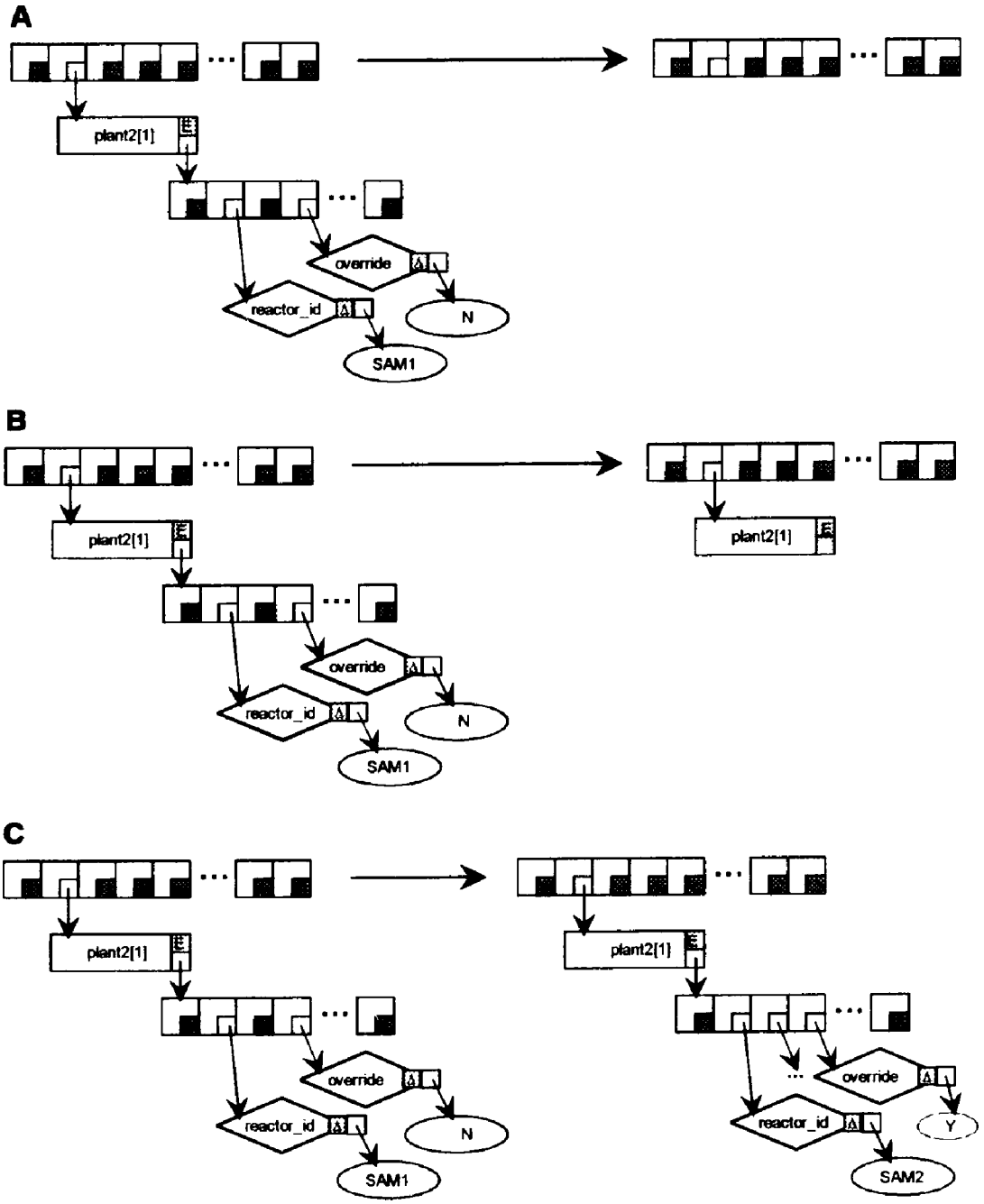


Figure 5. Example of Purging Explicit References Within Semantic Context.

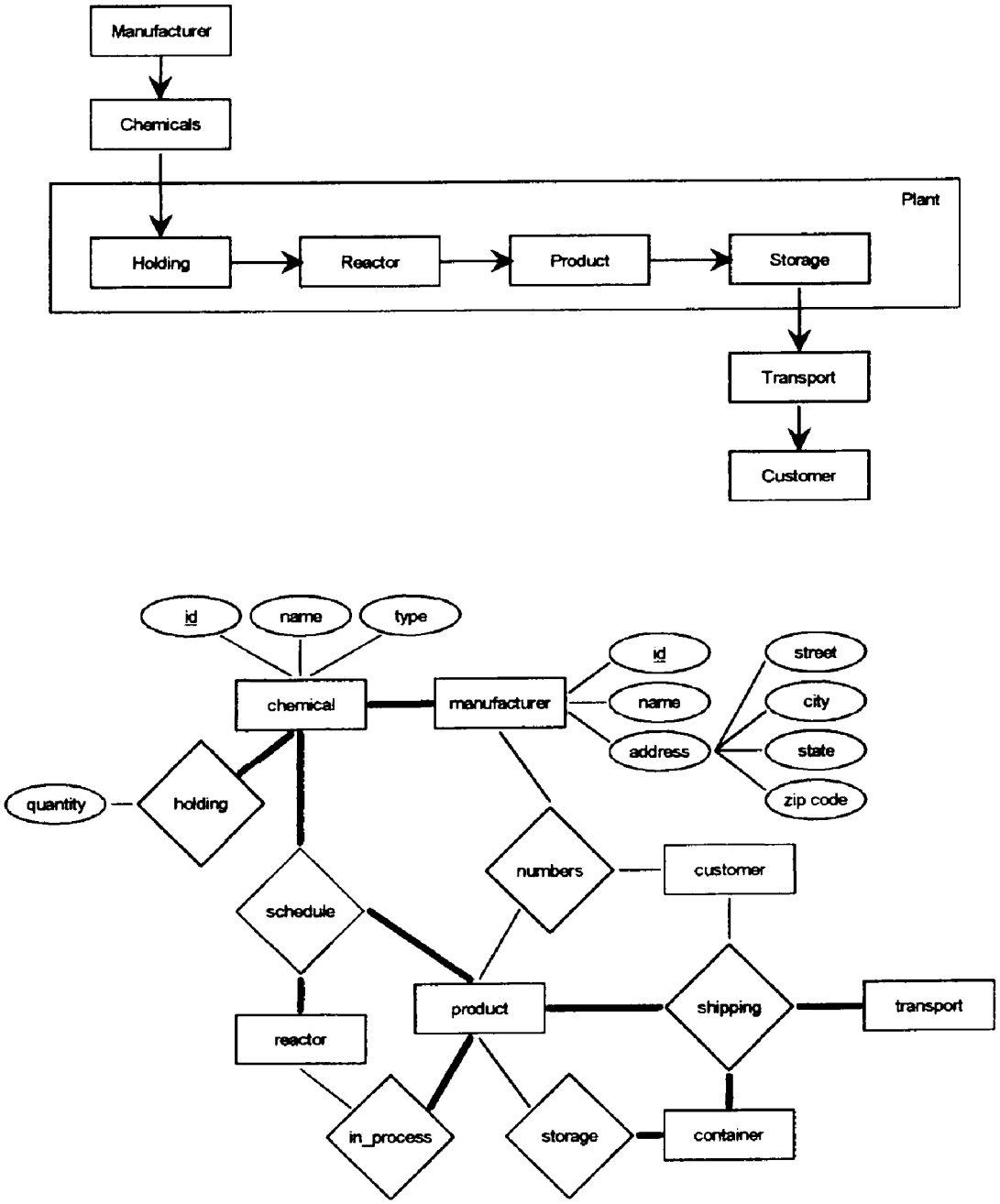


Figure 6. Sample Logical Design and E/R Diagram.

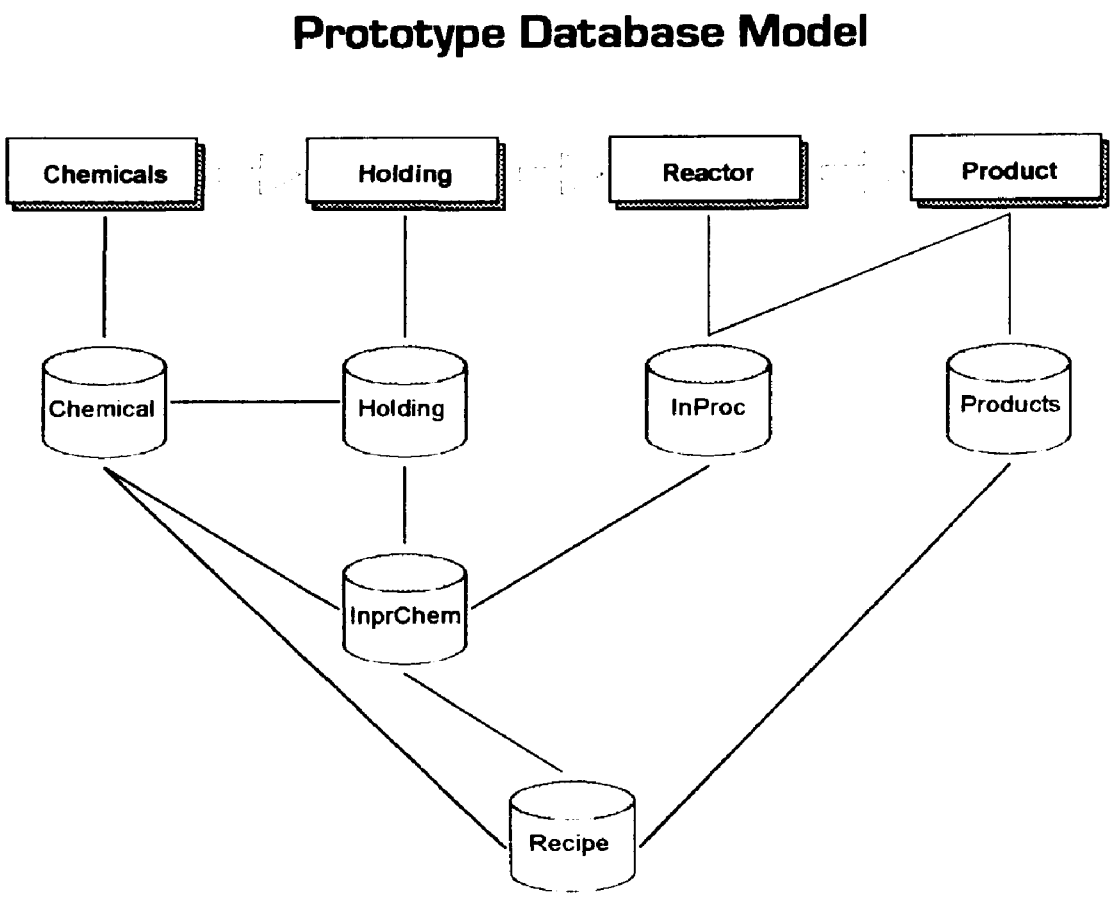


Figure 7. Logical Design Used By SDBMS Prototype.

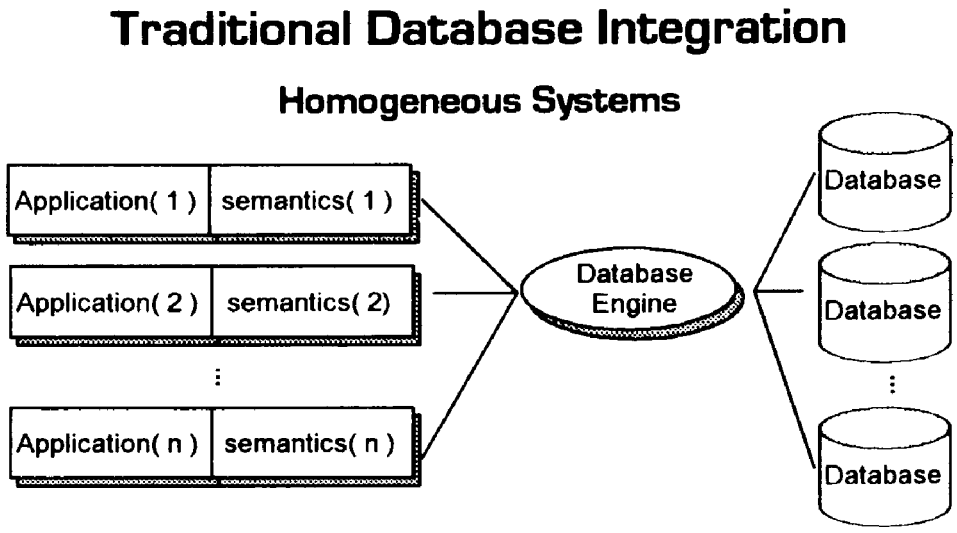


Figure 8. Traditional Database Integration--Homogeneous.

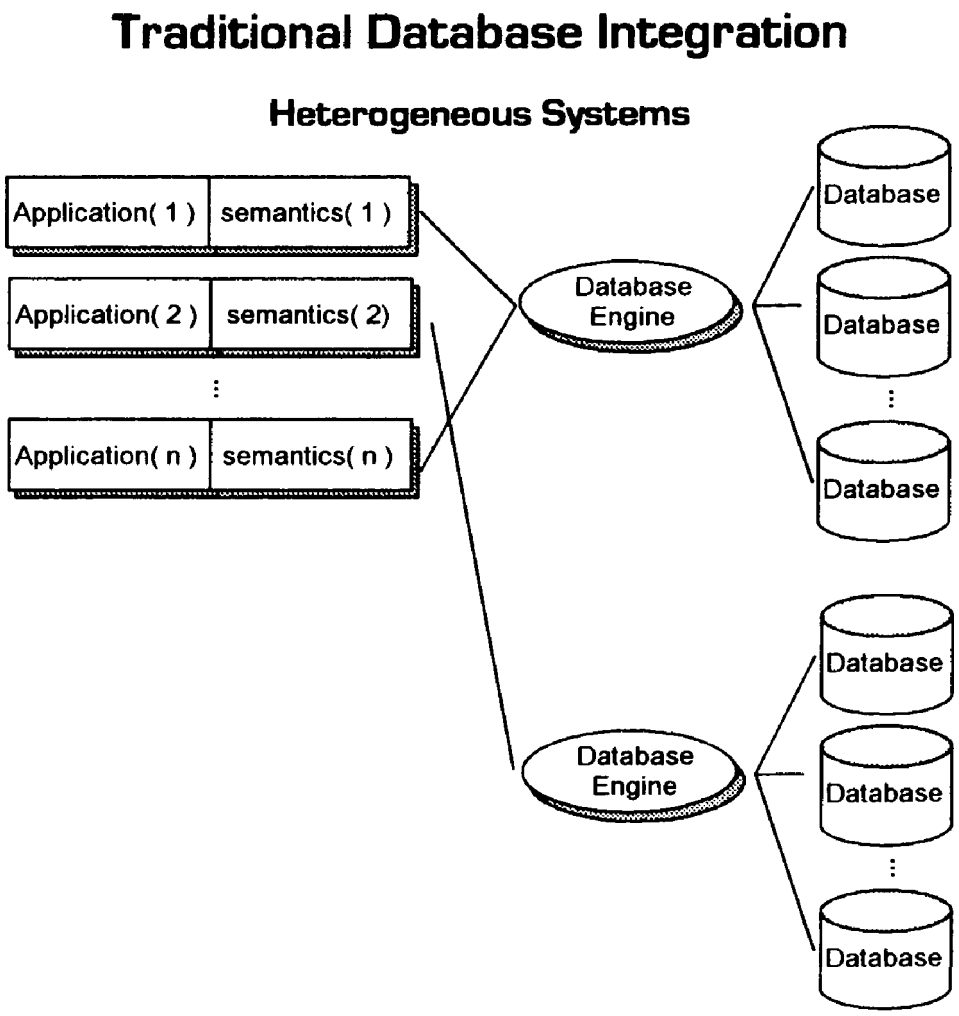


Figure 9. Traditional Database Integration--Heterogeneous.

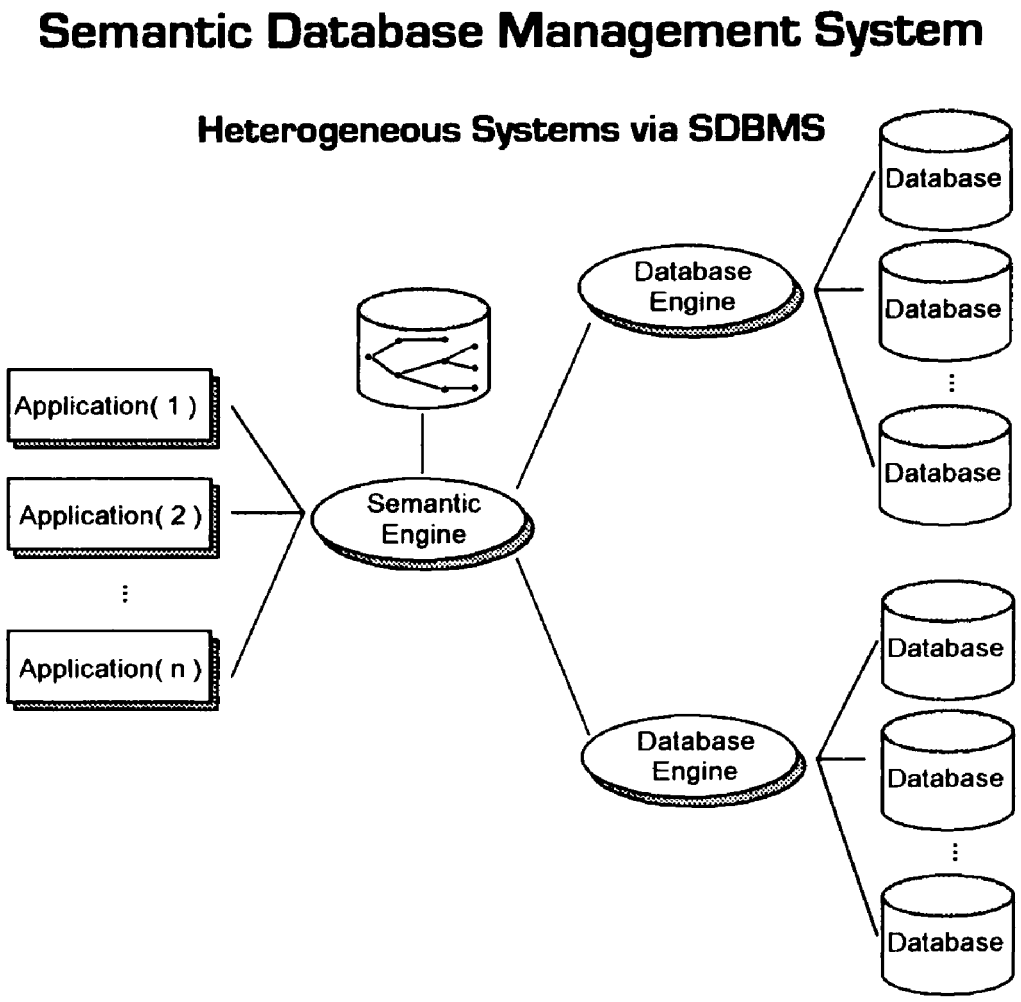


Figure 10. Database Integration Via SDBMS--Heterogeneous.

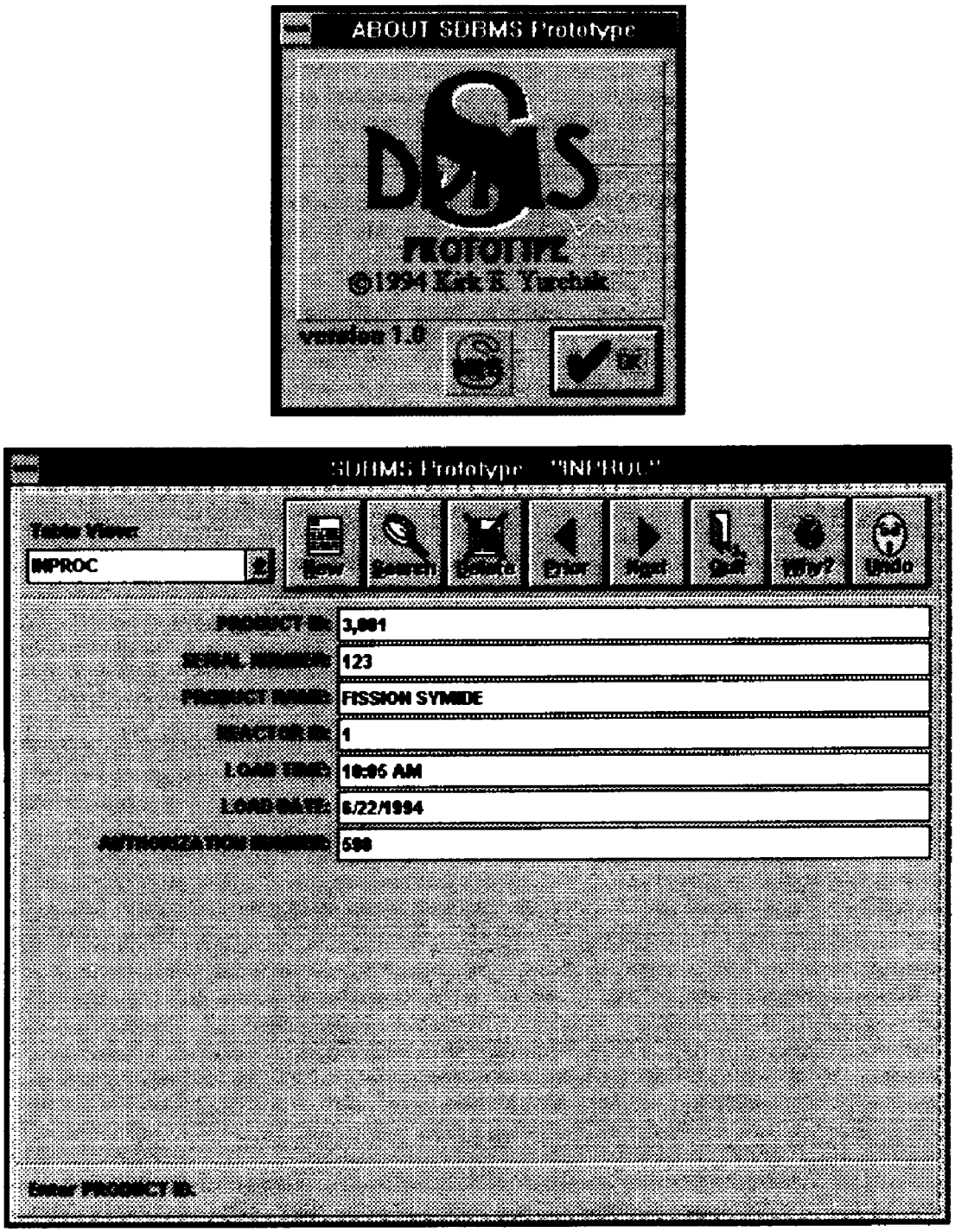
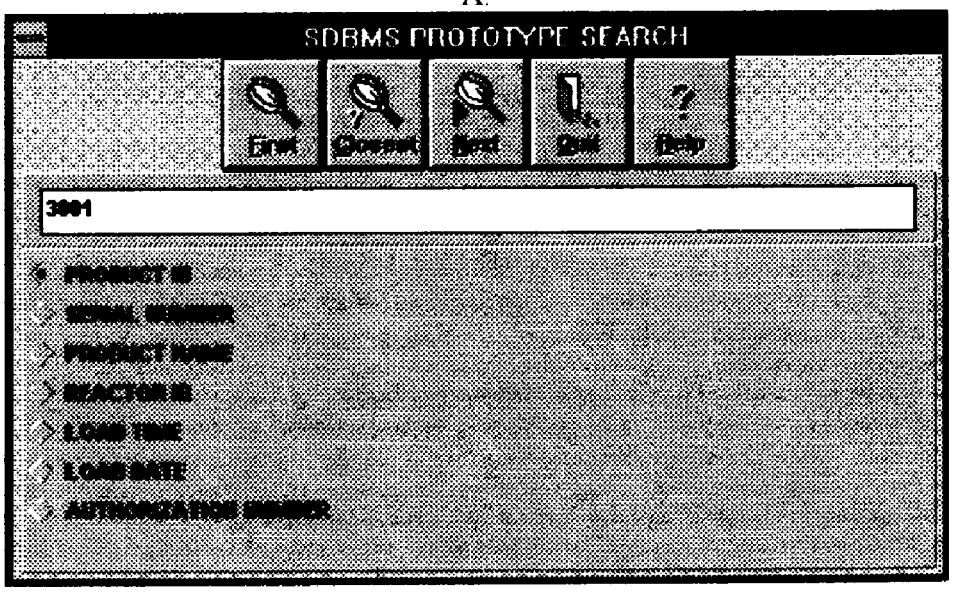
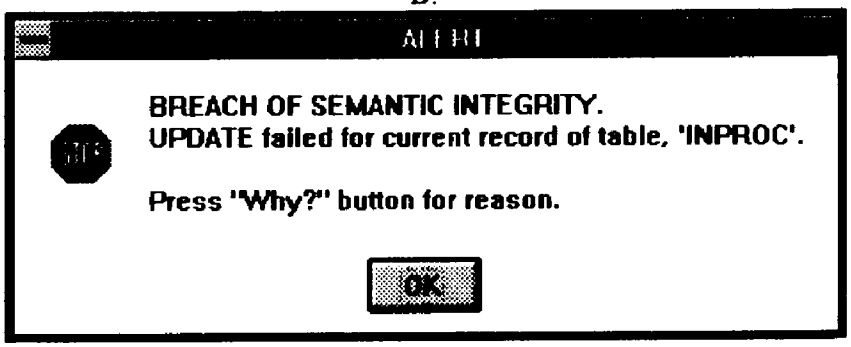


Figure 11. SDBMS Prototype--Main Window.

A.



B.



C.

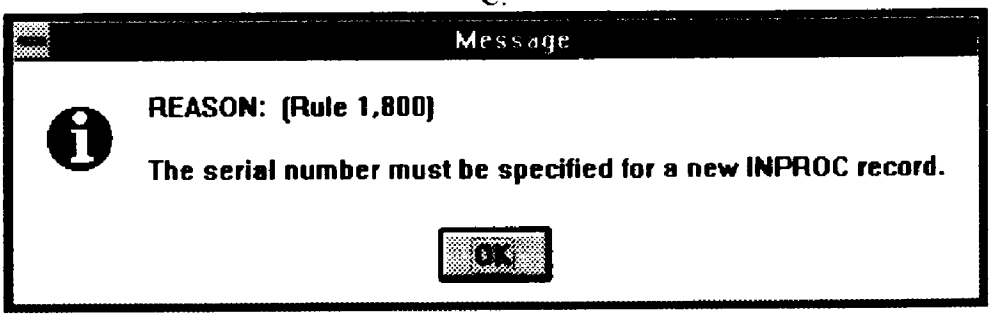


Figure 12. (A) SDBMS Prototype--Search Window. (B) SDBMS Prototype--Committal Breach Message Box. (C) SDBMS Prototype--Reason Message Box.

Bibliography

Brown, A. (1991). Object-Oriented Databases - Applications in Software Engineering. McGraw-Hill Book Company.

Butterworth, P., Otis, A. & Stein J. (1991). "The GemStone Object Database Management System." Communications of the ACM. Vol. 34. No. 10. October, 1991.

Date, C. J. (1990). An Introduction to Database Systems. (5th ed.). Addison-Wesley Publishing Company.

Deux, O. et al. (1991). "The O_2 System." Communications of the ACM. Vol. 34. No. 10. October 1991.

Geoffrion, A. M. (1992). "The SML Language for Structured Modelling: Levels 1 and 2; Levels 3 and 4." Operations Research. Vol. 40. No. 1. January-February 1992.

Giarratano, J. C. & Riley, G. (1989). Expert Systems - Principles and Programming. PWS-KENT Publishing Company.

Hughes, J. G. (1991). Object-Oriented Databases. Prentice-Hall International (UK) Ltd.

Lamb, C., Landis, G., Orenstein, J. & Weinreb, D. (1991). "The ObjectStore Database System." Communications of the ACM. Vol. 34. No. 10. October 1991.

Lippman, S. B. (1990) C++ Primer. Addison-Wesley Publishing Company.

Lohman, G., Lindsay, B., Pirahesh, H. & Bernhard Schiefer K. (1991). "Extensions to Starburst: Objects, Types, Functions, and Rules." Communications of the ACM. Vol. 34. No. 10. October 1991.

Meersman, R. A. & Sernadas, A. C. (editors). (1988). Data and Knowledge (DS-2). North-Holland.

Meyer, B. (1988). Object-Oriented Software Construction. Prentice-Hall International (UK) Ltd.

Piatetsky-Shapiro, G. & Frawley, W. J. (editors). (1991). Knowledge Discovery in Databases. The AAAI Press/The MIT Press.

Rich, E. & Knight, K. (1991) Artificial Intelligence. (2nd ed.). McGraw-Hill, Inc.

Rishe, N., Tai, D. & Li, Q. (1989). "Architecture for a Massively Parallel Database Machine." Microprocessing and Microprogramming. (Netherlands). Vol. 25. Iss. 1-5. January 1989.

Savnik, I. & Novac, F. (1989). "A Construction Database Model." Microprocessing and Microprogramming. (Netherlands). Vol. 27. Iss. 1-5. August 1989.

Sheu, P. C. (1989). "Describing Semantic Data Bases with Logic." The Journal of Systems and Software. Vol. 9. Iss. 1. January 1989.

Silberschatz, A., Stonebraker, M. & Ullman, J. (editors). (1991). "Database Systems: Achievements and Opportunities." Communications of the ACM. Vol. 34. No. 10. October 1991.

Stonebraker, M. & Kemnitz G. (1991). "The POSTGRES Next-Generation Database Management System." Communications of the ACM. Vol. 34. No. 10. October 1991.

Varvel, D. A. & Shapiro, L. (1989). "The Computational Completeness of Extended Database Query Languages." IEEE Transactions on Software Engineering. Vol. 15. No. 5. May 1989.

Zdonik, S. B. & Maier, D. (editors). (1990). Readings in Object-Oriented Database Systems. Morgan Kaufmann Publishers, Inc.

Appendix A

Advanced Sample Rule-Reduction Chains

1. {A} pant1.product_id(X) ∧ ¬(production1.product_id(X))
   ⇒ RejectValue(plant1.product_id)

   RULE: A/plant1
   SEARCH: production1.product_id = plant1.product_id
   TEST: ¬ε
   ACT: RejectValue(plant1.product_id)

2. {A} plant1.product_id(X) ∧ production1.product_id(X)
   ∧ production1.name(Y)
   ⇒ plant1.name(Y)

   RULE: A/plant1
   SEARCH: production1.product_id = plant1.product_id
   TEST: ε
   ACT: Set(plant1.name = production1.name)

3. {C} plant2.reactor_id(X) ∧ reactor_schedule.id(X)
   ∧ reactor_schedule.status('to-be-cleaned')
   ⇒ plant2.override('shut down')

   RULE: C/plant2
   SEARCH: reactor_schedule.id = plant2.reactor_id
   TEST: ε
   TEST: reactor_schedule.status = 'to-be-cleaned'
   ACT: Set(plant2.override = 'shut down')

4. {C} reactors.chemical_id(X) ∧ ¬ (chemicals.id(X))
   ⇒ Abort(reactors)

   RULE: C/reactors
   SEARCH: chemicals.id = reactors.chemical_id
   TEST: ¬ε
   ACT: Abort(reactors)

5. {C} plant1.product_id(*null*) ⇒ Abort(plant1)

RULE: C/plant1
TEST: plant1.product_id = <NULL>
ACT: Abort(plant1)

6. {C} plant1.product_id(X) ∧ production1.product_id(X) ∧
production1.serial_required('Y') ∧ plant1.serial(*null*)
⇒ AcquireValue(plant1.serial)

RULE: C/plant1
SEARCH: production1.product_id = plant1.product_id
TEST: ε
TEST: production1.serial_required = 'Y'
TEST: plant1.serial = <NULL>
ACT: AcquireValue(plant1.serial)

7. {C} plant1.chemical_id(X) ∧ chemicals.id(X) ∧
chemicals.volatile('Y')
⇒ plant1.special_handling('Y')

RULE: C/plant1
SEARCH: chemicals.id = plant1.chemical_id
TEST: ε
TEST: chemicals.volatile = 'Y'
ACT: Set(plant1.special_handling = 'Y')

8. {R} storage.chemical_id(X) ∧ chemical_removals.chemical_id(X)
∧ chemical_removals.instances(Y)
⇒ chemical_removals.instances(Y + 1) ∧
Update(chemical_removals)

RULE: D/storage
SEARCH: chemical_removals.chemical_id =
storage.chemical_id
TEST: ε
ACT: Set(chemical_removals.instances =
chemical_removals.instances +1)
ACT: Update(chemical_removals)

Appendix B

Advanced Sample Rule-Reduction Chains Used in SDBMS Prototype

100. {A} chemical.name[Δ](*null*) ∧ chemical.id(*null*) ∧
numbers.entity('chemical') ∧ numbers.number(X)
⇒ chemical.id(X+1) ∧ numbers.number(X+1) ∧
Update(numbers)

RULE: A/chemical.name
TEST: chemical.name[Δ] = *null*
TEST: chemical.id = *null*
SEARCH: numbers.entity = 'chemical'
TEST: ε
ACT: Set(chemical.id = numbers.number+1)
ACT: Set(numbers.number = numbers.number+1)
ACT: Update(numbers)

"The ID of a new chemical is acquired automatically by the system."

200. {A} ¬(chemical.id[Δ](*null*))
⇒ RejectValue(chemical.id)

RULE: A/chemical.id
TEST: chemical.id[Δ] ≠ *null*
ACT: RejectValue(chemical.id)

"Once the chemical ID is set, it may never change."

300. {A} chemical.id(X) ∧ chemical.name(*null*)
⇒ RejectValue(chemical.id)

RULE: A/chemical.id
TEST: chemical.name = *null*
ACT: RejectValue(chemical.id)

"A new chemical ID may not be set until the chemical NAME is set. The chemical ID is then automatically set by the system."

400. {A} chemical.type('explosive')
⇒ chemical.volatile('Y')

RULE: A/chemical.type
TEST: chemical.type = 'explosive'
ACT: Set(chemical.volatile = 'Y')

"Explosive-type chemicals are volatile."

450. {A} ¬(inproc.product_id[Δ](*null*))
⇒ RejectValue(inproc.product_id)

RULE: A/inproc.product_id
TEST: inproc.product_id[Δ] ≠ *null*
ACT: RejectValue(inproc.product_id)

"Once a product ID is assigned to an in-process record it may never change."

450. {A} ¬(inproc.product_id[Δ](*null*))
⇒ RejectValue(inproc.product_id)

RULE: A/inproc.product_id
TEST: inproc.product_id[Δ] ≠ *null*
ACT: RejectValue(inproc.product_id)

"Once a product ID is assigned to an in-process record it may never change."

475. {A} ¬(inproc.product_name[Δ](*null*))
⇒ RejectValue(inproc.product_name)

RULE: A/inproc.product_name
TEST: inproc.product_name[Δ] ≠ *null*
ACT: RejectValue(inproc.product_name)

"Once a product name is assigned to an in-process record it may never change."

500. {A} inproc.product_id(X) $\wedge$ $\neg$(products.id(X))
$\Rightarrow$ RejectValue(inproc.product_id)

RULE: A/inproc.product_id
SEARCH: products.id = inproc.product_id
TEST: $\neg\varepsilon$
ACT: RejectValue(inproc.product_id)

"A product ID which is referenced in the INPROC table must exist in the products table."

600. {A} inproc.product_id(X) $\wedge$ products.id(X) $\wedge$ products.name(Y)
$\Rightarrow$ inproc.product_name(Y)

RULE: A/inproc.product_id
SEARCH: products.id = inproc.product_id
TEST: ε
ACT: Set(inproc.product_name = products.name)

"A product ID implies a specific product name."

700. {C} inproc.product_id(*null*)
$\Rightarrow$ Abort(inproc)

RULE: C/inproc
TEST: inproc.product_id = *null*
ACT: Abort(inproc)

"The product ID field of an INPROC record must be non-null."

800. {C} inproc.product_id(X) $\wedge$ products.id(X) $\wedge$
products.authorization_required('Y') $\wedge$
inproc.authorization_number(*null*)
$\Rightarrow$ Abort(inproc)

RULE: C/inproc
SEARCH: products.id = inproc.product_id
TEST: ε
TEST: products.authorization_required = 'Y'
TEST: inproc.authorization_number = *null*
ACT: Abort(inproc)

"The defined product requires a valid authorization number."

900. {A} inprchem.chemical_id(X) ∧ chemical.id(X) ∧
chemical.volatile('Y')
⇒ inprchem.special_handling('Y')

RULE: A/inprchem.chemical_id
SEARCH: chemical.id = inprchem.chemical_id
TEST: chemical.volatile = 'Y'
ACT: inprchem.special_handling = 'Y'

"Volatile chemicals require special handling."

1000. {C} inprchem.chemical_id(*null*)
⇒ Abort(inprchem)

RULE: C/inprchem
TEST: inprchem.chemical_id = *null*
ACT: Abort(inprchem)

"The chemical ID field of an INPRCHEM record must be non-null."

1100. {C} inprchem.product_id(*null*)
⇒ Abort(inprchem)

RULE: C/inprchem
TEST: inprchem.product_id = *null*
ACT: Abort(inprchem)

"The product ID field of an INPRCHEM record must be non-null."

1200. {C} inprchem.serial_number(*null*)
⇒ Abort(inprchem)

RULE: C/inprchem
TEST: inprchem.serial_number = *null*
ACT: Abort(inprchem)

"The serial number field of an INPRCHEM record must be non-null."

1300. {A} products.name(X) $\wedge$ products.name[Δ](*null*) $\wedge$
products.id(*null*) $\wedge$ numbers.entity('product') $\wedge$
numbers.number(Y)
$\Rightarrow$ products.id(Y+1) $\wedge$ numbers.number(Y+1) $\wedge$
Update(numbers)

RULE: A/products.name
TEST: products.name[Δ] = *null*
TEST: products.id = *null*
SEARCH: numbers.entity = 'product'
TEST: ε
ACT: Set(products.id = numbers.number+1)
ACT: Set(numbers.number = numbers.number+1)
ACT: Update(numbers)

"Upon entry of a new product name, a new product ID is assigned automatically by the system."

1400. {A} $\neg$(products.id[Δ](*null*))
$\Rightarrow$ RejectValue(products.id)

RULE: A/products.id
TEST: products.id[Δ] $\neq$ *null*
ACT: RejectValue(products.id)

"Once a product ID is assigned it may never change."

1500. {A} products.id(X) $\wedge$ products.name(*null*)
$\Rightarrow$ RejectValue(products.id)

RULE: A/products.id
TEST: products.name = *null*
ACT: RejectValue(products.id)

"The product ID is assigned automatically by the system when the product name is entered."

1600. {A} products.type('explosive')
⇒ products.volatile('Y')

RULE: A/products.type
TEST: products.type = 'explosive'
ACT: Set(products.volatile = 'Y')

"Explosive-type products are volatile."

1700. {A} products.volatile('Y')
⇒ products.authorization_required('Y')

RULE: A/products.volatile
TEST: products.volatile = 'Y'
ACT: Set(products.authorization_required = 'Y')

"Volatile products require authorization."

1800. {C} inproc.serial_number(*null*)
⇒ Abort(inproc)

RULE: C/inproc
TEST: inproc.serial_number = *null*
ACT: Abort(inprchem)

"The serial number must be specified for a new INPROC record."

1850. {A} ¬(inprchem.product_id[Δ](*null*))
⇒ RejectValue(inprchem.product_id)

RULE: A/inprchem.product_id
TEST: inprchem.product_id[Δ] ≠ *null*
ACT: RejectValue(inprchem.product_id)

"Once a product ID is assigned to an in-process chemical record it may never change."

1860. {A} ¬(inprchem.serial_number[Δ](*null*))
⇒ RejectValue(inprchem. serial_number)

RULE: A/inprchem.serial_number
TEST: inprchem.serial_number[Δ] ≠ *null*
ACT: RejectValue(inprchem.serial_number)

"Once a serial number is assigned to an in-process chemical record it may never change."

1870. {A} ¬(inprchem.chemical_id[Δ](*null*))
⇒ RejectValue(inprchem.chemical_id)

RULE: A/inprchem.chemical_d
TEST: inprchem.chemical_id[Δ] ≠ *null*
ACT: RejectValue(inprchem.chemical_id)

"Once a chemical ID is assigned to an in-process chemical record it may never change."

1900. {A} inprchem.product_id(X) ∧ ¬(products.id(X))
⇒ RejectValue(inprchem.product_id)

RULE: A/inprchem.product_id
SEARCH: products.id = inprchem.product_id
TEST: ¬ε
ACT: RejectValue(inprchem.product_id)

"The product ID must exist in the products database."

2000. {A} inprchem.chemical_id(X) ∧ ¬(chemicals.id(X))
⇒ RejectValue(inprchem.chemical_id)

RULE: A/inprchem.chemical_id
SEARCH: chemical.id = inprchem.chemical_id
TEST: ¬ε
ACT: RejectValue(inprchem.chemical_id)

"The chemical ID must exist in the chemical database."

2100. {A} recipe.product_id(X) $\wedge$ $\neg$(products.id(X))
$\Rightarrow$ RejectValue(recipe.product_id)

RULE: A/recipe.product_id
SEARCH: products.id = recipe.product_id
TEST: $\neg\varepsilon$
ACT: RejectValue(recipe.product_id)

"A product ID which is referenced in the recipe database must exist in the products database."

2200. {A} recipe.chemical_id(X) $\wedge$ $\neg$(chemical.id(X))
$\Rightarrow$ RejectValue(recipe.chemical_id)

RULE: A/recipe.chemical_id
SEARCH: chemical.id = recipe.chemical_id
TEST: $\neg\varepsilon$
ACT: RejectValue(recipe.chemical_id)

"A chemical ID which is referenced in the recipe database must exist in the chemical database."

2300. {C} recipe.product_id(*null*)
$\Rightarrow$ Abort(recipe)

RULE: C/recipe
TEST: recipe.product_id = *null*
ACT: Abort(recipe)

"The product ID field of a recipe record must be non-null."

2400. {C} recipe.chemical_id(*null*)
$\Rightarrow$ Abort(recipe)

RULE: C/recipe
TEST: recipe.chemical_id = *null*
ACT: Abort(recipe)

"The chemical ID field of a recipe record must be non-null."

2500. {C} recipe.chemical_id(X) ∧ chemical.id(X) ∧
chemical.volatile('Y') ∧ recipe.product_id(Y) ∧
products.id(Y) ∧ ¬(products.volatile('Y'))
⇒ products.volatile('Y') ∧ Update(products)

RULE: C/recipe
SEARCH: chemical.id = recipe.chemical_id
TEST: ε
TEST: chemical.volatile = 'Y'
SEARCH: products.id = recipe.product_id
TEST: ε
TEST: products.volatile ≠ 'Y'
ACT: Set(products.volatile = 'Y')
ACT: Update(products)

"The inclusion of one or more volatile chemicals in a product's recipe makes that product volatile."

2600. {C} products.volatile('N') ∧ products.id(X) ∧ recipe.product_id(X) ∧
recipe.chemical_id(Y) ∧ chemical.id(Y) ∧ chemical.volatile('Y')
⇒ products.volatile('Y')

RULE: C/products
TEST: products.volatile = 'N'
SEARCH: recipe.product_id = products.id
TEST: ε
SEARCH: chemical.id = recipe.chemical_id
TEST: ε
TEST: chemical.volatile('Y')
ACT: Set(products.volatile = 'Y')

"The inclusion of one or more volatile chemicals in a product's recipe makes that product volatile."

2700. {C} inproc.product_id[Δ](*null*) ∧ inproc.product_id(X) ∧
recipe.product_id(X) ∧ recipe.chemical_id(Y) ∧
¬(holding.chemical_id(Y))
⇒ Abort(inproc)

RULE: C/inproc
TEST: inproc.product_id[Δ] = *null*
SEARCH: recipe.product_id = inproc.product_id
TEST: ε
SEARCH: holding.chemical_id = recipe.chemical_id
TEST: ¬ε
ACT: Abort(inproc)

"One or more of the chemicals required by the product's recipe is not in stock--product cannot be produced. Therefore, this product may not be dedicated to the in-process list at this time."

2800. {C} inproc.product_id[Δ](*null*) ∧ inproc.product_id(X) ∧
recipe.product_id(X) ∧ recipe.chemical_id(Y) ∧
recipe.quantity(Z) ∧ holding.chemical_id(Y) ∧
holding.quantity(<Z)
⇒ Abort(inproc)

RULE: C/inproc
TEST: inproc.product_id[Δ] = *null*
SEARCH: recipe.product_id = inproc.product_id
TEST: ε
SEARCH: holding.chemical_id = recipe.chemical_id
TEST: ε
TEST: holding.quantity < recipe.quantity
ACT: Abort(inproc)

"There is insufficient quantity of one or more of the chemicals required by the product's recipe--product cannot be produced. Therefore, the product can not be dedicated to the in-process list at this time."

2900. {C} inproc.product_id[Δ](*null*) ∧ inproc.product_id(X) ∧
recipe.product_id(X) ∧ recipe.chemical_id(Y) ∧
recipe.quantity(Z) ∧ holding.chemical_id(Y) ∧
holding.quantity(A) ∧ inproc.serial_number(B)
⇒ holding.quantity(A-Z) ∧ Update(holding) ∧
NewRecord(inprchem) ∧ inprchem.product_id(X) ∧
inprchem.serial_number(B) ∧
inprchem.chemical_id(Y) ∧ inprchem.quantity(Z) ∧
Insert(inprchem)

RULE: C/inproc
TEST: inproc.product_id[Δ] = *null*
SEARCH: recipe.product_id = inproc.product_id
TEST: ε
SEARCH: holding.chemical_id = recipe.chemical_id
TEST: ε
ACT: Set(holding.quantity = holding.quantity-recipe.quantity)
ACT: Update(holding)
ACT: NewRecord(inprchem)
ACT: Set(inprchem.product_id = inproc.product_id)
ACT: Set(inprchem.serial_number = inproc.serial_number)
ACT: Set(inprchem.chemical_id = recipe.chemical_id)
ACT: Set(inprchem.quantity = recipe.quantity)
ACT: Insert(inprchem)

"The inventory quantities of chemicals required by the product's recipe have been updated accordingly."

3000. {A} ¬(holding.chemical_id[Δ](*null*))
⇒ RejectValue(holding.chemical_id)

RULE: A/holding.chemical_id
TEST: holding.chemical_id[Δ] ≠ *null*
ACT: RejectValue(holding.chemical_id)

"Once a chemical ID is assigned it may never change."

3100. {A} holding.chemical_id(X) $\wedge$ $\neg$(chemicals.id(X))
$\Rightarrow$ RejectValue(holding.chemical_id)

RULE: A/holding.chemical_id
SEARCH: chemical.id = holding.chemical_id
TEST: $\neg\varepsilon$
ACT: RejectValue(holding.chemical_id)

"The chemical ID must exist in the chemical database."

3200. {A} $\neg$(holding.chemical_name[Δ](*null*))
$\Rightarrow$ RejectValue(holding.chemical_name)

RULE: A/holding.chemical_name
TEST: holding.chemical_name[Δ] $\neq$ *null*
ACT: RejectValue(holding.chemical_name)

"Once a chemical name is assigned to a holding record it may never change."

3300. {A} holding.chemical_id(X) $\wedge$ chemical.id(X) $\wedge$ chemical.name(Y)
$\Rightarrow$ holding.chemical_name(Y)

RULE: A/holding.chemical_id
SEARCH: chemical.id = holding.chemical_id
TEST: ε
ACT: Set(holding.chemical_name = chemical.name)

"A chemical ID implies a specific chemical name."

3400. {C} holding.chemical_id(*null*)
$\Rightarrow$ Abort(holding)

RULE: C/holding
TEST: holding.chemical_id = *null*
ACT: Abort(holding)

"The chemical ID field of a HOLDING record must be non-null."

3500. {C} holding.quantity(*null*)
⇒ Abort(holding)

RULE: C/holding
TEST: holding.quantity = *null*
ACT: Abort(holding)

"The quantity field of a HOLDING record must be a real number."

3600. {A} ¬(inproc.serial_number[Δ](*null*))
⇒ RejectValue(inproc.serial_number)

RULE: A/ inproc.serial_number
TEST: inproc.serial_number[Δ] ≠ *null*
ACT: RejectValue(inproc.serial_number)

"Once a serial number is assigned it may never change."

VITA

Mr. Yurchak was born in the city of Bethlehem, Pennsylvania, on December 15, 1968, to the parents Joleita W. and W. Russell Yurchak. He attended Lehigh University, graduating with High Honors, and was awarded the Bachelor of Science degree in Computer Science in June, 1990. He based his studies heavily on computer science, cognitive science, and psychology. During that time Mr. Yurchak obtained summer internships in the Knowledge Based Systems division of Air Products and Chemicals, Inc., of Trexlertown, Pennsylvania. He was involved in the research and development of several expert systems--one of which, a prototype expert system designed to ascertain and schedule the daily production of a chemical plant, resulted from private research sponsored by Air Products and conducted at Lehigh University as Mr. Yurchak's Senior Year Project.

Mr. Yurchak continued his education at Lehigh University engaging in graduate studies in artificial intelligence, database management systems, and advanced software engineering. He was awarded the Master of Science degree in Computer Science in May, 1992. During this time Mr. Yurchak worked as a research assistant sponsored by the Ben Franklin Technology Center and other private industries, researching and developing a Knowledge-Based Decision Support System which linked conventional relational databases, Computer-Aided Design graphical information, and various data-analysis packages together via expert systems technology to provide underground infrastructure assessment, facilities management, and environmental impact assessment. Work proceeded into the development of a *onecall* underground infrastructure assessment system for the pharmaceutical company, Merck, Sharpe, and Dohme.

Mr. Yurchak is currently Senior Vice President of Origination Alternatives, Inc. (OAI), based in Marlton, New Jersey, a financial services company. He is head of development for OAI and involved with engineering specialized proprietary mortgage origination software spanning the qualification, application, processing, tracking, and closing of residential loans. Mr. Yurchak is also currently consultant to a pharmaceutical returns company, Rx Returns, Inc., of Palm, Pennsylvania. He is engaged in the engineering of customized software to fully automate the company's warehouse through use of an expert database system. He is currently pursuing a Doctor of Philosophy degree in Computer Science at Lehigh University and expects his degree in October, 1994.