1988

Semantic Constraint Modeling in Database Using the Applicative Data Language.

Herman Hai-lou Lee
Louisiana State University and Agricultural & Mechanical College

Order Number 8819955

Semantic constraint modeling in database using the applicative data language

Lee, Herman Hai-Lou, Ph.D.

The Louisiana State University and Agricultural and Mechanical Col., 1988

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

PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark __✓__.

1. Glossy photographs or pages _____
2. Colored illustrations, paper or print _____
3. Photographs with dark background _____
4. Illustrations are poor copy _____
5. Pages with black marks, not original copy __✓__
6. Print shows through as there is text on both sides of page _____
7. Indistinct, broken or small print on several pages _____
8. Print exceeds margin requirements _____
9. Tightly bound copy with print lost in spine _____
10. Computer printout pages with indistinct print _____
11. Page(s) _____ lacking when material received, and not available from school or author.
12. Page(s) _____ seem to be missing in numbering only as text follows.
13. Two pages numbered _____. Text follows.
14. Curling and wrinkled pages _____
15. Dissertation contains pages with print at a slant, filmed as received _____
16. Other _____

_____

_____

SEMANTIC CONSTRAINT MODELING IN DATABASE USING THE APPLICATIVE DATA LANGUAGE

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirement for the degree of
Doctor of Philosophy

in

The Department of Computer Science

By
Herman Hai-Lou Lee
B.S., National Taiwan University, 1976
M.S., West Virginia Universtiy, 1980
May, 1988

This work is dedicated to the memory of my father, Wei-Chi Lee, and to my mother, Shan-Fei Wu Lee, who always believed that I could achieve any goal I set. And it is dedicated to my wife, Betty Wen-Bey Lee, and my son, Michael, who shared the bright and the dark moments.

ACKNOWLEDGMENTS

I wish to express my gratitude to Prof. Peter P.S. Chen for his advise and support to this work.

Special thanks are due to Prof. John A. Brewer, my minor professor in computer graphics who introduced me to the bright frontier of computer graphics and CAD/CAM; to Mr. S. Y. Zhu for many of our heated discussions on the mathematical side of this work; to Prof. Leslie Jones, who gave me advise on compound objects and non-first order normal forms.

I gratefully acknowledge the presence and comments of Prof. Donald Kraft, Prof. Ian Akyildiz, Prof. Sukhamay Kundu, Prof. Bush Jones, Prof. Kenneth Reid, and Prof. George Tracy, all who serve on my committee.

I wish to express my sincerely appreciation to Mr. A. Y. Yang who spent so much time correcting my mistakes and making the whole work near perfect.

I wish to thank Mr. Mounir Bsaibis and fellow hackers in Room 256 for many interesting conversations on LISP programming and "dirty" tricks on Unix.

And thank to modern technologies that put a PC on everyone's desk; it becomes a powerhouse in the proper hands.

I like to express sincere appreciation to my family for support and encouragement, especially to my wife and son, for their patience and understanding.

Herman Hai-Lou Lee
June 30, 1987.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

---> : mapping to.

<===> or ≡ : equivalent.

===> : implies.

$A,B,C,...,R_{AB},R_{AC},...$: base states (Italic Letters).

ADL: Applicative Data Language.

$A_i,A_1,A_2,...A_m$: attributes of entity sets.

Assign_plus: function for incremental computation, ς_2':= **Assign_plus**(ς_1).

Assign_minus: function for incremental computation, ς_2':= **Assign_minu**(ς_1).

C(N,E): the transition digraph of constraint C with node set N, edge set E.

CountS: entity space value function, μ:= **CountS**(ς).

DBST: Database State, all the data stored in the database.

Display: general output function, **Display**(ς, τ, μ, or β).

Dom(A_i),..,Dom(F_j),..: domains of entity attribute A_i or d_relaiton attribute F_j.

$e,e_i,e_1,..,e_n$: entities in an entity set.

eid_i:entity id number, the primary key of entity e_i.

E: entity sets.

|E| number of entities in an entity set.

$(E_{1,1} \times ... \times E_{1,q1})^*$: a compound dimension in a hyper entity space.

EID: the primary key of an entity set.

ERD: Entity Relationship Diagram.

ERM: Entity Relationship Model.

ExtendD: data space function, τ_2:= **ExtendD**(τ_1).

Extract: data type transformation function,τ:=**Extract**(ς).

F_1,F_j,F_p: attributes of d_relations.

FD: functional dependency.

FOL: First Order Logic.

G_set: the set of entity-set symbols in the **Group_byS** command. It cannot be implied and must be specified as part of the command.

Group_byS: entity space function, ς_2:= **Group_byS**(ς_1).

J_set: the set of symbols of entity sets in the **JoinS** command. For the **JoinS** operator, J_set is implied and equals to Sch(Ψ_1) ∩ Sch(Ψ_2).

JoinS: relation space function, ς_3:= **JoinS**(ς_1, ς_2). A ° B ° C: shorthand form of **JoinS**.

LBASE: An Entity Relationship Database Implemented in LISP.

Minus_part: function for incremental computation, ς_2':= **Minus_part**(ς_1').

Order(Ψ): total number of entity sets in Ψ.

P_set: the set of symbols of entity sets in the **ProjectS**, **ProjectD** commands. It cannot be implied and must be specified as part of the commands.

Plus_part: function for incremental computation, ς_2':= **Plus_part**(ς_1').

Power(Ψ): power set of Ψ.

ProductD: data space function, τ_2:= **ProductD**(τ_1).

ProjectS: entity space function, ς_2:= **ProjectS**(ς_1).

ProjectD: data space function, τ_2:= **ProjectD**(τ_1).

$\mathbf{PD}_{\leq}$,$\mathbf{PD}_{\geq}$,$\mathbf{PD}_{<}$,$\mathbf{PD}_{>}$,$\mathbf{PD}_{=}$: the d_relation predicates or the And_all predicates.

$\mathbf{PV}_{\leq}$,$\mathbf{PV}_{\geq}$,$\mathbf{PV}_{<}$,$\mathbf{PV}_{>}$,$\mathbf{PV}_{=}$: the value predicates.

$\mathbf{PS}_{\subseteq}$,$\mathbf{PS}_{\supseteq}$,$\mathbf{PS}_{\subset}$,$\mathbf{PS}_{\supset}$,$\mathbf{PS}_{=}$: the relation state predicates.

$r, r_i, r_1, \ldots, r_n$: relationship in a relationship set.

R: relationship sets.

$\|R\|$ number of relationships in a relationship set.

$s, s_i, s_1, \ldots, s_n$: entity vectors.

Sch(E): scheme of E, the set of attribute symbols.

Sch(R): scheme of R, the set of entity set symbols.

Sch(Ψ): scheme of Ψ, the set of entity space dimension symbols.

Sch(τ): scheme of τ, the set of d_relation attribute symbols.

SchE(Ψ): scheme of Ψ , which only includes all the singular dimension (primary entity set) symbols.

SelectS: entity space function, ς_2:= **SelectS**(ς_1). $(A)_{Predicate}$: shorthand form of **SelectS**.

SelectD: data space function, τ_2:= **SelectD**(ς_1).

SelectSS: entity space function, ς_3:= $\mathbf{SelectSS}_{\varsigma 2}(\varsigma_1)$.

Sign: function for incremental computation, ς_3':= $\mathbf{Sign}_{\varsigma 2'}(\varsigma_1)$.

Sign_change: function for incremental computation, ς_2':= **Sign_change**(ς_1').

SumD, **CountD**, **AvgD**, **MinD**, **MaxD**, **ValueD**: data space value functions, μ:= **SumD**(τ).

SumB, **CountB**, **AvgB**, **MinB**, **MaxB**: bag function that takes a bag as input and returns a single value.

$t, t_i, t_1, t_2, \ldots$: tuple in a d_relation.

Trace_bk: meta function that modify evaluation strategy (or transition digraph).

Unsign: function for incremental computation, ς_2:= **Unsign**(ς_1').

Ungroup: relation space function, ς_2:= **Ungroup**(ς_1).

$v_{i1}, \ldots, v_{ip}$: attribute values.

X': If X is a data structure, X' is the incremental data structures (ς' or τ'). If X is an operator, X' is the incremental entity space or data space operator.

X: a data structure in the current database state (after the update), X can be ς or τ.

$X^{\sim}$: a data structure in the previous database state (before the update), $X^{\sim}$ can be $\varsigma^{\sim}$ or $\tau^{\sim}$.

$\cup$, $\cap$, -: entity space function, state3:= f(state1, state2).

$\oplus$s: adding an entity vector to an old relation state.

$\oplus$t: adding a tuple to an old d_relation.

$\ominus$s: deleting an entity vector from an old relation state.

$\ominus$t: deleting a tuple from an old d_relation.

α: the realization function, $\varsigma := \varsigma' \ \alpha \ \varsigma'$ or $\tau := \tau' \ \alpha \ \tau'$.

$\Psi, \Psi_1, \Psi_2, \ldots, \Psi_n$: entity spaces.

$|\Psi|$ number of entity vectors in the entity space.

$\varsigma, \varsigma_1, \varsigma_2, \ldots, \varsigma_n$: entity states. $\varsigma \in \text{Power}(\Psi)$, or $\varsigma \subseteq \Psi$.

$|\varsigma|$ number of entity vectors in the relation state.

$\tau, \tau_1, \tau_2, \ldots, \tau_n$: d_relations.

$\mu, \mu_1, \mu_2, \ldots, \mu_n$: values.

$\beta, \beta_1, \beta_2, \ldots, \beta_n$: booleans.

θ: $<$, $>$, $\leq$, $\geq$, $=$, $<>$, value comparison.

δ: $\subseteq$, $\supseteq$, $\subset$, $\supset$, $=$s , set comparison.

ABSTRACT

There is a growing need to incorporate database integrity subsystems into large information systems in engineering design environments and real-time control and monitoring environments. The objectives of the integrity subsystem are to provide a user interface for constraint specification, to compile the specification into enforcement strategies, and to check data integrity at both compile-time and run-time.

The approach proposed by this research is to develop the conceptual view of the database using the Entity Relationship Model (ERM). Users' queries and semantic constraints can be specified by an ER-based data language, the Applicative Data Language (ADL). Any ADL constraint specification is compiled into both a compile-time and a run-time checking strategy for enforcement. The integrity subsystem, then, automatically maintains the consistency of data whenever there is a change in the database state.

The basic constructs of ADL are data structures, functions, and predicates. It takes advantage of the semantic clarification of objects and relationships in the Entity Relationship Model by doing, first, an object level computation and , then, a data element level computation. The object level computation determines how objects are associated with each other. The data element computation, on the other hand, examines the data values of those associated objects and derives new relations from these values. A semantic constraint, therefore, is formulated as a computation procedure that maps the current database state to a TRUE or FALSE value.

The computational syntax of ADL allows us to compile directly each constraint specification into a transition digraph for compile-time constraint checking. This research proposes the incremental computation strategy for efficient run-time constraint checking. The objective of the strategy is to do run-time constraint checking without full evaluation of the database. The entire computation procedure centers around the user's update. It propagates the incremental changes along the transition digraph to infer the effect of the update upon the new truth value of the semantic constraint.

This research concludes that ADL with its generality in semantic constraint modeling and its enforcement strategies at both compile-time and run-time is adequate as the architecture for an integrity subsystem supporting an Entity Relationship database.

CHAPTER I. INTRODUCTION

We are moving toward an information society, as John Naisbitt pointed in his best seller, "Megatrend". More and more businesses and industries regard information as one of their valuable assets and resources. How to effectively manage them and generate successful decisions has a dramatic impact on the productivity and profitability of the corporation.

We need information to make intelligent decisions. Production plants collect output data to control inventories. Engineers take a customer's specification to generate a product design. Banks collect security data to make investment decisions. These are typical examples in that we take in raw data to generate more useful data for our decision making.

The massive amount of data needs to be managed and stored in a database. We can structure our data according to a data model that the database supports so that people can share the same information resources. But with computers we can generate data far faster than we can by hand. Without proper control and management of the data that we create, we are heading for disaster. This is because merely storing data in a structured way does not always guarantee its correctness. If we use the data in a computation algorithm for decision making, no matter how accurate or sound the computation algorithm is, the validity of its output always depends on the correctness of its input.

This naturally leads us to the topic of this research, i.e. database integrity. Integrity in a database system can be broken down into several different levels. The domain integrity concerns the domain values of a particular attribute. For example, a person's age has to be between 0 and 200. Social security numbers have to be nine digits. Many existing database systems have the ability of maintaining domain integrities [Date83, Fern81].

The second level of integrity in a database system is the data model integrity. For example, in the relational data model, we can impose data dependencies and multi-value dependencies as constraints on the data. In the Entity Relationship Model (ERM), we can specify cardinalities of the involving entity sets with respect to a relationship set [Chen76,Chen77]. Many relational database systems provide support for data dependency constraints.

Finally, the semantic integrities (or constraints) represent more general constraints on data governed by the semantics of the enterprise. Examples of semantic constraints are given below:

In A Company Environment,

- Employees whose ages are below 30 should not exceed 20% of the total employees.
- Employees must earn less than 10 times the sale volume of their department if their department has a positive sale.

- Each manager must earn more than the average salary in his department.

In An Engineering Environment,

- A recent soil test at plant site B indicates that the load factor cannot exceed 30,000 lb/ft^2 at the foundation of each separation unit in the area.
- Due to space limitation at CHCl3 plant, the number of distillation towers, exchangers, or pumps cannot exceed 2 in each separation unit in the plant.
- Every distillation tower in the Ethylene plant should have at least a reboiler at the bottom and a condenser at the top.

Not many database systems support this type of integrities [Date83]. However, this research proposes a mechanism to model the semantic constraint based on the Entity Relationship Model.

1. CASE STUDIES

To better understand problems of database management and semantic constraint modeling, I had the opportunity to visit several companies that either develop large information systems to facilitate their day-to-day operations or sell these systems as their products. By analyzing the problems confronting their information systems, we can see the important implications of semantic constraint modeling in future database system design.

Case 1. A Communication Network Monitoring System

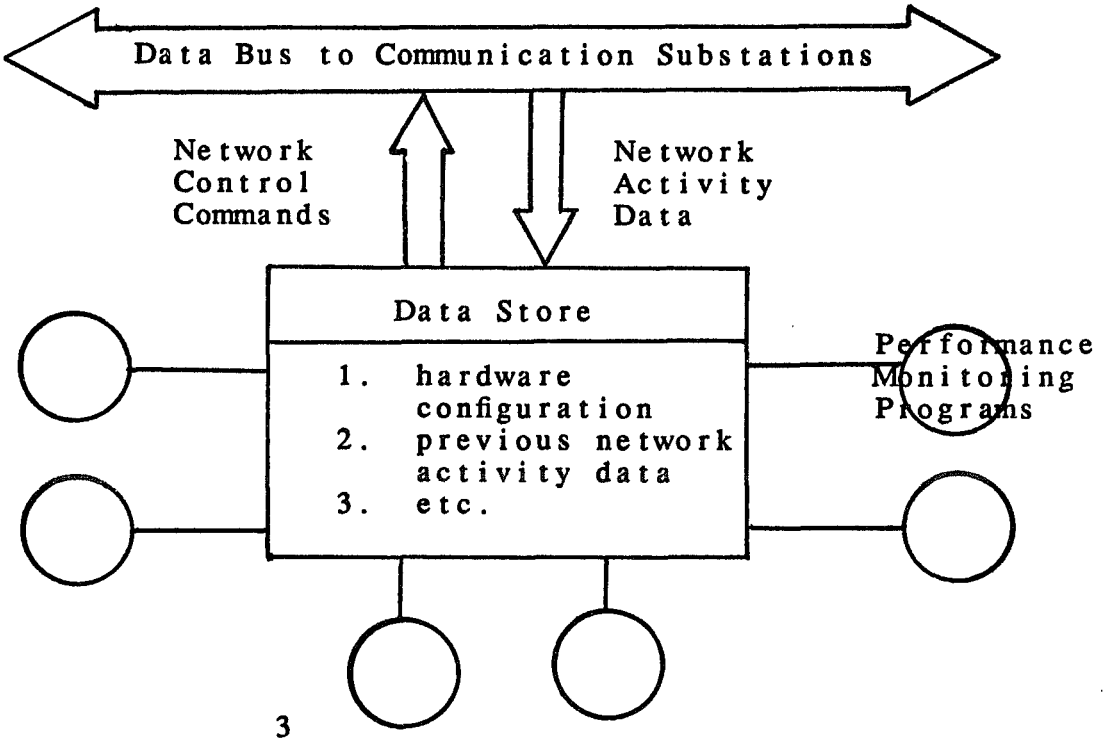
A company is engaged in developing a large telecommunication network monitoring system. The giant network covering the entire North American continent can transmit both digital and analog signals by various physical carriers. Each node on the network represents a communication substation that has a variety of communication switching systems and physical linkages. The configuration of the monitoring system is shown in Figure 1.1.1.

The objective of the monitoring system is to balance the work loads on various signal carriers and routing alternatives, and to maintain the communication capability at a certain level. Large amounts of network performance data are logged into the database at regular intervals. Each network performance evaluation program monitors a specific feature of the network. If there is an indication that the performance of a network node starts to degrade, appropriate actions will be taken.

Case 2. A C^3 I Electronic Warfare System

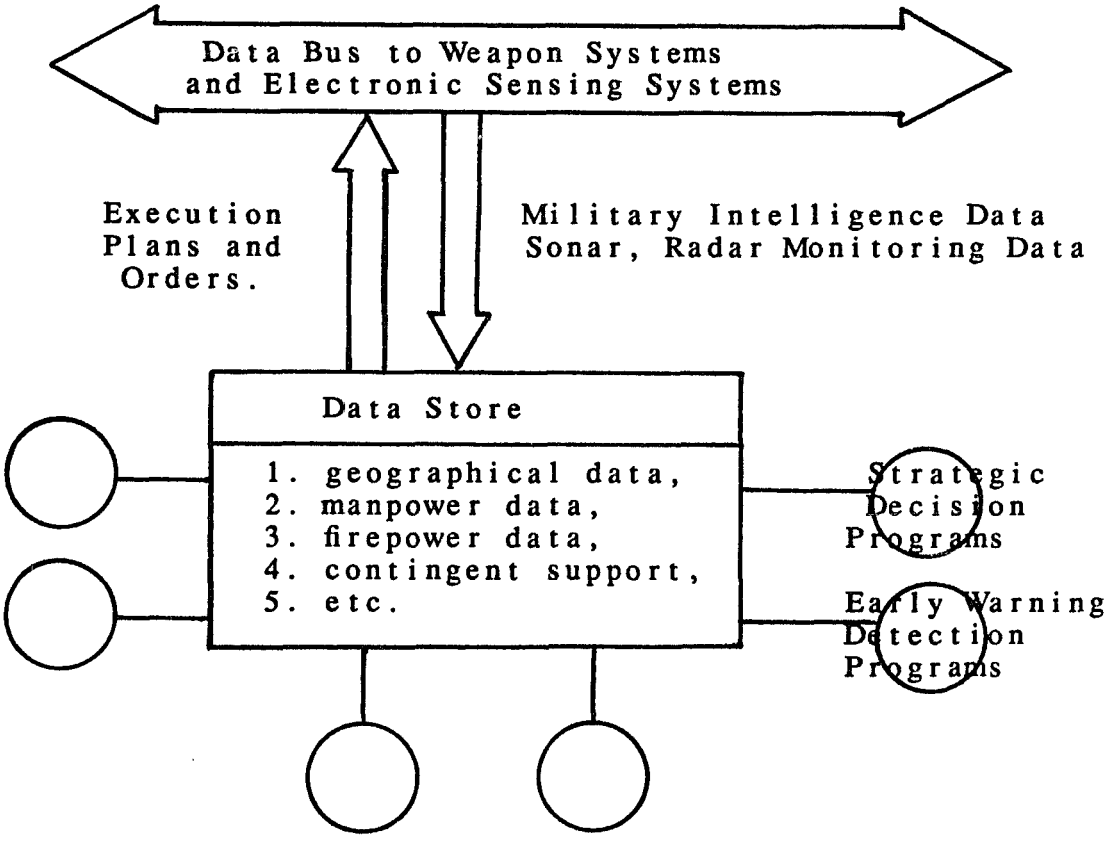
Many defense contractors are building military AI application systems for battle field management. It is known as the C^3I system (for command, control and communication) in the defense establishment. [Andr85, Baci85] A large database system (Figure 1.1.2) is typical in all these applications for storing

Figure 1.1.1. A Communication Network Monitoring System



3

Figure 1.1.2. A C I Electronic Warfare System



geographical data, personnel data, firepower data, and other strategic information. During battle engagement, an enormous amount of information coming from various sources tend to saturate the communication channels. It poses a real challenge to the information system to absorb, assimilate and quickly analyze the flood of data required for intelligent response and action.

Case 3. A Real-time Process Control System

A large oil refinery company is interested in developing a real-time process control expert system. [Moor84, Moor86, Arze86] In a typical process plant, there are several thousand measurements and alarms sent to the real-time control system at some regular intervals (Figure 1.1.3). The key to the success of such a real-time control system is to quickly identify process conditions which are potentially significant, and to invoke relevant course of actions to correct the situation. It is a typical pattern matching problem to identify trends and upsets in process conditions.

Case 4. An Engineering Design Information System

The use of computer in engineering design has gradually shifted from numerical simulations and design calculations to design information management in recent years [Jone83, Nava83, Mait85]. However, an adequate data model to model all the design data and activity is still being sought [Blah85, Canf85]. An integrated computer aided design (CAD) system must be able to manage various kinds of design data which are organized into design calculations, drawings, bills-of-material, and etc. (Figure 1.1.4) These data usually embody complex structures, tangled inter-relationships, and therefore cannot easily be modeled by traditional data models [Stal86], [Murt85, Boer85]. Many database design techniques that support business data processing are not adequate for the design of engineering databases, for example transaction management, and constraint checking [Boer85, East81].

One of the critical problems facing CAD databases is data integrity, because a design usually involves many engineers from quite different disciplines. A VLSI design team consists of system engineers working on the functionality of different modules, logic design engineers designing the logical gate components, and layout engineers putting together physical components on a PCB. A chemical plant design will involve process engineers defining the process equipment specifications, project engineers designing the mechanical details of the equipment, and civil engineers building the necessary foundations for the equipment. The problem is how to check consistencies among data across different engineering disciplines.

Figure 1.1.3. A Real-time Process Control System

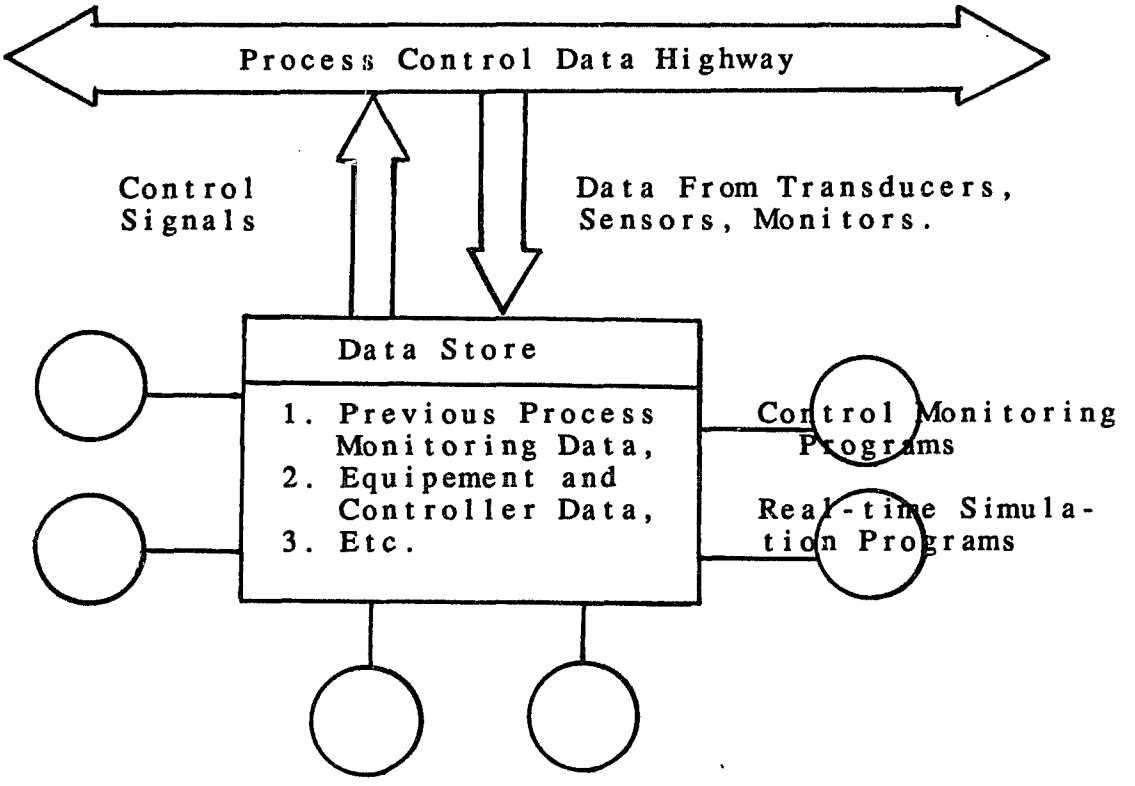
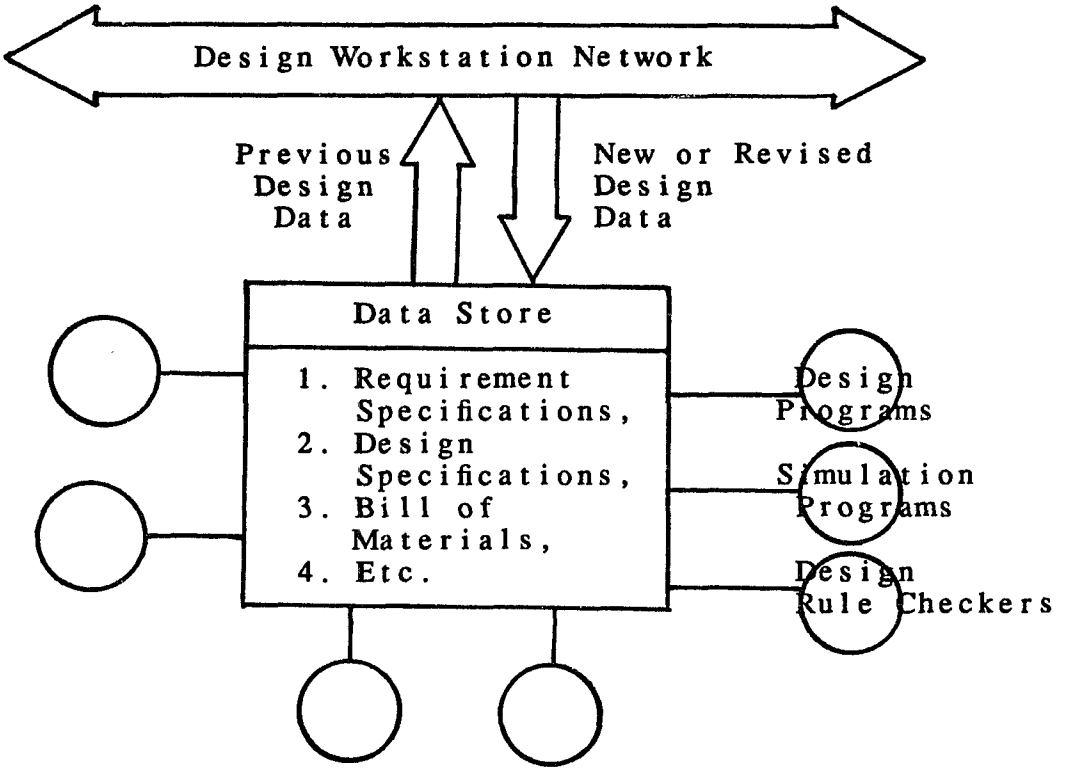


Figure 1.1.4. An Engineering Design Information System



2. GENERAL APPROACH TO SOLVE THE PROBLEM

Analyzing the above four large information systems, we can see there are some common problems confronting every one of them:

(i) Large amounts of data need to be organized and managed. Some of them have used commercially available database systems (DBMS) in their implementations. Some of them , unfortunately, still rely on old file management systems to manage their data. Those that do use a commercially available DBMS usually are not satisfied with the DBMS's run-time performance.

(ii) There is no common representation for data in the data store. None of them have built their databases according to the ANSI/X3/SPARC database architecture [Chen77]. When it comes to interfacing the application programs with the database, the physical file structures of records and fields are given to the application programmer. If the physical file structures change due to efficiency considerations, the data access part of the application program has to be revised.

(iii). There is little or no control at all of how the application programs may access or modify data in the database. Essentially, every application program can access and modify data in the database. There is no global coordination or control over the use or modification of data. If one "rogue" program tampers with the data, the whole database may be corrupted to such an extent that all the other application programs are rendered useless because of erroneous input data.

(iv). There is no common interface between application programs and the database. Each application program writes its own access procedures even though many of these codes are duplicates of others.

(v). One of the more critical issues of the first three cases we just discussed is the real-time performance of the information system. Large amounts of data pour in from various sources at regular intervals. All these data have to be assimilated, analyzed, and stored quickly for intelligent response and action. Efficiency hinges on two factors: first, old data has to be accessed quickly from the database and sent to the application programs; second, the application programs must detect patterns quickly by comparing the old data with the incoming data.

We believe all the above problems can be solved quite well from the database perspective. To tackle the above problems, first, we have to realize that the definition of semantic constraints covers a much broader spectrum than those often mentioned in the database literature. Almost all the application programs surrounding the database can be generally divided into two categories.

One category of application programs are those that generate new data from the raw data, e.g. Z1:= F1(X,Y). X and Y may be incoming data elements from outside information sources and Z1 is the data element that we are really interested in. Most likely, the X,Y,Z1 tuple has to be stored in the database, so there is actually a constraint on X,Y,Z1 of the form C1(X,Y,Z1). But another

application program may apply a totally different algorithm Z1':= F2(W,V) and modify Z1 to Z1'. Now, C1(X,Y,Z1') is no longer valid. But the database without an integrity subsystem may never be aware of this fact.

The other category of application programs are those that perform pattern matching; that is, if P(X,Y,Z), then take Action1. X and Y may be data elements already in the database and Z is the incoming data. Interestingly enough, P(X,Y,Z) is actually an assertion on the data in the database. If the assertion is true, a certain course of action will be taken. Needless to say, P(X,Y,Z) can be regarded as a semantic constraint in the database. The problem of pattern matching now becomes the problem of semantic constraint modeling in the database. Therefore, we can see that most of the application programs can be modeled as semantic constraints on data in the database.

We propose the following approach (Figure 1.2.1, Figure 1.2.2) to tackle the problems confronting these large information systems:

(i) Develop a high level conceptual view of the information system. We propose the Entity Relationship Model to represent the conceptual view of the database. [Chen83, Davi83]

(ii) Develop a data language that is based on the Entity Relationship Model. We propose the Applicative Data Language (ADL) that can be used to define semantic constraints, to query the current database, and to interface with application programs.

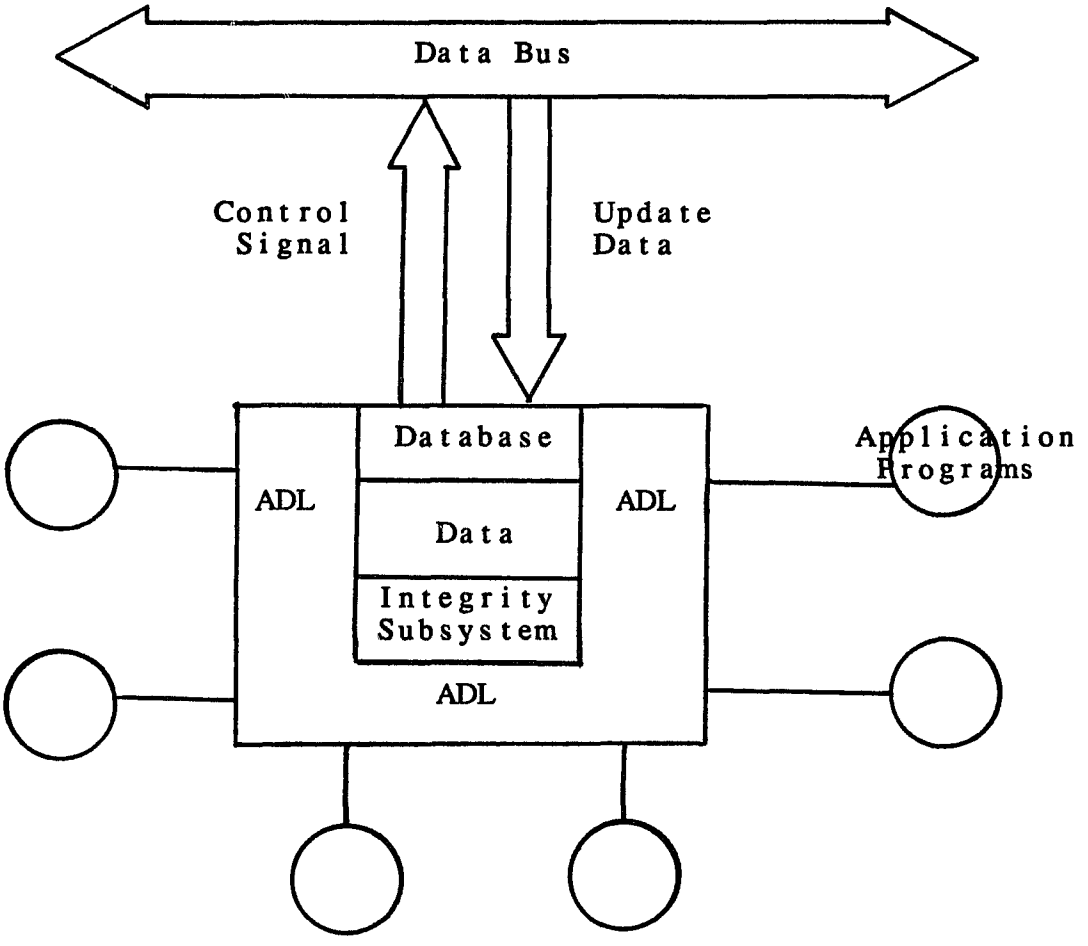
(iii) Develop constraint enforcement strategies to maintain the integrity of the database. Here, we propose the compile-time checking strategy and the run-time checking strategy of ADL. The checking of semantic constraints, thus, becomes automatic.

(iv) Map the Entity Relationship Model to the physical file structures [Chen77] and the ADL operators to physical access procedures. These mappings will accommodate the fact that the information system may consist of varieties of hardware configurations and physical database implementations. In such a distributed information environment, each one of the hardware and software configurations has only to map to one uniform representation. We implement ADL and its integrity subsystem on top of an Entity Relationship based database system (LBASE).

3. LITERATURE SURVEY

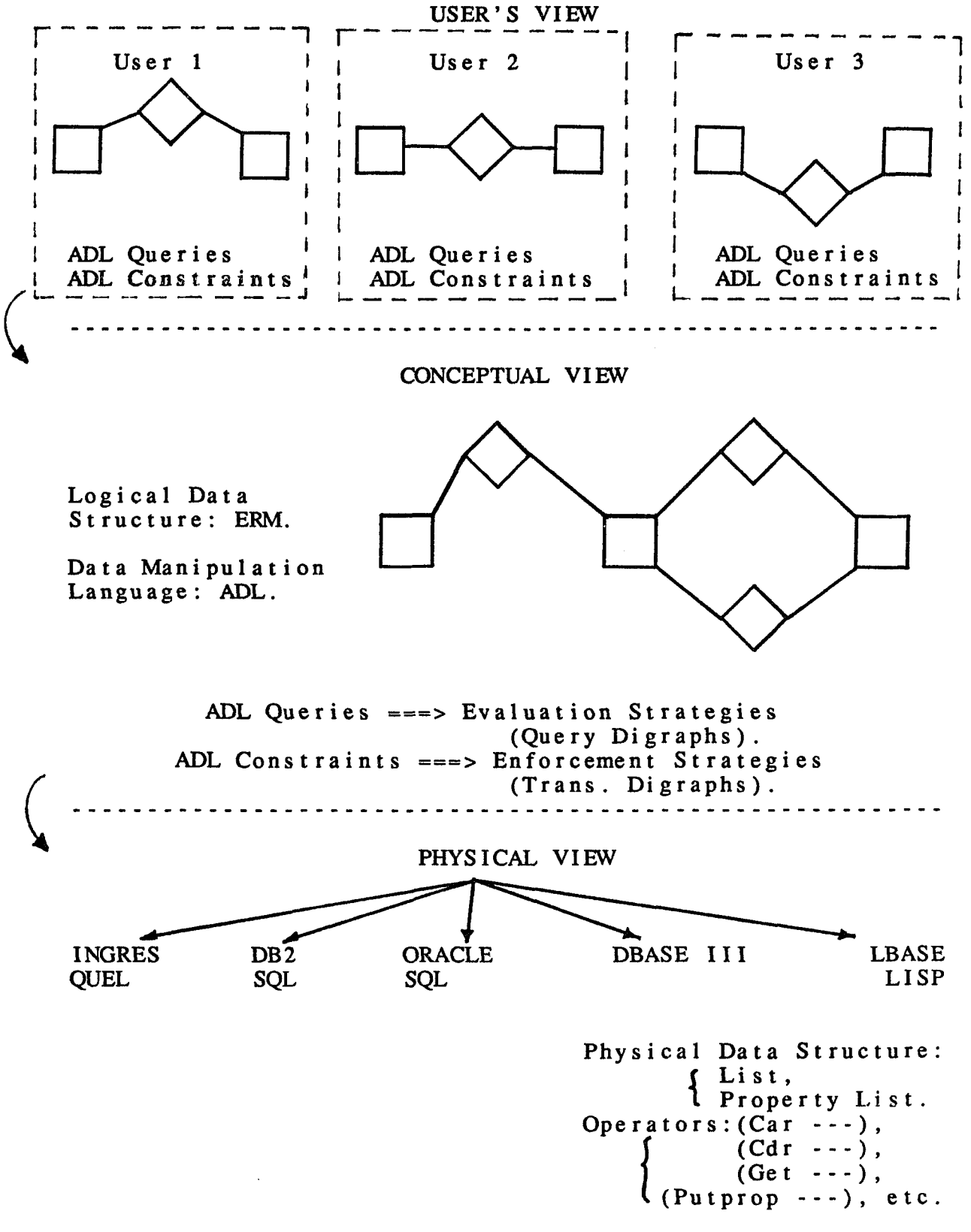
A literature survey on semantic constraints, database integrity, constraint enforcement, and constraint specification is carried out through out the research. What follows is an anatomy of the subjects and a comparison of ADL with earlier researches. The following literature provides an excellent review of database integrity and semantic constraints: [Eswa75, Date83, Fern81, Tsic82,

Figure 1.2.1. A Proposed Information System



Note:
1. The database is represented by the Entity Relationship Model.
2. The Applicative Data Lanague (ADL) will interface with the user as a query language, a constraint specification lanaguage, and a data access language.
3. An integrity subsystem is part of the overall architecture of the database system.

Figure 1.2.2. ADL's Role in ANSI/X3/SPARC Architecture



Aude86, Shep86, Flyn84, Morg86, Hamm76].

3.A. DATABASE INTEGRITY

Database integrity is the activity of maintaining the consistency or constraint of data in a database. It covers almost the entire database system activities and can be carried out by enforcing the following constraints:
(1). DOMAIN CONSTRAINT (or VALUE CONSTRAINT): A domain constraint restricts the value domains and value interpretations of data from the database, e.g., integer, real, character string, subrange, unit, etc. [Date83, Simo84]. Example: A person's age should range between 0 and 200 years old.
(2). DATA MODEL CONSTRAINT (or IMPLICIT CONSTRAINT): A data model constraint is the modeling primitive intrinsic to a data model, for example: Data Structures Implied by the Data Model, Functional Dependency, Multi-Value Dependency, Cardinality, Referential Dependency, or Existence Dependency, Key Uniqueness, Key Propagation after User Update [Tsic82, Abit85, Brod80, Shep86, Simo84, Date83]. Example: A person's social security number should determine a person's age and income.
(3). SEMANTIC CONSTRAINT (EXPLICIT CONSTRAINT): A semantic constraint models the semantics unique to an enterprise as a formalized assertion on data (see the next section for more detail). Example: A manager's salary should exceed the average salary of his subordinates.

On a semantic level, all the above constraint types look similar. Only when we translate the semantics into syntactic specifications do the above classification become meaningful. To model different database constraints requires different specification primitives and enforcement mechanisms. Currently commercial DBMS's can only specify and enforce domain constraints and very simple data model constraints. Very few can model the semantic constraint. It remains as one of the active research areas in database research [Date83].

In this research, an Applicative Data Language (ADL) is developed to provide a mechanism for modeling parts of the semantic constraints. The constraints modeled include local constraints, structural constraints, complex constraints, and aggregate constraints. ADL is different from earlier approaches of semantic constraint modeling in many aspects as discussed in the following sections.

3.B. CLASSIFICATION OF SEMANTIC CONSTRAINTS BY COMPUTATION TECHNIQUES

The specification of constraints represents a much easier task as compared to enforcement of these constraints. The diverse varieties of semantic constraints

call for vastly different data structures and computation algorithms to support their enforcement.

(1). DYNAMIC CONSTRAINTS (TRANSACTION CONSTRAINTS) are assertions on the old database state before the update and the new database state after the update [Furt81, Simo84, Date83]. Example: A person's age cannot decrease.

(2). TEMPORAL CONSTRAINTS are assertions on the data in a temporal database [Ehri84]. Example: When an engineer is promoted to "SENIOR ENGINEER", his salary should be raised by at least $4000.00 from his previous salary.

(3). DATABASE STATE CONSTRAINTS are assertions on the current database state. They can be further divided into:

(3.1). LOCAL CONSTRAINTS are assertions about attribute values of a single entity set [Lee87a, Lee87b, Brod80, Shep84, Shep86, Fenv85, Webe83, Fros83, Ston75, Hamm76]. Example: The total budget of employee's salary cannot exceed $3 million.

(3.2). STRUCTURAL CONSTRAINTS are assertions about inter-relationships among entity sets [Lee87a, Lee87b, Tabo83a, Tabo83b, Morg84, Morg86]. Example: Each department can manage at most five projects at any given time.

(3.3). COMPLEX CONSTRAINTS are assertions about both attribute values and inter-relationships of entity sets [Lee87a, Lee87b, Fros83, Brod80, Webe83, Bune79, Ston75, Brad84, Hamm76]. Example: Each department can only have a total employee-salary budget not exceeding its sales volume.

(3.4). AGGREGATE CONSTRAINTS (SET CONSTRAINTS) are assertions involving aggregate functions applied to a group of data [Lee87a, Lee87b, Fern81, Ston75, Naka83, Tabo83a, Tabo83b, Morg84, Morg86]. Example: The same as the above example.

(3.5). RELATION CLOSURE CONSTRAINTS are assertions involving relation closure computations of recursive relationships [Banc86]. Example: All the subparts of the engine block have to be made of stainless steel. (The data model of the enterprise includes a "PART" entity set and a "SUB_PART" recursive relationship set.)

There is no single language that can model all possible semantic constraints. ADL currently supports the following database state constraints: local constraints, structural constraints, complex constraints, and aggregate constraints. Many earlier researches can only support one particular type of database state constraints [Morg84, Morg86, Tabo83a, Tabo83b]. ADL differs from them in three respects: (1). ADL is based on a high level semantic data model, the ERM. (2). ADL can model a wider spectrum of semantic constraints and is not limited to a specific type of semantic constraints. (3). Besides specification of the semantic constraint, ADL also provides enforcement mechanisms that check the correctness of the constraint when it is first defined and when the database is

updated.

3.C. MODULES OF DATABASE SEMANTIC INTEGRITY SUBSYSTEM

To implement semantic constraints, a database system needs to incorporate a semantic integrity subsystem that is responsible for the specification, checking, compilation, and enforcement of semantic integrities. The subsystem is further divided into several functional modules. We will look at the functionality of each module as follows:

(1). CONSTRAINT SPECIFICATION MODULE

Constraint specification in which the user specifies semantic constraints with a predefined language is the first step of semantic constraint modeling. It has been the subject of many previous researches. Many of these earlier researches were based on the relational data model which has a very simple data structure and is limited in its semantic expressive power [Date83, Adib84, Nguy86a, Nguy86b, Brod80, Shep84, Shep86, Webe83, Ston75, Wils80, Ehri84, Hamm76, Thee81, Naka83]. Few take advantage of the latest development in semantic data models which uses the ERM as the underlying data structure [Tabo83a, Tabo83b, Morg84, Morg86]. Even those that do use ERM as the underlying data structure only specify the structure constraints mentioned in 3.B. ADL is also based on the ERM; but it can model a much wider range of semantic constraints.

(2). CONSTRAINT CONSISTENCY CHECKING MODULE

It checks the newly defined constraint to see: (a). if it is inconsistent with those already defined, (b). if it can be derived from other constraints. Since many constraint specification methods use First Order Logic (FOL), they can prove the contradiction or derivability of newly defined constraint with respect to existing constraints [Wojc84, Wils80]. Currently, ADL does not support consistancy checking. It relies on the user to check any inconsistency and redundancy. It is possible that in the future ADL can combine symbolic and algebraic computation techniques to perform consistency checking.

(3). CONSTRAINT COMPILATION MODULE

It compiles the constraint specifications into enforcement methods [Lee83b, Lee83c]. Many earlier approaches [Nico82, Hsu85, Ling84, Ston75] that use the logical specification method only compile the user's specification into a new specification with user's updates triggering conditions for constraint checking. Like all FOL type specifications, the recompiled specification in these approaches is only an extended specification which includes triggering conditions but not a detailed enforcement plan.

(4). CONSTRAINT ENFORCEMENT MODULE

It enforces semantic constraints both at compile-time and at run-time [Lee87b, Lee87c, Ling84, Hsu85, Webe83, Fros83, Bune79, Ston75, Bern80, Ehri84, Bert84].

(4.1). RUN-TIME TRIGGERING SUBMODULE: Each semantic constraint only has to be checked as a result of certain update transactions. It is, therefore, the responsibility of the run-time triggering submodule to monitor user updates and invoke proper checkings accordingly [Lee87c, Date83, Ditt86, Wils80 , Bert84].

(4.2). RUN-TIME EVENT SUBMODULE: When a constraint is violated as a result of an update, different actions will be taken depending on the constraint specification [Date83, Ditt86, Wils80, Bert84]. Possible actions include:

(4.2.1). GIVE WARNING, ROLL-BACK USER'S TRANSACTION: This type of constraints are called "passive constraints" because they do not take actions to correct the situation [Lee87c].

(4.2.2). CORRECT USER'S TRANSACTION: This type of constraints are called "active constraints" which correct the user's transaction to make the database consistent after the update [Date83], [Chen87].

(4.2.3). CORRECT DATABASE STATE: If the constraint involves not only data in an update but also data in the previous database, it is possible to modify the latter to correct the situation. This is also called an "active constraint" [Shep86].

3.D. COMPONENTS OF SEMANTIC CONSTRAINT SPECIFICATION

Each constraint specification can be divided into several components. Some can be optional (default values assumed); some are essential. Depending on the complexity of the semantic to be modeled, a constraint specification may incorporate all or some of following components:

(1). WHEN TO CHECK THE CONSTRAINT (CONSTRAINT INVOCATION, PRECONDITION): It is necessary to specify when to check the constraint if it is a dynamic constraint [Date83, Fenv85, Shep84, Shep86]. ADL currently does not model dynamic constraint. Therefore, it does not state preconditions in the constraint specifications.

(2). WHAT ARE THE CONSTRAINED DATA: Constraint data are data related to the semantic constraint. We have to either specify or compute the constrained data [Lee87a, Gard79, Thee81, Tabo83a, Tabo83b, Brod80, Shep84, Shep86, Ston75, Fenv85, Webe83, Fros83, Ehri84, Brad84, Hamm76, Morg84, Morg86]. ADL differs from all the other specification methods in several ways. First of all, it is based on the ERM which is a semantic data model. Second, it takes the computational approach in constraint specification instead of the FOL approach. Third, it incorporates an object level computation as the first step for finding constrained data.

(3). WHAT IS THE ASSERTION ON THE CONSTRAINED DATA: Once the constrained data is identified, we can make assertions about the constrained data [Lee87b, Gard79, Tabo83a, Tabo83b, Brod80, Shep84, Shep86, Ston75, Fenv85,

Webe83, Fros83, Ehri84, Brad84, Hamm76, Morg84, Morg86, Thee81].
(4). WHAT TO DO IN CASE OF VIOLATION: For more complicated constraints, it is necessary to specify the course of action to be taken if the constraint is violated [Shep84, Shep86, Thee81]. ADL assumes a simple default action for all the constraints, that is, rolling back user's transaction and warning the user.

3.E. CONCEPTUAL DATA MODEL USED IN CONSTRAINT SPECIFICATIONS

All constraint specification methods must have a conceptual data model as their underlying data structure. ADL differs from most other specification methods by using the ERM as its conceptual data model.

(1). RELATIONAL DATA MODEL: All FOL approaches of constraint specification use the relational data model because of its uniform and simple data structure [Abit85, Gard79, Nico78, Nico82, Ling84, Fenv85, Webe83, Bune79, Ston75, Bern80, Hamm76]. However, in recent years many semantic data models were developed to incorporate more semantic modeling capability.
(2). ERM IN DESIGN, RELATIONAL DATA MODEL IN SPECIFICATION: Few methods use the ERM in the design phase to capture the semantics and then transform everything into the relational data model in the performance phase for query formulation and constraint specification. [Naka83, Thee81].
(3). ENTITY-RELATIONSHIP MODEL: Since ERM has been widely accepted as a database design tool for capturing semantics of an enterprise, many recent researches have been trying to use the ERM as the underlying data structure for constraint specification [Lee87a, Tabo83a, Tabo83b, Morg84, Morg86, Furt81, Fros83, Brad84].

3.F. REPRESENTATION METHOD OF CONSTRAINT SPECIFICATION

There are several different approaches to specify semantic constraints.
(1). GRAPHICAL REPRESENTATION: Tabourier [Tabo83a,Tabo83b] and Brady [Brad84] extended the notions of using the Entity Relationship Diagram to specify various semantic constraints.
(2). LOGICAL REPRESENTATION: Many researchers extended the FOL to specify semantic constraints.

(2.1). DOMAIN CALCULUS: [Naka83].

(2.2). TUPLE CALCULUS: [Fros83, Ston75, Bern80, Furt81, Wojc84, Brod80, Nico78, Nico82,Ling84, Hsu85].

(2.3).INFERENCE RULE:[Adib84, Shep84,Shep86, Chol85].
The FOL type specification methods in many ways are limited by the simple syntax of FOL. Functions of any kind cannot be easily incorporated into these methods without some major modifications.
(3). COMPUTATIONAL REPRESENTATION: A more versatile approach in constraint specification is the computational approach where the constrained data are computed from the database [Lee87a, Lee87b, Bune79, Hamm76, Morg84, Morg86].

3.G. COMPONENTS OF CONSTRAINT ENFORCEMENT

Constraint enforcement involves both checking the existing data in the database when a constraint is first defined and checking the updated data after the constraint is in place.
(1). COMPILE-TIME CONSTRAINT CHECKING: Immediately after a constraint is defined, the system has to check all the existing data in the current database to see if they comply with the newly defined constraint. [Date83, Lee87c].
(2). RUN-TIME CONSTRAINT CHECKING: Once semantic constraints have been defined and the current database state is consistent with the defined constraints, a user may submit an update transaction to modify the database from one state to another. Run-time constraint checking insure that after the update is written to the database, it is still consistent with all the constraints. This is also called database monitoring. [Lee87c, Abit85, Malv81, Gard79, Lili82, Ling84, Bune79, Bern80, Lafu79, Nico82, Hsu85].

(2.1). IMMEDIATE CHECKING FOR A SINGLE UPDATE OPERATION: Check the set of constraints before or after a single update operation. Users may specify when to check a particular constraint (triggering conditions). [Date83, Ston75, Ling84, Webe83, Bune79].

(2.2). IMMEDIATE CHECKING FOR MULTIPLE UPDATE OPERATIONS (PART OF A TRANSACTION): Need to solve the scheduling and synchronizing problem between updating and checking operations [Simo84, Bert84].

(2.3). DELAYED CHECKING AFTER THE ENTIRE TRANSACTION: Defer checking until the entire update is completed [Lee87, Hsu85, Webe83, Lafu82].

ADL includes both compile-time and run-time enforcement methods at the conceptual level. They can be mapped to different physical implementation schemes (Figure 1.2.2). It uses the delayed checking technique to enforce run-time constraint checking, i.e. invoking the checking procedure after the entire transaction is submitted.

4. OBJECTIVES OF THE RESEARCH

The objective of this research is to develop a semantic constraint modeling mechanism in a database based on the Entity Relationship Model. The following features of the integrity subsystem will be investigated:

(i) Surveying the literature on the subjects of the types of semantic constraints, constraint specifications, constraint enforcement, and integrity subsystem architectures.

(ii) Developing a constraint specification language based on the Entity Relationship Model.

(iii) Developing a compile-time checking method that can be enforced on existing data in the database after a new semantic constraint is defined.

(iv) Developing a run-time constraint checking method that can be implemented more efficiently.

(v) Developing an integrity subsystem prototype to test the applicability of the constraint modeling methodology proposed in this research.

5. SCOPE OF THE RESEARCH

To achieve the above objectives, the boundaries of this research are delimited as follows:

(i). The constraint modeling mechanism proposed here only focuses on a conceptual level implementation of a semantic constraint subsystem in a database. It studies (1). how a constraint is specified, (2). how the specification is compiled into enforcement methods, (3). how the constraint is enforced at compile-time, and (4). how the constraint is enforced at run-time.

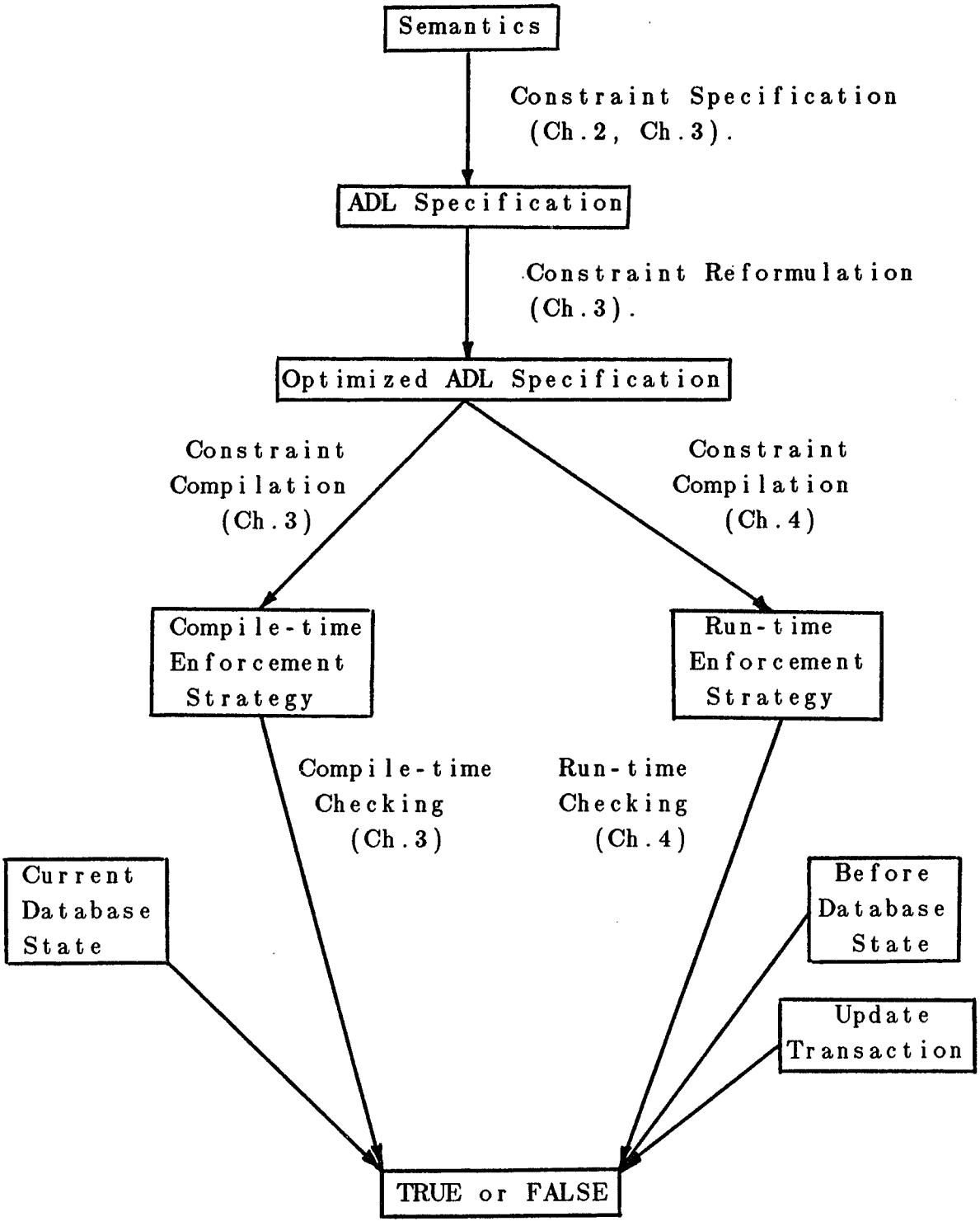
(ii). This research only concerns the modeling of database state constraints which are part of the semantic constraints. The database state constraints modeled include local constraints, structural constraints, complex constraints, and aggregate constraints. Constraints other than the ones just mentioned are not included in this research.

(iii). The consistency checking module of the constraint subsystem is not studied here. Therefore, the proposed system does not have the ability to check duplication or contradiction among defined semantic constraints.

(iv). Many implementation details of constraint modeling are not covered in this study. For example, how to schedule the checking of defined constraints (other than sequential), how to turn-on and off of a particular constraint, how to incorporate response actions for various constraints are not addressed in this study.

A diagrammatic view of the organization of the disseration is shown in Figure 1.5.1.

Figure 1.5.1. Organization Of The Dissertation



CHAPTER II. AN ER-BASED APPLICATIVE DATA LANGUAGE (ADL)

In recent years, database research has been looking into ways to assure the integrity of data stored in a database [Date81, Fern81, Tabo83a, Fros83]. People who implemented earlier CAD/CAM databases quickly realized that merely storing data in a well structured form did not necessarily improve the productivity of engineers. What's more important is that the data need to be checked and verified before it is used for decision making [Canf85, Mait85]. One way to maintain data consistency is to define semantic constraints with a high level data language and enforce it by a database subsystem.

Constraint modeling embodies two important tasks. First, it has to find the constrained data in the smallest possible domain. Second, within such a domain we make assertions on the constrained data. The methodology used to model semantic constraints in our research is called Applicative Data Language or ADL. It not only stands by itself as a complete query language because it has to meet the first task, but also has a rich set of predicates to complete the second task. The foundation of the language is made of three types of primitives: (1) data structures, (2) operators (or functions), and (3) predicates. By combining these three primitives in a proper manner according to our semantic perception, we can compute the truth value of the database with respect to the semantic constraint. Therefore, in this chapter we first define a data manipulation language, its data structure, and its operators.

In the past ten years, considerable effort has been given to use the Entity Relationship Model (ERM) as a database and information design tool [Chen76, Chen83]. Various attempts have been made to develop high level data manipulation languages based on ERM [Dehe76, Elma83, Mark83, Poon80, Shos78]. Many of these earlier languages are more or less based on a modified relational calculus adapted to the ERM. Atzeni and Chen proposed the "ER completeness" and "Simplified ER completeness" for ER calculus-like languages [Atze83]. They claimed that ER query languages developed prior to their paper did not satisfy their definition of completeness. Their ER-completeness cannot express queries that require comparison of unconnected objects in an ER diagram. Therefore, the ER-completeness has less expressive power than the relational completeness by Codd [Codd72].

Chen has proposed a specification for an algebraic language that is based on a directional binary ERM [Chen84]. It does not delineate the detail grammar constructs of such a language. Campbell and elta has proposed a graphical query language that is proved to be relational complete [Camp85]. But the graphical language does not have all the necessary operators to be a full fledged data language. The more recent attempt by Subieta and Missala has resulted in the language NETUL, a powerful ERM query language [Subi86]. It maps denotational

semantic queries directly into physical data structures of the database. It is a powerful language but with very complex data structures and syntax constructs. All these make it a harder language to use and to learn.

To meet our objective of semantic constraint modeling, we would like to develop a data language with the following features:

(1) It should support the logical data structures of the Entity Relationship Model.

(2) It should minimize the computation complexity by doing first an object level computation and then a data element level computation.

(3) It should handle the semantics of generalization, specialization, classification, and aggregation in many information systems.

(4) It should have a computational syntax so most application programs can use it as a data access language to interface with the database.

The data manipulation language of ADL proposed here is based on the concept of separating computations into two distinctive spaces (two-space approach). The two computation spaces represent two different levels of abstractions. In the entity space, we carry out object level computations by keeping each data record (an entity) as an indestructible object. We compute the relationships among data objects. The purpose is to identify constrained objects in the smallest possible domain. While in the data space, we look into the attribute values of entities and carry out computation at the data element level. To handle the vast varieties of semantic constraints, the two-space approach seems to be a better and more flexible method as compared to others [Hamm76, Tabo83a, Tabo83b, Morg84, Morg86].

The data manipulation language presented here takes advantage of the semantic modeling power of ERM. Since the abstractions of relationships and entities are distinctly modeled in the ERM, many calculations for finding out how objects are associated with each other can be done at object level in the entity space. This conceptual clarification of relationships and entities paves the way for the object level computations that are previously hard to implement in the relational data model.

This is because the semantics of data associations and data organizations are all buried in the same data structure (tables of tuples) in the relational data model. When we join two relations , we virtually have to join all the data elements together. It forces us to make early decisions on which attributes are eventually needed and which are not. This thinking process tends to interfere with the calculations of relationships among the data objects.

The proposed data manipulation language is presented in the following manner:

(1) Review the Data Structure of ERM (Section 1) -

ERM logically organizes data into entity sets and relationship sets. Most information about the real world are represented as attributes of entity and relationship sets in the database. Material for this review is drawn from Chen

[Chen76] and Maier [Maie84]. The logical view of the ERM is considered as the current database state from which we construct **base states** in the **entity spaces**. The constructed states, then, go through a series of entity space and data space computations to derive the truth value of the database with respect to a specific semantic constraint.

(2) Develop Data Structures, Operators, in Entity Space (Section 2) -

A state is the most important data structure in an entity space. We start with how to construct primary entity space and base states. We build higher order states and hyper states through entity space computations. All operators (functions) and predicates in the entity space use states as their arguments.

(3) Develop Data Structure, Operators, in Data Space (Section 3) -

In the data space, the entirety of data records is broken down into individual data elements. The basic data structure in the data space is called a **d_relation** (derived relations) which is similar to an ordinary relation in the relational data model, but are derived from the results of the entity spaces computations.

1. ENTITY RELATIONSHIP MODEL, A REVIEW

The material discussed here is drawn from Chen [Chen76], Maier [Maie84]. Some definitions have been modified and augmented to suit the present purpose of constraint modeling. For more detailed definitions of terms and theories, readers may refer to the original papers. The purpose of this chapter is just to provide the reader with a quick review of the ERM that represents the current database state.

#Definition 2.1.1: **Entity Set,Entity,Attribute,Sch(E),|E|**

An **entity set**, **E**, is the set of ordered tuples.

$E=\{(a_{i,1},..,a_{i,j},..,a_{i,m}) | 1\leq i\leq n,\ 1\leq j\leq m,\ a_{i,j}\in Dom(A_j)\}$. In which,

$(a_{i,1},..,a_{i,j},..,a_{i,m})$: an **entity**, we denote it as e_i.

$a_{i,j}$: the j-th **attribute** value of e_i.

Each position in the tuple corresponds to an attribute Aj (a symbol). The domain of an attribute is $Dom(A_j)$. The scheme of the entity is Sch(E) which is the set of attribute symbols, i.e. $\{A_1,.., A_j, .., A_m\}$. We denote |E| as the number of entities in E, i.e. $|E|=n$. ###

From the above definition, each tuple suggests a mapping from Sch(E) to $Dom(A_1) \cup .. \cup Dom(A_j) \cup .. \cup Dom(A_m)$ as

$$e_i \text{ mapping } \begin{cases} e_i(A_1) \text{ ---> } a_{i,1}, \\ \ldots \\ e_i(A_j) \text{ ---> } a_{i,j}, \\ \ldots \\ e_i(A_m) \text{ ---> } a_{i,m}. \end{cases}$$

Collectively, we can say: $e_i(Sch(E)) := e_i(\{A_1,..,A_j,..,A_m\}) := (a_{i,1},..,a_{i,j},..,a_{i,m})$ Therefore, entity set E can also be regarded as a set of all such mappings, $E := \{e_1,..,e_i,..,e_n\}$.

In order to uniquely identify an entity in a simple manner, we assume each entity set has a single attribute key, "EID", and impose the following Functional Dependency (FD) constraint for all the entity sets in the ERM.

FD

$$EID \longrightarrow A_1, \ldots, A_j, \ldots, A_m.$$

If entities cannot be uniquely identified by a single attribute key (need a composite key instead), the system needs to assign a surrogate key attribute for the entity set [Camp85, Doga86].

We sometimes use the following notation to express an entity set: $E(EID,A_1,...,A_j,...,A_m)$. It denotes the following things:

(1) $E = \{ (eid_i,a_{i,1},..,a_{i,j},..,a_{i,m}) \mid 1 \le i \le n,\ 1 \le j \le m,\ eid_i \in Dom(EID),\ a_{i,j} \in Dom(A_j)\}$

(2) Each id uniquely identifies an entity in E, i.e.

$eid_i = eid_j \Longrightarrow e_i = e_j$

(3) There exist n mappings $e_1,..,e_i,..,e_n$. Each is defined as: $e_i(EID)$ -> eid_i, $e_i(A_1)$ -> $a_{i,1}$,..., $e_i(A_m)$ -> $a_{i,m}$. Since there is a one-to-one mapping between e_i and eid_i, we can extend the notion of mapping to: $eid_i(A_1)$ ---> $a_{i,1}$, ... ,$eid_i(A_m)$ ---> $a_{i,m}$.

#Definition 2.1.2: **Relationship Set, Relationship, Sch(R), |R|**

A **relationship set, R**, is the set of ordered tuples.

$R = \{ (eid_{i,1},..,eid_{i,p},..,eid_{i,m}) \mid 1 \le i \le n, 2 \le p \le m,$
$eid_{i,1} \in Dom(EID_1)$ of E_1, ..,$eid_{i,m} \in Dom(EID_m)$ of $E_m\}$.

in which, $(eid_{i,1},...,eid_{i,p},...,eid_{i,m})$ is a **relationship** in R. We denote it as r_i. Each position in r_i corresponds to an associated entity of the relationship. $E_1,..,E_p,..,E_m$ are related entity sets of R. The scheme of R, **Sch(R)**, is defined as the set of all the symbols of the related entity sets, i.e. $Sch(R) := \{E_1,...,E_m\}$. $|R| = n$, the number of relationships in R. ###

From the above definition, each relationship r_i suggests a mapping from Sch(R) to $Dom(EID_1) \cup ... \cup Dom(EID_m)$.

$$r_i \text{ mapping } \begin{cases} r_i(E_1) \text{ ---> } eid_{i,1}, \\ \ldots \\ r_i(E_m) \text{ ---> } eid_{i,m}. \end{cases}$$

An equivalent view of R is simply a set of all such mappings, $R := \{r_1, .., r_i, ..,$

r_n}.

m=2, R is called a binary relationship set.

m>2, R is called an m-ary relationship set.

We may use the shorthand notation of $R(E_1,..,E_m)$ to express the following things.

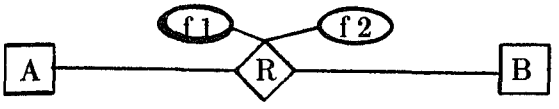
(1) $R = \{ (eid_{i,1},..,eid_{i,p},..,eid_{i,m}) \mid 1 \leq i \leq n,\ 2 \leq p \leq m,\ eid_{i,1} \in Dom(EID_1)$ of E_1, $..,eid_{i,m} \in Dom(EID_m)$ of $E_m\}$.

(2) Each r_i is uniquely identified by $E_1,...,E_m$. Therefore, $\{E_1,...,E_m\}$ is the primary key of R.

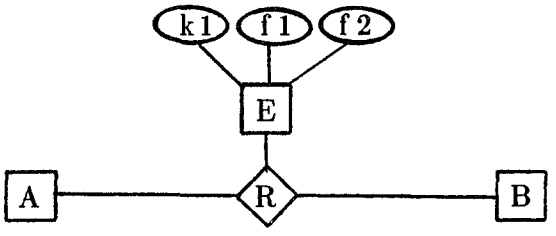
(3) Each r_i corresponds to a mapping from Sch(R) to $Dom(EID_1) \cup ... Dom(EID_m)$, i.e. $r_i(E_1) \text{---}> eid_{i,1},...., r_i(E_m) \text{---}> eid_{i,m}$.

Note that, in our definition of R, we assume that none of the relationship sets contain any attributes of their own. They merely represent the association of entities in the real world, which is a simplification of the original Entity Relationship Model. But such simplification will make the final ERM more readily transferable to other intermediate level data models, e.g. relational, network, and hierarchical. If the semantics requires a relationship set with its own attributes, we can make the following modification to adapt it to our simplified ERM.

Initial ERD:



Modified ERD:



"k1" is a primary or surrogate key which uniquely identifies the association between entities in A and B.

The ERM is considered as the logical data structure for the current database state. Next, we will develop an entity space model that is based on the ERM framework to compute relationships among entities. The basic data structure of the entity space model is constructed from the current database state.

An Entity Relationship Diagram (ERD) is a graphical representation of ERM. We assume the reader is already familiar with the notations of ERD.

2. DATA STRUCTURES AND OPERATORS IN THE ENTITY SPACE

Based on the simplified ERM, data associations (relationship sets) and data organizations (entity sets) have been distinctly modeled. We may take advantage

of this conceptual clarification by carrying out relationship calculations at the object level instead of at the data element level. This gives us the advantage of worrying not about the individual data which tends to be enormous in size, but about their higher abstraction, the entity.

Here, we will see how ADL utilizes the logical data structure of the ERM to calculate relationships among objects with entity space operators. The objective is to find the smallest set of objects that are related to the constraint. We first introduce the primary entity space in which all objects are singular objects. We then extend the definition to that of the hyper entity space in which some of the objects are compound objects. Both the primary and the hyper entity spaces are simply called entity spaces.

2.A. Data Structures In The Primary Entity Space

#Definition 2.2.1: **Primary Entity Space, Ψ, Sch(Ψ), Order(Ψ)**
Assume $\{E_1,...,E_m\}$ is a subset of the set of all the entity sets in the ERM. The Cartesian product space: E_1 x...x E_m is a **primary entity space**, Ψ, if for any E_i, E_j, i<>j, E_i and E_j are on a path in the ERD. **Sch(Ψ)**, the scheme of the entity space Ψ, is defined as the set of symbols of the involving entity sets, i.e., $Sch(\Psi)=\{E_1,...,E_m\}$. The **order** of the primary entity space Ψ, **Order(Ψ)**, is equal to $|Sch(\Psi)|$ or m, the number of related entity sets in the entity space. We define Ψ as an m-th order entity space. ###

Duplications of entity sets are allowed but marked by different symbols to signify different roles played by the same entity set in Ψ.

#Definition 2.2.2: **Entity Vector in the Primary Space**
An element of the relation space Ψ is called an entity **vector**,s_i. Let s_i be an entity vector:

$$s_i := <eid_{i,1},eid_{i,2},...,eid_{i,m}> \qquad ###$$

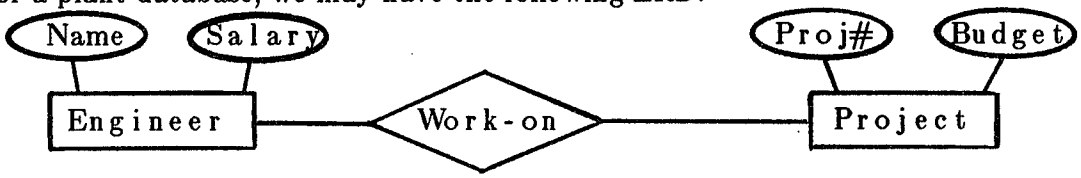
Note that $eid_{i,1},eid_{i,2},...,eid_{i,m}$ are simply the identifiers of associated entities. Each s_i suggests a mapping from Sch(Ψ) to $Dom(EID_1) \cup Dom(EID_2) \cup ... \cup Dom(EID_m)$ as follows:

$$s_i \text{ mapping} \begin{cases} s_i(E_1) \text{ ---> } eid_1, \\ .... \\ s_i(E_m) \text{ ---> } eid_m, \end{cases}$$

$|\Psi|$ is the total number of elements in Ψ and $|\Psi|$=n. From the definition, we know: $|\Psi| = |E_1| x ... x |E_m|$. As the number of entities in the entity sets increases, the size of Ψ multiplies quickly. However, we are not interested in Ψ, but interested in a very small subset of Ψ. This small subset of Ψ is implied by or computed from the current database.

#Example 2.2.1: Entity Space, Entity Vector

For a plant database, we may have the following ERD:



"Name" is the primary key (EID) of "Engineer"; "Proj#", the primary key of "Project". The current database contains the following facts:

Engineer	Name	Salary
	Herman	10,000
	Pete	70,000
	Ross	30,000
	Joe	25,000

Project	Proj#	Budget
	p101	20,000
	p102	99,000

Work-on	Engineer	Project
	Herman	p101
	Ross	p101
	Ross	p102

Since "Engineer" and "Project" are linked together by a path in the ERD, we can create an entity space Ψ = Engineer x Project (Figure 2.2.1) with the following properties:

Order(Ψ):=2,

Sch(Ψ):={Engineer, Project},

s_1 is a vector in the entity space and s_1 may be:

s_1:=<Herman,p101>.

The total number of such vectors in Ψ is $|\Psi|$

$|\Psi| = |\text{Engineer}| \text{ x } |\text{Project}| = 4 \text{ x } 2 = 8.$

To list them all, we would have

Ψ := {<Herman, p101>,<Herman, p102>,<Herman, p103>,
<Pete, p101>,<Pete, p102>,<Pete, p103>,
<Ross, p101>,<Ross, p102>,<Ross, p103>,
<Joe, p101>,<Joe, p102>,<Joe, p103>}###

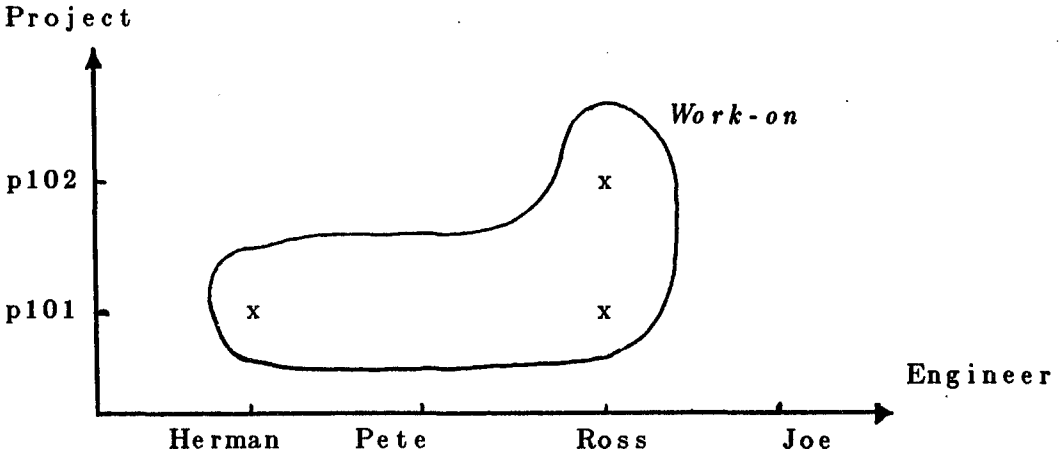
Note that in real applications we never compute the Cartesian product of the entity space. We only compute a subset of it.

#Definition 2.2.3: **Power(Ψ),Entity State ς, Sub-state, $|\varsigma|$.**

The power set of Ψ is **Power(Ψ)** which is the set of all possible combinations of entity vectors in Ψ. For any element $\varsigma \in$ Power(Ψ), ς is called a **state of** Ψ. And $\varsigma \subseteq \Psi$. The number of elements in a state is expressed as $|\varsigma|$ $|\varsigma| \leq |\Psi|$ Any subset of ς is a **sub-state** of ς. ###

The state in an entity space reflects the actual relationships among entities as implied by or computed from the current database at any one point in time

Figure 2.2.1. An Entity Space Of Ψ = Engineer x Project



Entity Space: Ψ = Engineer x Project.

Entity State: *Work_on* ={ <Herman, p101>, }.
<Ross, p101>,
<Ross, p102>

Entity Vector: <Herman, p101>.

Sch(Ψ)={ Engineer, Project }.

Order(Ψ)= 2.

Work-on ⊆ Ψ, or *Work_on* ∈ Power(Ψ)

(a snapshot). It is only a very small subset of all possible relationships that may exist among the entity sets.

#Definition 2.2.4: **E-state, R-state, Base States.**
(1). If there is an entity set E in the current database as:
$E = \{ (eid_i, a_{i,1}, .., a_{i,j}, .., a_{i,m}) \mid 1 \le i \le n,\ 1 \le j \le m,\ eid_i \in Dom(EID),\ a_{i,j} \in Dom(A_j) \}$,
we can construct a first order entity state E (an **E-state**) of the entity space Ψ, and $\Psi = E$, as follows:
$E = \{ <eid_i> \mid (eid_i, a_{i,1}, .., a_{i,j}, .., a_{i,m}) \in E \}$.
(2). If there is a relationship set U with $Sch(U) = \{E_1, ..., E_m\}$ in the current database as:
$U = \{ (eid_{i,1}, .., eid_{i,p}, .., eid_{i,m}) \mid 1 \le i \le n,\ 2 \le p \le m,$
$YMd_{i,1} \in Dom(EID_1)$ of E_1, $.., eid_{i,m} \in Dom(EID_m)$ of $E_m \}$,
we can construct an m-th ($m \ge 2$) order entity state U (an **R-state**) in the entity space Ψ, and $\Psi = E_1 \times E_2 \times ... \times E_m$, as follows:
$U = \{ <eid_{i,1}, .., eid_{i,p}, .., eid_{i,m}> \mid (eid_{i,1}, .., eid_{i,p}, .., eid_{i,m}) \in U \}$.
(3). All the E-states and R-states so constructed from the current database are called **base states**. ###

#Example 2.2.2: Entity State, E-state, R-state.
Continuing our Example 2.2.1 (Figure 2.2.1), we may have the following base states as implied by the current database:
Base States, E-states (in 1st order entity spaces)
Engineer := { <Herman>,<Pete>,<Ross>,<Joe> },
Project := { <p101>,<p102> }.
Base State, R-state (in a 2nd order entity space)
Work-on:= {<Herman,p101>,<Ross,p101>,<Ross,p102>}.
Since the entity space Ψ = Engineer x Project, we may write: *Work-on* $\subseteq \Psi$, *Work-on* $\subseteq$ Engineer x Project, or *Work-on* $\in$ Power(Ψ). Note that *Engineer*, *Project*, and *Work-on* are entity states. They are called base states because they are implied directly from the current database and require no computations. From these base states we can apply entity space operators to compute other states that reflect the semantic associations of objects. ###

2.B. Data Structures In The Hyper Entity space

When we try to use the primary entity space to model semantics among entities, we find that the data structure falls short of being complete. In particular, the semantics of aggregation, classification, generalization, and specialization cannot be adequately modeled by the primary entity space. To model this level of

semantic associations we need to group certain entities together as a set and associate the entire set with other entities (form a compound data structure). So in this section we introduce the concept of hyper entity space as an extension of the primary entity space.

In the hyper entity space, we also have to extend the operators to manipulate the compound data structures. Traditional relation algebra [Codd72, Ullm82], on the other hand, can only join, select, and project on single valued data structures (First Normal Form). Aggregate functions and operations are not part of the query language in the earlier versions of relational algebra. However, in the hyper entity space the aggregate function becomes a natural extension of entity space computations.

The semantics of "grouping" is very important in data modeling. It is a double-edged sword in terms of capturing data semantics (Figure 2.2.2). In one direction, it represents specialization and classification. In the other direction, it represents generalization and aggregation. The extended relation operators in the hyper entity space provide a way to operate on objects based on the semantics of "grouping". With these operators such semantics are not just static data definition (declarative semantics) any more. The user can actually "compute" new representations according to the generalization and specialization semantics (computational semantics).

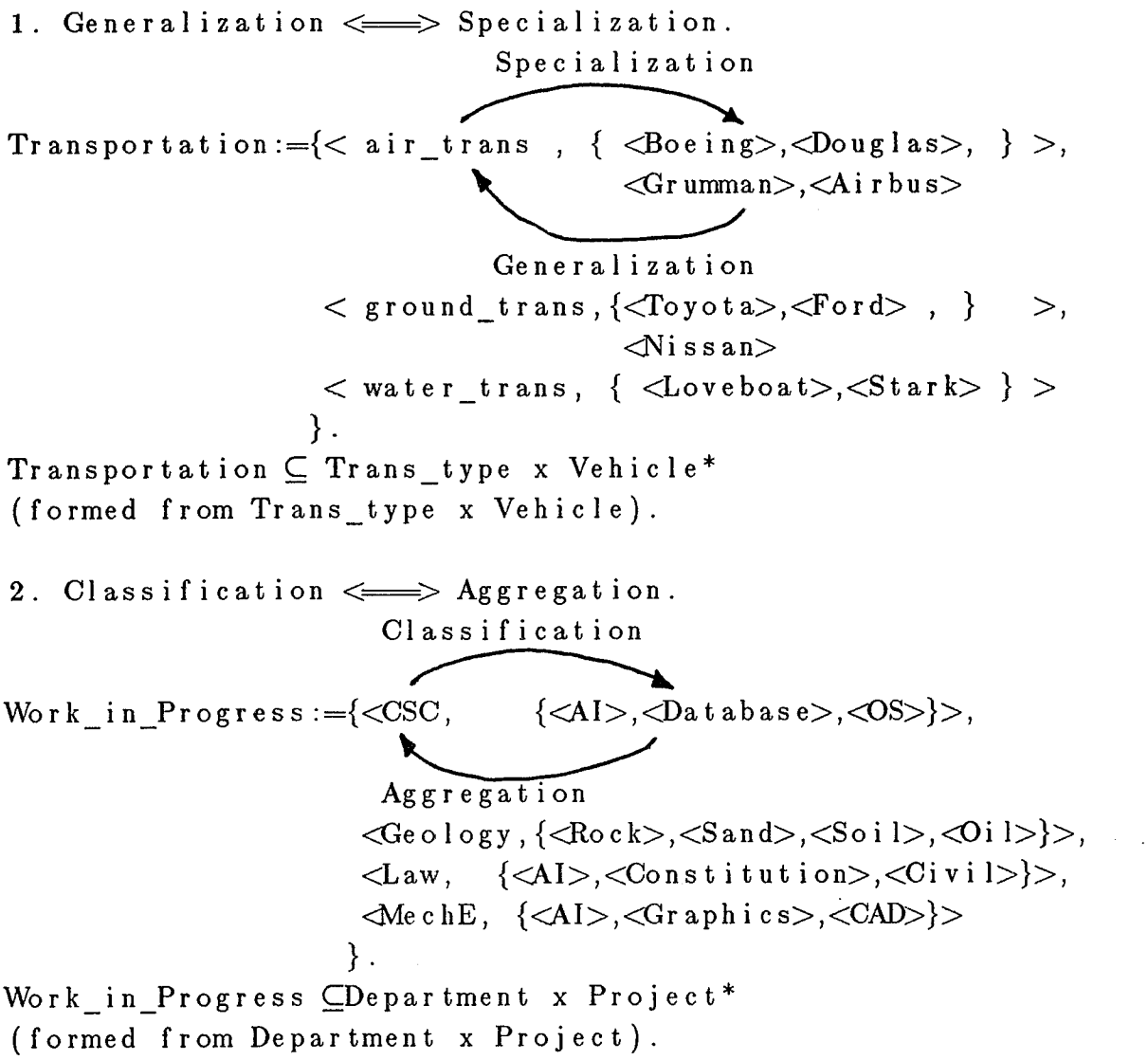
Gray [Gray84] has discussed the application of the Group_By operators to assemble data elements together before the application of aggregate functions like count, max, min, avg, etc. Thus, these aggregate functions become an extended part of the relational algebra. Merrett [Merr84] has proposed a "sigma join", Σ, which selects tuples that have common associations. More recent research [Roth85, Thom86, Jone86] in non-first normal form relations has generated tremendous interest in extending relation algebra to manipulate the compound data structures. However, all these models are based on the relational data model in which the concepts of objects and relationships are not distinctly modeled.

The concept developed here is a generalization of the Group_By concept of Gray [Gray84] and the sigma join concept of Merrett [Merr84]. We modify the **Group_by** operator to generate compound objects which contain sets of singular objects.

The primary entity space defined earlier is a Cartesian product space of entity sets. Each vector in the entity space consists of components corresponding to associated entities. Let's look at what is a hyper entity space.

#Definition 2.2.5: **Hyper Entity Space, Hyper Entity Vector**
A hyper entity space, Ψ, is the Cartesian product space of: $\Psi = E_1 \text{ x...x } E_p \text{ x Power}(\Psi_1) \text{ x ...x Power}(\Psi_r)$ or $\Psi = E_1 \text{ x..x } E_p \text{ x } (E_{1,1}\text{x..x}E_{1,q1})^* \text{ x .. x } (E_{r,1}\text{x...x}E_{r,qr})^*$, in which : $E_1,...,E_p$ are entity sets. $\text{Power}(\Psi_1),...,\ \text{Power}(\Psi_r)$

Figure 2.2.2. Using Hyper Entity Space To Model Generalization, Specialization, Classification, Aggregation.



or $(E_{1,1}x..xE_{1,q1})^*$,... , $(E_{r,1}x...xE_{r,qr})^*$ are the power sets of the primary entity spaces $\Psi_1,...,\Psi_r$, respectively. And, $\Psi_1 = E_{1,1}$ x...x $E_{1,q1}$,..., $\Psi_r = E_{r,1}$ x...x $E_{r,qr}$.

A hyper entity vector s_i in Ψ is of the following form: $s_i = <eid_{i,1}, ... , eid_{i,p}, \varsigma_{i,1},..., \varsigma_{i,r}>$, in which $eid_{i,1} \in Dom(EID_1)$ of E_1,.., $eid_{i,p} \in Dom(EID_p)$ of E_p, $\varsigma_{i,1} \in Power(\Psi_1)$,...,$\varsigma_{i,r} \in Power(\Psi_r)$. ###

The size of Ψ is potentially large. For example, if $|E_1|$=n1,..., $|E_p| = np$ and $|\Psi_1| =$ m1,..., $|\Psi_r| = mr$, then

$$|\Psi| = n1 \times ... \times np \times 2^{m1} \times ... \times 2^{mr}.$$

Fortunately, we only have to deal with a small subset of the hyper entity space at any given time.

#Definition 2.2.6: **Entity State in the Hyper Entity Space**

An entity state of a hyper entity space, ς, is a set of hyper entity vectors, i.e. $\varsigma \in Power(\Psi)$, $\varsigma \subseteq \Psi$, and Ψ is a hyper entity space. ###

#Definition 2.2.7: **Sch(Ψ), SchE(Ψ), Singular Dimension, Compound Dimension.**

If Ψ is a hyper entity space of the following form: $\Psi = E_1$ x..x E_p x $(E_{1,1}$ x..x $E_{1,q1})^*$ x..x $(E_{r,1}$ x..x$E_{r,qr})^*$, we define the scheme of Ψ as **Sch(Ψ)**:= $\{E_1$, ... ,E_p , $(E_{1,1}x..xE_{1,q1})^*$, ..., $(E_{r,1}x...xE_{r,qr})^*\}$ and **SchE(Ψ)**:= $\{E_1,...,E_p\}$. For primary entity space Ψ, Sch(Ψ)=SchE(Ψ). E_1, ..., E_p are called **singular dimensions**, $(E_{1,1}x..xE_{1,q1})^*$, ..., $(E_{r,1}x...xE_{r,qr})^*$, **compound dimensions**. Collectively, we call them just dimensions and represent them as $D_1,...,D_p,D_{p+1},...D_n$ with the following correspondence: Sch(Ψ) := $\{D_1$, ..., D_p, $D_{p+1},...,D_n\}$ and

$$\begin{aligned} D_1 &\equiv E_1, \\ &\cdots \\ D_p &\equiv E_p, \\ D_{p+1} &\equiv (E_{1,1}x\ldots xE_{1,q1})^*, \\ &\cdots \\ D_n &\equiv (E_{r,1}x\ldots xE_{r,qr})^*. \end{aligned}$$ ###

Finally, for each entity vector s_i in the hyper space: $s_i = <eid_{i,1}, ... , eid_{i,p}, \varsigma_{i,1},..., \varsigma_{i,r}>$, it suggests the following mapping for each s_i as:

$$s_i \text{ mapping } \begin{cases} s_i(D_1) \equiv s_i(E_1) \text{ ---> } eid_{i,1}, \\ \cdots \\ s_i(D_p) \equiv s_i(E_p) \text{ ---> } eid_{i,p}, \\ s_i(D_{p+1}) \equiv s_i((E_{1,1}x\ldots xE_{1,q1})^*) \text{ ---> } \varsigma_{i,1}, \\ \cdots \\ s_i(D_n) \equiv s_i((E_{r,1}x\ldots xE_{r,qr})^*) \text{ ---> } \varsigma_{i,r}. \end{cases}$$

#Definition 2.2.8: **Object, Singular Object, Compound Object**
Let's call all the values in s_i **objects**. $eid_{i,1},...,eid_{i,p}$ are **singular objects**; $\varsigma_{i,1}$, ..., $\varsigma_{i,r}$, **compound objects**. ###

#Example 2.2.3: Hyper Entity Space
Continuing our Example 1.2.1 of the Engineer, Work-on, Project, ERD, assume the current database state is the same as that in Example 1.2.1. The entity set Project implies the following entity space:

Project={<p101>,<p102>}.

The power set of entity space Project is:

Power(Project) = { Ø,{<p101>},{<p102>},{<p101>,<p102>}}.

We can define a hyper entity space (Figure 2.2.3): Ψ= Engineer x Power(Project) or Ψ = Engineer x Project*. In the current database, Ψ is

Ψ = { <Herman,Ø> , <Herman,{<p101>}> , <Herman,{<p102>}>,
<Herman,{<p101>,<p102>}>
<Pete, Ø> , <Pete, {<p101>}> , <Pete, {<p102>}>,
<Pete, {<p101>,<p102>}>
<Ross, Ø> , <Ross, {<p101>}> , <Ross, {<p102>}>,
<Ross, {<p101>,<p102>}>
<Joe, Ø> , <Joe, {<p101>}> , <Joe, {<p102>}>,
<Joe, {<p101>,<p102>}> }.

The schemes of Ψ are Sch(Ψ)={Engineer,Project*} and SchE(Ψ)={Engineer}. A state ς in Ψ as derived from the current database (we will talk about how to "compute" it later) is:

ς={ <Herman, {<p101>}> , <Ross, {<p101>, }> }.
<p102>

Assume an entity vector s_i in Ψ and s_i = <Ross, {<p101>, <p102>}>. This suggests the following mapping:

s_i(Engineer) ⟶ Ross
s_i(Project*) ⟶ {<p101>,<p102>} ###

2.C. Operators In The Entity Space Computation

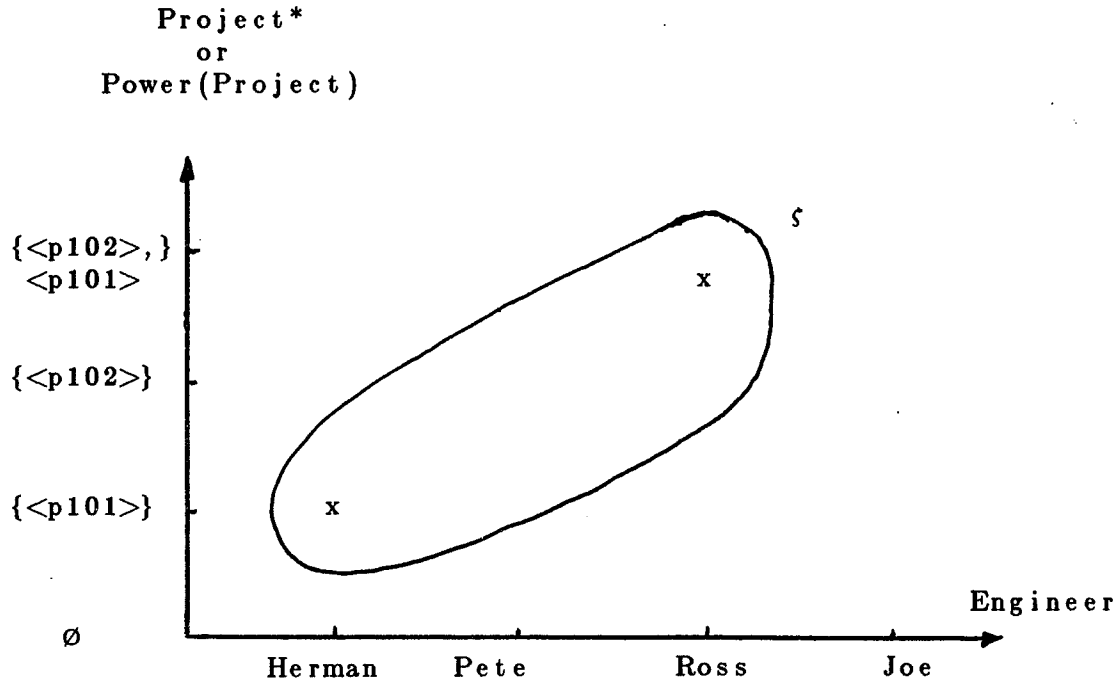
The Assignment Statement

The underlying structure of all the computations in ADL is the assignment statement.

A := *B* or
A := Expression.

There are two ways of using the assignment statements. The first is that we copy one data structure to another data structure so that the semantic meanings of the computations will more closely reflect the user's understanding of the

Figure 2.2.3. A Hyper Entity Space, Ψ = Engineer x Project*



Entity Space: Ψ = Engineer x Project*, or
Engineer x Power(Project).

Entity State: ς = { <Herman, {<p101>}>, }.
<Ross,{<p101>,<p102>}>

Entity Vector: <Ross, {<p101>,<p102>}>.

Sch(Ψ)={ Engineer, Project* }.

Order(Ψ)= 2.

$\varsigma \subseteq \Psi$, or $\varsigma \in$ Power(Ψ).

semantics. The second is that we compute a new data structure by the equation on the right hand side of the statement and assign the result to a data structure symbol on the left hand side of the statement. We will discuss this in detail after we have introduced all the operators.

The Operator Conventions of ADL

We use the functional (prefix) notations whenever possible to express operators of ADL; except where conventional usages of certain operators dominate, we use the infix notations. The operators are defined in the following format:

Operator $_{\text{Command}}$ (Operand1, Operand2).

The boldfaced letters differentiate an operator from its operands. The "Command" part of an operator is often less emphasized in the literature but is an important element in ADL's specification. By giving different commands to the same operator, we can stage a variety of different operations. We append suffixes "S", "D", "V", and "B" at the end of operator symbols to signify entity state operators, d_relation operators, value operators, and bag operators, respectively.

The Output Operator, Display

At the end of every ADL formula, we use the output operator to display the computation result.

Display (Data_structure).

The data structure can be either a state, a d_relation, a value, or a boolean. Assume the first two data structures are displayed in table formats. The boolean is always the output for a semantic constraint specification.

The Set Operators, $\cup$, -, $\cap$.

Since entity states are sets of entity vectors, we can apply set operators on two different states in the same entity space to obtain a new state.

#Definition 2.2.9: $\cup$, -

Two entity state $\varsigma_1, \varsigma_2 \subseteq \Psi$. $\varsigma_3 := \varsigma_1 \cup \varsigma_2$ iff

$s_i \in \varsigma_3 <===> (s_i \in \varsigma_1)$ or $(s_i \in \varsigma_2)$.

And $\varsigma_4 := \varsigma_1 - \varsigma_2$ iff

$s_i \in \varsigma_4 <===> (s_i \in \varsigma_1)$ and $(s_i \notin \varsigma_2)$). ###

Set intersection $\varsigma_1 \cap \varsigma_2$ can be defined as a macro of set differences, i.e.,

$\varsigma_5 := \varsigma_1 \cap \varsigma_2 := \varsigma_1 - (\varsigma_1 - \varsigma_2)$.

Since we use the set intersection more often than the set difference, it is worthwhile to treat set intersection as one of the fundamental operators.

The semantic meanings of these operators are equivalent to the logical "AND", "OR", and "NOT" in Predicate Logic.

The Join Operator, JoinS

JoinS is a function mapping two states of different entity spaces to a state in a new entity space.

#Definition 2.2.10: **JoinS, J_set.**

Assume $\varsigma_1 \subseteq \Psi_1$, $\varsigma_2 \subseteq \Psi_2$, and $Sch(\Psi_1) \cap Sch(\Psi_2) = \{D_{j1}, D_{j2}, .. D_{jn}\} <> \varnothing$. We can apply a join operation on ς_1 and ς_2 expressed as:

$\varsigma_3 :=$ **JoinS** $_{\{Dj1,Dj2,..,Djn\}} (\varsigma_1, \varsigma_2)$.

(1). $\{D_{j1}, ..., D_{jn}\}$ is the **J_set** of the **JoinS** operator which is equal to $Sch(\Psi_1) \cap Sch(\Psi_2)$.

(2). Let's look at the result of the **JoinS** operation. Let Ψ_3 be an entity space constructed from $Sch(\Psi_1) \cup Sch(\Psi_2)$, i.e., $Sch(\Psi_3) = Sch(\Psi_1) \cup Sch(\Psi_2)$. Assume $Sch(\Psi_1) - Sch(\Psi_2) := \{D_{11},...,D_{1p}\}$ and $Sch(\Psi_2) - Sch(\Psi_1) := \{D_{21},..,D_{2q}\}$. ς_3 is a state of Ψ_3 obtained as follows: If $\exists\, s_1 \in \varsigma_1$, $\exists\, s_2 \in \varsigma_3$ that satisfy:

$$s_1(D_{j1}) = s_2(D_{j1}),$$
$$\cdots$$
$$s_1(D_{jn}) = s_2(D_{jn}),$$

then there is an $s_3 \in \varsigma 3$ with :

$$s_3(D_{11}) = s_1(D_{11}),$$
$$\cdots$$
$$s_3(D_{1p}) = s_1(D_{1p}),$$
$$s_3(D_{j1}) = s_1(D_{j1}) = s_2(D_{j1}),$$
$$\cdots$$
$$s_3(D_{jn}) = s_1(D_{jn}) = s_2(D_{jn}),$$
$$s_3(D_{21}) = s_2(D_{21}),$$
$$\cdots$$
$$s_3(D_{2q}) = s_1(D_{2q}).$$ ###

$D_{j1}, D_{j2}, ..., D_{jn}$ are dimensions common to both Ψ_1 and Ψ_2. Since it is always the case that J_set is equal to the intersection of the $Sch(\Psi_1)$, $Sch(\Psi_2)$, a simplified notation of **JoinS** is $\varsigma_3 := \varsigma_1 \circ \varsigma_2$, assuming that we already know exactly the entity spaces to which ς_1, ς_2 belong.

We observe the following properties of **JoinS** operations:

(1). $Order(\Psi_3) < Order(\Psi_2) + Order(\Psi_1)$,

(2). $|\varsigma_3| \leq |\varsigma_1| * |\varsigma_2|$

(3). The **JoinS** operator is associative.

#Example 2.2.4: **JoinS** in the Hyper Entity Space

Assume $\varsigma_1 \subseteq \Psi_1$, $\Psi_1 = A \times (B \times C)^*$.

$\varsigma_1 = \{ <a1, \{<b3, c1>, \}>, <a2, \{<b1, c2>\}>, <a3, \{<b2, c2>\}>\}$.
$<b1, c4>$

And $\varsigma_2 \subseteq \Psi_2$, Ψ_2=D x (B x C)*.

ς_2= { <d1, { < b3, c1>,} > ,
< b1, c4>
<d4, { < b5, c2> } > ,
<d2, { < b2, c2> } > ;
<d5, { < b2, c2> } > }.

If ς_3 := **JoinS** $_{\{(BxC)^*\}}$ (ς_1 , ς_2), $\varsigma_3 \subseteq \Psi_3$, Ψ_3=A x D x (B x C)*. The result of the computation is:

ς_3:={ <a1, d1, { < b3, c1> } > ,
< b1, c4>
<a3, d2, { < b2, c2> } > ,
<a3, d5, { < b2, c2> } > } ###

The Select Operator, SelectS

The **SelectS** operator is used to select vectors in a given entity state, ς_1, to form form another state, ς_2. ς_2 is thus a sub-state of ς_1. All the entity vectors in the sub-state satisfy a predicate formula as specified by the **SelectS** operator. Therefore, we need to define what is a predicate formula first.

#Definition 2.2.11: **Variable, Valid Atom, Valid Formula**

(1). If $E_1,...,E_p$, $(E_{1,1}x..xE_{1,q1})^*$, ..,$(E_{r,1}x..xE_{r,qr})^* \in$ Sch(Ψ), we use E_1, ... , E_p, $(E_{1,1}x..xE_{1,q1})^c$, .., $(E_{r,1}x..xE_{r,qr})^c$ to represent **variables** in a predicate formula. Each variable stands for an element in an entity vector. For those singular dimensions, $A_1,A_2,..,A_m$ are attributes in the corresponding entity sets. Therefore, $E_1.A_1$ means the attribute value A_1 of entity variable E_1.

(2). The following are **valid atoms** -

(i) $E_i.A_j \; \theta$ c ,

(ii) $E_i.A_j \; \theta \; E_h.A_k$.

θ can be <,>,≤,≥,=,<>. c is a constant in Dom(A_j). Dom(A_j) and Dom(A_k) are compatible in data types for the comparison.

(iii) $|(E_{i,1}x...xE_{i,qi})^c| \; \theta$ d.

$(E_{i,1}x...xE_{i,qi})^c$ represents a compound object variable (a primary entity state) of an entity vector. $|(E_{i,1}x...xE_{i,qi})^c|$ means the number of elements (vectors) in the compound object, $(E_{i,1}x...xE_{i,qi})^c$. d is a constant.

(iv) $(E_{i,1}x...xE_{i,qi})^c \; \delta$ e.

δ can be $\subseteq,\supseteq,\subset,\supset$,and =s (set equal). e is a constant entity state (a set of primary entity vectors) in the primary entity space.

(3). **Valid formulas** are constructed as follows -

(i) An atom is a formula.

(ii) If f1 and f2 are formulas, f1 Λ f2, f1 V f2, ¬f1 are also formulas.

(iii) Nothing else is a formula. ###

A similar definition for valid formulas in ER-calculus can be found in [Atze83],[Tabo83b]. However, it is important to note that when we want to make assertions about a compound object in an entity vector, we need to treat the object as a whole. We cannot look into the element in the compound object and inquire about its attribute values.

#Definition 2.2.12: **SelectS**
Let $\varsigma_1 \subseteq \Psi_1$, we can apply **SelectS** on ς_1 expressed as:
ς_2:=**SelectS** $_{P(E1.A1,..,Ep.Ap,\ (E1,1x..xE1,q1)^c,..,(Er,1x..xEr,qr)^c)}$ (ς_1),in which
(1). $\varsigma_2 \subseteq \Psi_1$. ς_2 is of the same entity space as ς_1.
(2). $P(E_1.A_1,..,(E_{r,1}x..xE_{r,qr})^c)$ is a valid formula as defined above.
(3). ς_2 is obtained as the set of entity vectors that satisfy the predicate formula:
$\{<eid_{i,1},..,eid_{i,p},\varsigma_{i,1},..,\varsigma_{i,r}> | <eid_{i,1},..,eid_{i,p},\varsigma i,1,..,\varsigma i,r> \in \varsigma_1 \wedge v_1 := eid_{i,1}(A_1)$,.., v_m:=$eid_{i,m}(A_m) \wedge P(v_1,...,v_m,\varsigma_{i,1},..,\varsigma_{i,r}) = TRUE\}$ v_i's are attribute values of the corresponding entities. ###

We observe the following properties from the above definition:
(1). Both ς_1, $\varsigma_2 \subseteq \Psi_1$, so $Sch(\varsigma_1)=Sch(\varsigma_2)$.
(2). $\varsigma_2 \subseteq \varsigma_1$ and $|\varsigma_2| \leq |\varsigma_1|$

Most often the selection operators are used in the lower order entity spaces to minimize the computation complexity at the early stage of computation. It is similar to query optimization in relational algebra where selections should be performed as early as possible in the evaluation [Ullm82, Maie84].

Since, frequently, selections are performed on first order entity states, we use the following shorthand form: $(A)_{P(E1.A1,...,Em.Am)}$. It is equivalent to : **SelectS** $P(E1.A1,..,Em.Am)$ (A).

#Example 2.2.5: **SelectS** in the Hyper Entity space
Assume $\varsigma 2 \subseteq \Psi 2$, and $\Psi 2 = A \times (B \times C)^*$.
$\varsigma 2$={ <a1,{<b3, c1>,}>, <a2, {<b1, c2>}>, <a3, {<b2, c2>}>}
<b1, c4>
If ς_3:= **SelectS** $_{(B \times C)^c \models 2}$ (ς_2), and
ς_4:= **SelectS** $_{(BxC)^c \supseteq \{<b1,c2>\}}$ (ς_2), we would get
ς_3 := {<a1, {<b3, c1>,}>}, ς_4 :={<a2, {<b1, c2>}>}
<b1, c4>
Note how we use $(BxC)^c$ as a compound object variable in the predicate formulas of the **SelectS** operators. The assertions made on $(BxC)^c$ are always assertions about the compound object as a whole, not the individual element in the compound object. ###

The Project Operator, ProjectS

The **ProjectS** operator maps a state in a high order entity space to one in a lower order entity space.

#Definition 2.2.13: **ProjectS, P_set**

If $\varsigma_1 \subseteq \Psi_1$, we can apply the **ProjectS** operator on ς_1 expressed as: ς_2:= **ProjectS** $_{\{Dp1,Dp2,...,Dpm\}}$ (ς_1), in which

(1). $\{D_{p1},D_{p2},...,D_{pm}\}$ is called the P_set (the projection set). P_set $\subset$ Sch(Ψ_1). Unlike the J_set, the P_set cannot be implied and need to be explicitly given.

(2). From the P_set, we can construct an entity space Ψ_2 and Sch(Ψ_2)=P_set.

(3). We obtain ς_2 as follows: For $\forall\, v_i \in \varsigma_1$, we can have $s_i \in \varsigma_2$ and

$$s_i(D_{p1}) = v_i(D_{p1}),$$
$$s_i(D_{p2}) = v_i(D_{p2}),$$
$$\ldots$$
$$s_i(D_{pm}) = v_i(D_{pm}).$$ ###

We observe the following properties: (1). Order(Ψ_1)-1 $\geq$ Order(Ψ_2), and (2). $|\varsigma_1| \geq |\varsigma_2|$

#Example 2.2.6: **ProjectS** in the Hyper Entity space

Assume $\varsigma_1 \subseteq \Psi_1$, Ψ_1= A x B* x C*.

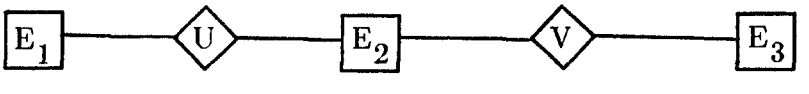
ς_1= { <a1, { <b3>, }, { <c1>, } > ,
<b1> <c4>
<a2, { <b1> }, { <c2> } > ,
<a3, { <b2> }, { <c2> } > ,
<a1, { <b3>, }, { <c3> } > }.
<b1>

ς_2 := **ProjectS** $_{\{A,B^*\}}$ (ς_1). So $\varsigma_2 \subseteq \Psi_2$, Ψ_2 = A x B*. The result of the computation is:

ς_2= { <a1, { <b3>, } > ,
<b1>
<a2, { <b1> } > ,
<a3, { <b2> } > } ###

The combination of the **JoinS** and the **ProjectS** operators plays a key role in sorting out relationships that are "expandable" to high order entity spaces. It provides a way to select entities according to their semantic associations with other entities instead of according to the their attribute values (as in **SelectS**). For example, if

Ψ_1=E_1,
Ψ_2=E_1 x E_2,
Ψ_3=E_1 x E_2 x E_3.



(Note that the equal signs in these expressions only denote the dimensions in

the various entity spaces. They do not imply any computation of the Cartesian product.)

$\varsigma_1 \subseteq \Psi_1$, and $\varsigma_1 := E_1$,

$\varsigma_2 \subseteq \Psi_1$, and $\varsigma_2 :=$ **ProjectS** $_{\{E1\}}$ $(E_1 \circ U \circ E_2)$,

$\varsigma_3 \subseteq \Psi_1$, and $\varsigma_3 :=$ **ProjectS** $_{\{E1\}}$ $(E_1 \circ U \circ E_2 \circ S \circ E_3)$,

Semantically, ς_1 includes all the entities in E_1. ς_2 consists of entities in E_1 that are associated to entities in E_2; and ς_3 includes those that are associated to an entity in E_2 and an entity in E_3. Therefore, $\varsigma_3 \subseteq \varsigma_2 \subseteq \varsigma_1$. Note that $(\varsigma_1 - \varsigma_2)$ are those entities in E_1 that have no association with any entity in E_2.

The Group By Operator, Group_byS

The **Group_byS** operator is the single most important operator in the hyper entity space since it constructs states in the hyper entity spaces from states in the primary entity spaces.

#Definition 2.2.14: **Group_byS**, **G_set**

If $\varsigma_1 \subseteq \Psi_1$ and Ψ_1 is a primary entity space, we can apply the **Group_byS** operator on ς_1 in the following format: $\varsigma_2 :=$ **Group_byS** $_{\{Eg1,...,Egp\}}$ (ς_1) or $\varsigma_2 :=$ **Group_byS** * (ς_1).

(1). $\{E_{g1},...,E_{gp}\}$ is called the **G_set**. It is part of the **Group_byS** command and must be specified explicitly by the user. When G_set is $\emptyset$, we use the second format for the **Group_byS** operation.

(2). If G_set = $\{E_{g1},...,E_{gp}\} <> \emptyset$ and $\Psi_1 = Eg_1 \times ... \times E_{gp} \times E_1 \times ... \times E_n$, then $\Psi_2 = E_{g1} \times ... \times E_{gp} \times (E_1 \times ... \times E_n)^*$. If G_set = $\emptyset$ and $\Psi_1 = E_1 \times ... \times E_n$, then $\Psi_2 = (E_1 \times ... \times E_n)^*$. Note that G_set $\subseteq$ Sch(Ψ_1) and G_set = SchE(Ψ_2). ς_2 is a state of Ψ_2.

(3). To obtain ς_2, the **Group_byS** operator proceeds as follows:

Case 1. G_set <> $\emptyset$

(i) Partition ς_1 into disjoint sub-states ς_{11}, ..., ς_{1n} (or classes). In each class ς_{1i}, $\forall$ $s_i \in \varsigma_{1i}$, has the same value of $<s_i(E_{g1}),...,s_i(E_{gp})>$.

(ii) Each disjoint class of $\varsigma_{11},...,\varsigma_{1n}$ corresponds to an entity vector in ς_2 derived as follows:

Let ς_{1i} be a class, s_i any vector in ς_{1i}, and t_i the corresponding vector in ς_2. First, $\varsigma_{1i}^* :=$ **ProjectS** $_{(Sch(\Psi 1)\text{-}G\_set)}(\varsigma_{1i})$. Then, $t_i := <s_i(E_{g1}),...,s_i(E_{gp}),\varsigma_{1i}^*>$. ς_2 is simply the set of all such t_i's. Each t_i is indexed by $<s_i(E_{g1}) ... s_i(E_{gp})>$, the grouping index. The ς_{1i}^* is the compound object.

Case 2. G_set = $\emptyset$

Let ς_1 be a primary entity state in $E_1 \times ... \times E_n$. If $\varsigma_2 :=$ **Group_byS** * (ς_1), then $\varsigma_2 \subseteq \Psi_2$, $\Psi_2 = (E_1 \times ... \times E_n)^*$, and $\varsigma_2 := \{<\varsigma_1>\}$. ###

From the above definition, it is easy to prove that (1). $|\varsigma_2| \leq |\varsigma_1|$ and (2). order(Ψ_2)=|G_set|+1.

#Example 2.2.7: The **Group_byS** Operator

Let Ψ_1=A x B x C, $\varsigma_1 \subseteq \Psi_1$. If ς_2= **Group_byS** $_{\{A\}}$ (ς_1), then Ψ_2=A x (B x C)*, $\varsigma_2 \subseteq \Psi_2$. We may assume that ς_1 at the instance level is:

ς_1 :={ <a1, b3, c1> ,<a2, b1, c2> ,
<a1, b1, c4> ,<a3, b2, c2> }

According to the partition rule, we can partition ς_1 into the following disjoint classes:

ς_{11} := {<a1, b3, c1>,<a1, b1, c4>}
ς_{12} := {<a2, b1, c2>}
ς_{13} := {<a3, b2, c2>}

The corresponding vectors in ς_2 are :

ς_2={<a1, {<b3, c1>,}>, <a2, {<b1, c2>}>, <a3, {<b2, c2>}>}
<b1, c4>

Note that in each entity vector the compound object is indexed by the grouping index, e.g. {<b3,c2>,<b1,c4>} is indexed by <a1>.

If ς_3:= **Group_byS** * (ς_1), then

ς_3 := {<ς_1>} := {< {<a1, b3, c1> ,<a2, b1, c2> ,
<a1, b1, c4> ,<a3, b2, c2>} >}. ###

The **Group_byS** operator carries a fundamental semantic meaning, i.e., grouping. The entities are classified by their common associations with a particular entity vector (the G_set). We call all the elements in the G_set the grouping index. After grouping, hyper entity state in the hyper entity space is formed.

The Value Operator, CountS

The value operator transforms a state into a value. There is only one such operator in the entity space, i.e. **CountS**. It counts the number of vectors in a state. Throughout this paper we use $\mu,\mu_1,...,\mu_n$ to represent values.

#Definition 2.2.15: **CountS**

μ:= **CountS**(ς) means that μ equals the number of vectors in ς (an integer). ###

We may also use the following shorthand form to represent **CountS**: ♯

3. DATA STRUCTURES AND OPERATORS IN THE DATA SPACE

While in the entity space, most calculations are to derive the smallest entity state that reflects the semantic associations of entities. However, many user's queries and semantic constraints require further calculations at the data elements level. Thus, we will look at the data structure and operators in the data space for such computations.

3.A. Data Structure In The Data Space

We start with the basic data structure in the data space, the d_relation.

#Definition 2.3.1: **Relation Scheme, D_relation, and Tuple, Data Space,** $\#\tau$. Assume $F_1, ..., F_p$ are attribute names. Corresponding to each attribute name F_j is a value set $Dom(F_j)$, $1 \leq j \leq p$, called the domain of F_j. A d_relation τ is a bag of tuples $\{(v_{i1},...,v_{ip}) | v_{i1} \in Dom(F_1), ..., v_{ip} \in Dom(F_p)\}$ and each $t_i = (v_{i1},...,v_{ip})$ is a tuple of the d_relation. The relation scheme of τ, $Sch(\tau)$, is a finite set of attribute names $\{F_1,...,F_p\}$. A data space is simply the Cartesian product of the value domains of τ, i.e. $Dom(F_1) \times Dom(F_2) \times ... \times Dom(F_p)$. $\#\tau$ is the number of tuples in τ. ###

Attribute names are sometimes simply called attributes. Each tuple t_i suggests a mapping from $Sch(\tau)$ to $Dom(F_1) \cup ... \cup Dom(F_p)$ with the restriction that for each mapping $t_i \in \tau$, $t_i(F_j)$ must be in $Dom(F_j)$, $1 \leq j \leq p$.

$$t_i \text{ mapping } \begin{cases} t_i(F_1) \text{ ---> } v_{i1}, \\ .... \\ t_i(F_p) \text{ ---> } v_{ip}. \end{cases}$$

We further assume τ is a bag, i.e. it allows duplicates in the set. Duplicates have the same mappings from $Sch(\tau)$ to $Dom(F_j)$s but coexist in τ. Assume that sets like {1, 2, 4, .., n}, {a, b, .., z} are singular sets which are sets of integers, reals, and character strings. A singular set can be the domain of an attribute. But more important is that the sets of bags from the singular sets can also be the domain of an attribute. Therefore, the d_relation can be a non-first normal form relation in which the components of a tuple can be bags. For those attributes whose domains are the sets of bags we append a "*" at the end of their names.

#Example 2.3.1: Non-first normal form d_relation.
$Sch(\tau)$= {A,B*}. Dom(B*) is the set of all bags from 1 to 100. Dom(A) is the set of character strings.

τ	A	B*
t_1	Joe	{10,2,7,7}
t_2	Betty	{4,4}
t_3	Mike	{5,1}

t_1=(Joe, {10,2,7,7}), t_2=(Betty, {4,4}), t_3=(Mike, {5,1}). For t_1 we can have the following mapping:

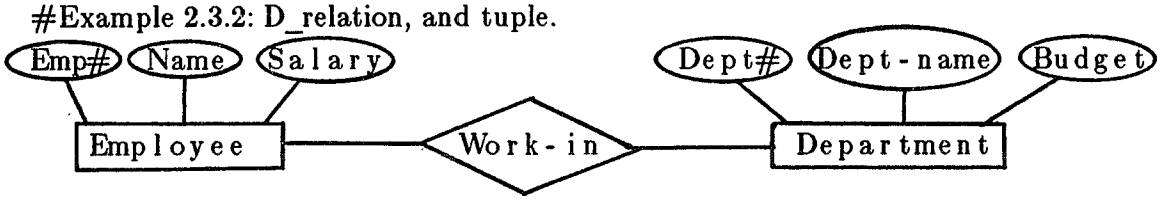
$t_1(A) \longrightarrow$ Joe,
$t_1(B^*) \longrightarrow \{10,2,7,7\}$. ###

The reader who is familiar with relational data model should find similarities in the above definitions and those in the relational data model; only now we allow

duplicates to exist.

In the entity space computations, we have successfully identified those entities that are related to the user's query or the semantic constraint. Now, we are in a position to probe into the attribute values of those entities. Certainly, we are not interested in all the attribute values of all those entities. Therefore, the d_relation is a derived or computed relation from the current database state. The attributes of the d_relation are those of the involved entity sets in the computed entity state.

#Example 2.3.2: D_relation, and tuple.



Query: Find all the employees who work in the toy department, their name, their salary, and their department budget.

First, we do an entity space computation by finding a state in the entity space Employee x Department.

$\varsigma := Employee \circ Work\text{-}in \circ (Department)_{\text{Dept-name='toy'}}$

We are now interested in building a d_relation, τ, whose scheme is {Emp-name, Emp-salary, Dept-budget} and extract the values of the attributes from the entity sets Employee and Department. A tuple in τ may be

t_i = (Joe, 20,000, 80,000)

It suggests the following mapping:

$$\begin{cases} t_i(\text{Emp-name}) \longrightarrow \text{Joe}, \\ t_i(\text{Emp-salary}) \longrightarrow 20{,}000, \\ t_i(\text{Dept-budget}) \longrightarrow 80{,}000. \end{cases}$$

###

Note that in the entity space we deal with entities in their entirety. In the data space, however, we break the entity apart, look at its data values, and operate on the data values. In Example 2.3.2, the d_relation τ will take its first two attributes from the attributes of Employee and its third attribute from the attributes of Project.

After we have defined what is a d_relation, let's look at how to derive it. All the d_relations are obtained by using the **Extract** operator to extract attributes from a state in an entity space. Therefore, **Extract** is a data type transformation operator which transforms a state into a d_relation. From then on, the computations are shifted to data space computations.

3.B. Operators In The Data Space Computation

The Transformation Operator: Extract

#Definition 2.3.2: **Extract**

τ:= **Extract** $_{\{F1<\text{-}E1.A1,..,Fp<\text{-}Ep.Ap\}}$ $^{(\varsigma)}$

$\Psi = E_1 \times ... \times E_p$, $\varsigma \subseteq \Psi$, $Sch(\tau) = \{F_1,...,F_p\}$. $\forall E_i \in Sch(\Psi)$. A_i is the attribute of E_i. In essence, we are renaming the attributes of entity sets involved in ς to attributes of the d_relation τ.:

(1). We impose the following preconditions on **Extract**. ς is either a primary entity state or a hyper entity state. If it is a hyper entity state, the compound dimension of the hyper entity space can only be the power set of a one-entity-set entity space. That is, hyper entity spaces with dimension (A)*, or (B)* are eligible for the **Extract** operator; but those with dimension (AxB)*, or (AxBxC)* are not.

(2). We compute each $t_i \in \tau$ from each $s_i \in \varsigma$ and $s_i = <o_{i,1},..,o_{i,m}>$. $o_{i,1},...o_{i,m}$ are either singular or compound objects. Therefore, we have two different cases of mapping from the objects in s_i to the values in t_i.

Case 1. If $o_{i,j}$ is a singular object, then $o_{i,j} = eid_{i,j}$

$t_i(F_j) = eid_{i,j}(A_j)$.

The $Dom(F_j)$ is a singular set. F_j is said to be a value attribute.

Case 2. If the $o_{i,j}$ is a compound object, due to the restriction stated in (ii),

$o_{i,j} = \{<eid_{i,j,1}>, ..., <eid_{i,j,k}>\}$

$t_i(F_j) = \{ eid_{i,j,1}(A_j),...,eid_{i,j,k}(A_j) \}$.

$Dom(F_j)$ now corresponds to a power set of a singular set. F_j in this case is a compound attribute. ###

From the above definition, $|\tau| = |\varsigma| = n$, assuming there are n elements in ς. Therefore, **Extract** will not remove duplicate tuples in the resulting d_relation.

Note that the E_i's shown on the extraction list are only subsets of $Sch(\Psi)$. The attribute A_i's are only parts of the attributes in the entity sets. The point is to extract the minimal set of data that will be used in later computations.

#Example 2.3.3: Continuation of Example 2.3.2

τ := **Extract** $_{\{\text{Emp-name} <\text{-} \text{Employee.name, Emp-salary} <\text{-}}$

Employee.salary, Dept-budget <- Department.budget$\}$ $^{(\varsigma)}$.

Therefore, $Sch(\tau)$ = { Emp-name , Emp-salary, Dept-budget} and τ can be assumed as follows:

Emp-name	Emp-salary	Dept-budget
Joe	20,000	80,000
Herman	12,000	80,000
Ross	15,000	80,000 ###

#Example 2.3.4: Extracting from a hyper entity state.

Query: Find all the salary records in each department. We can proceed with the following formulation of the query:

ς_1:= *Department* ° *Work-In* ° *Employee*,

ς_2:= **Group_byS** {Department} (ς_1),

Note that ς_2 is a state in the hyper entity space Ψ_2=Department x Employee* where Employee* is a compound dimension of the hyper entity space.

τ_1:= **Extract** {Dept-name<-Department.name,
Salary*<-Employee*.salary } (ς_2),
Display (τ_1).

Possible values for τ_1 can be:

Dept-name	Salary*
Toy	{25000, 32000, 25000}
Shoe	{15000, 18000}

Salary* is now a compound attribute. ###

The Extend Operator, ExtendD

The extend operator extends the relation scheme by computing a new attribute of the d_relation from existing attributes.

#Definition 2.3.3: **ExtendD**

τ_2:= **ExtendD** {Fnew:= f(F1,...,Fm)} (τ_1).

(1). Assume Sch(τ_1)= {F_1, ..., F_m, F_{m+1},..., F_p}, Sch(τ_2) = {F_1 ,..., F_m, F_{m+1} ,..., F_p, F_{new}}.

(2). f(F_1,...,F_m) is a numerical function composed of +, -, *, /, **SumB**, **CountB**, **AvgB**, **MinB**, or **MaxB** with F_1 ,...,F_m as its arguments.

(3). For each $s_i \in \tau_1$, s_i=(v_1, ..., v_m, v_{m+1}, ...,v_p), and v_{new} := f(v_1,...,v_m), we can find a corresponding $t_i \in \tau_2$ with: t_i = (v_1,...,v_m,v_{m+1},...,v_p,v_{new}). ###

A set of functions that work on the compound attributes of a d_relation is worthy of mentioning here. These functions are **SumB**, **CountB**, **AvgB**, **MaxB**, **MinB** and are defined as follows (A "bag" means that the value of a compound attribute is a bag in a d_relation tuple):

SumB(A^c): Compute the sum of a bag.

CountB(A^c): Compute the number of elements in a bag.

AvgB(A^c):Compute the average of all the numbers in a bag.

MinB(A^c): Compute the minimum of a bag.

MaxB(A^c): Compute the maximum of a bag.

Note that A* is the attribute name; but A^c is its variable value of a d_relation tuple.

#Example 2.3.5: Continuation of Example 2.3.3.

τ_2:= **ExtendD** {Emp-Raise:=0.1 x Emp-salary} (τ).

Sch(τ_2) = {Emp-name, Emp-salary, Dept-budget, Emp-raise}. So τ_2 is (based on Example 2.3.3 data):

Emp-name	Emp-salary	Dept-budget	Emp-raise	
Joe	20,000	80,000	2,000	
Herman	12,000	80,000	1,200	
Ross	15,000	80,000	1,500	###

#Example 2.3.6: Continuation of Example 2.3.4.

Query: Find out the average salary of each department. Using the data provided in Example 2.3.4.

τ_3:= **ExtendD** {Avg-salary:= **AvgB**(Salary^c)} (τ_1),

τ_4:= **ProjectD** {Dept-name, Avg-salary} (τ_3).

/***ProjectD** will be defined later */

τ_4

Dept-name	Avg-salary
toy	27,333
shoe	16,500

###

The Selection Operator, SelectD

The selection operator will select tuples satisfying certain conditions in a d_relation to form a new d_relation.

#Definition 2.3.4: **SelectD**

τ_2:= **SelectD** $_{P(F1,...,Fm)}$ (τ_1).

(1). $Sch(\tau_1)=Sch(\tau_2)= \{F_1,...,F_p\}$.

(2). $P(F_1,...,F_m)$ is a predicate formula involving the attribute values of each tuple in τ_2. Assume $F_1,...,F_m$ stand for the value variables of each tuple. The predicate formula is built from valid atoms.

Valid Atoms:

(i) $F_i \ \theta$ c. F_i is a value variable of tuple t. c is a constant. θ is $<$, $>$, $\leq$, $\geq$, or $=$.

(ii) $F_i \ \theta \ F_j$. F_i and F_j are different value variables of tuple t. θ is $<$, $>$, $\leq$, $\geq$, or $=$.

Valid Formula: (The same as that in Definition 2.2.11)

(3). τ_2 is calculated as follows: Set $\tau 2$ to $\emptyset$. For each $t_1 \in \tau_1$,t_1= (v_1,..., v_m, v_{m+1}, ..., v_p). If $P(v_1,...,v_m)$ = TRUE, we add t_1 to τ_2. ###

The Projection Operator, ProjectD

The projection operator is used to reduce the attributes (columns) in a d_relation. It is similar to the **ProjectS** operator in the entity space but has quite different semantic meanings. **ProjectS** is used to project a high order entity space to a low order entity space. Therefore, duplicate relationships mean the same thing and are removed accordingly. Here, however, the **ProjectD** merely reduces the number of columns in a d_relation. Each tuple in the d_relation represents a unique entity in the database holding values as indicated by the tuple.

Duplicates in d_relations mean different entities holding the same value by mere coincidence, so duplicate tuples are not removed even after the projection.

It is usually the case that we first compute a new attribute by using **ExtendD** and, then, we eliminate useless attributes by using the **ProjectD** operator.

#Definition 2.3.5: **ProjectD**

τ_2:= **ProjectD** $_{\{F1,...,Fm\}}$ (τ_1).

(1). $Sch(\tau_2)=\{F_1,...,F_m\}$, $Sch(\tau_1)=\{F_1,...,F_m,F_{m+1},...,F_p\}$. $Sch(\tau_2) \subseteq Sch(\tau_1)$.

(2). τ_2 is computed as follows: Set $\tau_2=\emptyset$. For each $t_1 \in \tau_1$, create $t_2 = (v_1,...,v_m)$ with $v_1=t_1(F_1)$,..., $v_m=t_1(F_m)$. Add t_2 to τ_2.

###

The Cartesian Product Operator, ProductD

One objective of the ERM is to give a detailed description of relationships among entities so that users' queries and semantic constraints can be defined by using existing associations (or relationship sets) among various entity sets. This is an idealistic assumption upon which the entity space computations are based. Therefore, so far the "Cartesian Product" operation has been carefully avoided in the entity space computation. The Cartesian product will simply match anything in one set to anything in the other set no matter whether such matches have any semantic meaning or not. It is the most expensive operation of all the operators in relation algebra [Ullm82].

But at times, we want to associate data together because of their compatibility in data types. The θ-join in extended relational algebra is an example of this kind of operation [Merr84]. Such associations may or may not be modeled distinctly as relationships between the entity sets in the predefined ERM. Only when there is no existing means to associate two data sets together do we use the Cartesian product of the two data sets.

Note that the Cartesian product operator introduced here is only for the completeness of the Application Data Language as a high level query language. For semantic constraint modeling, the "Cartesian Product" can be avoided altogether by formulating the problem using aggregate value comparisons (see APPENDIX C for more detail) [Bern80]. Therefore, we only present the operator as part of the query language but not as part of the semantic constraint specification language.

#Definition 2.3.6: **ProductD**

τ_3:= **ProductD** (τ_1, τ_2), in which

(1) $Sch(\tau_3) = Sch(\tau_2) \cup Sch(\tau_1)$.

(2) τ_3 is a d_relation formed as the Cartesian product of τ_1, τ_2.

###

The Value Operators, SumD, CountD, AvgD, MinD, MaxD, ValueD.

The value operators transform a d_relation to a numerical value.

#Definition 2.3.7: **SumD, CountD, AvgD, MinD, MaxD.**

Assume μ is a value (integer or real). τ is a d_relation with Sch(τ)={A}, a single attribute d_relation. Dom(A) is a set of integers or reals. Then, we can apply the following value operators to compute a value from this d_relation.

μ= **SumD** (τ),
μ= **CountD** (τ),
μ= **AvgD** (τ),
μ= **MinD** (τ),
μ= **MaxD** (τ). ###

When applying these operators, one must remember that the d_relation has to be a single attribute relation.

Often, we just want to have a single value extracted from the entire database for further computations. For example, τ:= {(100)} is a d_relation; and we need to transform τ into a single value. We can use **SumD** to do it. But **SumD** does not give us any warning if τ:= {(40),(60)}. **SumD** will still give us the same answer as if the result comes from a single tuple. It may be worthwhile to define a special operator for this type of data structure transformation. It will give us warning if more than one tuple exist in the d_relation.

#Definition 2.3.8: **ValueD**

If τ is a d_relation with a single attribute and a single tuple, we can define the following operator, **ValueD** as: if τ={(c)}, then μ := **ValueD** (τ) := c. ###

3.C. Operators For Value Computations

Numerical computations represent an important part of engineering calculations. All previous computations in the entity space and in the data space are set-theoretical computations. In the entity space, the compuation involves the set of entity vectors (state); in the data space, it involves of tuples (d_relation). In a commercial database we may display a computed d_relation as an answer to the user's query. In engineering databases, however, we may want to use the values derived from the entity space and the data space to do further computations. Actually, once data are in the value form, they can be subjected to any numerical computation. But, here we only consider four basic value operators, i.e., $\mu3$:=$\mu1$+$\mu2$, $\mu3$:=$\mu1$-$\mu2$, $\mu3$:=$\mu1$*$\mu2$, $\mu3$:=$\mu1$/$\mu2$.

3.D. An Operator Macro Facility

There are certain combinations of primitive operators that are heavily used in query and constraint formulations. It is worthwhile to define them as macros that represent certain sequences of primitive operations. Here, we define such a macro facility as a way to simplify the notation of query and constraint formulations.

Macro **Macro_name** $_{\text{command1,...,commandm}}$ $(Op_1,...,Op_n)$=

$\mathbf{P_1}$ $_{\text{command1}}(Op_1,..(((..\mathbf{P_m}$ $_{\text{commandm}}$ $(Op_n)))))$

Macro_name is the name of the Macro operator. $Command_1,..,$ $command_m$ are commands. $Op_1,...,Op_n$ are operands. $\mathbf{P_1},...,\mathbf{P_m}$ are primitive operators ,i.e. **JoinS**, **SelectS**, **ProjectS**, **Group_byS**.

#Example 2.3.7: Set intersection as a macro

Macro $\cap$ $(\varsigma_1, \varsigma_2) = \varsigma_1 - (\varsigma_1 - \varsigma_2)$. ###

#Example 2.3.8: Relation "divide" as a macro

We can define the relation "divide" operator as a macro with the following preconditions:

Assume $\varsigma 1 \subseteq \Psi 1$, $\varsigma 2 \subseteq \Psi 2$,

(1). $\Psi 1$, $\Psi 2$ are primary entity spaces.

(2). $Sch(\Psi 2) \subseteq Sch(\Psi 1)$.

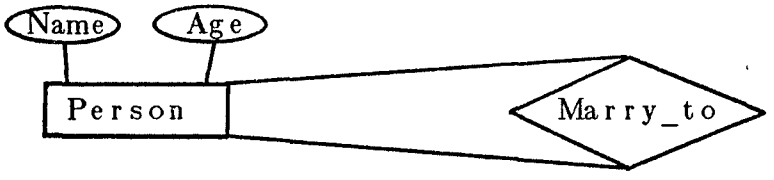
If $\varsigma 1$, $\varsigma 2$ satisfy 1. and 2., we can now define the divide operation as follows:

Macro **Divide** $(\varsigma 1, \varsigma 2)$ = **ProjectS** $_{(Sch(\Psi 1)\text{-}Sch(\Psi 2))}$ **(JoinS (Group_byS** $_{(Sch(\Psi 1)\text{-}Sch(\Psi 2))}$ $(\varsigma 1)$, **Group_byS** * $(\varsigma 2)$)). ###

3.E. Semantic Meaning Of The Assignment Statements In ADL

As we have mentioned earlier there are two meanings for the assignment statements in ADL. The first is to copy one data structure to another so that the computation procedures will more closely reflect the user's understanding of the semantics. The second is that we compute a new data structure by an equation and assign the result to a data structure symbol. From the examples we have seen so far, we can easily identify the second meaning of the assignment statements. Now let's look at how an assignment statement is used to convey the semantic meaning of computation procedures.

#Example 2.3.9hAssignment Statements in ADL



Assume in the "Marry_to" relationship we put a husband in the first argument and his wife, in the second. Query: Give the Johnson's age as well as his wife's age.

ADL: ς:=(*Person*)$_{\text{Person.name='Johnson'}}{}^{\circ}_{Marry\_to\ ^{\circ}Person}$,

τ:=**Extract** {Hus-Age<-Person.age, Wife-Age<-Person.age} (ς),

Display(τ).

We can see there is ambiguity in the formulation of the query since in the **Extract** statement it is not really clear whether the first person.age and the second one really mean the same thing. It also gives the user the burden of remembering the semantic meaning of symbols by their positions in the formula, e.g. the first Person relates to husband; the second Person relates to the wife in ς.

Therefore, it may be better to create identical data structures for the purpose of clarifying the interpretations from semantics to ADL formula. In the physical implementation, such identical data structures are only virtual and their values can be found by pointers to the original structure. A better formulation is:

ADL: *Husband*:= *Person*,

Wife:= *Person*,

ς:= (*Husband*)$_{\text{Husband.name='Johnson'}}{}^{\circ}_{Marry\_to\ ^{\circ}Wife}$,

τ:= **Extract** {Hus-Age<-Husband.age, Wife-Age<-Wife.age} (ς),

Display(τ).

Semantically, the latter one more closely reflects the user's understanding of the query. Also note that *Husband* and *Wife* can be virtual data structures that point to the same data structure *Person*.

4. SUMMARY

This paper proposes a data manipulation language ADL that is based on a two-space approach. In the entity space, we carry out object level computation to find the smallest entity state that contains all the entities needed for user queries or constraints. Then, in the data space, we carry out computations at the data element level to derive new relations. The syntax of the Applicative Data Language (ADL) is given in APPENDIX A. We proved ADL is relational complete in APPENDIX B. Examples of ADL as a query language are given in APPENDIX C.

The most important data structure in the entity space is the state which corresponds to a set of entity vectors. Each entity vector represents a valid association of entities implied by the current database state. Through a series of relationship computations using $\cup$, $\cap$, -, **JoinS**, **SelectS**, **ProjectS**, and **Group_byS**, we can find a computed state for the user's query or the semantic constraints.

A hyper entity state contains entity vectors whose values can be compound objects. The hyper entity state provides a convenient way of modeling the semantics of generalization, specialization, classification, and aggregation using the compound data structure.

After we have identified associated entities in the entity space computation, we move on to the data space computation by manipulating the d_relations that are sets of tuples. The answer to the user's query may very well be a computed d_relation. Or we may transform the d_relation into values and carry out further computations on values.

ADL is developed as a constraint specification language using the computational syntax. We believe that ADL is adequate in achieving its design objective. The virtue of ADL as a data language is its simplicity in data structure and operators as compared to other high level data language like NETUL [Subi86]. The applicative nature of ADL makes it handy to be blended in application programs as a data access language. It is easy to learn for anyone who has been using a procedural programming language such as Pascal, Fortran, Basic, or LISP.

Each data manipulation language corresponds to a computational model of semantics. ADL has the flexibility of being an open-ended model. As we demand more and more semantic complexities to be handled by the language, we can create new data structures and new operators. But remember, the more complex the problem we try to model, the greater the difficulty we are going to encounter in expanding the model.

Another thing worth mentioning is that many query languages require a one sentence syntax in which the entire formula has to be put into one contiguous block of character strings, e.g. NETUL, SQL, QUEL [Subi86, Date81, Ullm82]. This creates an extra burden for the user because the final form of the sentence can only be decided after all its sub-parts have been well thought out. There is uncertainty at the beginning of the formulation since little is known about all the substructures of the formula. For complex queries, these languages demands layers and layers of nesting structures. It is analogous to a big chunk of unstructured software with no functions and no subroutines.

ADL, however, in this respect is much like a structured programming language. It allows users to think one step at a time and to do a step-wise refinement of formulating the query. The entire query is a paragraph which contains several sentences. A well-formulated query is a well-orchestrated paragraph. The added advantage of this approach is that the syntax for both the interactive query language and the data access language (in application programs) can be almost identical.

The intermediate results of the computation in ADL by no means imply any physical creations of intermediate data structures during the course of the computation. The computation procedures only reflect the users' semantic

perception of the problem. In physical implementation such a computation procedure can be optimized and transformed into another representation.

CHAPTER III. SEMANTIC CONSTRAINT MODELING BY ADL

A major concern of engineering design involves the verification of design data. People who implemented or studied earlier CAD databases soon realized that existing commercial databases lack a systematic approach to verify the design data [Mait85, Katz85, Stal86, Murt85, Boer85]. Most existing database systems rely solely on the application programs to maintain data integrity. Such an approach has serious drawbacks of program redundancies and global inconsistency [Hamm76, Lafu79, Bert84]. There is a definite need to incorporate data integrity checking as an integral part of the database system [Date83, Fern81].

Bertino [Bert84] pointed out that a database integrity subsystem is composed of four modules: (1). a language for constraint definition, (2). integrity constraint processor, (3). integrity checking strategy definer, and (4). integrity constraint enforcer. Many researchers have proposed the First Order Logic (FOL) or Relational Calculus as a semantic integrity (or constraint) specification language [Ston75, Nico78, Nico82, Ling84, Hsu 85, Fros83, Webe83]. All these approaches are based on the relational data model because relations can be equivalent to the predicates in FOL. Thus, these approaches provide an elegant and concise way of specifying certain semantic constraints. However, they are mainly specification languages that give little hints as to how to enforce the defined constraints. Also, functions of any form are not a natural part of the FOL. Therefore, most of these approaches did not incorporate functions in their enforcement strategy [Nico78, Nico82, Ling84, Hsu85, Fros83, Ston75].

In recent years, there are three ERM (Entity Relationship Model) based constraint specification languages. The Occurrence Structured Model (EROS) proposed by Tabourier and elta [Tabo83a, Tabo83b] is a graphical constraint specification language which only specifies semantic constraints that make assertions on the relationships among entities. There are six basic constructs for the language: NEGATION, AFFIRMATION, SCARD, VCARD, IMPLICATION, and CONNECT. Constraints on attribute values of entities or relationships cannot be modeled by EROS. There is no enforcement strategy discussed for EROS.

Morgenstern [Morg84, Morg86] proposed a constraint specification method called Constraint Equations (CE). Objects in the Constraint Equation generally have to be on a navigation path in the Entity Relationship Diagram (ERD). The grouping of objects in the semantic constraints is modeled as "Path Intersection". Since all the related objects have to be on a path in the ERD, the Constraint Equation Method is even more limited in its modeling power than the Occurrence Structure Model. [Tabo83a,83b]

Nakano proposed a Logic-Oriented ER Model (LOER) [Naka83] for semantic constraint specification. But the model actually requires an ERM schema to be

transformed into a relational schema before constraints can be specified by a modified domain relational calculus. The added external constructs to the First Order Logic to model constraints using aggregate functions cause great confusions over the issue of quantifiers. LOER uses a truth table in the evaluation space (a Cartesian product of all domain variables) to do constraint checking, which is not a very efficient approach when the variable domains are exceedingly large.

The concept of constraint modeling proposed in this paper is to model constraints as a computation procedure. The first part of the procedure is to find the constrained data through entity space and data space computations. And the second part of the procedure is to make assertions on the constrained data. In [Lee87a], we discuss the Applicative Data Language (ADL) as the data manipulation language to find the constrained data. Now, in this paper we will look at how semantic constraints are modeled by making assertions on the constrained data.

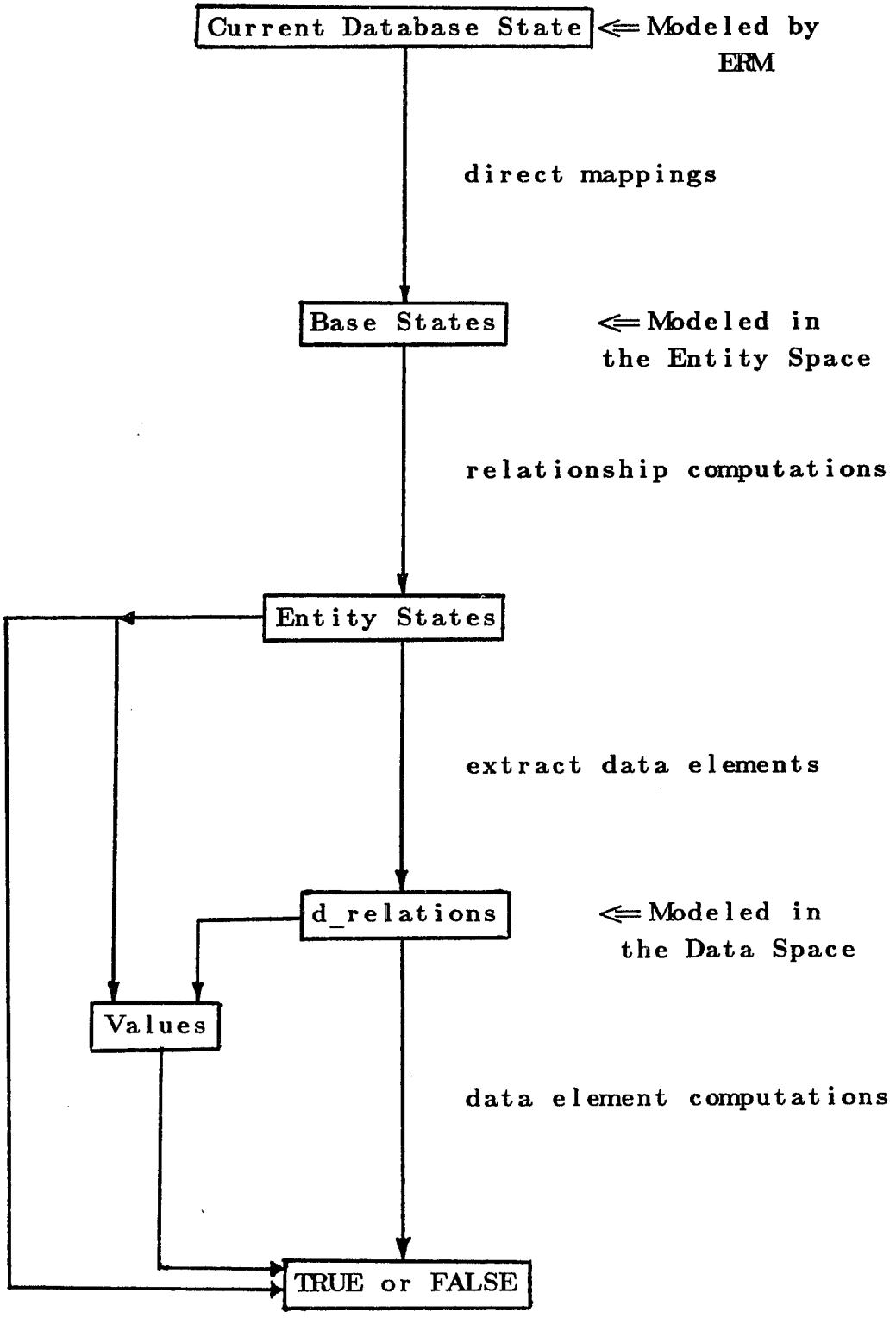
The difference between ADL as a query language and ADL as a constraint specification language is that the latter embodies a rich set of predicates to make assertions on the constrained data. The data manipulation part of ADL is simply to identify the constrained data. It is the assertion part of ADL that completes the specifications of semantic constraints.

On the conceptual level, ADL maps the current database state through a series of operators and predicates to a TRUE or FALSE value with respect to a specific assertion (Figure 3.0.1). The semantics of assertions are translated into computation procedures involving different data structures, operators, and predicates.

One of the special features of ADL is the diverse constraints that can be modeled by it. There are basically three types of constraints. **Structural constraints** [Tabo83a] consists of assertions on the relationships among entities or objects. For this type of constraints the database needs to compute the relationship among entities from known relationships. The computations are **relation-intensive** operations similar to the entity space computations. The second type of constraints, however, makes assertions on attribute values of entities. The scope of such constraints is limited to only one entity set. The computations involved often operate on integers, reals or character strings. Therefore, computations for this kind of constraint, called **local constraints** in [Hamm76], are **data-intensive** operations. The third type of constraints called **complex constraints** require relation-intensive operations, followed by data-intensive operations.

In section 1 we introduce the basic boolean operators and value predicates. Entity space predicates that are applied in structural constraints are discussed in section 2. In section 3 we look at how local constraints can be modeled using data space predicates. In section 4, we point out the limitation of ADL as a

Figure 3.0.1. Constraint Modeling By ADL



constraint specification language from the standpoint of constraint enforcement. Optimizing constraint specifications through reformulation is covered in section 5. Finally, in section 6 we present a compile-time checking strategy for semantic constraints specified by ADL. The run-time constraint checking strategy of ADL will be discussed in [Lee87c]. Abundant examples of using ADL as a constraint specification language are given in APPENDIX D. These examples are drawn from [Tabo83a, Tabo83b, Morg84, Morg86, Hamm76, Nico82, Ston75].

1. BOOLEAN OPERATORS AND ASSERTIONS ON VALUES

It is likely that an assertion may contain several sub-assertions linked together by the boolean operators. The boolean variables are represented as β, β_1, ... , β_n throughout this paper. The boolean operators are Λ (and) , V (or), and $\neg$(not).

(1) $\beta_3 := \beta_1 \Lambda \beta_2$,
(2) $\beta_3 := \beta_1 \mathbf{V} \beta_2$,
(3) $\beta_3 := \neg\beta_1$.

In the entity space, we have value operator **CountS** to transform a state into a value. In the data space, we have **CountD**, **SumD**, **AvgD**, etc. to transform d_relations into values. Once we have values, we can make assertions on them by using value comparison predicates: $\mathbf{PV}_{<}$, $\mathbf{PV}_{>}$, $\mathbf{PV}_{\leq}$, $\mathbf{PV}_{\geq}$, $\mathbf{PV}_{=}$, $\mathbf{PV}_{<>}$ (not equal). They are applied in the following formats (μ_1,μ_2 are either value variables or constants).

(1) $\beta := \mathbf{PV}_{<} (\mu_1, \mu_2)$,
(2) $\beta := \mathbf{PV}_{>} (\mu_1, \mu_2)$,
(3) $\beta := \mathbf{PV}_{\leq} (\mu_1, \mu_2)$,
(4) $\beta := \mathbf{PV}_{\geq} (\mu_1, \mu_2)$,
(5) $\beta := \mathbf{PV}_{=} (\mu_1, \mu_2)$,
(6) $\beta := \mathbf{PV}_{<>} (\mu_1, \mu_2)$.

The suffix "V" means that it is a value predicate.

2. ASSERTIONS IN THE ENTITY SPACE

For simple **structural constraints**, which only make assertions on the relationships among data, all the computations required are in the entity space. To complete the definition of a structural constraint, we make assertions on the final computed states.

2.A. Entity Space Predicates

The Set Predicates: $\mathbf{PS}_{\subseteq}$, $\mathbf{PS}_{\supseteq}$, $\mathbf{PS}_{\subset}$, $\mathbf{PS}_{\supset}$, $\mathbf{PS}_{=}$.

Since states in the entity spaces are sets of entity vectors, traditional set

predicates can be applied to the computed states. They are in the following formats: s_1 is a user specified entity vector (a constant). ς_1, ς_2 are entity states.
(1) β:= $\mathbf{PS}_{\subseteq}$ (ς_1, ς_2),
(2) β:= $\mathbf{PS}_{\supseteq}$ (ς_1, ς_2),
(3) β:= $\mathbf{PS}_{=}$ (ς_1, ς_2),
(4) β:= $\mathbf{PS}_{\subset}$ (ς_1, ς_2),
(5) β:= $\mathbf{PS}_{\supset}$ (ς_1, ς_2).
Note: (1) $s_1 \in \varsigma_1$ is equivalent to $\{s_1\} \subseteq \varsigma_1$. Therefore, we eliminate "$\in$" from our predicate primitives. (2) Either ς_1 or ς_2 can be a constant entity state specified by the user. In that case, the binary predicates can be regarded as unary predicates to simplify constrain enforcement. (3) The suffix "S" means that it is an entity state predicate.

2.B. Summary of Operators and Predicates in the Entity space.

For a structural constraint, whose assertion is about relationships among entities, all the computations can be done in the entity space. The operators [Lee 87a] and predicates summarized below can be used to model structural constraints.

OPERATORS (or FUNCTIONS):

Unary state function- state:=f(state1) , include
SelectS, **ProjectS**, **Group_byS**.

Binary state function- state:=f(state1,state2),
include $\cup$, $\cap$, -, **JoinS**.

Aggregate function- value:=f(state1), include **CountS**.

Numerical function- value:=f(value1, value2), include
+,-,*,/.

Boolean function- boolean:=f(boolean1,boolean2),
include $\wedge$, $\vee$.
boolean:=f(boolean1), include $\neg$.

PREDICATES:

State predicates- boolean:=p(state1,state2), include
$\mathbf{PS}_{\subseteq}$, $\mathbf{PS}_{\supseteq}$, $\mathbf{PS}_{\subset}$, $\mathbf{PS}_{\supset}$, $\mathbf{PS}_{=}$.
(Note that "$\in$" can be transformed into $\subseteq$).

Value predicates- boolean:=p(value1, value2), include
$\mathbf{PV}_{<}$, $\mathbf{PV}_{>}$, $\mathbf{PV}_{\leq}$, $\mathbf{PV}_{\geq}$, $\mathbf{PV}_{=}$, $\mathbf{PV}_{<>}$.

2.C. Structural Constraints

In APPENDIX D we have many interesting examples of using ADL to specify semantic constraints published in several related papers on the subject. Here, we

give a few examples of the structural constraints. When we formulate a semantic constraint, two things are needed: one is the ERD of the enterprise; the other, the BNF syntax specification of ADL. Specifying a constraint is equivalent to translating semantics into a computation procedure of ADL. With ERM as the logical data mode and ADL as the specification langauge, the task is easier.

Since semantics are so complex and varied, there is no single methodology that can mechanically translate all semantics into ADL formulas. But the following guidelines are helpful:

(1). Analyze the semantic constraint in detail to identify: first, what are the objects involved in the constraint; second, how the objects are related to each other in the constraint and according to the ERD. This understanding is crucial for us to formulate computation strategies in the entity space.

(2). Decide what are the constrained data and how to compute the constrained data from the database. If the constraint is about the relationship of objects, we only have to do entity space computations to find out the constrained objects. If the constraint is about object's attribute values, we may need to do the data space, as well as entity space, computations to isolate the constrained data.

(3). After we have identified and computed the constrained data in the database, we can apply predicates on the constrained data to complete the specification.

#Constraint 3.2.1. (See the ERD of Figure D.2 in APPENDIX D) [Tabo83a]
Semantic: If a person x is child of a person y who is brother or sister of a person z, who is a parent of a person t, we expect that x, t are cousins.

$_p$: means the person playing the role of the parent,
$_c$: means the person playing the role of the child.

ADL: $Person1_c := Persone2_p := Person3_p := Person4_c := Person,$
$Person5 := Person6 := Person,$
$\varsigma_1 := Person1_c \circ Parent \circ Person2_p \circ Sibling \circ Person3_p \circ Parent \circ Person4_c,$
$\varsigma_2 := Person5 \circ Cousin \circ Person6,$
$\beta := \mathbf{PS}_{\subseteq}(\mathbf{ProjectS}_{\{Person1c, Person4c\}}(\varsigma_1), \varsigma_2),$
$\mathbf{Display}(\beta).$ ###

#Constraint 3.2.2. (See the ERD of Figure D.4 in APPENDIX D) [Morg84]
Semantics: The projects directed by a department are those worked on by all the employees in the department.

ADL: $\varsigma_1 := D \circ Have \circ E,$
$\varsigma_{11} := \mathbf{Group\_By S}_{\{D\}}(\varsigma_1),$
$\varsigma_2 := D \circ Have \circ E \circ Work\_On \circ P,$
$\varsigma_{22} := \mathbf{Group\_By S}_{\{D,P\}}(\varsigma_2),$
$\varsigma_3 := \mathbf{ProjectS}_{\{D,P\}}(\mathbf{JoinS}_{\{D,E^*\}}(\varsigma_{11}, \varsigma_{22})),$

$\varsigma_4 := D\,{}^{\circ}Direct\,{}^{\circ}P$,
$\beta := \mathbf{PS}_{=}(\varsigma_3, \varsigma_4)$,
Display (β). ###

The above example shows how the **Group_byS** operator aggregates all the employees together in a department. The group of employees is later compared with another group of employees for equality, since the semantics calls for finding those projects that are worked on by **ALL** the employees in the department.

Both examples clearly demonstrate the unique characteristics of structure constraints. That is, they only use entity space operators and predicates in their specifications. They make assertions on the interrelationships among objects and attribute values of objects are never involved in the calculation (except where selection of objects is required).

3. ASSERTIONS IN THE DATA SPACE

For local constraints, where constrained data logically belong to one entity set, almost all computations are done in the data space.

3.A. Data Space Predicates

#Definition 3.3.1: **The And_all Predicates, $\mathbf{PD}_{>}$, $\mathbf{PD}_{<}$, $\mathbf{PD}_{\geq}$, $\mathbf{PD}_{\leq}$, $\mathbf{PD}_{=}$.**
Assume that τ is a d_relation and is subjected to the following preconditions:
1. Sch(τ)={A,B}, i.e. τ can only be a two-attribute d_relation. Each attribute should be a value attribute.
2. Dom(A) and Dom(B) are compatible domains whose elements can be compared with. We define $\mathbf{PD}_{>}(\tau)$ is TRUE, if $\forall\, t_i=(v_{i1}, v_{i2}) \in \tau$, $v_{i1} > v_{i2}$ is TRUE. It is FALSE, otherwise. Similar definitions for $\mathbf{PD}_{<}(\tau)$, $\mathbf{PD}_{\geq}(\tau)$, $\mathbf{PD}_{\leq}(\tau)$, $\mathbf{PD}_{=}(\tau)$. ###

The And_all predicate is to apply the predicate to each tuples of a d_relation and "AND" all the results together. If any of the tuple returns FALSE, the whole result is false.

#Example 3.3.1: The And_All Predicates.
Assume Sch(τ)={A,B} and τ is :

τ	A	B
	5.0	5.5
	4.5	4.5
	3.0	3.2

Then, $\mathbf{PD}_{\leq}(\tau)$ is TRUE; $\mathbf{PD}_{<}(\tau)$ is FALSE. ###

3.B. Interfacing With Application Programs

The And_all predicate can be extended to cover external predicates and programs that do complicated pattern matching or constraint checking on data in the database beyond the capability of ADL. Therefore, the extended And_all operator is the key operator that interfaces the database with the application programs.

Generally, we can envision that all the computations in an information system can be divided into two different levels of complexity. At the lower level, we do data access computation on structured data (ADL and the ERM database, for example). Part of the semantics that deals with finding the relevant data is modeled in this low level of computation. At the higher level of complexity, we may have to use some advanced and complicated computation algorithms to generate more useful data for decision making, e.g. finite element analysis for structure design, fussy logic reasoning for deduction, statistical analysis for pattern recognition. These are just a few examples that are beyond the capability of a general purpose database language.

However, ADL can extend the And_all predicate to bridge the gap of the two levels of computation complexity. For instance, if we have an external integrity checking program that checks the consistency of variables x,y,z or a pattern matching program that monitors the pattern of variables x,y,z, we can define an external predicate of the form **P**(x,y,z). Through the entity space and the data space computations, we can filter out from the database a d_relation, τ_1, with attributes X,Y,Z. Each tuple in τ represents a valid association of (x,y,z). We can then subject the whole d_relation to the external program for more detailed analysis that is beyond the capability of ADL. The operator we use for this interface is the And_all predicate, i.e. $\mathbf{PD}_{\mathbf{p}}(\tau)$. If all the tuples in τ_1 satisfy the external predicate, **P**, the result is true. If any one of them is false, the result is false.

It is important to realize that the integrity subsystem supporting ADL is not just a passive mechanism that only sends data upon the request of the external programs. When we use the ADL specifying a semantic constraint, we are actually defining a pattern in the database to be monitored automatically by the integrity subsystem. When the database is transformed from one state to another, these low level pattern matching activities are supervised by the integrity subsystem. The integrity subsystem will awake a sleeping process (for example the external program, **P** , just mentioned) if there are some changes in the predefined pattern due to user updates. For example, if τ_1 is changed to τ_2, P will be invoked by the integrity subsystem to check the new development in τ_2. More detailed discussion of how to do run-time constraint checking is covered in [Lee87c].

3.C. Summary of Operators and Predicates in the Data Space.

OPERATOR THAT TRANSFORMS DATA STRUCTURES

d_relation:=f(state), include **Extract**.

OPERATORS (or FUNCTIONS)

Unary d_relation function- d_relation:= f(d_relation1),
include **SelectD**, **ProjectD**, **ExtendD**.
(Note that **ProjectD** does not remove duplicates)

Value function- value:=f(d_relation), include **SumD**,
CountD, **AvgD**, **MaxD**, **MinD**,**ValueD**.

Numerical function- value:=f(value1, value2), include
+,-,*,/.

Boolean function- boolean:=f(boolean1,boolean2),
include $\wedge$, $\vee$.
boolean:=f(boolean1), include $\neg$.

PREDICATES:

D_relation predicates- boolean:=p(d_relation), include
$\mathbf{PD}_{>}$, $\mathbf{PD}_{<}$, $\mathbf{PD}_{\geq}$, $\mathbf{PD}_{\leq}$, $\mathbf{PD}_{=}$.

Value predicates- boolean:=p(value1, value2), include
$\mathbf{PV}_{<}$, $\mathbf{PV}_{>}$, $\mathbf{PV}_{\leq}$, $\mathbf{PV}_{\geq}$, $\mathbf{PV}_{=}$, $\mathbf{PV}_{<>}$.

3.D. Local Constraints and Complex Constraints

Many semantic constraints discussed in various published papers are modeled by ADL in APPENDIX D. Local constraints are those constraints that only involve one entity set. Generally, local constraints carry out most of their computations in the data spaces with no operations in the entity spaces (except the **SelectS** operator).

#Constraint 3.3.1. (See the ERD of Figure D.8 in APPENDIX D) [Ston75]
Semantics: Harding must make more than twice the average employee salary.

ADL: $\varsigma_1 := (E)_{\text{E.name='Harding'}}$,
$\tau_1 :=$ **Extract** {H_Sal<- Emp.salary} (ς_1),
$\varsigma_2 := E$, $\tau_2 :=$ **Extract** {Emp_Sal<- Emp.salary} (ς_2),
$\mu_1 :=$ **ValueD** (τ_1), $\mu_2 :=$ **AvgD** (τ_2),
$\beta :=$ $\mathbf{PV}_{>}$ $(\mu_1, (2.0 * \mu_2))$,
Display(β). ###

Complex Constraints always involve more than one entity set with computations in both the entity space and the data space. Many complex constraints can be seen in APPENDIX D.

#Constraint 3.3.2. (See the ERD of Figure D.8 in APPENDIX D) [Ston75]
Semantics: Employee must earn less than ten times the sales volume of their department if their department has a positive sales.

ADL: ς_1 :=*Emp 'Work_in' (Dept)* Dept.sale>0',
τ_1 := **Extract** {Emp_Sal<-Emp.salary,
Dept_Sal<-Dept.sale} (ς_1),
τ_{11} :=**ProjectD** {Emp_sal,Rate}
(**ExtendD** {Rate:=10*Dept.sale} (τ_1)),
β:= $PD_<$ (τ_{11}),
Display(β). ###

In summary, ADL models semantic constraints as computation procedures that involve data structures, operators, and predicates. Each constraint specification maps the current database state (DBST) to a boolean value of TRUE or FALSE. It defines a pattern in the database to be monitored by the integrity subsystem. The ADL takes a stepwise approach in query formulation and constraint specification. It first computes relationships among entities (an object level computation); and , then, computes new relations by their data values (a data element level computations).

4. LIMITATION OF ADL AS A CONSTRAINT MODELING LANGUAGE

Even though ADL is a powerful semantic constraint specification language (see APPENDIX D.), there are cases that cannot be adequately modeled by ADL in its current form. We can certainly extend ADL to capture these special cases by modifying either existing data structures or existing operators. But, there are other considerations that may refrain us from doing so.

Specifying a constraint is much less of a problem than enforcing a constraint. We can always define fancy operators for a particular type of semantic constraints. But to implement these operators and their cooperating data structures can be a difficult problem no matter what the physical implementation scheme is. It may be better off to leave extremely complex and rarely occurred semantic constraints to application programs rather than complicating the entire language.

When being confronted by a complicated semantic constraint, we need to ask ourselves the following questions:
(1). Can we model it by a sequence of complicated computation procedures instead of a new data structure or a new operator ? For complicated and often used computation procedures we can always use the "Macro" facility to define new macros from existing operators. It is very important to capture what is fundamentally missing in the existing operator and data structure before rushing ahead to create new ones. It is easy to define mountains of operators. Each only has

a specific purpose.

(2). If the semantic requires fundamentally different operations and data structures, we must make a decision about whether to modify the existing data structures or to create new operators. While modifying an existing data structure, we also have to extend all the current operators to accommodate the new data structure. Adding a new operator, on the other hand, is limited in its effect to the overall structure of the language.

(3). What are the added complexities for the compile-time and the run-time constraint checking as a result of introducing a new data structure or a new operator ? Semantic constraints are specified for the mere purpose of enforcing them. Therefore, we must take into consideration of how a complex constraint should be enforced.

(4). Is it really worthwhile to add the complexity to the language just for this particular type of semantic constraints? There is always a trade-off point between complexity and efficiency. We must make a fair evaluation of where this point might be.

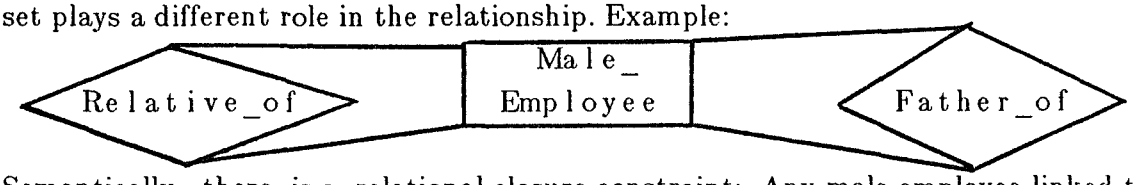
While sorting out semantic constraints in various published papers (see APPENDIX D), we find the following cases cannot be modeled adequately by ADL in its current version from the standpoint of efficient constraint enforcement.

(1). Dynamic Constraints

Dynamic constraints are assertions on the data before and after user update. Example: The employee's salary cannot decrease. It requires the database system to keep two copies of the same data. One is before the update; the other is after the update. Assertions made on the two copies of data are called dynamic constraints. Possible solutions to model dynamic constraints in ADL include incorporating a time dimension for data in the entity space model. For any dynamic constraint, the final entity space can be $E_1 \times E_2 \times \ldots \times E_n \times$ TIME. The TIME dimension may just include two elements for simple dynamic constraints, i.e. NEW and OLD.

(2). Relational Closure Constraints

In ERM, we can define a relationship between the same entity set. Each entity set plays a different role in the relationship. Example:



Semantically, there is a relational closure constraint: Any male employee linked to another male employee through successive association of the Father_of relationship is a relative of the other employee. Excellent review of how to solve the relational closure problem can be found in [Aho 74], [Merr84], [Banc86].

(3). Functional Dependency Constraints

Functional dependencies are generally considered not a semantic constraint but a

data model integrity constraint. [Date82], [Tsic82] To check the functional dependency constraint, the database subsystem needs to have the ability to sort a relation by its primary key (can be multiple attributes) and then compare values in the non-key attributes. ADL has no operator that will sort a d_relation. Thus, it cannot enforce the functional dependency.

To solve each of the above cases might be a research topic all by itself. It is possible that ADL can be expanded in the future to consider all these cases.

5. REFORMULATION OF CONSTRAINT SPECIFICATION

A constraint specified by a user cannot be directly bsed as a computation procedure for constraint checking, Because the user's specification is a direct translation from the semantics. Efficiency for execution is not a mtate concern. The specification may contain useless statements, equivalent expressions, wrong statement sequence, etc. It is thus the job of the integrity subsystem to detect such inefficiency and reformulate the specification if required.

#Example 3.5.1: Inefficient Constraint Specification.
We may have the following ADL constraint specification by a user.

$\varsigma_1 := A\,{}^{\circ}R_{AB}\,{}^{\circ}B,$ - - - - (1)
$\varsigma_2 := B\,{}^{\circ}R_{BC}\,{}^{\circ}C,$ - - - - (2)
$\varsigma_4 := \varsigma_1 \circ \varsigma_2,$ - - - - (3)
$\varsigma_4 := \mathbf{ProjectS}_{\{A\}}(\varsigma_1),$ - - - - (4)
$\beta_1 := \mathbf{PS}_{\subseteq}(\varsigma_4, \varsigma_5),$ - - - - (5)
$\varsigma_5 := \mathbf{ProjectS}_{\{B\}}(\varsigma_2),$ - - - - (6)
$\beta_2 := \mathbf{PS}_{\supset}(\varsigma_4, \varsigma_6),$ - - - - (7)
Display (β_1).

Note that the above specification meets every syntax rule of ADL. But, it is not efficient. ς_4 has been assigned twice in statements (3),(4). Statement (7) is a useless statement. And also in statement (7), ς_6 never has any value at all. Statement (5) should be put after statement (6). To eliminate such inefficiencies we need to tighten up the syntax rules with more restrictions. ###

#Definition 3.5.1: **A Well Formed Specification.**
An ADL constraint specification is a well formed specification if the following conditions are met.
(1). Each variable on the left hand side of the assignment statement should appear only once on the left hand sides of all the assignment statements.
(2). The last **Display** function of the specification should have the boolean variable, β_n, of the last assignment statement as its operand.
(3). Also, β_n is the only variable in the entire specification that never appears

on the right hand side of any assignment statement.
(4). A variable can only be used on the right hand side of a statement after it appears on the left hand side of a previous statement.
(5). If equivalent expressions appear in more than one place in a specification, they should all be replaced with a variable symbol. An assignment statement that assigns a common expression to the variable will be placed before all these statements.
###

To remove all the inefficiencies in a constraint specification, the integrity system needs to do a context analysis besides syntax analysis to spot the patterns shown in Example 3.5.1.

A semantic constraint specified by the user also suggests a computation sequence according to the following definition.

#Definition 3.5.2: **Order of Computation**
The **order of computation** for a particular constraint specification is the computation sequence of all the operators according to the following rules:
(1). If statement 1 precedes statement 2, all operators in statement 1 precede those in statement 2.
(2). In the same statement, the operator in the inner most parentheses has the highest priority. The operator in the second inner most parentheses has the second highest priority and so on.
(3). If several associative binary operators are of equal precedence and operate on more than two operands in sequence, the order of operations is from left to right. If two expressions are operands of a binary operator, the left one is evaluated first.
(4). In the same expression, the precedence rules for each category of operators are as follows: entity space operators preceding data space operators, data space operators preceding value operators, value operators preceding boolean operators.
(5). For value operators, *,/ precede +,-.
(6). For boolean operators, "¬" precedes "Λ". "Λ" precedes "V".
###

According to the above ordering definition, each ADL constraint specification only has one unique sequence of computation.

Optimization is important to constraint enforcement because it has to be evaluated and invoked to guarantee the correctness of data as the database changes from one state to another. The time invested on optimizing the constraint computation is well spent in the long run since optimization only has to be done once, while the constraint checking may be carried out many times. This is especially true during run-time constraint checking since the constraint is invoked much more frequently and the response time is at a premium. To achieve better time or space efficiency in constraint computation, the integrity subsystem must

optimize the constraint specification during constraint compilation.

The term "optimization" may be misleading here since most optimization strategies discussed in literature do not always guarantee to find the "best" solution from all possible alternatives [Ullm83, Maie84, Gray84]. They are more or less only "heuristics" that are based on presumptions of certain physical implementation schemes and database states. This is in part because there are so many interacting factors to consider in optimization:

(1). The physical implementation of the database- Whether or not the data records are indexed may govern how the operation is carried out. For example, a selection can be computed by scanning the entire d_relation a tuple at a time. However, if an index exists for the attribute on which the selection is performed, we can compute the selection through index look up. Pointers from tuples in one d_relation to tuples in another can be maintained to expedite the **"JoinS"** operations.

(2). The current state of the database- If we have to **JoinS** several entity states together, it may be wise to start with the smallest state first and progress to the largest. This tends to minimize the sizes of the intermediate results.

(3). The software and hardware configurations- Optimization can be accomplished at more than one level. At the database level, we can optimize the computation sequences in terms of the Data Manipulation Language like ADL. At the system software level, we have to consider the system file structures and their supporting facilities. Finally, at the system hardware level, it is important to understand the memory access method and the number of disk drives available. The range of optimization that can be done depends a lot on the amount of information available on a particular database and the software and hardware systems that support it.

When all these factors must be considered, it is not an easy task to find the "best" evaluation strategy. But, we can always do a local optimization based on the limited information at hand. The optimization strategies that we shall discuss will not depend a lot on the physical implementation of the database, for to do so we would have to discuss various file structures and management techniques, trade-offs between secondary storage access and computation time. General strategies that will reduce the sizes of intermediate results and the number of operations are introduced. Therefore, we concentrate on optimization at the database level.

The general optimization strategies of ADL are as follows:

(1). Perform **SelectS** in the entity space computations, **SelectD** in the database space computations, as early as possible. This will minimize the computation space in later operations.

(2). Eliminate useless dimensions in states by **ProjectS**, and useless attributes in d_relations by **ProjectD** in every step of the computation sequence. Always keep

a minimal set of data for later computations.
(3). Extract only those attributes that are needed in the data space computations from the result of the entity space computations.
(4). Find out equivalent expressions in a computation sequence and replace them with one common expression so that it only has to be evaluated once. ADL is designed specifically to facilitate building of common expression as an intermediate result that is assigned to a variable symbol. We can use the variable symbol in as many places as the semantics require in later computations.

The technique to realize the above strategies is constraint reformulation. The purpose of reformulation is to replace a less time or space efficient expression with an equivalent but more efficient expression. The criteria for efficiency is judged by factors discussed earlier, i.e. the physical implementation, the current state of the database, the software and hardware configurations of the supporting system.

There are four data structures in ADL: values, booleans, states, and d_relations. The equivalences of the first two data structures are easy to understand. But, we need to look at what are equivalent states, equivalent d_relations, and equivalent expressions.

#Definition 3.5.3: **Equivalent State, Equivalent D_relation, Equivalent Expression**

Two entity states ς_1, ς_2 are equivalent, if ς_1 and ς_2 are of the same entity space Ψ, and $\forall\ s_1 \in \varsigma_1 ===> s_1 \in \varsigma_2$, and $\forall\ s_2 \in \varsigma_2 ===> s_2 \in \varsigma_1$. s_1 and s_2 are mappings from Sch(R) to domains of entity identifiers. We express the equivalence as $\varsigma_1 \equiv \varsigma_2$.

Two d_relations τ_1, τ_2 are equivalent, if τ_1 and τ_2 are of the same relation scheme, i.e. $Sch(\tau_1)=Sch(\tau_2)$, and $\forall\ t_1 \in \tau_1 ===> t_1 \in \tau_2$, occurrences of t_1 in τ_1 = occurrences of t_1 in τ_2, and $\forall\ t_2 \in \tau_2 ===> t_2 \in \tau_1$, occurrences of t_2 in τ_2 = occurrences of t_2 in τ_1. t_1 and t_2 are mappings from $Sch(\tau_1)$ to the domains of attributes. The equivalence is expressed as $\tau_1 \equiv \tau_2$.

Two ADL expressions Exp_1, Exp_2 are formulas of ADL operators. Exp_1 and Exp_2 are equivalent if:
(1). Each expression has the same set of input variables $(X_1,...,X_n).X_1,...,X_n$ are data structure variables of ADL.
(2). Exp_1 and Exp_2 result in two equivalent data structures (values, booleans, states, or d_relations) when $(X_1,...,X_n)$ is instantiated to any set of values $(v_1,.\ ,v_n)$ that meet all the preconditions of both Exp_1 and Exp_2. We express the equivalence as $Exp_1 \equiv Exp_2$. ###

#Lemma 3.5.1. : Transitive Property of Equivalence
(1). If $\varsigma_1 \equiv \varsigma_2$, $\varsigma_2 \equiv \varsigma_3 ===> \varsigma_1 \equiv \varsigma_3$.
(2). If $\tau_1 \equiv \tau_2$, $\tau_2 \equiv \tau_3 ===> \tau_1 \equiv \tau_3$.

(3). If $Exp_1 \equiv Exp_2$, $Exp_2 \equiv Exp_3$ ===> $Exp_1 \equiv Exp_3$.
Proof: From Definition 3.5.2, the proof is straight forward. ###

In Appendix E, we provide a list of equivalent ADL expressions that can be proved from the above definition of equivalent expressions and the definitions of ADL operators. Many of them are adapted from relation algebra optimization techniques discussed in [Ullm83, Maie84, Gray84]. Such equivalent expressions have two purposes: (1). Identify equivalent expressions in a constraint and replace them with a common expression to eliminate redundant evaluations. (2). Based on information available, the integrity subsystem can reformulate a constraint with an equivalent expression to optimize the space or time efficiency of constraint evaluation.

#Theorem 3.5.1: For well formed specification, if Constraint Specification 2 is obtained through a series of reformulations by equivalent expressions listed in Appendix E from Constraint Specification 1, Constraint 2 and Constraint 1 are equivalent.
Proof: We can always transform a constraint specification into a single expression by :
(1). Identify the last predicate statement, Statement 1, as indicated in Definition 3.5.1 items (2) and (3).
(2). Replace any variable symbol in the right hand side of Statement 1 by the assignment statement for the variable according to Definition 3.5.1 item (4).
(3). Repeat Step 2 until no variable exists in the right hand side of the Statement 1 except base state symbols.
(4). We can represent the entire specification by the expression, Exp_1, at the right hand side of Statement 1.

Assume now we start replacing equivalence expressions in Exp_1 by using rules in Appendix E to get the final expression Exp_n that corresponds to Statement 2.

Exp_1,..., Exp_i, Exp_{i+1}, ...,Exp_n.

If we can prove $Exp_i \equiv Exp_{i+1}$ through substitution, by Lemma 3.5.1 and induction $Exp_1 \equiv Exp_n$.
Case 1. Exp_i and Exp_{i+1} are themselves equivalent expressions in Appendix E. Then, $Exp_i \equiv Exp_{i+1}$.
Case 2.Exp_i and Exp_{i+1} contain equivalent subexpressions, $Exp_{i,s}$, $Exp_{i+1,s}$. $Exp_{i,s} \equiv Exp_{i+1,s}$. We can replace the equivalent subexpressions by variable symbols X_i, X_{i+1}. So Exp_i becomes $Exp_i(X_i,Y)$; Exp_{i+1} becomes $Exp_{i+1}(X_{i+1},Y)$. Y are the common variables to both Exp_i and Exp_{i+1}. $X_i := Exp_{i,s}$, $X_{i+1} := Exp_{i+1,s}$. But $X_i \equiv X_{i+1}$ for all the variables in $Exp_{i,s}$ and $Exp_{i+1,s}$. Therefore, (X_i,Y) and (X_{i+1},Y) are the same. Thus, $Exp_i \equiv Exp_{i+1}$. If $Exp_1 \equiv Exp_n$, Statement 1 and Statement 2 will give the same truth value for

the same set of input variables. ###

As a result of optimization and reformulation, the system generates a new constraint specification, still in terms of ADL operators, with a fix number of operators and in a unique computation sequence (Definition 3.5.2). All the common expressions in the specification are identified and replaced with a unique variable symbol.

6. COMPILE-TIME CONSTRAINT CHECKING METHOD

After a constraint is optimized, immediately, the DBMS has to check the validity of the current DBST with respect to the constraint. This is called compile-time constraint checking which subjects the whole database to the verification algorithm. We invoke compile-time constraint checking when:
(1). The database is first established.
(2). A constraint is first defined.
(3). A user controlled compile-time constraint checking is invoked after a large batch of data are updated.

The compile-time constraint checking is expensive because large amounts of data need to be accessed and computed. But the computation process is straight forward if an applicative constraint specification language like ADL is used.

In order to carry out constraint enforcement, we need a representation that reflects the process of computations. Even though ADL represents a conceptual level constraint specification, it is not mechanical or efficient enough for computer implementation. Ullman [Ullm82] used a parse tree for the representation of database queries. Buneman and elta [Bune79] represented the derivation procedures of computed relations by a construction diagram. Here, we introduce an acyclic digraph to represent the constraint checking process.

6.A. Constraint Computation: The Transition Digraph

#Definition 3.6.1: **Transition Digraph**
Each Constraint specified by the ADL has a corresponding Transition Digraph C(N,E) in which C is the constraint label, N the set of nodes, and E the set of directed edges. Each node of the graph represents either a state(ς), a d_relation(τ), a value(μ), or a boolean(β). The edge of the digraph represents either an operator or a predicate. (Note that the final **Display** function is omitted from the transition digraph for simplicity.) ###

We can build a transition digraph from an optimized constraint specification (as discussed in the last section) according to the following algorithm.

#Algorithm 3.6.1: **Building Transition Digraph from A Constraint**

Specification.

(1). Mark each operator with a sequence number according to the order in the computation sequence determined by Definition 3.5.2. {The **Display** operator is ignored.}

(2). {We build the transition digraph from left to right, from bottom to top.}

For each operator in the computation sequence do:

(2.1) The operand nodes of the operator should either already exist, or not exist yet. If it does not exist yet, the operand must be a base state or a constant. If the operand is a base state, create a new node marked by its base state symbol. If the operand is a constand, create a constant node with a circle enclosing the constant value.

(2.2) Create a new node and mark it by a proper data structure variable symbol designated as the result of the operation.

(2.3) Draw directed arcs from the operand nodes to resulting node. Mark the arc by the operator symbol. ###

#Definition 3.6.2: **A Well Formed Transition Digraph.**

A transition digraph built from a well formed specification according to Algorithm 3.6.1 is called a **well formed transition digraph**. #Definition 3.6.3: **Sink Node, Source Node**

A node that only has in-arcs is a sink node. Any node that only has out-arcs is a source node. ###

The sink node is always a boolean node. The source nodes correspond to the base states specified in the constraint.

#Theorem 3.6.1: A well formed transition digraph should have at least one source node, and exactly one sink node.

Proof: A well formed specification according to Definition 3.5.1 should have exactly one boolean variable that does not appear on the right hand sides of all the assignment statements. Such a boolean variable will be a sink node according to Algorithm 3.6.1 and Definition 3.6.3. According to the variable substitution method suggested in Theorem 3.5.1, there is at least one operation that computes the value of the sink node. It requires at least one base state as the operand of the operation after all the variables are substituted. Since base states are source nodes from the database, we would have at least one source node in the transition digraph according to Algorithm 3.6.1 and Definition 3.6.3. ###

Figure 3.6.1 to Figure 3.6.3 are examples of the transition digraphs for well formed constraints specified in ADL.

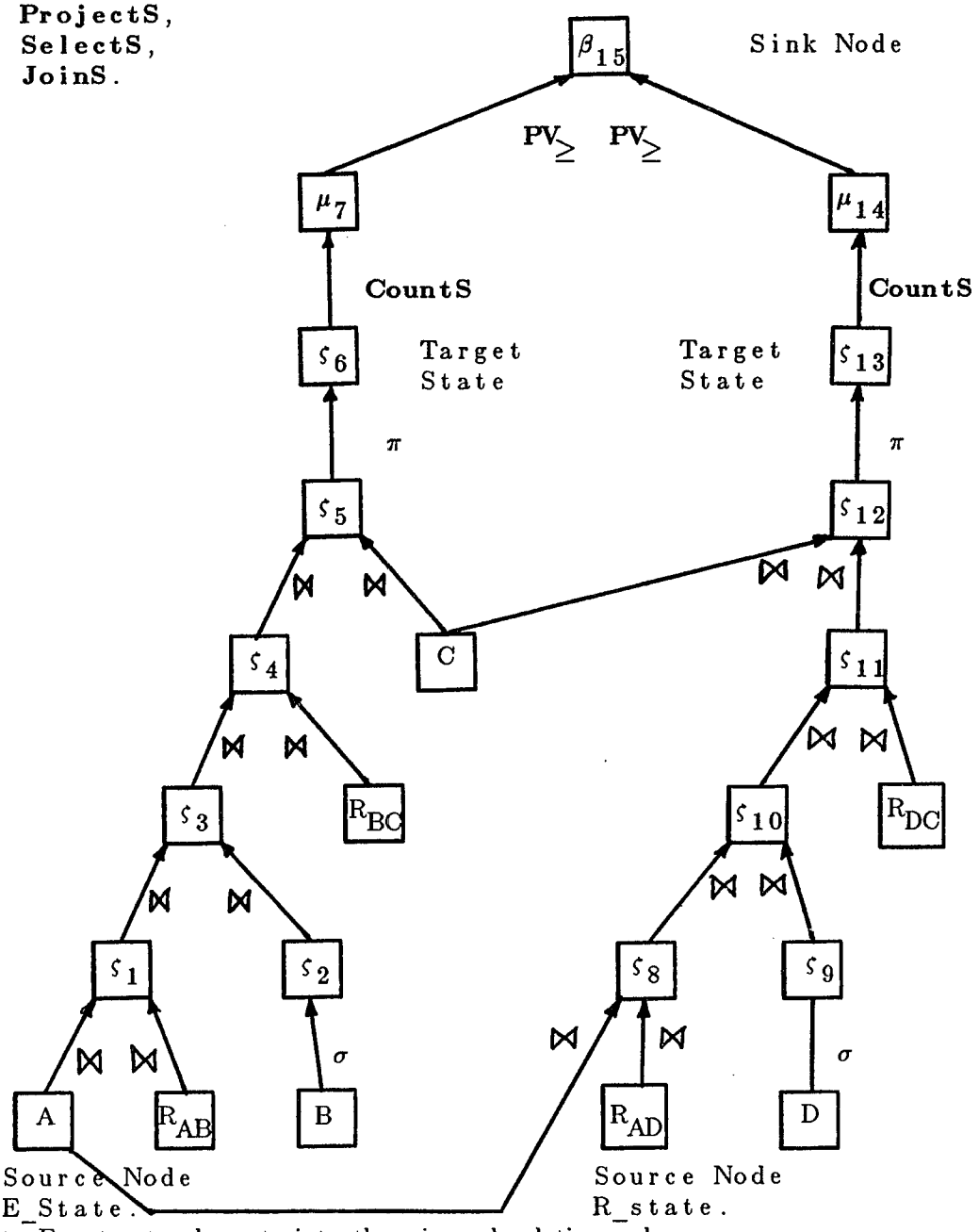
It is a deterministic procedure to find out how a particular node in the transition digraph is derived from its predecessors. We will give the definition of such a procedure first and derive the algorithm later.

Figure 3.6.1. Transition Digraph for Structural Constraint

Structural Constraint :

ς_5:=A*R_{AB}*(B) $_{B\geq 'b2'}$*R_{BC}*C,
ς_6:= **ProjectS** $_{\{A,C\}}$ (ς_5),
ς_{12}:=A*R_{AD}*(D) $_{D\geq 'd2'}$*R_{DC}*C,
ς_{13}:= **ProjectS** $_{\{A,C\}}$ (ς_{12}),
β_{15}:= $PV_{\geq}$(ς_6, ς_{13}).

π: **ProjectS**,
σ: **SelectS**,
$\bowtie$: **JoinS**.

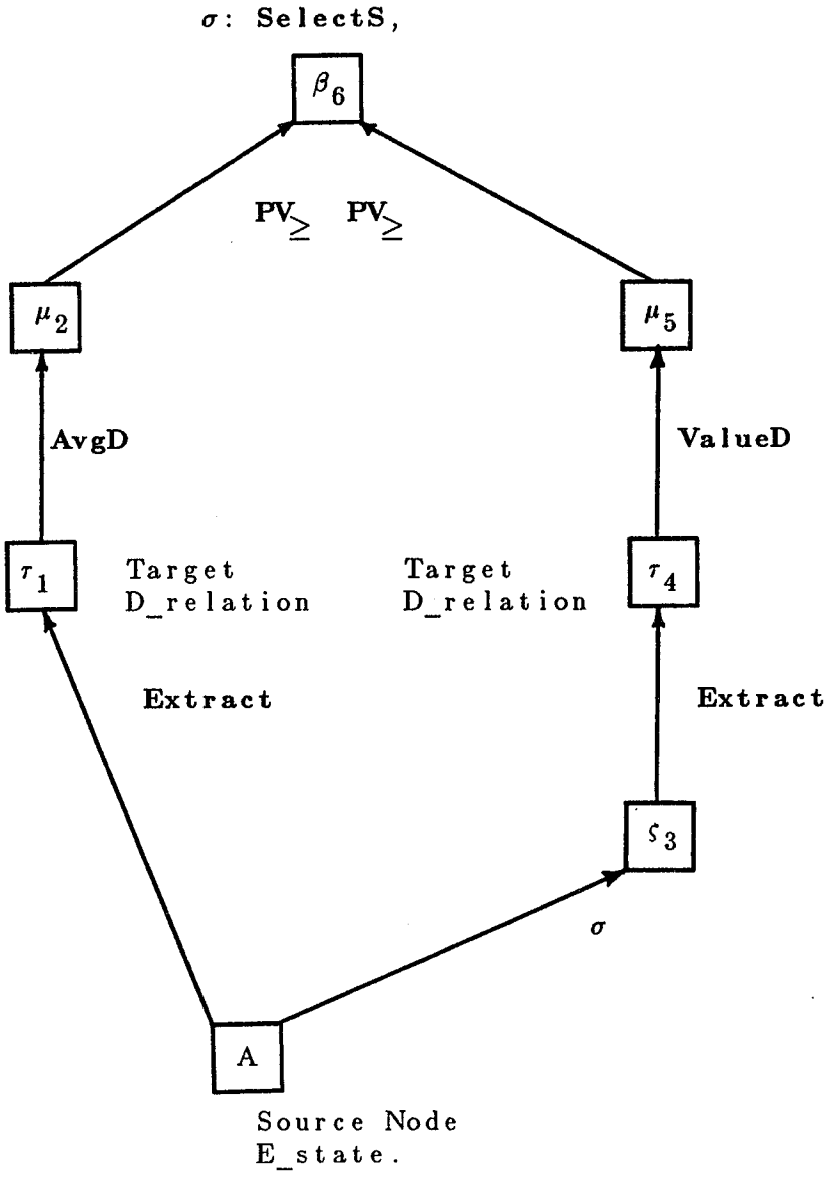


Note: For structural constraints, there is no d_relation node.

Figure 3.6.2. Transition Digraph for Local Constraint

Local Constraint :

τ_1:= **Extract** $_{\{F1<\text{-}A.f1\}}$ (A),
μ_2:=**AvgD**(τ_1),
τ_4:=**Extract** $_{\{F2<\text{-}A.f1\}}$ (**SelectS** $_{A='a1'}$ (A)),
μ_5:=**ValueD**(τ_4),
β_6:= **PV**$_{\geq}$(μ_2, μ_5).



Note: For local constraints, there is only a single source node.

Figure 3.6.3. Transition Digraph for Complex Constraint

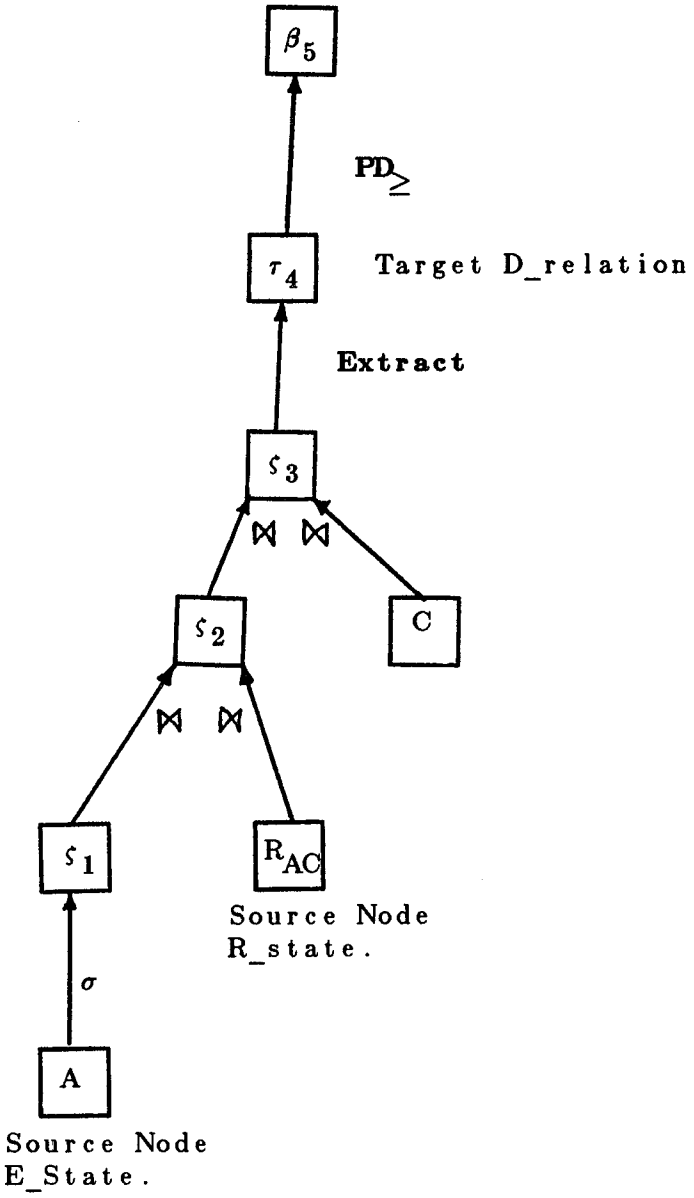
Complex Constraint:

ς_3:= (A) $_{A\geq}$ 'a2' $^{\bullet}R_{AC}$ $^{\bullet}C$

τ_4:= **Extract** $_{\{F1<\text{-}A.f1,\ F2<\text{-}B.f1\}}$ (ς_3),

β_5:= $\mathbf{PD}_{\geq}$ (τ_4).

π: **ProjectS**,
σ: **SelectS**,
$\bowtie$: **JoinS**.



#Definition 3.6.4: **Subgraph of a Node in a Transition Digraph, Subgraph (node).**
Subgraph of a node n_i, Subgraph(n_i), is a subgraph of the transition digraph which includes all the nodes and the edges that lead to n_i starting from the source nodes. ###

Subgraph(node) represents the part of the transition digraph that computes the value of that particular node. For example, Figure 3.6.4 shows the proper subgraph of the transition digraph in Figure 3.6.1.

We further define the following attributes of a node in the transition digraph for our discussion of algorithms. To express the attribute of a node we simply use the Pascal syntax for record, i.e., "node.attribute". A transition digraph can be completely characterized by its nodes with their attributes.

#Definition 3.6.5: **Attributes of Nodes**
node.label- the label of the node, e.g. ς_5.
node.order- the order of evaluation for the node in the entire computation sequence.
node.new_value- the newly computed value of the node, e.g. TRUE.
node.type- the value type of the node, e.g. state, d_relation, value, or boolean.
node.pre_op- the set of the previous operation (function or predicate symbols) that creates this node, e.g. {**ProjectS**}. For any source node, node.pre_op:=Ø.
node.pre- the set of node labels that derive this node, e.g. $\{\varsigma_2, \varsigma_3\}$. For source nodes, node.pre=Ø.
node.post- the set of node labels that are derived from this node, e.g. $\{\varsigma_{10}, \beta_2, \mu_2\}$. For the sink node, node.post=Ø.
node.base- the set of labels of all the base states (both the E_states and R_states) in subgraph(n_i). It represents all the base states needed for the computation of the node value, e.g. $\{A,B,C,R_{AB},R_{BC}\}$
node.E_state- the set of labels of all the E_states in subgraph(n_i), e.g. {A,B}. Note that node.E_state $\subseteq$ node.base. ###

#Lemma 3.6.1: node.pre_op should have exactly one element for any non-source node in a well formed transition digraph.
Proof: According to Algorithm 3.6.1, each non-source node is create as the result of a unique operation in the computation sequence. Therefore, there is exactly one operation that computes the value of that node. According to Definition 3.6.5, node.pre_op for the non-source node should have exactly one element. ###

In the transition digraph there are certain nodes to which we should pay special attention. They represent the final results of entity space computations or data space computations just before the final assertion is made. We define such nodes as

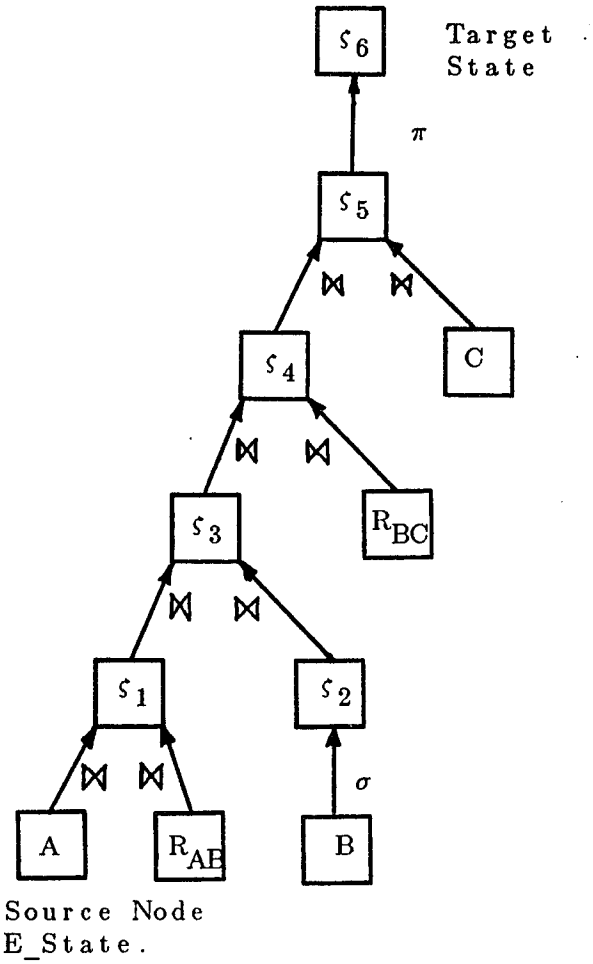
Figure 3.6.4. Proper Subgrpah of Transition Digraph In Figure 3.6.1.

Structural Constraint :

$\varsigma_5 := A * R_{AB} * (B)_{B \geq 'b2'} * R_{BC} * C,$
$\varsigma_6 := \textbf{ProjectS}_{\{A,C\}} (\varsigma_5),$
$\varsigma_{12} := A * R_{AD} * (D)_{D \geq 'd2'} * R_{DC} * C,$
$\varsigma_{13} := \textbf{ProjectS}_{\{A,C\}} (\varsigma_{12}),$
$\beta_{15} := PV_{\geq}(\varsigma_6, \varsigma_{13}).$

Subgraph (ς_6) of Transition Digraph in Figure 3.6.1 is:

π: **ProjectS**,
σ: **SelectS**,
$\bowtie$: **JoinS**.



target states or target d_relations.

#Definition 3.6.6: **Target states and target d_relations**
If n.type = state and (∃ m ∈ n.post (m.type = value V m.type = boolean)) then n represents a target state. If n.type=d_relation and (∃ m ∈ n.post (m.type = value V m.type = boolean)) then n is a target d_relation. ###

#Theorem 3.6.2: Any well formed transition digraph should have at least one target state or d_relation.
Proof: According to Definition 3.5.1, there is at least one boolean variable and one predicate statement in every well formed specification, so there is at least a boolean node and several other nodes (corresponding to the operands in the predicate statement) computing the final boolean node. If the boolean variable is computed from states or d_relations, according to Definition 3.6.6 we would have at least one target state or one target d_relation. If the boolean variable is computed from values or other boolean variables, we can repeat the searching process until a state or d_relation leading to the value or boolean variables is found. If no such state or d_relation can be found, then a boolean node or a value node is a source node by itself. In other words, the value variable or the boolean variable appears on the right hand side of a statement, but never appears on the left hand side. It contradicts to the original assumption that the digraph is a well formed transition digraph. ###

Now, we are in a position to define the algorithm that builds the subgraph of node n_i. Assume the original transition digraph is C(N,E). N is the set of node n_i. Each n_i has attributes as defined above. E is the set of directed edges (n_i,n_j).

#Algorithm 3.6.2: **Building the subgraph of a node in the transition digraph.**

```
function subgraph (n_i): digraph;
   begin
   create( Cs(Ns,Es) ); Ns:=Ø ; Es:=Ø ;
   build_subgraph(n_i,Cs(Ns,Es));
   subgraph:=Cs(Ns,Es)
   end.
procedure build_subgraph (n_i, Cs(Ns,Es));
   begin
   Ns:= Ns ∪ { n_i };
   if( n_i.pre=Ø ) then
      return
   else
      for (each nj in ni.pre) do
```

```
        begin
        Es:=Es ∪ {(ni,nj)};
        build_subgraph(nj,Cs(Ns,Es))
        end;
    end.
```

Note: Create(Cs(Ns,Es)) is a procedure that builds an empty digraph data structure. ###

6.B. Analysis of The Transition Digraph.

By observing the differences of structural, local and complex constraints, we will find recurring patterns of transition digraphs.

#Theorem 3.6.3: A well formed transition digraph is an acyclic digraph.
Proof: Assume there exists a cycle in the transition digraph of the form $X_0 \Rightarrow X_1 \Rightarrow \ldots \Rightarrow X_n \Rightarrow X_0$, where $X_1, \ldots, X_n$ are nodes in the transition digraph. Assume the operation from X_0 to X_1 is Op_1; operation from $X_n \Rightarrow X_0$ is Op_n. According the path $X_0 \Rightarrow X_1 \Rightarrow \ldots \Rightarrow X_n \Rightarrow X_0$, we find Op_1 precedes Op_n. However, according to the path $X_n \Rightarrow X_0 \Rightarrow X_1$, we conclude Op_n precedes Op_1. The two conclusions are contradictory to each other. Therefore, there cannot be a cycle in well formed transition digraph and it has to be an "acyclic" transition digraph. ###

A few observations will reveal that the nodes of the digraph can be distinctly grouped together according to their similarity and complexity of computations. We call such grouping, clusters:

(1). The BV_cluster: the set of all boolean nodes and value nodes.
(2). The DR_cluster: the set of all d_relation nodes.
(3). The TS_cluster: the set of all entity state nodes (including the target state nodes).
(4). The ER_cluster: the set of all E_state nodes and R_state nodes.

Note that the computations always proceed from the ER_cluster to the VB_cluster. Further, the computation sequences for different types of constraints can be summarized as follows:

(1). STRUCTURAL CONSTRAINTS
ER_CLUSTER ⇒ TS_CLUSTER ⇒ VB_CLUSTER
(2). LOCAL CONSTRAINTS:
ER_CLUSTER ⇒ DR_CLUSTER ⇒ VB_CLUSTER,
or
ER_CLUSTER ⇒ TS_CLUSTER ⇒ DR_CLUSTER ⇒ VB_CLUSTER.
(3). COMPLEX CONSTRAINTS:
ER_CLUSTER ⇒ TS_CLUSTER ⇒ DR_CLUSTER ⇒ VB_CLUSTER.

Computations in the BV_cluster are trivial once we know all the target states and d_relations leading to it. The difficult part of the computation is in the TS_cluster and the DR_cluster. Here, we have to do set (of vectors or tuples) computations instead of discrete value computations. Therefore, how to compute the target states or d_relations is usually the crucial part of constraint computation.

Once we have a compile-time enforcement strategy, the integrity subsystem proceeds to do a full evaluation of the database with respect to the constraint. For such compile-time checking, the sequence of computation can follow the node.order attribute. It is the same as the sequence of constructing the transition digraph from the constraint specification.

#Algorithm 3.6.3: **Full evaluation (or compile-time checking) of a constraint according to post-order traversal of the transition digraph.**

```
procedure full_eval(C(N,E));
  begin
  partition the nodes into a set of source nodes
                  and a set of non-source nodes;
  for each node in the source node set
          node.new_value:= E_state or R_state;
  sort the nodes in the non-source node set according
          to the node.order attributes;
  evaluate each node in the non-source node set
        according the the sort sequence;
  result:=sink_node.new_value;
  if( result= FALSE ) then
    begin
    give warning to the database user;
    roll back the transaction;
    other recovery actions
    end
  else
    current DBST is OK w.r.t. C(N,E)
  end.                                    ###
```

#Example 3.6.1: Constraint Computation

For the ERD shown in Figure 3.6.5, we define the following Constraint :

$$\varsigma_3 := (B)_{B \geq 'b3'} \circ R_{AB} \circ A,$$
$$\varsigma_4 := \mathbf{ProjectS}_{\{A\}} (\varsigma_3),$$
$$\varsigma_7 := (A)_{A \geq 'a2'} \circ R_{AC} \circ C,$$
$$\varsigma_8 := \mathbf{ProjectS}_{\{A\}} (\varsigma_7),$$

Figure 3.6.5. ERD For Example 3.6.1

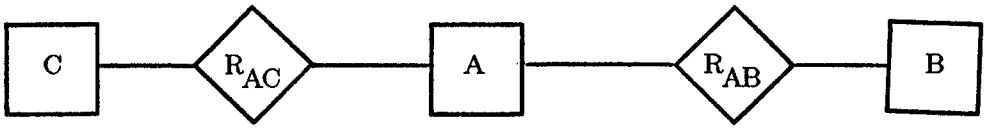
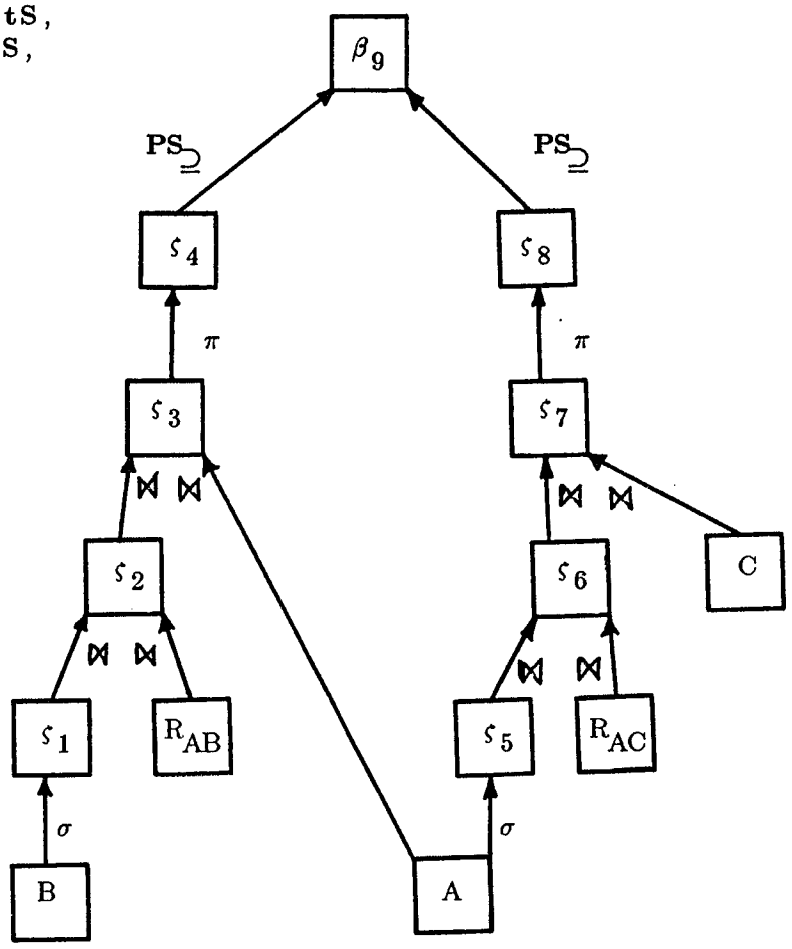


Figure 3.6.6. Transition Digraph For Example 3.6.1

Constraint :

$\varsigma_3 := (B)_{B \geq 'b3'} {}^{\bullet} R_{AB} {}^{\bullet} A,$
$\varsigma_4 := \mathbf{ProjectS}_{\{A\}} (\varsigma_3),$
$\varsigma_7 := (A)_{A \geq 'a2'} {}^{\bullet} R_{AC} {}^{\bullet} C,$
$\varsigma_8 := \mathbf{ProjectS}_{\{A\}} (\varsigma_7),$
$\beta_9 := \mathbf{PS}_{\supseteq} (\varsigma_4, \varsigma_8).$

π: **ProjectS**,
σ: **SelectS**,
$\bowtie$: **JoinS**.



β_9:= $\mathbf{PS}_{\supseteq}$ (ς_4, ς_8).

The transition digraph of the above constraint is shown in Figure 3.6.6.

The base states of the constraints are:

A:={ <a1>,<a2>,<a3>,<a4>,<a5> },

B:={ <b1>,<b2>,<b3>,<b4> },

C:={ <c1>,<c2>,<c3> },

R_{AB} := { <a1,b2>,<a1,b3>,<a2,b1>,<a3,b3>,<a4,b4>,<a5,b3>},

R_{AC} := { <a1,c2>,<a3,c2>,<a4,c1>,<a4,c4> }

Part of the transition states of the constraint are:

ς_3:={ <a1,b3>, <a3,b3>, <a4,b4>, <a5,b3> },

ς_7:={ <a3,c2>, <a4,c1>, <a4,c4> },

The target states of the constraint are:

ς_4:={ <a1>, <a3>, <a4>, <a5> },

ς_8:={ <a3>, <a4> },

And the final boolean value of the sink node is:

β_9:= $\mathbf{PS}_{\supseteq}$ (ς_4, ς_8) := TRUE. ###

Note that we can think of the calculation procedure as the following mappings:

$\varsigma_4 \subseteq$ A, or ς_4:=f1(A,B, R_{AB}),

$\varsigma_8 \subseteq$ A, or ς_8:=f2(A,C, R_{AC}).

f1, f2 are composite functions of **ProjectS**, **JoinS**, **SelectS**, which work as filters to filter out the target states ς_4 and ς_8 from the base states A,B,...,R_{AB},.... We, then, make assertions about ς_4 and ς_8.

7. SUMMARY

The Applicative Data Language (ADL) models the semantic constraint as a computation procedure. It first computes the constrained data in the entity space , as well as in the data space. Then, assertions are made on these constrained data. Users can formulate a semantic constraint by using the three basic constructs of ADL: data structures, functions, and predicates. We have introduced the first two constructs in [Lee87a]. Here, we discuss the last construct in constraint specification: predicates.

We can make assertions on states in the entity space and on d_relations in the data space. Predicates can be applied to values derived from states or d_relations. Once we have sub-assertions, we can build more complicate assertions by the boolean operators.

However, due to implementation consideration at the present time, there are cases that cannot be modeled by the current version of ADL. Dynamic constraints are assertions made on certain data before and after an update transaction. Transitive closure constraints require recursive computations of the transitive closure. The functional dependency constraints which require a

sorting function to check dependencies are generally considered as data model constraints instead of semantic constraints. All three cases cannot be modeled by ADL at the present time.

Each user defined constraint specification need to be further optimized for computation efficiency before being compiled into enforcement methods. Such optimization is achieved by reformulating user defined expressions into equivalent but more efficient expressions (Appendix E).

The advantage of ADL as constraint specification language is that the specification actually implies a computation procedure to verify data integrity. We can compile the constraint specification into a transition digraph which can be used for compile-time constraint checking.

In APPENDIX D, we use ADL to model some of the constraints mentioned in published literature [Tabo83a, Tabo83b, Morg84, Morg86, Hamm76, Nico82, Ston75].

Table 3.7.1 is a comparison ADL with three other ERM based constraint specification methods: EROS, CE and LOER. The primitives in ADL [Lee 87a] is much simpler than those in EROS [Tabo83a, Tabo83b], yet more expressive in modeling power. EROS is developed specifically for modeling structure constraints. Attributes of entities or relationships are not utilized in the specification of semantic constraints. Even though EROS can specify relational closure constraints, little detail is given as to how it can be maintained.

The rigidity of CE's syntax [Morg84, Morg86] greatly reduces its ability to handle vast varieties of semantic constraints. The structural constraint has to be specified as a single sentence equation or predicate. On each side of the equation, objects involved in the expression have to be linked by a path in the ERD. There is little implementation details of how to enforce a constraint specified by the CE.

LOER [Naka83] actually requires a schema in ERM to be transformed into a schema in the relational model before constraints can be specified by a modified domain relational calculus. The added external constructs to the First Order Logic to model constraints with aggregate functions cause great confusions over the issue of quantifiers. LOER uses a truth table in the evaluation space (a Cartesian product of all domain variables) to do constraint checking, which is not a very efficient approach when the variable domains are exceedingly large.

We claim that ADL as a semantic constraint specification language has the following advantages:

1. It is based on a high level data model, ERM.
2. It can model a wide spectrum of semantic constraints: local, structural, and complex constraints.
3. The ADL specification is augmented by both the compile-time and the run-

Table 3.7.1. Comparison of ERM-based Semantic Constraint Models.

Comparison	ADL	EROS	CE	LOER
Ref.	This Paper	[Tabo83a] [Tabo83b]	[Morg84] [Morg86]	[Naka83]
Modeling Language	Procedural	Graphical, Specifi.	Procedural	Logical, Domain Calculus
Local Constraint	YES	NO	NO	YES
Structure Constraint	YES	YES	YES	YES
Dynamic Constraint	NO*	YES	NO	NO
Negation Constraint	YES	YES	NO	YES
Recursive Constraint	NO*	YES	NO	NO
Binary Relation	YES	YES	YES	(Use Relational Data Model, not ERM)
N-ary Relation	YES	NO	NO	As above
Using Attr. To Select Entity	YES	NO	NO	YES
Primitives For Structural Constraints.	JoinS SelectS ProjectS Group_byS ∪, ∩, - and etc.	NEGATION AFFIRMATN. SCARD VCARD IMPLICATN. CONNECT	CONN. PATH PATH INTERSECT.	∧ ∨ ¬ ===> <===>

*Due to implementation consideration at the present time.

time enforcement strategies.

4. The semantic constraint is modeled as a computation procedure that can be refined step by step in a well structured manner.

5. The data structures, functions, and predicates can be expanded to model more difficult semantics in the real world. But we must evaluate the added enforcement complexities before such modifications.

CHAPTER IV. RUN-TIME CONSTRAINT ENFORCEMENT STRATEGY FOR ADL

Semantic constraint modeling in database becomes increasingly importance in recent years as a result of interest in CAD database, and AI and database integration [Fenv85a, Fenv85b, Lafu79, Shep86, Morg86, Date83, Tsic82, Fern81]. A database system need to incorporate an integrity checking subsystem that automatically insures the consistency of data in the database [Hamm76, Bert84]. Many researchers concerned with the overall architecture of such an integrity subsystem [Eswa75, Wils80, Neum82, Adib85, Flyn84, Wojc84, Doga85, Crem83, Shep86]. Others studied the user interfaces that were necessary for the integrity subsystem [Fenv85, Lafu82, Ditt86].

The first utility that must be provided by a constraint subsystem is the constraint specification language. Many ad hoc specification languages just extend existing data manipulation languages to specify semantic constraints [Ston75, Fenv85]. The most prevalent semantic specification languages are those based on First Order Logic (FOL) or Relational Calculus [Fros83, Webe83, Bern80, Gard79, Naka83, Nico82, Nico78, Ehri84, Hsu85]. This type of languages usually take advantage of the assertion specifying power of FOL and the uniform data structure of the relational data model. However, since functions are not a natural part of FOL, many of these languages have to be extended beyond that of FOL to specify complex constraints that involve aggregate functions [Naka83, Webe83].

Another approach for semantic constraint specification is the computational approach that consists of three basic constructs: data structures, functions, and predicates [Bune79, Lee87a, Lee87b]. Lee and Chen proposed a comprehensive data language called Applicative Data Language (ADL) that can model various semantic constraints as computation procedures [Lee 87b]. ADL divides the computation of finding data to two different levels of abstraction. In the entity space, it carries out the object level computation, while in the data space it carries out the data element level computation.

There are two procedures for semantic constraint enforcement. One is the compile-time enforcement which is activated after a new constraint is defined [Lee 87b, Naka83]. The compile-time constraint enforcement subjects all the data in the current database to the constraint checking algorithm. It is a very expensive operation in terms of the total data involved in the computation.

The second enforcement procedure is the run-time enforcement which represents a more important problem in constraint modeling. It is activated when a user submits an update transaction to the integrity subsystem. Assuming the data in the current database are consistent, the subsystem has to check the update against all the defined constraints and make sure that they will not be

violated after the update is realized in the database. The problem challenging many researchers is how to do run-time constraint checking effectively based on our knowledge of the semantic constraints [Hsu85, Fros83, Webe83, Bern80, Nico82, Lin84, Bune79, Lafu79].

The FOL approaches taken by many researchers solved the problem along the following directions. Frost and elta [Fros83] simplified the data model to a Simplified Binary Relation model so that semantic constraints can be limited to disjunctive forms of literals. Bernstein and elta [Bern80] and [Bern81] limited the types of constraints to those of two free assertions. Update could only be in a single relation type. Hsu [Hsu85], Nicolas [Nico82] and elta recompiled specific sets of constraints into FOL formulas that include update actions. Upon user's updates, certain run-time constraints would be triggered to check the consistency of data. But such recompilation procedures only worked for FOL formulas with the prefix parts matching certain patterns and updates limited to certain types of updates.

Buneman and elta [Bune79] proposed an avoidance algorithm to do trivial acceptance of the unrelated update with respect to a specific constraint. It is not a rigorous computation algorithm; but, for the simple constraints that it intends to handle, it is sufficient.

This research proposes a comprehensive run-time constraint checking strategy for any semantic constraint defined by ADL. Since ADL is based on a high level data model, the Entity Relationship Model (ERM), it can model a variety of semantic constraints, e.g., local, structural, and complex constraints [Lee87b]. As a result, the run-time constraint checking strategy proposed here is more general and less restrictive as compared to those of others [Hsu85, Nico82, Ston75, Webe83, Fros83]. Also, because ADL works with the higher abstraction of data (entity or relationship), we are able to untangle the computation complexity of run-time constraint checking algorithms and not to be overwhelmed by the massiveness of data.

Though few papers [Hsu85], [Nico82], [Bune79] tried to establish some formalisms representing update transactions, they were not sophisticated enough to represent the general effects of the updates on constraint computations. This research, however, defines a data structure that can represent not only the update transaction, but also its effects on the computations of semantic constraint checkings.

If the database state (DBST) is proven correct after the compile-time constraint checking [Lee87b], the user may change the current database state (DBST) to a new DBST by an update transaction. The problem is how to verify that such transaction will not result in a DBST violating the defined constraints. Certainly, compile-time checking method will be too costly for this purpose since we already know the old DBST is correct. This paper proposes a

run-time constraint checking strategy called the incremental computation strategy.

In section 1, we define new data structures that represent user's update transactions and changes in the intermediate results of constraint computation. We also define some new operators that can be applied on these new data structures. The mechanism of run-time constraint checking is represented by an incremental transition digraph. Operators in the entity space and the data space now have to be redefined as incremental operators derived from existing operators. Section 2 introduces three computation techniques for the derivation of incremental operators, i.e. partial evaluation, partial realization, and historic record keeping. Section 3 gives examples of the incremental computation strategy. Section 4 discusses some theoretical aspects of the computation strategy. Finally, in section 5 we conclude our discussion by pointing out some advantages of the incremental computation strategy.

First, let's take a look at the semantic differences of update actions. An update transaction in the database means three things- "delete", "insert", and "modify". In terms of constraint enforcement, we can think of "modify" as a two step process. It first deletes an old data from the DBST; then, adds a new data to the DBST. If the update is "delete", we delete something from the old database state. The data should **already exist** in the old DBST. If the update is "insert", we add something that **not yet exists** in the old DBST. The semantic difference of "insert" and "delete" plays an important role in later definitions and algorithms for constraint enforcement.

1. RUN-TIME CONSTRAINT CHECKING: INCREMENTAL COMPUTATION

After the database has been established, users will constantly request update transactions to modify the current DBST. The integrity subsystem has to keep track of such run-time activities to insure that the new DBST still complies with all the integrity constraints. The compile-time checking algorithm [Lee87b], however, is a very expensive way to maintain run-time constraint checking because an update transaction only affects a very small part of the database. The vast majority of data in the database are virtually unaffected by the update. Therefore, we need a more efficient strategy for run-time constraint checking.

The target states and target d_relations (defined in [Lee87b]) are our center of attention for run-time constraint checking. The objective is to monitor the incremental changes to the target states and target d_relations in the transition digraph after an update. If the old database state is true and we know all the incremental changes to the target states and d_relations, we can make confident inference about whether or not the updated database is true.

The transactions that will be discussed in this paper are those of adding entities or relationships to or deleting entities or relationships from the old DBST.

1.A. Signed Data Structures and Operators

Since the entity space operations and the data space operations of ADL [Lee87a] are based on set theoretical computations, any update to the current database state will only change the intermediate results of the computation incrementally. If the state changes at all as a result of the update, it either adds new entity vectors or deletes existing entity vectors from the old entity state. Such incremental changes are the results of the discrete computation algorithms in the entity space and the data space. The fact that the exact extent of the incremental change is theoretically computable is crucial to the development of the incremental computation strategy for run-time constraint checking. But even it is theoretically possible, that does not mean it can be done easily.

To tackle the problem of run-time constraint checking, we need to introduce new data structures that represent the update transactions and new operators that compute the effects of such transactions. The data structures defined in [Lee87a, Lee87b] represent **unsigned data structures** which include entity spaces, states, vectors, data spaces, d_relations, and tuples. We are now extending the framework to define signed data structures.

#Definition 4.1.1: **Plus Vector, Minus Vector, Signed Vector**
If $s_i = <eid_{i,1}, eid_{i,2}, ..., eid_{i,m}>$ is a vector in the entity space $\Psi = E_1 \times E_2 \times ... \times E_m$, then $\oplus <eid_{i,1}, eid_{i,2}, ..., eid_{i,m}>$ and $\ominus <eid_{i,1}, eid_{i,2}, ..., eid_{i,m}>$ are the **plus** and the **minus vectors** in Ψ, respectively. Together, they are called **signed vectors** in the entity space Ψ, expressed as s_i'. ###

#Definition 4.1.2: **Signed State**
A signed state, ς', is a set of signed vectors in the same entity space. ς' satisfies the following conditions:
(1). If $\oplus <eid_{i,1}, eid_{i,2}, ..., eid_{i,m}> \in \varsigma'$, then
$\ominus <eid_{i,1}, eid_{i,2}, ..., eid_{i,m}> \notin \varsigma'$.
(2). If $\ominus <eid_{i,1}, eid_{i,2}, ..., eid_{i,m}> \in \varsigma'$, then
$\oplus <eid_{i,1}, eid_{i,2}, ..., eid_{i,m}> \notin \varsigma'$. ###

The conditions stated in the definition simply point out that, for the same unsigned vector, its plus vector and its minus vector cannot coexist in a signed state, ς'.

#Definition 4.1.3: **Plus Tuple, Minus Tuple, Signed Tuple**
If $t_i = (v_{i,1},..,v_{i,p})$ is a tuple in the data space $Dom(F_1)$ x ... x $Dom(F_p)$, then $\oplus(v_{i,1},..,v_{i,p})$ and $\ominus(v_{i,1},..,v_{i,p})$ are the **plus** and the **minus tuple** in the data space, respectively. Together they are called **signed tuples**, expressed as t_i'. ###

#Definition 4.1.4: **Signed D_relation**
A **signed d_relation,** τ', is a bag of signed tuples in the same data space. ###

Note that since d_relations are bags of tuples, it is possible to have signed tuples of the same values but different in signs.

We define the following operators that transform unsigned data structures to signed ones and vice versa, or operate solely on the signed data structures.

(1) Operators That Transform An Unsigned State to A Signed State.

#Definition 4.1.5: **Sign** $_{\varsigma'}$ **(ς), Assign_plus (ς), Assign_minus (ς),**

(a) ς_3':= **Sign** $_{\varsigma_2'}$ (ς_1)

ς_3' and ς_1 are both of the same entity space ,Ψ = Ax...xGx...xM, ς_2' is of a subspace of Ψ, i.e. Ψ_1=Ax...xG. ς_3' is computed as follows:(i) Set ς_3' to Ø. (ii) For each $s_i=<a_i,..,g_j,..,m_k>$ in ς_1, if the corresponding $\oplus<a_i,..,g_j> \in \varsigma_2$', then s_i'= $\oplus<a_i,..,g_j,..,m_k>$ is added to ς_3'. If the corresponding $\ominus<a_i,..,g_j> \in \varsigma_2$', then s_i'= $\ominus<a_i,..,g_j,..,m_k>$ is added to ς_3'. If none of the above, proceed to the next vector in ς_1. (iii) Repeat (ii) until ς_1 is exhausted.

(b) ς_2':= **Assign_plus** (ς_1)

ς_2' and ς_1 are of the same entity space. ς_2' is computed as follows: (i) Set ς_2' to Ø. (ii) For each $s_i=<a_i,..,g_j,..,m_k> \in \varsigma_1$, add $\oplus<a_i,..,g_j,..,m_k>$ to ς_2'. (iii) Repeat (ii) until ς_1 is exhausted.

(c) ς_2':= **Assign_minus** (ς_1)

ς_2' and ς_1 are of the same entity space. ς_2' is computed as follows: (i) Set ς_2' to Ø. (ii) For each $s_i=<a_i,..,g_j,..,m_k> \in \varsigma_1$, add $\ominus<a_i,..,g_j,..,m_k>$ to ς_2'. (iii) Repeat (ii) until ς_1 is exhausted. ###

(2) Operator That Transforms A Signed State To An Unsigned State or A Signed D_relation to An Unsigned D_relation:

#Definition 4.1.6: **Unsign (ς') or Unsign(τ')**

(a) ς_2:= **Unsign** (ς_1').

ς_2 and ς_1' are of the same entity space. ς_2 is computed as follows: (i) Set ς_2 to Ø. (ii) For each $\ominus<a_i,..,g_j,..,m_k>$ or $\oplus<a_i,..,g_j,..,m_k> \in \varsigma_1$', add $<a_i,..,g_j,..,m_k>$ to ς_2. (iii) Repeat (ii) until ς_1' is exhausted.

(b) τ_2:= **Unsign** (τ_1').

τ_2 and τ_1' are of the same data space. τ_2 is obtained as follows: (i) Set τ_2 to Ø.

(ii) For each $\oplus(t_i,..,u_j,..,v_k)$ or $\ominus(t_i,..,u_j,..,v_k) \in \tau_1'$, add $(t_i,..,u_j,..,v_k)$ to τ_2.
(iii) Repeat (ii) until τ_1' is exhausted. ###

(3) Operators That Change A Signed State or A Signed D_relation:
#Definition 4.1.7: **Plus_part (ς'), Minus_part (ς'), Sign_change (ς'),** or **Plus_part (τ'), Minus_part (τ').**

(a) ς_2':= **Plus_part** (ς_1')
ς_2' and ς_1' are of the same entity space. ς_2' is built from all the plus vectors in ς_1' and nothing else.
(b) τ_2':= **Plus_part** (τ_1')
τ_2' and τ_1' are of the same data space. τ_2' is built from all the plus tuples in τ_1' and nothing else.
(c) ς_2':= **Minus_part** (ς_1')
ς_2' and ς_1' are of the same entity space. ς_2' is built from all the minus vectors in ς_1' and nothing else.
(d) τ_2':= **Minus_part** (τ_1')
τ_2' and τ_1' are of the same data space. τ_2' is built from all the minus tuples in τ_1' and nothing else.
(e) ς_2':= **Sign_change** (ς_1')
ς_2' and ς_1' are of the same entity space. ς_2' is computed as follows: (i) Set ς_2' to $\emptyset$. (ii) For each $s_i \in \varsigma_1'$, if $s_i'=\ominus<a_i,..,g_j,..,m_k>$, add $\oplus<a_i,..,g_j,..,m_k>$ to ς_2'. If $s_i'=\oplus<a_i,..,g_j,..,m_k>$, add $\ominus<a_i,..,g_j,..,m_k>$ to ς_2'. (iii) Repeat (ii) until ς_1' is exhausted. ###

#Example 4.1.1: Incremental Computation Operators. Assume the following ς and ς':
ς:= {<a2, b3>,<a2, b1>,<a3, b2>,<a1, b1>},
ς':= {$\oplus$<a2>,$\oplus$<a4>,$\ominus$<a3>}.
(1) Operators that transform unsigned states to signed states:
ς_1':=**Sign** $_{\varsigma'}$ (ς), ς_1'={$\oplus$<a2, b3>,$\oplus$<a2, b1>,$\ominus$<a3, b2>}.
ς_2':=**Assign_plus** (ς), ς_2'={$\oplus$<a2,b3>,$\oplus$<a2,b1>, $\oplus$<a3,b2>,$\oplus$<a1,b1>}.
ς_3':=**Assign_minus** (ς), ς_3'={$\ominus$<a2,b3>,$\ominus$<a2,b1>, $\ominus$<a3,b2>,$\ominus$<a1,b1>}.
(2) Operator that transforms signed states to unsigned states:
ς_4:= **Unsign** (ς'), ς_4={<a2>,<a4>,<a3>}.
(3) Operators that change one signed state to another signed state:
ς_5':=**Plus_part** (ς'), ς_5'={$\oplus$<a2>,$\oplus$<a4>}.
ς_6':=**Minus_part** (ς'), ς_6'={$\ominus$<a3>}.
ς_7':=**Sign_Change** (ς'), ς_7'={$\ominus$<a2>,$\ominus$<a4>,$\oplus$<a3>}.###

Using the primitive operators, we can define an important operator macro that will be used extensively in run-time constraint checking.

#Definition 4.1.8.: **The Realization Operator, α**
The realization operator, α, is a macro of the following format: $\varsigma_3 := \varsigma_2\ \alpha\ \varsigma_1'.\ \varsigma_1'$, ς_2, and ς_3 are of the same entity space. ς_3 is computed as:

$\varsigma_3 := (\varsigma_2$ - **Unsign** (**Minus_part** (ς_1'))) $\cup$
Unsign (**Plus_part** (ς_1')). ###

1.B. Modeling the Semantics of Update Transactions

Now, we will look at how the signed data structure can be used to model the update transactions which consist of inserting entities or relationships, deleting entities or relationships, and modifying entities or relationships. The updated entities or relationships must be in some predefined entity sets or relationship sets in the data dictionary [Lee87a]. As updates take place, the extensions of the entity sets and relationship sets change while the intensions remain the same [Tsic82]. We model "modification" as a "deletion" followed by an "insertion". Therefore, at the present time we only consider insertions and deletions.

#Definition 4.1.9: **Update_element.**
An update_element, u', is a discrete unit of the update transaction translated according to the following rules into a signed vector:
(1) If we insert an entity $e_i = (eid_i, a_{i,1}, ..., a_{i,m})$ to an entity set E (it must not exist already), we represent it as a signed vector $\oplus <eid_i>$ in the entity space Ψ where $\Psi = E$. [Lee87a]
(2) If we delete an entity $e_j = (eid_j, a_{j,1}, ..., a_{j,m})$ from an entity set E (it must exist already), we represent it as a signed vector $\ominus <eid_j>$ in the entity space Ψ where $\Psi = E$.
(3) If we insert a relationship $(eid_{i,1}, ..., eid_{i,m})$ to a relationship set R (it must not exist already), we represent it as a signed vector $\oplus <eid_{i,1}, ..., eid_{i,m}>$ in the relationship entity space Ψ where $\Psi = E_1$ x...x E_m.
(4) If we delete a relationship $(eid_{j,1}, ..., eid_{j,m})$ from a relationship set R (it must exist already), we represent it as a signed vector $\ominus <eid_{j,1}, ..., eid_{j,m}>$ in the relationship entity space Ψ where $\Psi = E_1$ x...x E_m. ###

Since inserting something that is already there or deleting something that is not there is semantically meaningless, we state explicitly in the translation rules to inhibit such things happening.

#Definition 4.1.10: **Update_set, U'**
An update_set, U', is a set of update_elements. ###

#Definition 4.1.11: **Before State, Current State, Before D_relation, Current D_relation**

Any state value that represents an E_state, R_state, or computed state before an update takes place is called the **before state**. It is express as $\varsigma^{\sim}$. Any state value that represents the corresponding E_state, R_state, or computed state after the update takes place is called the **current state**. It is represented as ς without any superscript. We define the **before d_relation**, $\tau^{\sim}$, and the **current d_relation**, τ, in the same way. ###

#Definition 4.1.12: **Incremental State and Incremental D_relation**
If $\varsigma^{\sim}$, ς are the before state and current state with respect to an update_set U', the **incremental state**, ς', with respect to the U' is computed as follows:

$\varsigma' :=$ **Assign_plus** $(\varsigma - \varsigma^{\sim}) \cup$ **Assign_minus** $(\varsigma^{\sim} - \varsigma)$.

Similarly, the **incremental d_relation** τ' is defined as

$\tau' :=$ **Assign_plus** $(\tau - \tau^{\sim}) \cup$ **Assign_minus** $(\tau^{\sim} - \tau)$.

ς' is a signed state; τ', a signed d_relation. ###

#Theorem 4.1.1: If U' = Ø, i.e., no update is taken place, then ς' and τ' are Ø.
Proof: According to Definition 4.1.11 for $\varsigma^{\sim}$ and ς, if there is no update, $\varsigma^{\sim} = \varsigma$. Thus,

$\varsigma' :=$ **Assign_plus** $(\varsigma - \varsigma^{\sim}) \cup$ **Assign_minus** $(\varsigma^{\sim} - \varsigma) :=$ Ø.

Similarly, $\tau^{\sim} = \tau$, $\tau' :=$ Ø. ###

Theorem 4.1.1 simply points out that if there is no update, all the incremental states and incremental d_relations are Ø.

#Theorem 4.1.2: If ς'=Ø, then $\varsigma = \varsigma^{\sim}$. If τ'=Ø. then $\tau = \tau^{\sim}$.
Proof: Since ς'=Ø, $|\varsigma'|$=0. Therefore, $|$**Assign_plus**$(\varsigma - \varsigma^{\sim}) \cup$ **Assign_minus**$(\varsigma^{\sim} - \varsigma)| = 0$. |**Assign_plus**$(\varsigma - \varsigma^{\sim})|$=0 and |**Assign_minus**$(\varsigma^{\sim} - \varsigma)|$=0. From the definition of **Assign_plus** and **Assign_minus**, we know |**Assign_plus**$(\varsigma - \varsigma^{\sim})| = |\varsigma - \varsigma^{\sim}| = 0$ and |**Assign_minus**$(\varsigma^{\sim} - \varsigma)| = |\varsigma^{\sim} - \varsigma| = 0$. If $|\varsigma - \varsigma^{\sim}| = |\varsigma^{\sim} - \varsigma| = 0$, we conclude $\varsigma = \varsigma^{\sim}$. In a similar fashion, we can prove if τ'=Ø. then $\tau = \tau^{\sim}$. ###

The above theorem implies that if ς' is Ø with respect to an update_set, U', we can conclude the before state and the current state are the same. That is, the update does not change the content of the entity state.

#Theorem 4.1.3: If $\oplus <e_{i,1}, \ldots, e_{i,m}> \in \varsigma'$, then $<e_{i,1}, \ldots, e_{i,m}> \in \varsigma$ and $<e_{i,1}, \ldots, e_{i,m}> \notin \varsigma^{\sim}$. If $\ominus <e_{i,1}, \ldots, e_{i,m}> \in \varsigma'$, then $<e_{i,1}, \ldots, e_{i,m}> \in \varsigma^{\sim}$ and $<e_{i,1}, \ldots, e_{i,m}> \notin \varsigma$.
Proof: If $\oplus <e_{i,1}, \ldots, e_{i,m}> \in \varsigma'$, according to the definition of ς', $<e_{i,1}, \ldots, e_{i,m}> \in (\varsigma - \varsigma^{\sim})$. Therefore, $<e_{i,1}, \ldots, e_{i,m}> \in \varsigma$ and $<e_{i,1}, \ldots, e_{i,m}> \notin \varsigma^{\sim}$. If $\ominus <e_{i,1}, \ldots, e_{i,m}> \in \varsigma'$, then $<e_{i,1}, \ldots, e_{i,m}> \in (\varsigma^{\sim} - \varsigma)$. Therefore, $<e_{i,1}, \ldots, e_{i,m}> \in \varsigma^{\sim}$ and $<e_{i,1}, \ldots, e_{i,m}> \notin \varsigma$. ###

Theorem 4.1.3. suggests semantically that if s_i' is a plus vector in ς', its unsigned

counter part must exist in the current state and not exist in the before state.

#Theorem 4.1.4: If ς, $\varsigma^{\sim}$, ς' are the current state, the before state and the incremental state with respect to an update, they satisfy the equation: $\varsigma := \varsigma^{\sim}\ \alpha\ \varsigma'$.

Proof: $\varsigma := (\varsigma \cap \varsigma^{\sim}) \cup (\varsigma - \varsigma^{\sim})$

$:= (\varsigma^{\sim} - (\varsigma^{\sim} - \varsigma)) \cup (\varsigma - \varsigma^{\sim})$

$:= (\varsigma^{\sim} - \mathbf{Unsign}\ (\mathbf{Minus\_part}(\varsigma'))) \cup \mathbf{Unsign}\ (\mathbf{Plus\_part}\ (\varsigma'))$

$:= \varsigma^{\sim}\ \alpha\ \varsigma'$.

Please refer the definition of ς' and α for further details. ###

Theorem 4.1.4 simply states the fact that if we know the before state and the incremental state, we can compute the current state. Both Definition 4.1.12 and Theorem 4.1.4 point out that $\varsigma^{\sim}$, ς, and ς' are dependent variables. Knowing any two of the them, we can compute the third.

#Definition 4.1.13: **Incremental E_state, Incremental R_state**
If we partition the update_elements in an update_set, U', according to their corresponding entity sets and relationship sets. The partitions thus obtained are called **incremental E_states** and **incremental R_states**, respectively. ###

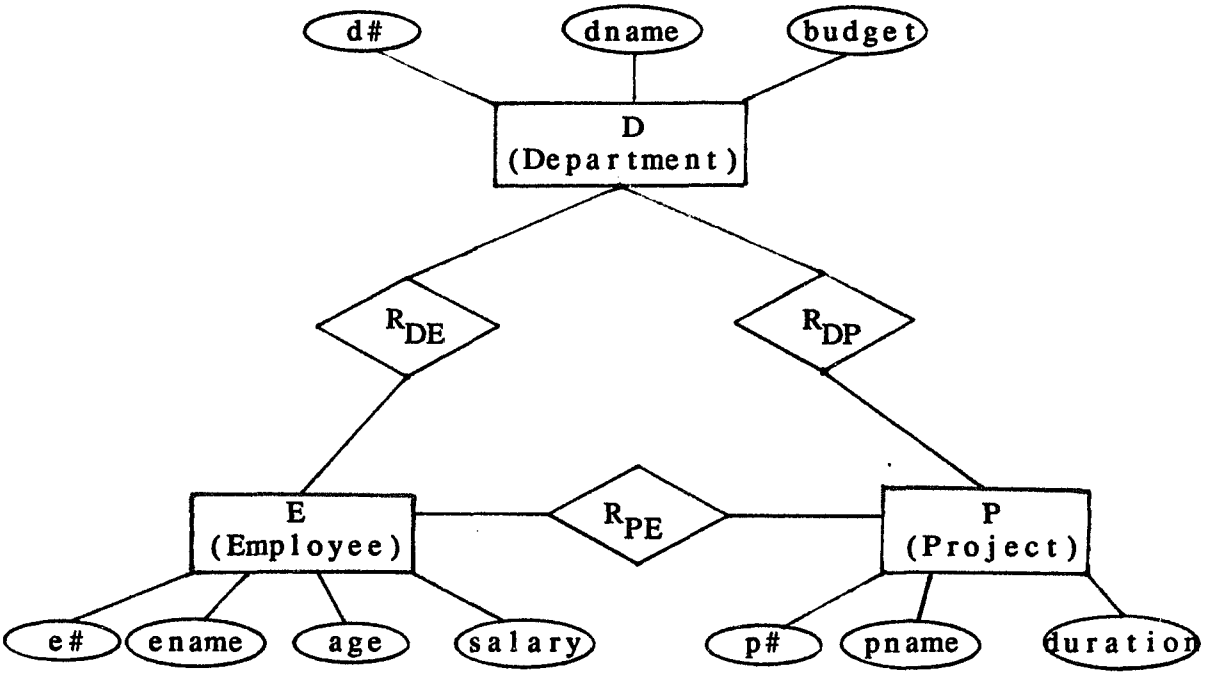
#Definition 4.1.14: **Base(U')**
If U' is an update_set, **Base**(U') is a function that returns the set of symbols of the incremental E_states and incremental R_states involved in U'. ###

#Example 4.1.2: Update_element, Update_set, Incremental E_state, Incremental R_state, Before State, Current State, **Base**(U').
If we have a database about a company and part of the database is shown in Figure 4.1.1 with its database state before an update. Figures 4.1.2-4.1.4 show semantic constraints imposed on the database.

Suppose now, we want to update the database through the following transaction:
(1) Add Employee (e101, Paul, 35, 28000) to the Employee entity set, E.
(2) Add Relationship (d1, e6) to the R_{DE} relationship set.
(3) Delete Employee (e5, Steve, 35, 33000) from the Employee entity set, E.

According to Definition 4.1.9, we can represent the update transaction as update_elements. Correspondingly, we have:
(1) $\oplus$ <e101> to the entity space Ψ where $\Psi = E$.
(2) $\oplus$ <d1, e6> to the relationship entity space Ψ where $\Psi = D \times E$.

Figure 4.1.1. Part of A Company Database



D

d#	dname	budget
d1	csc	100,000
d2	me	200,000
d3	che	300,000
...		
...		

E

e#	ename	age	salary
e1	Tom	25	28,000
e2	Mike	24	25,000
e3	Mary	23	30,000
e4	Dave	32	37,000
e5	Steve	35	33,000
e6	John	40	40,000
...			
...			

P

p#	pname	duration(year)
p1	ai	2
p2	db	3
p3	os	2
p4	robotic	4
p5	oil	2
...		
...		

R_{DE}

d#	e#
d1	e1
d1	e2
d2	e4
d3	e5
...	
...	

R_{PE}

p#	e#
p1	e1
p2	e2
p4	e3
p5	e4
...	
...	

R_{DP}

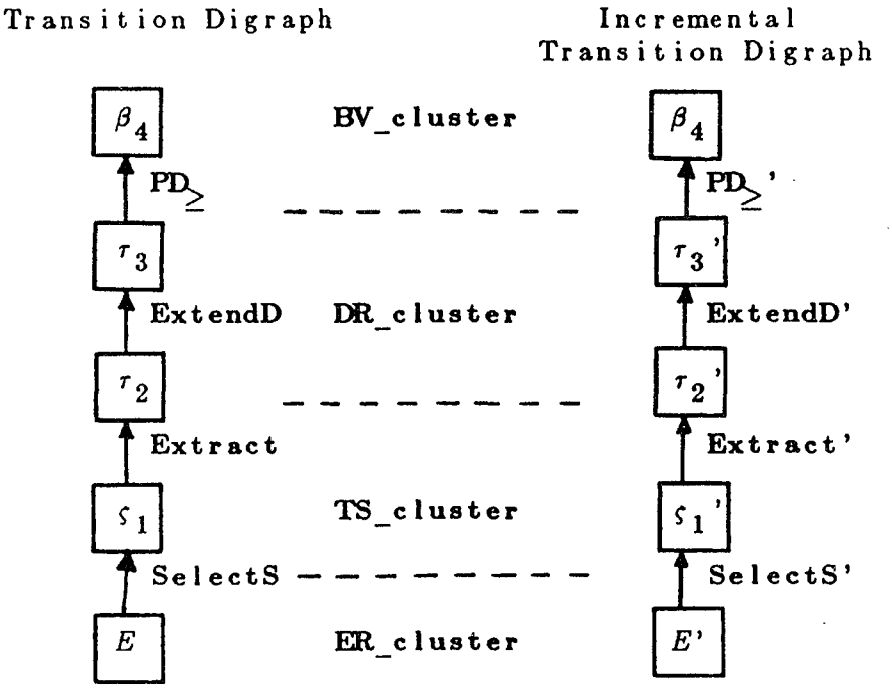
d#	p#
d1	p1
d1	p2
d1	p3
d2	p4
d3	p5
...	
...	

Figure 4.1.2. Semantic Constraint and Incremental Transition Digraph 1

Assuming that part of the database is shown in Figure 4.1.1. The instances are the database state before the update.

Semantic Constraint: Employees who are over 26 should earn at least $30,000/year.

ADL Specification: $\varsigma_1 := (E)_{E.age > 26'}$
$\tau_2 := \text{Extract}\ \{Esal <- E.salary\}\ (\varsigma_1),$
$\tau_3 := \text{ExtendD}\ \{Limit := 30,000\}\ (\tau_2),$
$\beta_4 := PD_{\geq}(\tau_3).$



Note:
1. Nodes in the BV_cluster are still values and booleans in the incremental transition digraph.
2. Computations in the BV_cluster of an incremental transition digraph are just regular boolean and value computations.
3. Nodes in the ER_cluster, TS_cluster, and DR_cluster now become incremental data structures in the incremental transition digraph.
4. All nodes in the ER_cluster of an incremental transition digraph are either incremental E_states or incremental R_states.
5. Computations in the ER_cluster, TS_cluster, and DR_cluster now are incremental computations.

Figure 4.1.3. Semantic Constraint and Incremental Transition Digraph 2

Assuming that part of the database is shown in Figure 4.1.1. The instances are the database state before the update.

Semantic Constraint: Those employees whose ages are below 30 should not exceed 20% of total employees.

ADL Specification: $\varsigma_1 := (E)_{E.age < 30}$,
$\mu_2 := \mathbf{CountS}(\varsigma_1)$,
$\mu_3 := \mathbf{CountS}(E)$,
$\mu_4 := 0.2 * \mu_3$,
$\beta_5 := \mathbf{PV}_{<}(\mu_2, \mu_4)$.

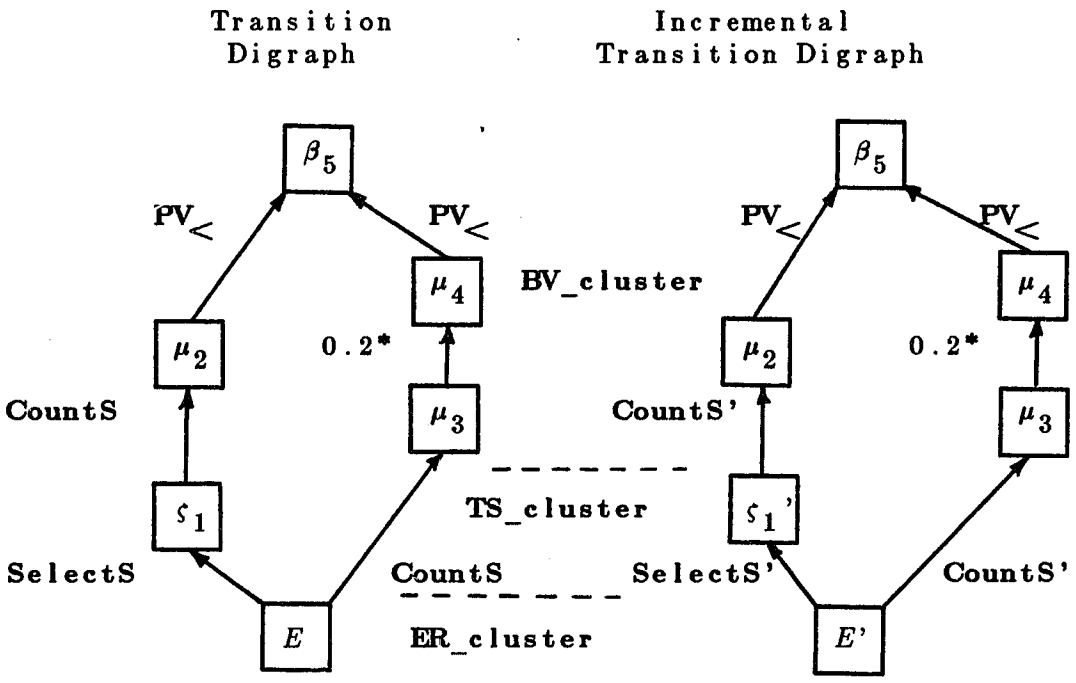
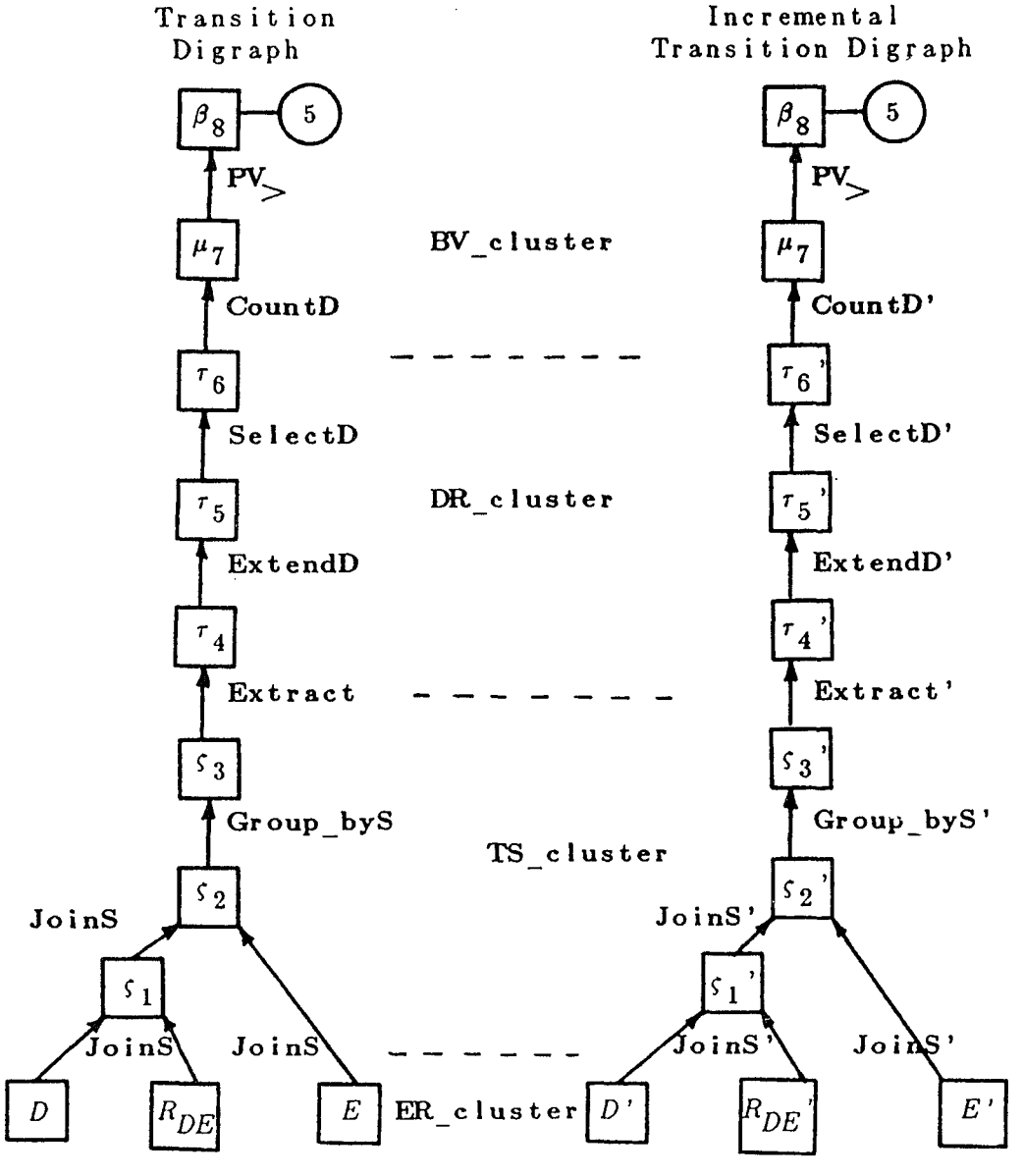


Figure 4.1.4. Semantic Constraint and Incremental Transition Digraph 3

Assuming that part of the database is shown in Figure 4.1.1. The instances are the database state before the update.

Semantic Constraint: The company should have more than 5 departments of which average employee ages are over 30.

ADL Specification:
$\varsigma_2 := D * R_{DE} * E.$
$\varsigma_3 :=$ Group_byS $_{\{D\}}$ (ς_2),
$\tau_4 :=$ Extract $_{\{Eage^* <- E^*.age\}}$ (ς_3),
$\tau_5 :=$ ExtendD $_{\{AvgAge := AvgB(Eage^c)\}}$ (τ_4),
$\tau_6 :=$ SelectD $_{AvgAge > 30}$ (τ_5),
$\mu_7 :=$ CountD (τ_6),
$\beta_8 :=$ $PV_>(\mu_7, 5)$.



(3) $\ominus$<e5> from the entity space Ψ where Ψ = E.

According the Definition 4.1.10, the update_set, U',is: U' := {$\oplus$<e101>, $\oplus$<d1, e6>, $\ominus$<e5>}. We can partition U' according to the corresponding entity and relationship sets: E' := {$\oplus$<e101>, $\ominus$<e5>}, R_{DE}' := {$\oplus$<d1,e6>}. E' and R_{DE}' are thus the incremental E_state and the incremental R_state with respect to U'. And **Base**(U') := {E',R_{DE}'}.

The before state and the current state regarding the "Employee" entity set with respect to U' are :

$E^{\sim}$:= {<e1>,<e2>,<e3>,<e4>,<e5>,<e6>, . . . ,<e100>} .

E := {<e1>,<e2>,<e3>,<e4>, <e6>, . . . ,<e100>,<e101>} .

1.C. Run-time Constraint Checking: The Incremental Computation Strategy.

The key to the success of run-time constraint checking is to do as little computation as possible while we are still able to infer the correct answer. If the constraint, for example, is a structural constraint [Lee87b] and the target states are ς_1, ς_2, the entity space calculation is to map the current DBST to the target states. i.e. ς_1:=f1(DBST), ς_2:=f2(DBST). f1, f2 are composite functions of entity space functions, $\cap$, $\cup$, -, **JoinS**, **ProjectS**, **SelectS**, **Group_ByS** [Lee87a]. And, then, apply the entity space predicate about ς_1, ς_2 as P(ς_1, ς_2). P can be any of the entity space predicates, $\mathbf{PS}_{\subseteq}$, $\mathbf{PS}_{\supseteq}$, $\mathbf{PS}_{\subset}$, $\mathbf{PS}_{\supset}$, $\mathbf{PS}_{=}$. [Lee87b] The result is either TRUE or FALSE.

When we update the database, we have two corresponding database states. One is the before database state, DBST$^{\sim}$; and the other, the current database states, DBST. Conceptually, the problem is of the following:

Known Conditions-

f1(DBST$^{\sim}$)=$\varsigma_1^{\sim}$, f2(DBST$^{\sim}$)=$\varsigma_2^{\sim}$ and P($\varsigma_1^{\sim}$, $\varsigma_2^{\sim}$) is true.

Update:= U'.

Question-

Is P(ς_1, ς_2) TRUE or FALSE ?

One of the sure way to solve the problem, but may not be the most efficient way, is to let the update realize in DBST$^{\sim}$ and, then, invoke a full evaluation (compile-time checking [Lee87b]) to compute ς_1 and ς_2 again. That is to say-

ς_1:= f1 (DBST); {Full evaluation of ς1 from current DBST.}

ς_2:= f2 (DBST); {Full evaluation of ς2 from current DBST.}

And the answer is P(ς_1, ς_2). But, this full evaluation strategy is expensive considering the fact that only a small portion of the database is updated. We are also not using the fact that the before database state, DBST$^{\sim}$, is consistent with the constraint already.

This paper, however, proposes an incremental computation strategy to

solve the same problem but with less computation complexity. The key idea is essentially of the following:

(1). We maintain the same computation structure or computation sequence as in the full evaluation. Only that we now replace the ∩, ∪, -, **JoinS**, **ProjectS**, **SelectS**, **Group_ByS** with their incremental counter parts, ∩', ∪', -', **JoinS'**, **ProjectS'**, **SelectS'**, **Group_ByS'**. These **incremental functions** are procedures or subroutines that take incremental states as their operands and generate another incremental state as their output. Therefore, the original computation procedure, f1, now become f1'; f2 become f2', so that-

ς_1':= f1'(U'); {ς_1', the incremental state of $\varsigma_1^{\sim}$.}

ς_2':= f2'(U'); {ς_2', the incremental state of $\varsigma_2^{\sim}$.}

(2) We replace the predicate P with its incremental counter part, P', an **incremental predicate**. P' takes incremental states as its operands. It is designed in such a way that: if $P'(\varsigma_1', \varsigma_2') \wedge P(\varsigma_1^{\sim}, \varsigma_2^{\sim}) \Longrightarrow P(\varsigma_1, \varsigma_2)$.

Since we have already been assured that $P(\varsigma_1^{\sim}, \varsigma_2^{\sim})$ is true, we only have to evaluate $P'(\varsigma_1', \varsigma_2')$ to infer whether $P(\varsigma_1, \varsigma_2)$ is TRUE or FALSE. Notice how tacitly we have avoided full evaluations of ς_1, ς_2 in the incremental computation strategy by concentrating the computation around update and its corresponding incremental changes.

By using equivalent expressions, a formal ADL constraint specification may be optimized for efficient run-time constraint checking. Assuming that such an optimization step is already carried out [Lee87b].

#Definition 4.1.14: **Normal Constraint Specification, Incremental Constraint Specification**

For any ADL constraint specification using the functions and predicates defined in [Lee87a],[Lee87b] is called the **normal constraint specification**. An **incremental constraint specification** is obtained from a normal constraint specification by the following rules.

(1) Replace the operators in the normal specification with the corresponding incremental operators as shown in Figure 4.1.5.

(2) Replace any entity state symbol in the normal specification with its corresponding incremental state symbol.

(3) Replace any d_relation symbol in the normal specification with its corresponding incremental d_relation symbol. ###

We now represent the run-time enforcement strategy by an acyclic directed graph called the incremental transition digraph which is constructed from the incremental constraint specification.

#Definition 4.1.15: **Incremental Transition Digraph**.

An incremental transition digraph is a digraph C(N,E) built from the

Figure 4.1.5. Normal ADL Operators and Their Incremental Operators

Normal ADL Operators	Incremental Operators
1. Entity Space Functions	
$\cup$	$\cup'$
$\cap$	$\cap'$
-	-'
JoinS	**JoinS'**
ProjectS	**ProjectS'**
SelectS	**SelectS'**
Group_byS	**Group_byS'**
CountS	**CountS'**
2. Entity Space Predicates	
$PS_{\subset}$	$PS_{\subset}'$
$PS_{\subseteq}$	$PS_{\subseteq}'$
$PS_{\supseteq}$	$PS_{\supseteq}'$
$PS_{\supset}$	$PS_{\supset}'$
$PS_{=}$	$PS_{=}'$
3. Entity space to Data Space Function	
Extract	**Extract'**
4. Data Space Functions	
SelectD	**SelectD'**
ExtendD	**ExtendD'**
ProjectD	**ProjectD'**
CountD	**CountD'**
SumD	**SumD'**
AvgD	**AvgD'**
MaxD	**MaxD'**
MinD	**MinD'**
ValueD	**ValueD'**
5. Data Space Predicates	
$PD_{>}$	$PD_{>}'$
$PD_{\geq}$	$PD_{\geq}'$
$PD_{<}$	$PD_{<}'$
$PD_{\neq}$	$PD_{\neq}'$
$PD_{\leq}$	$PD_{\leq}'$
$PD_{=}$	$PD_{=}'$

Note: Definitions of incremental operators are given in APPENDIX F.

incremental constraint specification according to the following rules:
(1). Mark each operator with a sequence number according to the computation sequence and precedence rules discussed in [Lee87b]. {The **Display** operator is ignored.}
(2). {We build the transition digraph from left to right, from bottom to top.}

For each operator in the computation sequence do:

(2.1) The operand nodes of the operator should either already exist, or not exist yet. If it does not exist yet, the operand must be an incremental E_state, incremental R_state, or constant. If it is a constant, create a constant node with a circle enclosing the constant value. Else, create a new node marked by its incremental E_state or R_state symbol.

(2.2) Create a new node and mark it by a proper data structure variable symbol designated as the result of the operation.

(2.3) Draw directed arcs from the operand nodes to resulting node. Mark the arc by the operator symbol. ###

Examples of incremental transition digraphs for run-time constraint checking are shown in Figure 4.1.2- 4.1.4.

#Definition 4.1.16: **Attributes of Nodes In the Incremental Transition Digraph.**

node.label- the label of the node, e.g. $\varsigma 5'$.

node.order- the order of evaluation for the node in the entire computation sequence.

node.new_value- the newly computed value of the node, e.g. TRUE.

node.last_value-the last value of evaluation of this node.

node.pre_op- the set of the previous operation (function or predicate symbols) that compute this node, e.g. {**ProjectS'**}. For any source node, node.pre_op:=Ø.

node.pre- the set of node labels that derive this node, e.g. {$\varsigma 2'$, $\varsigma 3'$}. For source nodes, node.pre=Ø.

node.post- the set of node labels that are derived from this node, e.g. {$\varsigma 10'$, $\beta 2$, $\mu 2$}. For the sink node, node.post=Ø.

node.base- the set of labels of all the incremental E_states and R_states that are needed in deriving this node, e.g. $\{A',B',C',R_{AB}',R_{BC}'\}$

node.E_state- the set of labels of all the incremental E_states that are needed in deriving this node, e.g. $\{A',B'\}$. Note that node.E_state $\subseteq$ node.base. ###

1.D. Computation Through the Incremental Transition Digraph.

The incremental transition digraph represents a run-time checking method.

It provides a predefined structure to carry out the run-time constraint computations. When an update is requested, we have to instantiate all the nodes in the incremental transition digraph with appropriate values corresponding to the update. Some node values in the digraph can be determined easily just by checking their relevance to the update.

#Definition 4.1.17: **Marked Node.**
A marked node, n_i, with respect to an update_set, U', is any node on the incremental transition digraph that has n_i.base $\cap$ **Base**(U') $<>$ $\emptyset$.
###

Now, let's look at how the run-time constraint checking is carried out.

#Algorithm 4.1.1: **Global Run-time Constraint Checking.**
(1) Translate the semantic of an update transaction according to Definition 4.1.9 into an Update_set, U'.
(2) Partition the update_elements of U' by their belonging entity spaces into individual incremental E_states and incremental R_states. Thus, we also have determined the value of **Base**(U').
(3) Repeat the following steps for each incremental transition digraph registered in the database until exhausted (sequential checking).

(3.1). Find the sink node of the incremental transition digraph (the final boolean node), β.

(3.2). If β is marked (Definition 4.1.17), go to the local run-time constraint checking algorithm for this particular incremental transition digraph.

(3.2). Go to the next incremental transition digraph. Note that step (3.2) determines if U' is relevant to the constraint at all. If β is marked, then it is possible that the truth value of the constraint might be affected by U'. However, if β is unmarked, U' is totally irrelevant to the constraint. The following algorithm check the individual constraint that have been identified in Algorithm 4.1.1 as a possible candidate affected by U'.

#Algorithm 4.1.2: **Local Run-time Constraint Checking.**
This algorithm is carried out when a particular incremental transition digraph is identified as relevant to U'.
(1) Reset the new_value attribute of each node to null.
(2) Instantiate new values for the nodes in the incremental transition:

(2.1) If the node is unmarked (Definition 4.1.17) -
If the node is a value node, μ_i,
or a boolean node, β_j -
μ_i.new_value := μ_i.last_value,
or β_j.new_value := β_j.last_value.

If the node is an incremental E_state, R_state, state, or d_relation node, n_k, n_k.new_value:= Ø..

(2.2) If the node is marked and the node is an incremental E_state or R_state, n_l, n_l.new_value:= the corresponding partition of U'.

(3) For those node values remain undefined (must be marked), we have to go through the computation sequence to compute their new_values. Exactly how each new_value is computed is discussed in the next section and APPENDIX F.

###

Figure 4.1.6-4.1.8 show how local run-time constraint checkings are carried out for semantic constraints specified in Figure 4.1.2-4.1.4.

2. INCREMENTAL FUNCTIONS AND PREDICATES

In the incremental computation strategy, the same computation structure as that of the normal constraint specification is reserved. We only replace the operators and data structures with their incremental counter parts to obtain the incremental constraint specification from which we build the incremental transition digraph. In this section we will look at how the incremental functions and predicates are defined or computed.

First, not all ADL operators [Lee87a, Lee87b] have their incremental counter parts. Those ADL operators that work on values and booleans still remain the same in the incremental computation (shown as operators in the BV_cluster in Figure 4.1.2-4.1.4). Only those that transform states or d_relations to another states, d_relations, values , or booleans have their corresponding incremental operators (Figure 4.1.5).

The incremental operators delineated in APPENDIX F are presented in two forms: one form is purely definitional; the other, computational. Simple incremental operators like **SelectS'**, **Extract'**, **ExtendD'**, **ProjectD'**, **SelectD'** are presented by giving their definitions of how these operations are performed. They are actually the same as the normal ADL operators except that now they work on signed data structures. The rest of the incremental operators in APPENDIX F are presented as computation procedures or subroutines of normal ADL operators and operators of signed data structures (see Definition 4.1.5 - 4.1.8).

Ideally, the incremental operators should only take incremental states or incremental d_relations as their inputs. But, however, just based on such input is not enough to carry out the computation. It requires three common computation techniques to achieve the goal:

(1). Partial Evaluation- to get partial information from the before state or before d_relation.

Figure 4.1.6. Incremental Computation of Semantic Constraint 1

Assuming that part of the database is shown in Figure 4.1.1. The instances are the database state before the update.

The constraint specification, transition digraph, and incremental transition digraph are shown in Figure 4.1.2. The values shown here are corresponding values of the nodes in the transition digraph (full evaluation) and the incremental transition digraph (incremental computation).

Full Evaluation Before The Update	Incremental Computation	
	Case 1	Case 2
β_4 := TRUE	β_4 := TRUE	β_4 :=FALSE
τ_3 := {(37000,30000) (33000,30000), (40000,30000), ...}	τ_3' := {⊖(33000,30000)}	τ_3' := {⊖(37000,30000), ⊕(28000,30000)}
τ_2 := {(37000), (33000), (40000), ...}	τ_2' :={⊖(33000)}	τ_2' :={⊖(37000), ⊕(28000)}
ς_1 := {<e4>, <e5>, <e6>, ...}	ς_1' :={⊖<e5>}	ς_1' :={⊖<e4>, ⊕<e101>}
E:={<e1>, <e2>, <e3>, <e4>, <e5>, <e6>, ...}	E' :={⊖<e5>}	E' :={⊖<e4>, ⊕<e101>}
	Update U' := {⊖<e5>, ⊕<d1, e3>}	Update U' := {⊖<e4>, ⊕<e101>, ⊖<d1 p1>}

Note:

(1). In Case 2 of the incremental computation, the update is to add entity (e101, Paul, 35, 28000) to the entity set, E, delete entity (e4, Dave, 32, 37000) from E, and delete relationship (d1, p1) from R_{DP}. Only the first two update_elements are relevant to this constraint.

(2). The result of the Case 2 incremental computation is FALSE. Therefore, the system has to give warnings to the user that his update violates this particular semantic constraint and rolls back the update transaction.

(3). We only show the result of each step of the incremental computation. Exact detail of how to carry out the operation is explained in APPENDIX F.

Figure 4.1.7. Incremental Computation of Semantic Constraint 2

Assuming that part of the database is shown in Figure 4.1.1. The instances are the database state before the update.

The constraint specification, transition digraph, and incremental transition digraph are shown in Figure 4.1.3 The values shown here are corresponding values of the nodes in the transition digraph (full evaluation) and the incremental transition digraph (incremental computation).

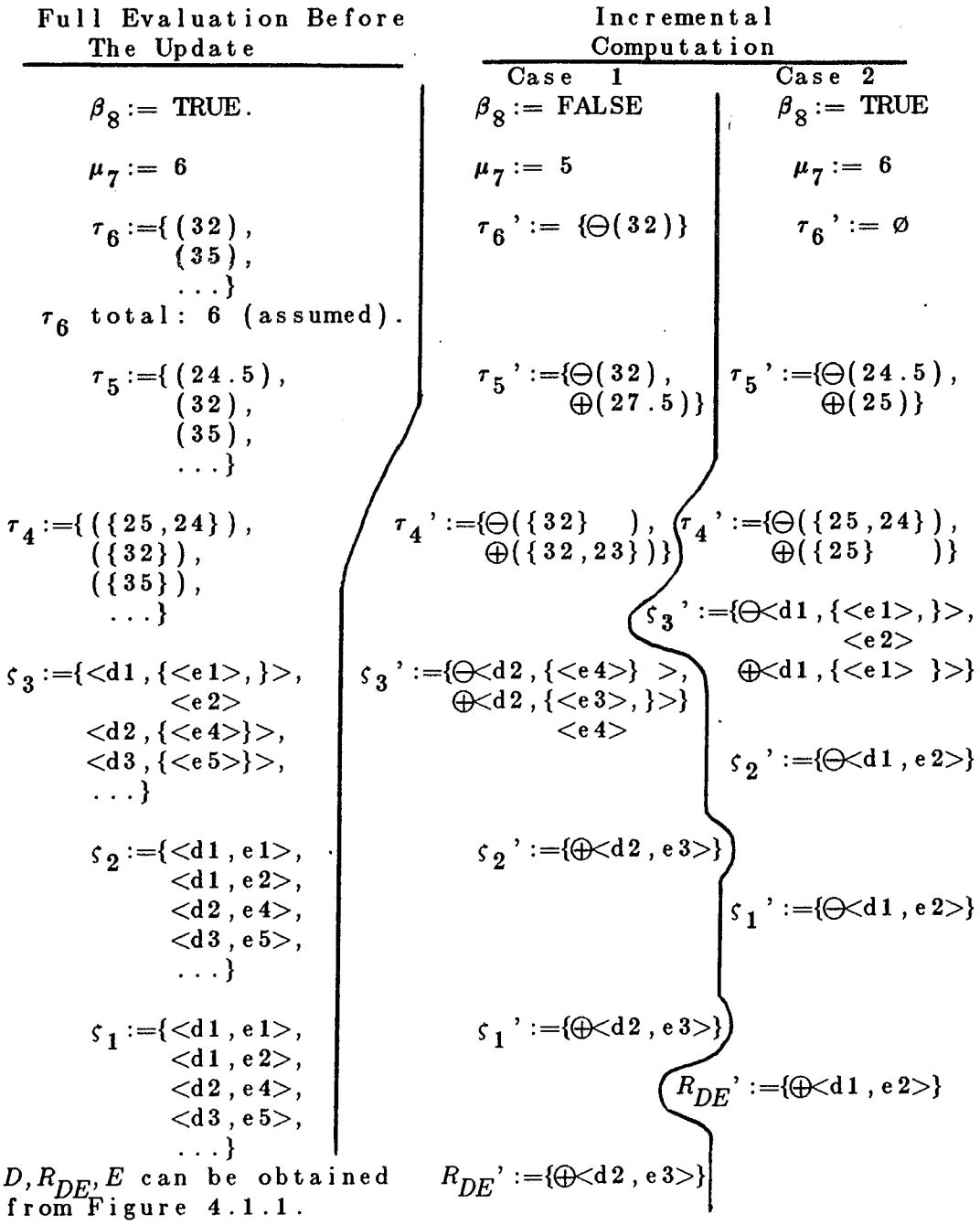
Full Evaluation Before The Update	Incremental Computation	
	Case 1	Case 2
Certain values are assumed.		
β_5 := TRUE	β_5 := TRUE	β_5 := FALSE
μ_4 := 20	μ_4 := 19.8	μ_4 := 19.8
μ_3 := 100	μ_3 := 99	μ_3 := 99
μ_2 := 19	μ_2 := 19	μ_2 := 20
ς_1 := {<e1>, <e2>, <e3>, ...}	ς_1 ' :=∅.	ς_1 ' :={⊕<e102>}
ς_1 total: 19.		
E :={<e1>, <e2>, <e3>, <e4>, ... <e100>}.	E' :={⊖<e4>}	E' :={⊖<e5>, ⊖<e6>, ⊕<e102>}
E total: 100.	Update U' :={⊖<e4>, ⊕<d11>}	Update U' :={⊖<e5>, ⊖<e6>, ⊕<e102>, ⊖<p2, e2>}

Note: In Case 2 of the incremental computation, entity (e102, Harry, 28, 33000) is added to the entity set, E.

Figure 4.1.8. Incremental Computation of Semantic Constraint 3

Assuming that part of the database is shown in Figure 4.1.1. The instances are the database state before the update.

The constraint specification, transition digraph, and incremental transition digraph are shown in Figure 4.1.4. The values shown here are corresponding values of the nodes in the transition digraph (full evaluation) and the incremental transition digraph (incremental computation).



(2). Partial Realization- to let the update realized in a sub-state or sub-d_relation to carry out computation.

(3). Historic Recording Keeping- to keep track of the last evaluation results to compute the current values. We will examine each one of them in detail here.

One important principle of deriving the incremental operators is the principle of referential independence. For example, if ς_3':=**Op**'(ς_1', ς_2') where **Op**' is an incremental operator, the computation of ς_3' should solely be based on the current values of ς_1' and ς_2' not on how they are derived. This referential independence principle will allow the incremental operators to be more general and not dependent on the specific order or combination of operators. Therefore, any semantic constraint that is definable by ADL can be compiled into an efficient run-time checking, the incremental computation strategy, and not subjected to any other restriction.

2.A. Computation of Values and Booleans in the Incremental Transition Digraph.

As shown in Figure 4.1.2.-4.1.4., in the incremental transition digraph, value nodes and boolean nodes always assume their actual values as if the Update is realized. There is no such thing as incremental value or incremental boolean in the computation. However, we need somehow to transform the results obtained in incremental states and incremental d_relations into values and booleans. This is the task of the **target incremental functions** and **target incremental predicates**. Literally, we only carry out the incremental computation up to the target incremental functions and predicates. Once values and booleans are derived, we resume regular computation of values and booleans.

#Definition 4.2.1: **Target Incremental Predicates, and Target Incremental Functions.**
Any incremental function or predicate that transforms an incremental state or d_relation to a value or a boolean is called a target incremental function or a target incremental predicate. ###

According to this definition, we can identify the target incremental predicates and functions of ADL as:

1. Target Incremental Functions and Target Incremental Predicates in the Entity space:

   Functions: **CountS'**.

   Predicates: $\mathbf{PS}_{\subseteq}$', $\mathbf{PS}_{\supseteq}$', $\mathbf{PS}_{\subset}$', $\mathbf{PS}_{\supset}$', $\mathbf{PS}_{=}$'.

2. Target Incremental Functions and Target Incremental Predicates in the Data

Space:

Functions: **SumD'**, **CountD'**, **SumD'**, **MinD'**, **MaxD'**, **ValueD'**.

Predicate: $\mathbf{PD}_{<}$', $\mathbf{PD}_{>}$', $\mathbf{PD}_{\leq}$',$\mathbf{PD}_{\geq}$',$\mathbf{PD}_{=}$'.

Once we have evaluated all the nodes immediately following the target incremental predicates and the target incremental functions, the rest of the computations become trivial.

2.B. Partial Evaluation (For Entity Space Computation Only).

A few observations of the computation in APPENDIX F will reveal that not all incremental computations are straight forward if we only know the incremental states in the entity space computation. There are many instances that we need to know part of the before entity state in order to calculate the next incremental state.

#Example 4.2.1: Difference Between **ProjectS'** and **ProjectS**

Assuming **ProjectS'** is an incremental operator in the entity space, ς_1':= { $\ominus$<a1,b1> }, ς2':=**ProjectS'** $_{\{A\}}$ (ς_1'), ς_2~:= **ProjectS** $_{\{A\}}$ (ς_1~). Intuitively, we would assume **ProjectS'** is the same as **ProjectS**, so

ς_2':= { $\ominus$<a1> }.

This is indeed a very good first guess. But in reality, it is not always the right answer. The problem stems from the fact that ς_2' has to be a valid incremental state as any other incremental states or incremental d_relations. If we take the projection on ς_1' to get ς_2', we are not really sure whether or not <a1,b1> is the only vector responsible for the existence of <a1> in ς_2~. For example, if <a1,b2> is also in ς_1~ besides <a1,b1>, then deleting <a1,b1> will not result in the deletion of <a1> from ς_2~. Therefore, ς_2':=$\emptyset$. To compute this answer, we will have to look into ς_1~ to find out all the vectors with "a1" as its first argument. This leads us to the next topic: partial evaluation or context directed evaluation. The name is derived from the fact that we are not interested in all the vectors in ς_1~, but only a small subset of ς_1~. Exactly which subset that we are looking for depends on the kind of operators and the incremental states. ###

Before we explain how to do partial evaluation in the entity space using a trace back operator, we need one more function besides those already mentioned in the entity space [Lee87a].

#Definition 4.2.2: **SelectSS and Filter State**

The set select function **SelectSS** is of the following format: ς2:= **SelectSS** $_{\varsigma}$ (ς1), with ς_1, $\varsigma_2 \subseteq \Psi_1$ and $\varsigma \subseteq \Psi_2$. Ψ_1= A x...x G x H x...x L, Ψ_2= A x...x G.

ς_2 is obtained by: For each $<a_i,...,g_j,h_k,...,l_m> \in \varsigma_1$, if $<a_i,...,g_j> \in \varsigma$, we add $<a_i,...,g_j,h_k,...,l_m>$ to ς_2. Any of the A ,.. ,G, H,.. , L can be a compound

dimension. ς is called a **filter state**. ###

SelectSS enables us to sort out desirable vectors from a specific entity state, ς_1, by means of a filter state, ς. If ς is related to an update and ς_1 is a before state (Definition 4.1.11), the result of **SelectSS** is a substate of the before state that are needed for the incremental computation.

#Example 4.2.2: **SelectSS**
If ς_1:= {<a1,b1>,<a2,b2>,<a2,b3>,<a3,b4>}, and ς:= {<a1>,<a3>}, then ς_2:= **SelectSS** $_{\varsigma}(\varsigma_1)$:= {<a1,b1>,<a3,b4>}. ###

Now, we are in a position to discuss the most important operator for partial evaluation, the trace back operator, **Trace_bk**. Most operators we have seen so far are defined by their effects on data structures. They do not give details of how the operations are carried out. The trace back operator, however, is a meta operator that not only defines the result of the operation, but also implies an evaluation strategy of how the result should be computed. Therefore, we define **Trace_bk** in terms of its effect and its evaluation strategy. We define an evaluation strategy of a computed state ς by its subgraph, **Subgraph**(ς), in the entire transition digraph [Lee87b].

#Definition 4.2.3: **Trace_bk, Partial Evaluation**
The trace back operator **Trace_bk** is an operator that defines a mapping from one state to another in the entity space and also defines an evaluation strategy for computing the result. ς_2:= **Trace_bk** $_{\varsigma}(\varsigma_1)$ has two meanings-
(1). The result of the operation, ς_2, is exactly the same as ς_2:= **SelectSS** $_{\varsigma}(\varsigma_1)$.
(2). **Partial Evaluation** is the process of evaluating **SelectSS** $_{\varsigma}(\varsigma_1)$. The **Trace_bk** operation is carried out according to one of the following partial evaluation methods.

(a). **Dynamic Partial Evaluation**: We can do partial evaluations at run-time, i.e., compute the subset of ς_1 by modifying the evaluation strategy of ς_1. **Subgraph**(ς_2):= **Partial_eval**(ς, **Subgraph**(ς_1)). **Partial_eval** is an optimization function defined in APPENDIX G. Since we are only evaluating a subset of ς_1 according to the filter state ς, **Partial_eval** optimizes and modifies the evaluation strategy of **Subgraph**(ς_1), and returns it as **Subgraph**(ς_2). If ς_2 is only a small subset of ς_1, **Subgraph**(ς_2) should cost much less to evaluate than **Subgraph**(ς_1).

(b) **Static Partial Evaluation**: We can keep a historic record of last evaluation result of ς_1. Whenever a subset of ς_1 is needed, we can just open up the historic record to filter out the subset, ς_2, without doing the re-evaluation. It trades the storage space for run-time efficiency. The historic record needs to be updated after each update is committed. The overhead of managing the historic records can be large. ###

Of the two partial evaluation methods, each has its drawbacks and its advantages. Dynamic partial evaluation has the advantage of easy implementation and requires very little overhead. But it can be very expensive if the incremental computation calls for many partial evaluations. Static partial evaluation, on the other hand, gives a more stable computation cost for any number of partial evaluations. But, the overhead of managing the historic records can be a burden to the integrity subsystem. It is possible for the database administrator to decide certain semantic constraints using the dynamic partial evaluation, while others using the static partial evaluation.

The **Partial_eval** function mentioned in the dynamic partial evaluation, in essence, minimizes the computation cost for the **Trace_bk** operation. It adds new nodes as filtering points in **Subgraph**(ς_1) to sort out desirable vectors and push down the selections as far down the digraph as possible. Detail algorithm for the **Partial_eval** is presented in APPENDIX G.

So far, we have defined **Trace_bk** purely from a syntactical point of view. Semantically, however, the **Trace_bk** operator is always applied to the before states. That is, it is always used in the following format:

$\varsigma_{11}\tilde{}$:= **Trace_bk** $_{\varsigma}$ ($\varsigma_1\tilde{}$).

And $\varsigma_{11}\tilde{} \subseteq \varsigma_1\tilde{}$. ς is an unsigned entity state obtained from ς' by ς:= **Unsign**(ς'). Once we have $\varsigma_{11}\tilde{}$, we may calculate another incremental state by: ς_2':= Op'(ς', $\varsigma_{11}\tilde{}$). This is exactly the purpose of partial evaluation.

We assume that the **static partial evaluation** method is used through out the remaining of this paper.

#Example 4.2.3.: Partial Evaluation

The partial evaluation technique is only for incremental computations in the entity space. Let's look at how the partial evaluation is used to compute **Group_ByS'**. In Figure 4.1.4 and Figure 4.1.8, Case 1 of the incremental computation, we have ς_2':={$\oplus$<d2,e3>}. The next computation is to find ς_3', ς_3':= **Group_byS'**$_{\{D\}}$(ς_2'). By examining the operation of **Group_byS** and ς_2', we find that more information about the before state of ς_2', i.e., $\varsigma_2\tilde{}$, is needed in order to compute ς_3'. But we are not interested in all the $\varsigma_2\tilde{}$, only a subset $\varsigma_{22}\tilde{}$ of it. In this subset $\varsigma_{22}\tilde{}$, all the vectors should have the same "d2" as their first arguments. According to APPENDIX F, we can follow the computation procedure as (some variable names are changed to adapt to this example):

Known- ς_2':={$\oplus$<d2,e3>}, **Unknown-** ς_3'.

ς_4:= **Unsign** (ς_2') := {<d2, e3>}, - - - - - - - - (1)

ς_5:= **ProjectS** $_{\{D\}}$ (ς_4) := {<d2>}, - - - - - - - - (2)

$\varsigma_{22}\tilde{}$:= **Trace_bk** $_{\varsigma_5}$ ($\varsigma_2\tilde{}$) := {<d2, e4>}, - - - - - - - - (3)

Note that steps (1) and (2) compute the filter state, ς_5, from the incremental state. The kind of operation that this operator will perform, in this case the

Group_byS, determines how the filter state is computed from the incremental states. Once we have the filter state, it is used in step (3) to sort $\varsigma_{22}^{\sim}$ from $\varsigma_2^{\sim}$. We complete the computation:

$\varsigma_{33}^{\sim}$:= **Group_byS** $_{\{D\}}$ $(\varsigma_{22}^{\sim})$:= {<d2, {<e4>}>}, - - - (4)
ς_9' := **Assign_minus** $(\varsigma_{33}^{\sim})$:= {⊖<d2, {<e4>}>}, - - - (5)
ς_{22} := $\varsigma_{22}^{\sim}$ α ς_2' := {<d2, e4>,<d2, e3>}, - - (6)
ς_{33} := **Group_byS** $_{\{D\}}$ (ς_{22}) := {<d2, {<e3>,<e4>}>}, - - (7)
ς_{10}' := **Assign_plus** (ς_{33}) := {⊕<d2, {<e3>,<e4>}>}, - (8)
Finally, ς_3' := $\varsigma_9' \cup \varsigma_{10}'$:= {⊖<d2, {<e4>}>,
⊕<d2, {<e3>,<e4>}> }. - (9)

Steps (1) through (9) are exactly the same computation sequence for Group_byS' in APPENDIX F. If we analyze what are the exact input for Group_byS', we will find that ς_3':= **Group_byS'** $(\varsigma_2', \varsigma_{22}^{\sim})$. ###

2.C. Partial Realization (For Entity space Computation Only)

The partial realization technique for incremental computations is based on the notion that any update only affects a small part of the before entity state.

#Theorem 4.2.1: If ς_1', ς_1, and $\varsigma_1^{\sim}$ are the incremental state, the current state, and the before state with respect to an update, and $\varsigma_{11}^{\sim} \subseteq \varsigma_1^{\sim}$, then:

ς_s:= $\varsigma_{11}^{\sim}$ α ς_1', $\varsigma_s \subseteq \varsigma_1$.

Proof: By Theorem 4.1.4 and Definition 4.1.8,

ς_1 := $\varsigma_1^{\sim}$ α ς_1' := ($\varsigma_1^{\sim}$ - **Unsign(Minus_part(ς_1'))** ∪
Unsign(Plus_part(ς_1'))

ς_s := $\varsigma_{11}^{\sim}$ α ς_1' :=($\varsigma_{11}^{\sim}$ - **Unsign(Minus_part(ς_1'))** ∪
Unsign(Plus_part(ς_1'))

Since $\varsigma_{11}^{\sim} \subseteq \varsigma_1^{\sim}$, by the definitions of set operations, we thus prove $\varsigma_s \subseteq \varsigma_1$. ###

#Definition 4.2.4: **Partial Realization**

If ς_1', ς_1, and $\varsigma_1^{\sim}$ are the incremental state, the current state, and the before state with respect to an update, $\varsigma_{11}^{\sim} \subseteq \varsigma_1^{\sim}$, and ς_{11}:= $\varsigma_{11}^{\sim}$ α ς_1', we say ς_{11}, a subset of ς_1 (Theorem 4.2.1) is obtained through **partial realization** of ς' on $\varsigma_{11}^{\sim}$, a subset of $\varsigma_1^{\sim}$. ###

The best way to understand how the partial realization is applied to the incremental computation is to go through an actual example.

#Example 4.2.4: Partial Realization

As we have seen in Example 4.2.1 we need do partial evaluation to compute ς_2'

where ς_2':= **ProjectS'** $_{P\_set}$ (ς_1'). The entire computation sequence is shown in APPENDIX F as follows:

ς_4 := **Unsign**(ς_1'), - (1)
ς_5 := **ProjectS** $_{P\_set}$ (ς_4), - - - - - - - - - - - - - - - - (2)
$\varsigma_{11}^{\sim}$:= **Trace_bk** $_{\varsigma 5}$ ($\varsigma_1^{\sim}$), - - - - - - - - - - - - - - - - (3)
$\varsigma_{22}^{\sim}$:= **ProjectS** $_{P\_set}$ ($\varsigma_{11}^{\sim}$), - - - - - - - - - - - - - - - - (4)
ς_{11} := $\varsigma_{11}^{\sim}$ α ς_1', - - - - - - - - - - - - - - - - - - - (5)
ς_{22} := **ProjectS** $_{P\_set}$ (ς_{11}), - - - - - - - - - - - - - - - - (6)
ς_2' := **Assign_plus**(ς_{22} - $\varsigma_{22}^{\sim}$) $\cup$
Assign_minus($\varsigma_{22}^{\sim}$ - ς_{22}). - - - - - - - - - - - - (7)

Note that steps (1) and (2) are to find the filter state, ς_5 which is used in (3) to do partial evaluation. By the definition of **Trace_bk** $\varsigma_{11}^{\sim} \subseteq \varsigma_1^{\sim}$. From $\varsigma_{11}^{\sim}$, we use the normal **ProjectS** to get $\varsigma_{22}^{\sim}$, a subset of $\varsigma_2^{\sim}$. Step (5), we apply **partial realization** of ς_1' to $\varsigma_{11}^{\sim}$, the subset of $\varsigma_1^{\sim}$. The result is ς_{11}, a subset of ς_1. Again, we apply the normal **ProjectS** operator to ς_{11} to obtain ς_{22}, a subset of ς_2. Then ς_2' is calculated from ς_{22} and $\varsigma_{22}^{\sim}$ in step (7). According to Definition 4.1.12, originally ς_2' should be:

ς_2':= **Assign_plus**(ς_2 - $\varsigma_2^{\sim}$) $\cup$ **Assign_minus**($\varsigma_2^{\sim}$ - ς_2).

Comparing it with equation (7), we see ς_2 is replaced by its subset ς_{22} and $\varsigma_2^{\sim}$ by its subset $\varsigma_{22}^{\sim}$. But, ς_{22} is computed from ς_{11} in step (6); $\varsigma_{22}^{\sim}$ is from $\varsigma_{11}^{\sim}$ in step(4). And, ς_{11} is computed from $\varsigma_{11}^{\sim}$ and ς_1' by partial realization. ###

The tactics of partial realization is that, instead of compute ς_2' from its full set ς_2 and $\varsigma_2^{\sim}$, we look at their subset ς_{22} and $\varsigma_{22}^{\sim}$. If $\natural\varsigma_{22}\natural$ or $\natural\varsigma_{22}^{\sim}\natural << \natural\varsigma_2\natural$ or $\natural\varsigma_2^{\sim}\natural$ we can expect considerable savings in computation cost. Since most updates only affect a small subset of the state, we may take advantages of this fact by letting the update realized in a small subset, checking the outcome of the update, and then inferring its overall effects.

2.D. Historic Record Keeping

Historic records are data recorded after the last evaluation of a node in an incremental transition digraph. It can be a value, a boolean, a state, or d_relation depending largely on the kind of operators involved. The purpose of the historic record is to facilitate the incremental computation so that run-time evaluation cost can be minimized. The place where the historic record is widely used is the evaluation of aggregate functions (see APPENDIX F).

#Example 4.2.5: Historic Record Keeping
According to Figure 4.1.3 and 4.1.7 Case 2, E':= {$\ominus$<e5>, $\ominus$<e6>,$\oplus$<e102>}. And μ_3:= CountS'(E'). The node corresponding to μ3 keeps a historic record

μ_3.last_value. According to APPENDIX F, computation for CountS' is:

μ_3.new_value := μ_3.last_value + |**Unsign** (**Plus_part**(*E'*))|
- |**Unsign** (**Minus_part**(*E'*))|
:= 100 + 1 - 2 := 99. ###

The task of evaluating incremental predicates is not an easy one. The major problem is when the predicate is last evaluated as FALSE. Then, we may also need to keep a historic record for the cause of the failure.

#Example 4.2.6: Predicate Is Evaluated As FALSE

Assume $\varsigma 1$, $\varsigma 2$ are states. ς_1~:={ <a1>, <a2>, <a3>, <a4>}, and ς_2~:={<a1>, <a4>} We know $\mathbf{PS}_{\subseteq}(\varsigma_1$~,$\varsigma_2$~) is FALSE. Now, if we make an update of ς_1~ and ς_2~ to correct the problem: ς_1':= {$\ominus$<a2>} and ς_2':={$\oplus$<a2>}, we need to do partial evaluations first.

ς_5 := **Unsign**(ς_1') $\cup$ **Unsign**(ς_2') := {<a2>},
ς_{11}~ := **Trace_bk** $_{\varsigma 5}$ (ς_1~) := {<a2>},
ς_{22}~ := **Trace_bk** $_{\varsigma 5}$ (ς_2~) := ∅,

Let ς_1' and ς_2' be partially realized in their corresponding substates:

ς_{11} := ς_{11}~ α ς_1' := ∅,
ς_{22} := ς_{22}~ α ς_2' := {<a2>},

so $\mathbf{PS}_{\subseteq}(\varsigma_{11},\varsigma_{22})$ is TRUE. Therefore, we may conclude $\mathbf{PS}_{\subseteq}(\varsigma_1,\varsigma_2)$ is also TRUE. But, in reality, it is not so. The problem is that information based on ς_1' and ς_2' alone is not enough to conclude the answer if the predicate is last evaluated as FALSE. Because <a3> is one of the reason causing the failure, but it does not appear anywhere in the update. Therefore, there is no way the update can relate to <a3> through partial evaluation. ###

It may seem like that we have no alternative but to do a full evaluation of the boolean node if the last evaluation is FALSE. But if run-time performance is all that matters, we may store additional values as attributes of the boolean node. This alternative may be well justified if run time response is more important than storage spaces.

To avoid full evaluation for cases where the last evaluation is FALSE, we can record as attributes the elements that are responsible for the failure. Note that such historic records are only necessary when the last evaluation is FALSE. Let's define the new attributes (historic records) for each boolean node as :

node.fail if it is a unary predicate,

node.fail1, node.fail2 if it is a binary predicate. The kind of records that will be kept in these attributes depends on the type of predicates.

#Example 4.2.7: Historic Records for $\mathbf{PS}_{\subseteq}$,$\mathbf{PS}_{\supseteq}$,$\mathbf{PS}_{\subset}$,$\mathbf{PS}_{\supset}$,$\mathbf{PS}_{=}$.

If node.last_value= FALSE then we keep the following historical records.

(1). If $\mathbf{PS}_{\subseteq}(\varsigma_1$~,$\varsigma_2$~):= FALSE,

node.fail1 := ($\varsigma_1^{\sim}$ - $\varsigma_2^{\sim}$), node.fail2:= ø.

(2). If $\mathbf{PS}_{\supset}(\varsigma_2^{\sim},\varsigma_2^{\sim})$:= FALSE,
node.fail1 := ø, node.fail2:= ($\varsigma_2^{\sim}$ - $\varsigma_1^{\sim}$).

(3). If $\mathbf{PS}_{\subset}(\varsigma_1^{\sim},\varsigma_2^{\sim})$:= FALSE,
node.fail1 := ($\varsigma_1^{\sim}$ - $\varsigma_2^{\sim}$), node.fail2:= ($\varsigma_2^{\sim}$ - $\varsigma_1^{\sim}$).

(4). If $\mathbf{PS}_{\supset}(\varsigma_1^{\sim},\varsigma_2^{\sim})$:= FALSE,
node.fail1 := ($\varsigma_1^{\sim}$ - $\varsigma_2^{\sim}$), node.fail2:= ($\varsigma_2^{\sim}$ - $\varsigma_1^{\sim}$).

(5). If $\mathbf{PS}_{=}(\varsigma_1^{\sim},\varsigma_2^{\sim})$:= FALSE,
node.fail1 := ($\varsigma_1^{\sim}$ - $\varsigma_2^{\sim}$), node.fail2:= ($\varsigma_2^{\sim}$ - $\varsigma_1^{\sim}$).

We may also store the set of tuple values that cause failure as a historic record for the incremental predicate $\mathbf{PD}_{<}$', $\mathbf{PD}_{\geq}$', $\mathbf{PD}_{<}$', $\mathbf{PD}_{>}$', $\mathbf{PD}_{=}$'. More detailed computations that involve historic records are shown in APPENDIX F. ###

3. PERFORMANCE EVALUATION

We would like to compare the cost of doing run-time constraint checking to that of doing compile-time constraint checking. Since many FOL approaches [Hsu 85], [Nico82] cannot specify detail computation steps, efficiency of run-time constraint checking of these approaches is usually based on the ratio of run-time checking domains over compile time checking domains. However, for ADL, performance evaluation of constraint enforcement can be measured more accurately. If the "comparison" operator is the most expensive operator of all the computation, we can measure the total number of comparisons involved in the computation and compare different alternatives. If , however, secondary storage access is the bottleneck, we can measure the number of disk accesses required and make comparisons. Such quantitative comparisons cannot be achieved easily by the FOL approaches.

To have a better understanding of what is involved in the compile-time constraint checking, as well as in the run-time constraint checking, we will work out some simple constraint computations both at compile-time and at run-time. Then, we expand the concept to more realistic situation where there are hundreds or thousands of data records to be checked in the database against a defined semantic constraint.

#Example 4.3.1: Compile-time Constraint Checking. Semantic Constraint :

ς_3 := (B) $_{B\geq' b3'}{}^{\circ}R_{AB}{}^{\circ}A$,
ς_4 := **ProjectS** $_{\{A\}}$ (ς_3),
ς_7 := (A) $_{A\geq' a2'}{}^{\circ}R_{AC}{}^{\circ}C$,
ς_8 := **ProjectS** $_{\{A\}}$ (ς_7),
β_9 := $\mathbf{PS}_{\supseteq}$(ς_4, ς_8).

In Example 3.6.1, we have seen how the above ADL specification is compiled into a run-time checking method: the transition digraph shown in Figure 3.6.6. Readers may want to review Example 3.6.1 for details of the compile-time constraint checking. ###

#Example 4.3.2: Incremental Computation, Single Update_element.

Now, if we have an update with a single update_element $\ominus$<a4>, we have to use the run-time constraint checking method represented by the incremental transition digraph in Figure 4.3.1.

U':={$\ominus$<a4>}. **Base**(U'):={A'}. Therefore, the base states in the incremental transition digraph now become: A':={$\ominus$<a4>}.

Since B', ς_1', ς_2', R_{AB}', R_{AC}', and C' are unmarked states, they are all equal to ø. B':=ø, ς_1':=ø, ς_2':=ø,
R_{AB}':=ø, R_{AC}':=ø, C':=ø.

ς_3' := **JoinS'** (ς_2', A') := {$\ominus$<a4, b4>},
ς_5' := (A') $_{A \geq 'a2'}$:= { $\ominus$<a4> },
ς_7' := **JoinS'** (ς_6', C') := {$\ominus$<a4, c1>,$\ominus$<a4, c4>},
ς_4' := **ProjectS'** $_{\{A\}}$ (ς_3') := {$\ominus$<a4>},
ς_8' := **ProjectS'** $_{\{A\}}$ (ς_7') := {$\ominus$<a4>},
β_9 := **PS$_{\supseteq}$'** (ς_4', ς_8') := TRUE. ###

#Example 4.3.3: Incremental Computation, Single Update_element.

If U':= {$\ominus$<a2>}, we can have base states as:
A':= {$\ominus$<a2>}, B':= ø, R_{AB}':=ø, R_{AC}':=ø, C':=ø.
ς_1':= ø, ς_2':=ø, ς_5':= {$\ominus$<a2>}, ς_6':=ø.
ς_3':=ø, ς_4':=ø, ς_7':=ø, ς_8':= ø.
β_9:= last_value := TRUE. ###

Note that the two target states ς_4' and ς_8' are all empty sets. Therefore, β will remain the same as the last evaluation result which is TRUE. The update eventually has no impact on the final result of the constraint because the update_element has been filtered out at the earlier stage of the computation.

#Example 4.3.4:Incremental Computation, Multiple Update_elements.

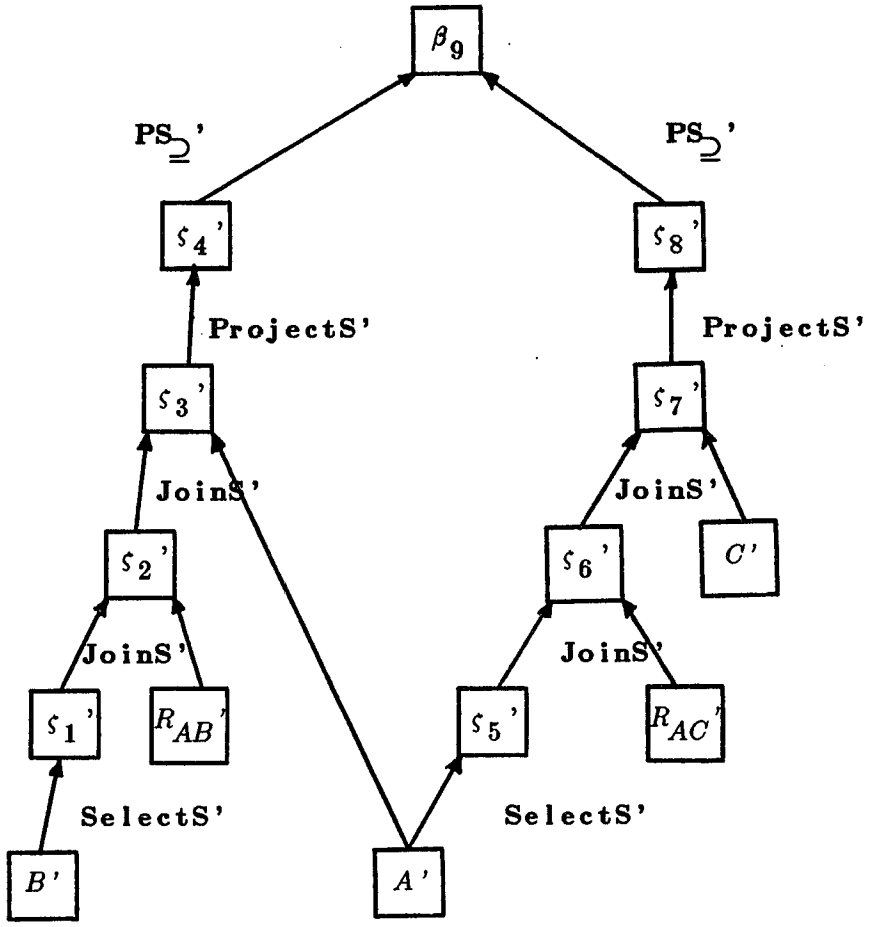
If U':= {$\oplus$<a2,b3>,$\ominus$<a4>}, we can have base states as: **Base**(U'):= {R_{AB}',A'}
A':= {$\ominus$<a4>}, R_{AB}':= {$\oplus$<a2,b3>}, R_{AC}':= ø, B':= ø, C:= ø.
ς_1':=ø, ς_3':= {$\oplus$<a2,b3>,$\ominus$<a4,b4>},
ς_4':= {$\oplus$<a2>,$\ominus$<a4>},
ς_5':= {$\ominus$<a4>}, ς_7':= {$\ominus$<a4,c1>,$\ominus$<a4,c4>},
ς_8':= {$\ominus$<a4>},
β_9:= **PS$_{\supseteq}$'** (ς_4', ς_8') := TRUE. ###

The reason that incremental computation can handle multiple updates easily

Figure 4.3.1. Incremental Transition Digraph For Example 4.3.2

Constraint :

$\varsigma_3 := (B)\ {}_{B\geq}{}'b3'\ {}^{\cdot}R_{AB}{}^{\cdot}A,$
$\varsigma_4 := \mathbf{ProjectS}_{\{A\}}(\varsigma_3),$
$\varsigma_7 := (A)\ {}_{A\geq}{}'a2'\ {}^{\cdot}R_{AC}{}^{\cdot}C,$
$\varsigma_8 := \mathbf{ProjectS}_{\{A\}}(\varsigma_7),$
$\beta_9 := PS_{\supseteq}(\varsigma_4, \varsigma_8).$



is because all incremental operators deal with sets instead of individual elements. Therefore, the incremental computation strategy compute the **net result** of the update transaction. It is a better approach since during the update transaction each individual update may violate many constraints temporarily. But after everything is considered, the net result of the update should still be consistent.

Now let's consider a more realistic case where the database contains hundreds of data records. Each operation requires external access to the secondary storage. Therefore, many operators in the constraint computation have to process one block of data at a time. We assume the access to the secondary storage is the bottleneck of the computation process. The number of block reads and block writes is used as the criteria of performance comparison. [Ullm82, Aho 82]

To provide a quantitative comparison of performances, we have to consider the physical implementation of the data structures, functions, and predicates of ADL. It is necessary to map the conceptual data model and its data manipulation language to the file structures and the algorithms at the physical level, respectively. For the example given below, we provide a very simple physical implementation scheme of the ERM and its ADL operators in APPENDIX H.

#Example 4.3.5: Block Access of Run-time Constraint Checking.
According to the physical implementation scheme in APPENDIX H, we may assume that the physical file structures of the database are those shown in Table 4.3.1.

We can go through the compile-time checking algorithm (Figure 3.6.6) and calculate the number of block accesses. Now, assume that a user submits an update of $\{\ominus<a101>,\ominus<a105>\}$, we can use the run-time checking algorithm (Figure 4.3.1) to find out the number of block accesses. The result is shown in Table 4.3.2. The evaluation costs for run-time constraint checking depend on whether the partial evaluation is dynamic or static. It may be misleading to believe that static partial evaluation is better because the read and write access to manage the historic records are not included in the computation cost in Table 4.3.2. ###

4. THEORY OF INCREMENTAL COMPUTATION

The theoretical background of the incremental computation methods is briefly discussed in this section. The fundamental concept is that most of the operations and predicates in the entity space as well as in the data space are based on the set theoretic computations that are discrete in nature. If we add data to the database, we are, in effect, adding distinct elements to sets. Such incremental changes should be traceable as long as the computation procedure is

Table 4.3.1. Physical File Structures of A Database

Entity Set	Data Record File				Dense Index File	
	No. of Rec.	Byte/Rec	Rec./Blk	Blk/file	Index/Blk,	Blk/file
A	200	24	10	20	32	7
B	400	28	9	45	32	13
C	300	32	8	38	32	10

Relationship Set	Data Record File			
	No. of Rec.	Byte/Rec	Rec./Blk	Blk/File
R_{AB}	5000	8	32	157
R_{AC}	4000	8	32	125

Table 4.3.2. Run-time and Compile-time Block Access Comparisons.

	Read Access (No. of Blks)	Write Access (No. of Blks)
Complie-time Constraint Checking	**9162**	**8797**
Run-time Constraint Checking with Dynamic Partial Evaluation	**2098**	**1007**
Run-time Constraint Checking with Static Partial Evaluation	**534**	**70**

deterministic in nature. The question is how to isolate those objects that are related to the update in a fast and efficient manner and, then, make a decision about whether or not the update is acceptable with limited information on hands.

There are basically two types of operations in the incremental computations that are considered non-trivial. They are:

(1). f(set, or sets) = set,

(2). f(set, or sets) = boolean,

Let's examine them one by one.

4.A. Operations For Set To Set Transformation: f(set, or sets) = set.

If we can determine the effect of an update on a set computation, the computation itself must be partitionable.

#Definition 4.4.1: **Partition**(S_1, rule)
Partition(S_1,Rule) is function that performs partition according to Rule on set S_1. The value returned by the functions is $\{S_{11}, S_{12},..., S_{1n}\}$ with: (1) $S_{1i} \cap S_{1j} = \emptyset$ for all 1i<>1j. (2) $S_{11} \cup S_{12} \cup ... \cup S_{1n} = S_1$. ###

#Definition 4.4.2: **Partitionable Unary Set Operator**
f(S_1):= S_2, S_1 and S_2 are sets. f is an unary set operator. f is called a partitionable unary set operator if it has the following properties.
(1) **Partition** (S_1, Rule) = $\{S_{11},...,S_{1n}\}$ means that there is a partition rule partitioning S_1 into disjoint subsets $S_{11},...,S_{1i},...,S_{1n}$.
(2) For each disjoint set,$S_{21}=f(S_{11})$,..., $S_{2i}=f(S_{1i})$,..., $S_{2n}=f(S_{1n})$. $S_{2i} \cap S_{2j} = \emptyset$ for all 2i<>2j and $S_{21} \cup ... \cup S_{2i} \cup ... \cup S_{2n} = S_2$. That is, S_{2i}, S_{2j} are disjoint subsets of S_2. ###

The second property points out an important fact. That is, if f is a partitionable unary set operator, we can compute S_2 either form the full set S_1 or from the disjoint subsets of S_1.

#Example 4.4.1: Partitionable Unary Set Operator
If **Double** and **Total** are two set operators on sets of integers. **Double** forms another set by doubling each element in its operand set. **Total** adds all the elements together and form a single element set. So,

Double ({1, 2, 3})= {2, 4, 6}. **Total**({1, 2, 3})={ 6 }.

According to the above definition **Double** is a partitionable unary set operator, while **Total** is not. ###

Now, if we modify the set S_1, we represent the new and old sets as S_1 and $S_1^{\sim}$, S_2 and $S_2^{\sim}$. So we have

$S_2^{\sim} := \mathbf{f}(S_1^{\sim})$, $S2 := \mathbf{f}(S_1)$,

If f is partitionable, we can partition S_1 and $S_1^{\sim}$ according to:

$G^{\sim} := \mathbf{Partition}(S_1^{\sim}, Rule) := \{S_{11}^{\sim}, ..., S_{1n}^{\sim}\}$

$G := \mathbf{Partition}(S_1, Rule) := \{S_{11}, ..., S_{1m}\}$,

The invariant part of the partition is $G \cap G^{\sim}$. And the variant part of the partitions is $(G-G^{\sim}) \cup (G^{\sim}-G)$.

#Theorem 4.4.1: Assume, S_1, $S_1^{\sim}$, S_2, $S_2^{\sim}$, G, $G^{\sim}$ are shown as above.(1) $\forall S \in G \cap G^{\sim} \Longrightarrow f(S) \subseteq (S_2^{\sim} \cap S_2)$.(2) $\forall S \in ((G-G^{\sim}) \cup (G^{\sim}-G)) \Longrightarrow f(S) \subseteq ((S_2-S_2^{\sim}) \cup (S_2^{\sim}-S_2))$.

Proof: (1) $S \in G \Longrightarrow \exists S_{1i}, S_{1i}=S \Longrightarrow f(S_{1i})=f(S)=S_{2i} \subseteq S_2$. Also $S \in G^{\sim} \Longrightarrow \exists S_{1j}^{\sim}, S_{1j}^{\sim}=S \Longrightarrow f(S_{1j}^{\sim})=f(S)=S_{2j}^{\sim} \subseteq S_2^{\sim}$. Therefore, $f(S) \subseteq (S_2^{\sim} \cap S_2)$ (2) By using Definition 4.4.2, we can prove when $S \in (G-G^{\sim})$, $f(S) \subseteq (S_2-S_2^{\sim})$. When $S \in (G^{\sim}-G)$, $f(S) \subseteq (S_2^{\sim}-S_2)$. It is thus proved.

Semantically, the above theorem points out that if we are interested in finding out the difference between S_2 and $S_2^{\sim}$, we may look at the difference between G and $G^{\sim}$.

The basis of partial evaluation is the concept of partitionable set operators because when there is an update, we only have to look at the variant part, $(G-G^{\sim}) \cup (G^{\sim}-G)$, and find out the incremental changes in this group of disjoint subsets. We can be assured that subsets in the invariant part of the partition, $G \cap G^{\sim}$, will never change.

The concept of partial realization is also based on the idea of partitionable set operators. Since each partition is a disjoint set from the others, the update is allowed to be realized on the variant part of the partition without worrying about the effects on the invariant part. Comparing the differences of the variant part before and after the update, we can infer the overall changes of the update on the entire set.

#Definition 4.4.3: **Complete Partition for Unary Set Operator**

If f is partitionable and a Rule of **Partition**(S, Rule) results in the maximum number of disjoint subsets of S, the Rule is called a **complete partition rule** for f and the partition, a **complete partition**. No other rules can satisfy (1) and (2) of Definition 4.4.2 and yet generate more disjoint subsets of S_1. ###

The reason that we are interested in the complete partition is because we try to divide the operation to as little unit operations as possible. So if part of the set is modified, we can examine the smallest extent of the partition that is affected.

#Example 4.4.2: Complete Partition Rules

Rule 1: The set of each element in S1 forms a class. Rule 1 is a complete partition

rule of the operator **Double** discussed in Example 4.4.2. Rule 1 is also a complete partition rule for **SelectS**.

The complete partition rule is mostly dependent on the kind of operation. For example, for ς_2:=**Group_byS** $_{G\_set}$ (ς_1) and ς_2:=**ProjectS** $_{P\_set}$ (ς_1), the partition of ς_1 is by the following rule.

Rule 2: If G_set or P_set :={A,..,H}, $\varsigma_1 \subseteq \Psi_1$, and G_set or P_set $\subseteq$ Sch(Ψ_1), group all the entity vectors in ς_1 with the same values $<a_i,...,h_k>$ into one disjoint subset. Therefore, for **Group_byS** and **ProjectS** the partition is based on the G_set or the P_set, respectively.

The **Extract** operator and the d_relation operators, **ExtendD**, **ProjectD**, and **SelectD** are also partitionable functions with partition rules being Rule 1. Note that the **ProjectD** uses Rule 1 is because it allows duplicates. ###

#Definition 4.4.4: **Partitionable Binary Set Operator**

f($<S_1,S_2>$) := S_3, S_1,S_2,S_3 are sets. f is a binary set operator. f is called a **partitionable binary set operator** if it has the following properties.

(1). **Partition** ($<S_1, S_2>$, Rule)= { $<S_{11}, S_{21}>$, ..., $<S_{1n}, S_{2n}>$ } means that there is a partition strategy, partitioning S_1, S_2, into disjoint subsets S_{11}, S_{12},..., S_{1n}, S_{21}, S_{22},..., S_{2n}, and pairing them together into set tuples.

(2). For each $<S_{1i},S_{2i}>$, S_{31}= f($<S_{11},S_{21}>$), S_{32}= f($<S_{12}, S_{22}>$) ,..., S_{3n}= f($<S_{1n},S_{2n}>$). $\varsigma_{3i} \cap \varsigma_{3j} = \emptyset$ for all 3i<>3j and $S_{31} \cup S_{32} \cup ... \cup S_{3n} = S_3$. This is, S_{3i}, S_{3j} are disjoint subsets of S_3. ###

#Definition 4.4.5: **Complete Partition for Binary Set Operator**

If f is a partitionable binary set operator and a Rule of **Partition**($<S_1,S_2>$, Rule) results in the maximum number of disjoint pairs of $<S_1,S_2>$, the Rule is called a **complete partition rule** for f and the partition, a **complete partition**. No other rules can satisfy (1) and (2) of Definition 4.4.3 and yet generate more disjoint pairs of $<S_1, S_2>$. ###

#Example 4.4.3: Complete Partition

Rule 3: If S_1 and S_2 form a set pair $<S_1, S_2>$, we partition $<S_1, S_2>$ as follows:

(1).The set of each element in S1 forms a disjoint subset.

(2).The set of each element in S2 forms a disjoint subset.

(3). For $\forall$ x $\in$ ($S_1 \cap S_2$), <{x},{x}> forms a disjoint pair. For $\forall$ y $\in$ (S_1-S_2), <{y},$\emptyset$ > forms a disjoint pair. For $\forall$ z $\in$ (S_2-S_1), <$\emptyset$, {z}> forms a disjoint pair.

For all the set operators in the entity space, $\cap$, $\cup$, -, the complete partition rule is simply **Rule 3**.

For ς_3:=**JoinS** J_Set (ς_1, ς_2) the partition rule is:

Rule 4: First, partition ς_1 and ς_2 according to the values corresponding to the

J_Set. Match the classes which have the same values of elements in the J_Set in pairs in the two partitions. For those that does not have a matching counter part, match it with ø. ###

#Example 4.4.4: Complete partition rule for **JoinS**
ς_3:= **JoinS** $_{\{A\}}$ (ς_1, ς_2),
ς_1:= {<a1,b2>,<a1,b3>,<a2,b1>},
ς_2:= {<a1,c2>,<a2,b3>,<a3,c2>},
Partition(<ς_1, ς_2>, **Rule 4**):= { <{<a1,b2>,<a1,b3>},{<a1,c2>}>, <{<a2,b1>},{<a2,c3>}>, < ø, {<a3,c2>}> }
Rule 4 is the one given above. ###

4.B. Operations for Set to Boolean Transformation: f(set, or sets) = boolean.

The word 'partitionable' loses its appeal when we discuss operations of set-to-boolean transformations, i.e., predicates. We can still partition the sets as we do in the set transformation operation, but we encounter a problem with predicates. The problem may not be that serious if the predicate is always evaluated as TRUE (the final sink node of the transition digraph). But, as reality tells us, we cannot always expect so.

An unary predicate may have the following property: P(S_1):= β_1, P is a unary predicate on sets. S_1 is a set.
(1). **Partition** (S_1, Rule)= {S_{11},...,S_{1n}} means that there is a partition strategy partitioning S_1 into disjoint subsets S_{11},S_{12},...,S_{1n}.
(2). For each class S_{1i}, β_{11}=P(S_{11}),β_{12}=P(S_{12}),..., β_{1n}=P(S_{1n}). and $\beta_{11} \Lambda \beta_{12} \Lambda ... \Lambda \beta_{1n} = \beta_1$.

If β_1 is TRUE, we know immediately all β_{11},...,β_{1n} are TRUE. But if β_1 is FALSE, we are not sure about β_{11},...,β_{1n} any more. The best we can say is that at least one of them is FALSE.

Again, if we make an update, the new and the old partitions are G and G~. The invariant part of the partitions are G$\cap$G~; the variant part, (G-G~)$\cup$(G~-G). If the original value of β_1~ is TRUE, by applying P to (G-G~) $\cup$ (G~-G) we can tell if β_1 is still true or not. There is no need to access the invariant part, i.e., G$\cap$G~. But if β_1~ is FALSE, just by evaluating the variant part of the partition is not enough to tell the new value of β_1. We also need to know something about the invariant part of the partitions. The historic record (the "fail" attributes) for each boolean node in the incremental transition digraph is thus created for this purpose.

5. SUMMARY

The incremental computation strategy is build on the foundation of signed data structures which are formalisms of user updates or incremental changes. The incremental computation, thus, becomes an algebraic operation on both the signed and the unsigned data structures. The primary objective of the incremental computation is to minimize the computation cost of the run-time constraint checking.

Every semantic constraint defined by ADL can be recompiled into an incremental transition digraph that represents a run-time constraint checking mechanism. Inputs to the incremental computation procedure are user updates (signed data structure), historic records (unsigned data structure), and a subset of the previous database state (unsigned data structure obtained through partial evaluation). The computations simply take all these data into consideration and infer if a semantic constraint is still valid after the update is realized in the database. The incremental operators are defined in detail in APPENDIX F.

Each incremental operator can be thought of as a composite function of previously defined functions and predicates. It calls for the following computation techniques:(1).partial evaluation,(2).partial realization, and (3).historic record keeping. Partial evaluation is to identify and evaluate the part of the database that is directly related to the update. Since evaluation is costly, the main concerns here is to limit the evaluation to those that are absolutely necessary for the computation. Partial realization is to find a disjunct subset of data from the database and let the update be realized in this disjunct subset. We then observe the possible effects of the update within this subset. Historic records of the last evaluation results are created to avoid expensive re-evaluations.

As a run-time constraint checking strategy, the incremental computation strategy has the following advantages:

(1). It represents a rigorous computation strategy for run-time constraint computation. It is based on a high level data model, the Entity Relationship Model (ERM) and a high level constraint specification language, the Applicative Data Language (ADL). Any semantic constraint defined by ADL can be re-compiled into a run-time constraint checking method.

(2). It defines formal data structures to represent user's updates and their effects on the intermediate results of constraint computation. Run-time constraint checking, thus, becomes algebraic operations on such newly defined data structures.

(3). It is an efficient method for run-time constraint checking, since all the computations center around the user update which is usually a very small part of the overall database. Only data that is directly related to the update is evaluated (partial evaluation and partial realization).

(4). It identifies historic records that are deemed necessary for the incremental computations to avoid expensive re-evaluations of the previous database state. The historic records can be maintained after the update is realized.
(5). It allows multiple objects to be updated and the incremental computation is only carried out once. The execution order of the update transaction is not important in the computation.

CHAPTER V. IMPLEMENTATION OF AN INTEGRITY SUBSYSTEM

The purpose of implementing an integrity subsystem is to test: (1). the ability of ADL as a constraint modeling language, (2). ADL's compile-time checking method, and (3). ADL's run-time checking method. We build the integrity subsystem as a part of an Entity- Relationship based database system implemented in LISP (called LBASE). The ERM proposed in CHAPTER II is used as the conceptual view of the database. Entity sets and relationship sets are mapped to different physical data structures in the database. The fact that ADL is based on a high level data model, ERM, has the advantage of allowing the user to formulate his queries and constraints totally based on his understanding of the conceptual schema, not on any of the physical data structures. Each ADL formula is then compiled into data access procedures that compute the answer from the physical implementation of the database.

With the above defined objectives, we simplify LBASE by choosing its physical data structures closely resembling ADL's logical data structures. As a result, interpretation from the conceptual level to the physical level becomes straight forward. At the physical level of LBASE, most information about objects are stored as property lists of LISP atoms. Relations among objects are stored as multiple-layered lists of surrogate keys of the atoms. Mapping from the high level ADL data language to the physical data manipulation is realized through LISP procedures.

Any database computation usually consists of two parallel parts, i.e., the symbolic (or intensional) computation and the instance (or extensional) computation [Tsic82]. The symbolic computation gives the interpretation of the data computed. For example, entity names, relationship names, and attribute names are the symbolic part of the data language. In order to give proper interpretation o3hthe computed results, we need to do symbolic manipulation at the same time we do instance computation. A good example of this is the "joining" of two relation states. At the instance level we know the operation of how two states can be joined together. And at the symbolic level the DBMS has to compute the union of the symbol sets of the two relation states as the new symbol set for the joined relation state.

We choose LISP [Wile84, Fode83] to implement LBASE for the following reason:

(1). LISP is an applicative language. [MacL83, Wins84] The "applicative" nature of ADL matches closely to the language principle of LISP. ADL operators are build from primitive functions and predicates.

(2). LISP is an excellent prototyping language for new language development and new algorithm testing. The "interactive" nature of LISP and its debugging facility give the user great convenience in program development and prototyping

[Hatt85, Darl82].

(3). The "recursive" nature of LISP simplifies the development of algorithms that involve graph structures (in our case, the transition digraph and the incremental transition digraph). To do computation on such graph data structures, non-recursive programming languages usually generate much more codes than recursive programming languages.

(4). The symbolic manipulation power of LISP makes the symbolic operation in database computation an easier task. As pointed out earlier, all operations in the database computation involve both the symbolic computation and the instance computation.

The development environment for the current version of LBASE is as follows:

Language: Franz LISP,
Computer: VAX 11/780,
Operating System: Unix,
Location: Computer Science Department,
Louisiana State University.

In section 1, we introduce the user interface facilities of LBASE. Section 2 gives an overview of the architecture of the integrity subsystem of LBASE. The basic physical data structures of LBASE is discussed in section 3. Section 4 discusses the implementation of ADL operators. How to do compile-time checking as well as run-time checking is explained in section 5.

1. USER INTERFACE

It is not the purpose of the this implementation to develop a full fledged database system to rival those commercially available databases. LBASE as a database system, however, does provide the following interfaces: (1). ER Editor, (2). Data Editor, (3). Constraint Editor, and (4). Query Processor. It is a menu driven system except where the user has to input long strings of characters or numbers. These interface facilities are considered essential in many large database systems.

The ER editor is equivalent to the data definition languages in many database systems [Date81, Ullm82]. It is an implementation of the ERM approach in constructing the conceptual view of the database [Chen76, Chen83, Davi83]. From a system design point of view, the ERM is well suited for the top-down design methodology because the user only has to identify the entity set and the relationship set in the enterprise in the first phase of the logical design. Details like attributes can be assigned to the relationships and entities in the second phase of the logical design.

The ER editor supports this top-down design methodology. The user can create an entity set or a relationship set with no attributes at the earlier phase

of the design and add attributes to the entity sets and relationship sets later. The ER-editor menu gives the user options such as printing logical views of the database, printing information about the entity sets and the relationship sets, adding, deleting, and renaming entity sets and relationship sets.

When the user invokes the ER-editor, he is accessing the intensional part (or the symbolic part) of the database. The semantic or conceptual view of the database is well represented as proper data structures in LBASE by using the ER editor. Data structures for storing the entities and the relationships are also created at the same time.

While entity sets and relationship sets are defined by the ER editor, no data values are yet stored in the extensional part of the data structures. The purpose of the data editor is to access the extensional part (instances) of the database. It allows the user to insert, delete, or modify the data stored in the current DBST. LBASE makes the data entry process as flexible as possible. User can either enter data row by row or column by column. The data entry process iterates itself until the user gives a termination signal.

Data entered earlier can also be modified using the data editor. Internally, each modification is represented as a deletion followed by an addition and will also be treated as so in run-time constraint checking.

The constraint editor allows the user to define new semantic constraint, to modify existing constraint, and to delete existing constraint. The user formulates semantic constraints using ADL and an ERD of the enterprise. Once the user finishes his input, the editor will check the existing data in the database to see if the data complies with the newly defined constraint. Each semantic constraint is identified by a constraint sequence number. The user has to refer to such a number for editing a constraint.

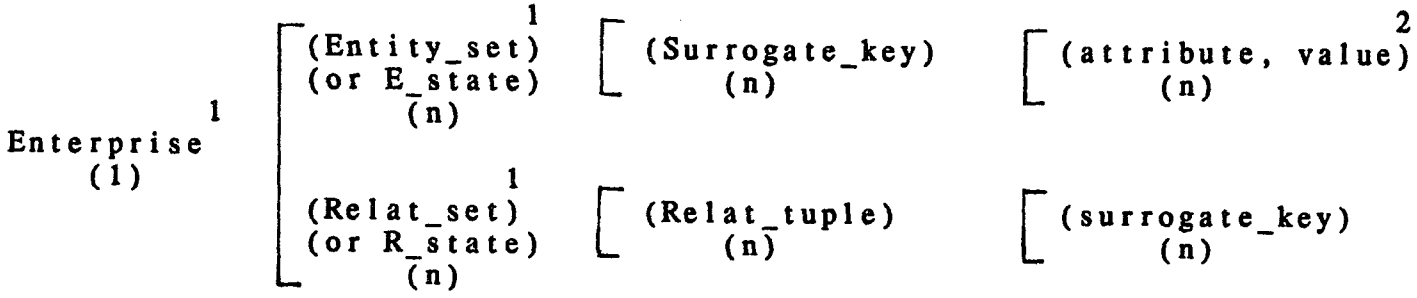
The query processor is to retrieve data from the database. Again, the user can formulate his queries using ADL and the ERD of the enterprise. There is no need for him to be aware of the physical implementation of the database. User queries are compiled into a transition digraph. The query transition digraph is used only once for evaluation and the result is shown on the user's terminal with a predefined format.

2. DATA STRUCTURES OF LBASE

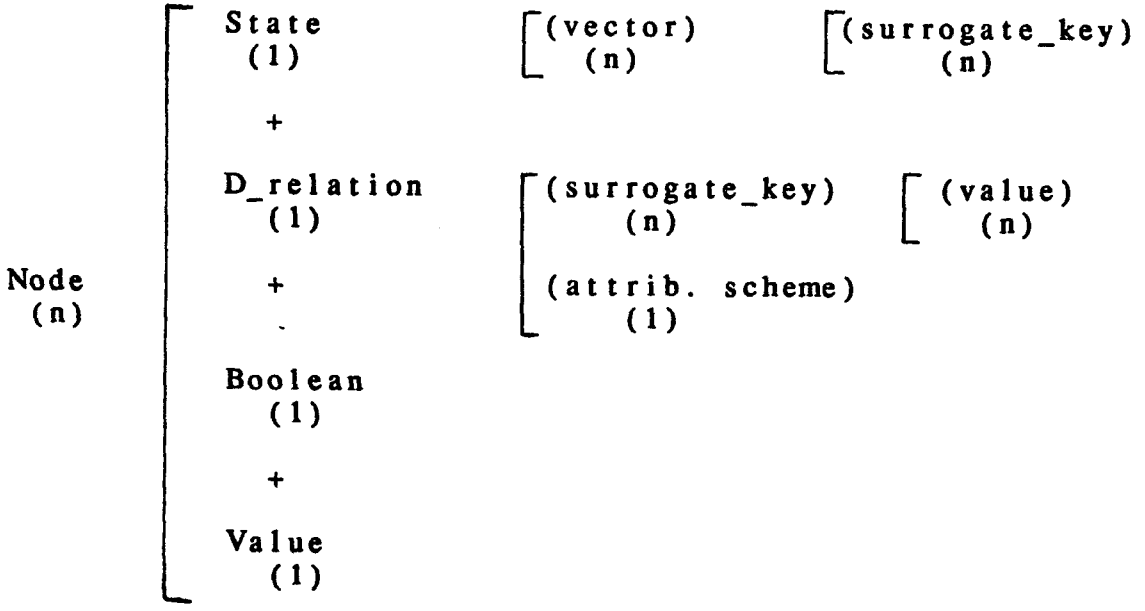
LBASE builds its high level data structures from two fundamental LISP data structures, i.e., lists (usually multiple layers of lists) and property lists. The data structures of LBASE can be generally divided into two parts, i.e., the static part, and the dynamic part (Figure 5.2.1). The static data structures represent the current state of the database. All the data in the database, both intensional and extensional information, are stored in the static data structure. But during query

Figure 5.2.1. Warnier/Orr Diagram of LBASE Data Structures

Static Data Structure: Representing the Current Database State.



Dynamic Data Structure: Representing Computation Results.



1. Characteristic data like surrogate key prefix are stored as property lists.
2. The attribute value pairs are stored as property lists of surrogate keys.

processing or constraint checking, new data structures are created and destroyed constantly. Such temporary data structures are the dynamic data structures of LBASE. They are derived from the static data structures as intermediate results of the computation.

In the static data structure, data about a particular enterprise is stored in one "category". For example, in a university enterprise the database may contain data about students, teachers, and classes. A company enterprise may deal with personnel records, budgetary information, and project data. Each enterprise consists of two separate lists. One is the entity set list which contains all the entity set symbols; the other the relationship set list that contains all the relationship set symbols.

Each entity set symbol is itself a list of surrogate keys. Descriptive data about the entity set are stored either as property lists or compound lists. Attribute values of each entity are stored as property lists belonging to the entity surrogate key. The list of all such surrogate keys thus becomes the value of the entity set.

Each relationship set comprises a symbolic list and a surrogate key list (two sublists). Descriptive data about the relationship set are stored in the symbolic list. The arity of the surrogate list is the same as the arity of the relationship set. It stores the instances of relationships.

The static data structures of LBASE, i.e., environment, entity sets (corresponding to the E_states), relationship sets (corresponding to the R_state), and attributes, represent the current database state. When a given query or constraint is evaluated, the database system has to compute the answer based on the static data structures. In the computation processes temporary data structures will be created and destroyed dynamically.

The intermediate computation result can be treated as a computation node. The characteristic data of each node such as the operation that create this node and the operands that derive this node are stored in several property lists of the node. The value of the node can be any of the four possible data types of ADL, i.e. states, d_relations, booleans, and values. The later two data types are the same as the build-in LISP data types. Each primary relation state is represented as a compound list of a symbolic list and a surrogate key list. When compound objects are involved in a hyper relation state, multiple layered lists are constructed. A d_relation is represented as a list of tuples.

Signed data structures mentioned in the incremental computation are basically the same as their unsigned counter parts. The only difference is that any signed data structure will bear a "Sign" attribute of +1 or -1. The incremental states and incremental d_relations are simply collections of such signed data structures.

3. FUNCTIONS AND PREDICATES OF LBASE

Separating the computation space into the relation space and the data space computation in ADL gives a clear advantage to the implementation of ADL because object-oriented programming techniques can be readily adapted. In the relation space computation we treat each entity as an object. In the physical implementation, however, the objects are usually represented as system generated surrogate keys (or pointer). All the relation space operators (except the **SelectS**) only operate on these surrogate keys instead of any values pointed by the keys. It is usually the case that after relation space computation is completed the number of objects is restricted to limited few.

Then, we extract attribute values from the objects to do further computation in the data space. The number of data involved at this data element level is usually drastically reduced because of earlier reduction in the number of objects in the relation space computation.

The strong binding between relation space computation and object-oriented programming has many important implications. Without such object-oriented approach, we will then be forced to do data element level computation from start to end, which usually generate large amounts of data as intermediate results. In a distributed database environment, transferring such large amount of data is definitely undesirable. The object-oriented approach is one way to minimize the amount of data transfer and the ADL strongly supports such an approach in its implementation. Similar efficiency consideration is also observed in many real-time control systems where the response time is critical. The object-oriented computations augmented by proper physical file structures may solve the efficiency problem.

Some of the expensive operators like **JoinS** and **Group_byS** require, first, sorting the nested list data structure; then, from the sorted data structure we can create a new data structure.

Implementation of data space operators are done by transforming the property list of d_relations from one form to another. They are usually easier to implement than those relation space operators.

Incremental operators and incremental predicates are usually more difficult to implement since they usually involve more comparisons and manipulations than the regular operators and predicates in ADL. But the incremental operators and incremental predicates usually handle data that are either update or data related to the update.

4. ARCHITECTURE OF THE INTEGRITY SUBSYSTEM

There are many architecture proposed in literature for an integrity subsystem

in a database [Hamm76, Bert84, Simo84, Lili82, Wils80, Eswa75]. To meet the objective of our implementation, we propose an integrity subsystem architecture shown in Figure 5.4.1.

The integrity subsystem provides mechanisms to prevent inconsistencies caused by inaccuracy and misunderstanding in users' update transactions. Every time when the user submits a new semantic constraint specification, the integrity subsystem immediately compile the ADL formulas into two enforcement methods. The transition digraph for compile-time constraint checking is evaluated right after the compilation of the constraint. The incremental transition digraph for run-time constraint checking strategy is evaluated upon an user's update.

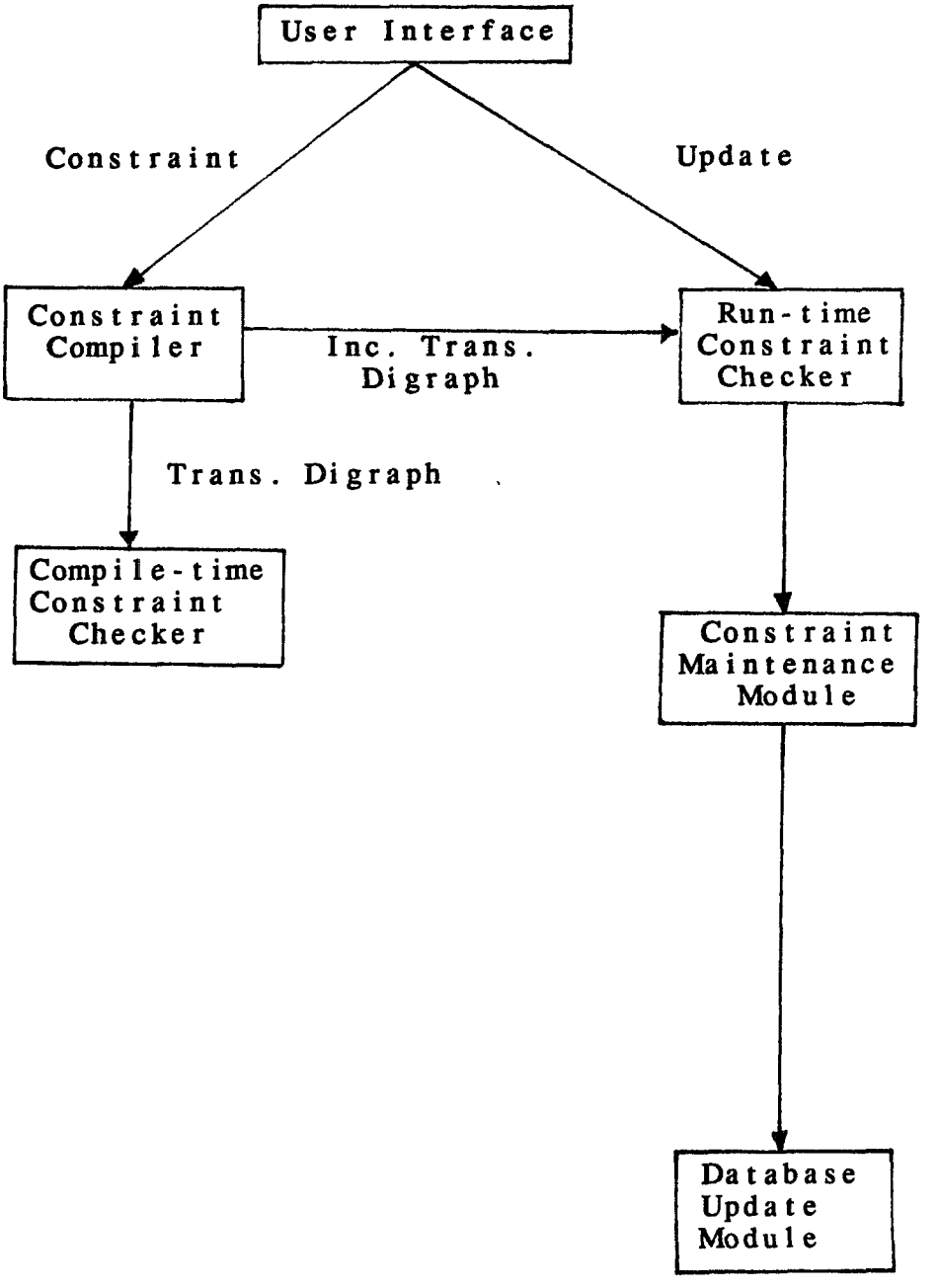
The user may activate the integrity checking subsystem at two different occasions. One is when the user uses ADL to define new semantic constraints. The other is when the user requests update transactions. In the first instance, the ADL specification of a semantic constraint taken from the user's terminal is compiled into a transition digraph and an incremental transition digraph. The former is sent to the compile-time constraint checker for immediate evaluation using the current data. Part of the intermediate computation results need to be incorporated into the incremental transition digraph as historic records.

When an user submits an update, the run-time constraint checker is invoked. Assuming that each update session completed by an user represents an integral part of data editing. The modified data are stored temporarily in a buffer area as signed data structures. The run-time constraint checker will then compare the signed data structures with each incremental transition digraph to see if the update is relevant to the particular constraint. If it is, the incremental transition digraph is evaluated to find out its new truth value. When all the incremental transition digraphs have been checked and there is no violation, the update data is sent to the constraint maintenance module where historic records are updated and maintained. Finally, the database update module will make the final commitment to the update by permanently modifying existing data structures. If any of the constraint is violated, the run-time constraint checker will notify the user that the update violates certain semantic constraints and the update needs to be re-edited.

5. COMPILE-TIME AND RUN-TIME CONSTRAINT CHECKING

The basic mechanism for the compile-time constraint checking is the transition digraph which is represented as a linked list of nodes [Aho 82, Horo84]. The evaluation of the transition digraph actually follows a topological sort sequences [Aho 82, Horo84]. Those nodes that are source nodes are evaluated first before any of the derived nodes is evaluated. The computation proceeds from the bottom of

Figure 5.4.1. Architecture of An Integrity Checking Subsystem



the transition digraph to the top of the digraph. The final node to be evaluated is always a boolean node (the sink node).

Before the run-time constraint checking, user update are sorted into incremental E_states and R_states. The incremental states are then assigned to the source nodes of relevant incremental transition digraph. Those nodes that are not marked (not affected by the current update) become empty sets. The incremental computations involve both the signed data structures and the unsigned data structures. The latter is usually obtained through partial evaluation of the old database state (before the update is committed).

There are basically two ways to do partial evaluation. One way is to dynamically modify the transition digraph of the compile-time checking method. The purpose of partial evaluation is to find a small subset of the database that is related to the current update. The dynamic partial evaluation method computes such a subset as required during the incremental computation. Alternatively, we can maintain a historic record at each node of the incremental transition digraph to keep track of the last full evaluation result (an unsigned data structure). Then, when partial evaluation is required, it only needs to filter out the relevant part of the historic records. We call this partial evaluation method the static partial evaluation. It will certainly speed up run-time constraint checking by using the static partial evaluation. But it takes storage space to store the historic record at each node. After the update is committed, extra overhead is required to maintain the historic records. This implementation of LBASE uses the static partial evaluation strategy.

6. SUMMARY

The implementation of an Entity-Relationship database system is to test the constraint modeling capability of ADL and its compile-time and run-time constraint enforcement methods. The database system is written is LISP because: (1). LISP is an applicative language and so is ADL; (2). LISP is an excellent prototyping language; (3). LISP allows recursive programming with which many graph algorithms can be developed easily; (4). The symbolic manipulation power of LISP facilitates many database computations involving the data definition part of the database. The database system developed is called LBASE.

LBASE provides four user interface facilities that are considered essential to many database systems, i.e., (1). ER editor, (2). data editor, (3). constraint editor, and (4). query processor. The ER editor is equivalent to the data definition languages of many database systems. The data editor allows the user to enter, delete, or modify data. Semantic constraints are entered through the constraint editor. Interactive queries are processed by the query processor.

Data structures of LBASE are divided into two groups. The static data

structures such as enterprise, relationships, entities, and attributes represent the current database state. The dynamic data structures such as computation nodes (states, d_relations, booleans, and values) are created and destroyed during the computation of queries and constraint checkings. ADL supports the object-oriented programming techniques during its implementation because most relation space operators deal only with objects instead of data values.

The architecture of LBASE's integrity subsystem consists of a constraint compiler, a compile-time constraint checker, a run-time constraint checker, and a constraint maintenance module. The integrity subsystem is activated when a constraint is newly defined, it is later modified, or an user submits an update transaction. The subsystem provides a mechanism to prevent inconsistencies caused by inaccuracy and misunderstanding in users' update transactions.

CHAPTER VI. CONCLUSIONS AND RECOMMENDATIONS

1. CONCLUSIONS OF THIS RESEARCH

We conclude from this research the following:

1.A. For database systems where consistency of data is important and patterns of data are constantly analyzed, the following approach is suggested to model the information system: (1) Develop the conceptual view of the information system using the Entity Relationship Model, (2) Use the Applicative Data Language for query formulations and semantic constraint specifications, (3) Compile the semantic constraint specification into an optimized transition digraph and an optimized incremental transition digraph, (4) Carry out the compile-time constraint checking using the transition digraph and the run-time constraint checking using the incremental transition digraph.

1.B. In this research a data manipulation language , the Applicative Data Language (ADL) , which is based on the conceptual view of the database is developed. The user can formulate his queries and constraints based on his understanding of the conceptual view instead of the physical view of the database. The Applicative Data Language has the following features: (1) It is built using three basic constructs, i.e. data structures, functions, and predicates. (2) It takes advantage of the semantic clarification of objects and relationships in the ERM to do, first, an object level computation and , then, a data element level computation. (3) The semantics of generalization, specialization, aggregation, and classification can all be formalized and modeled with compound objects. ADL models these semantics as computation procedures in the hyper relation space by operators like: **Group_byS**, **JoinS**, **SelectS**, etc. (4) The applicative nature of ADL and its computational syntax make it a good data access language for application programs interfacing the database. It also supports a structured step-by-step approach in query formulation and constraint specification.

1.C. Since ADL models semantic constraints as computation procedures, there are three types of semantic constraints classified by the data objects involved in their formulations. They are local, structural, and complex constraints. Each constraint specification can be viewed as a mapping from the database state to a TRUE or FALSE value through the ADL computation procedure. The process of constraint specification is the process of translating semantics to syntactic specifications.

1.D. An ADL constraint specification can be compiled into enforcement methods.

The first step of the compilation is to optimize the specification by substituting less efficient expressions with equivalent but more efficient expressions. Result of the reformulation is governed by the physical implementation scheme, the current database, and the software and hardware configuration. The second step of the compilation is to translate optimized well formed specifications into transition digraphs and incremental transition digraphs. The transition digraph is the enforcement method for the compile-time constraint checking where the entire database is subjected to the computation algorithm. The incremental transition digraph is for run-time constraint checking.

1.E. An efficient computation strategy is developed for run-time constraint checking, i.e. the incremental computation strategy. The strategy is based on signed data structures formalizing the user's update and incremental operators computing the effects of the update. It is efficient because the entire computation strategy centers around the update which is usually a very small part of the database. It propagates the incremental changes from the bottom of the incremental transition digraph to the top of the digraph to find the new truth value of the semantic constraint after such an update takes place. The incremental computation strategy calls for the following computation techniques: (1) partial evaluation, (2) partial realization, and (3) historic record keeping.

1.F. ADL and its enforcement strategies for semantic constraint modeling have been tested in an Entity Relationship database (LBASE). The separation of object level computation from the data element level computation in ADL strongly supports object-oriented programming in the implementation of such an integrity subsystem. In the physical implementation, entity space computation is carried out by manipulating the pointers and surrogate keys of data records. After relevant data records are identified, data values of the records are accessed for the data space computation. The efficiency of this object-oriented programming technique has strong implications to many distributed database environments and run-time control and monitoring environments.

1.G. From the result of this research, ADL and its enforcement strategies are recommended as the underlying architecture of an integrity subsystem in a database system.

2. RECOMMENDATIONS FOR FUTURE RESEARCH

We recommend the following directions for future research:

2.A. ADL's semantic constraint modeling capability should be extended further to

cover dynamic constraints and relational closure constraints. The difficulties involved in extending ADL to cover more semantics lies not so much with its data structures or operators, but with its enforcement strategies, especially, the added complexity in run-time constraint checking.

2.B. For real-time control and monitoring information systems, ADL need to consider further: (1) scheduling problems of constraint checking, (2) response actions for constraint violations. Currently, all the constraints are of equal importance in ADL and all responses display either TRUE or FALSE of the checking result. This is certainly not the case in a real-time control and monitoring system. Certain constraints or patterns should be given higher priorities for checking than others. If the constraints are violated or patterns are matched in a real-time environment, the integrity subsystem should invoke various action plans to correct the situations or to give warnings to the user. So instead of **Display**(a) at the end of a constraint specification, we can say **Action7**(a). The course of action will, then, depend on the result of the pattern in the current database state.

2.C. We should study the cases where the databases can tolerate inconsistent data in a "statistically controled" manner. This relates to the randomness, entropy, and noise levels of all things in nature. While we can specify things by giving exact numbers, but reality always tells us that things are not exactly the same or exact as we would like them to be. A good example of this is a manufacturing database where sample data are constantly monitored and entered into the database and are compared with product specifications. Instead of taking immediate actions when one sample is "out of spec", the integrity subsystem may wait for further evidence to indicate that the production process may need corrective measures. This also has important implications in real-time control and monitoring systems. Instead of overloading the integrity subsystem with run-time data checking, it can carry out the checking intermittently on incoming data assuming that any pattern takes some time to develop. Once the integrity subsystem becomes suspicious of the on-going activity, it may take more frequent analyses of the data.

2.D. Semantic constraint modeling in database paves the way for the next generation of database and software development. To most people, the database is still regarded a passive depository of data, where they can find facts of the enterprise. Linkage between application programs and databases are usually poorly defined and loosely coupled. However, with a semantic integrity subsystem, we can see a closely coupled information system with application programs and databases bounded together in one coherent way. Application

programs like expert systems and simulation packages can all share the data in a well controled and orchestrated manner. Therefore, we strongly recommend expanding and promoting applications of semantic constraint modeling methodology in future information system development efforts.

BIBLIOGRAPHY

[Abit85]. Abiteboul, Serge; Victor Vianu, "TRANSACTIONS AND INTEGRITY CONSTRAINTS", Proc. of 4th ACM SIGACT-SIGMOD, Oregon, March, 1985.

[Adib85]. Adiba, M.; G.T. Nguyen, "HANDLING CONSTRAINTS AND META-DATA ON A GENERALIZED DATA MANAGEMENT SYSTEM", EXPERT DATABASE SYSTEMS, Larry Kerschberg, Benjamin/Cummings Publ. Company.

[Aho,74]. Aho, A.V.; Hopcroft, J.E.; Ullman, J.D., "THE DESIGN AND ANALYSIS OF COMPUTER ALGORITHMS", Reading, MA; Addison-Wesley.

[Aho,82]. Aho, A.V.; Hopcroft, J.E.; Ullman, J.D., "DATA STRUCTURES AND ALGORITHMS", Addison-Wesley Publishing Company, 1982.

[Andr85]. Andriole, S.J., "ARTIFICIAL INTELLIGENCE AND NATIONAL DEFENCE: AN AGENDA AND A PROGNOSIS", APPLICATION IN AI, S.J. ANDRIOLE, EDTR, PETROCELLI BOOKS.

[Appo85]. Appollonio, V.D., and elta, "THE INTEGRATION OF THE NETWORK AND RELATIONAL APPROACHES IN A DBMS", Proc. of the 4th British National Conf. on Databases, July 1985.

[Arze86]. Arzen, Karl-Erik, "EXPERT SYSTEMS FOR PROCESS CONTROL", Application of AI in Engineering Problems, 1st Intl. Conf., U.K., Springer-Verlage, 1986.

[Atwo85]. Atwood, T. M., "AN OBJECT-ORIENTED DBMS FOR DESIGN SUPPORT APPLICATIONS", COMPINT, Computer Aided Technologies, 1985, IEEE CH2136-0.

[Atze83]. Atzeni, P.; Chen, P., "COMPLETENESS OF QUERY LANGUAGES FOR THE ENTITY RELATIONSHIP", ENTITY RELATIONSHIP APPROACH TO INFORMATION MODELING AND ANALYSIS; P. Chen, Edtr; North-Holland, 1983.

[Aude86]. Aude, J.S.; Jilary J Kahn, "A DESIGN RULE DATABASE SYSTEM TO SUPPORT TECHNOLOGY-ADAPTABLE APPLICATIONS", 23rd Design Automation Conf.; 1986 IEEE 0738-100X.

[Baci85]. Baciocco, Jr., Albert J., "ARTIFICIAL INTELLIGENCE AND C3I", APPLICATION IN AI, S.J. ANDRIOLE, EDTR, PETROCELLI BOOKS.

[Banc86]. Bancilhon, Franois; Raghu Ramakrishnan, "AN AMATEUR'S INTRODUCTION TO RECURSIVE QUERY PROCESSING", Proc. of SIGMOD '86, ACM, Washington D.C., May, 1986, Carlo Zaniolo, Edtr..

[Bato84]. Batory, D.S.; Alejandro P. Buchmann, "MOLECULAR OBJECTS, ABSTRACT DATA TYPES, AND DATA MODELS: A FRAMEWORK", PROC. OF THE 10TH INTL. CONF. ON VERY LARGE DATA BASES,P 172.

[Bell82]. Bell, John;, "DATA MODELING OF SCIENTIFIC SIMULATION PROGRAMS", PROC. OF ACM SIGMOD INTL. CONF. ON MANAGEMENT OF DATA, JUNE 1982.

[Bena86]. Benayoune, M. and P. E. Preece, "METHODOLOGY FOR THE DESIGN OF DATABASES FOR ENGINEERING APPLICATIONS", CAD, Vol. 18, No. 5, 1986.

[Bern80]. Bernstein, P.; Blaustein, B.; Clark E., "FAST MAINTENANCE OF SEMANTIC INTEGRITY ASSERTIONS USING REDUNDANT AGGREGATE DATA", Proc. of 6th Intl. Conf. On VLDB, 1980.

[Bern81]. Bernstein, P.A.; Blaustein, B.T;, "A SIMPLIFICATION ALGORITHM FOR INTEGRITY ASSERTIONS AND CONCRETE VIEWS", Proc. of COMPSAC'81, IEEE Comp. Soc. 5th Int. Computer Software and Application Conf.,p90-99,1981.

[Bern76]. Bernstein, Philip A., "SYNTHESIZING THIRD NORMAL FORM RELAIONS FROM FUNCTIONAL DEPENDENCIES", Trans. On Database Systems, Vol.1, No.4, Dec. 1976.

[Bert84]. Bertino, E.; D. Apuzzo, "INTEGRITY ASPECTS IN DATA BASE MANAGEMENT SYSTEMS", IEEE TRENDS AND APPLICATIONS: MAKING DATABASE WORK, 1984, CH2053-7, p43-52.

[Blah85]. Blaha, M.R. ; and elta, "A PROCESS ENGINEERING VIEW OF DATABASE MANAGEMENT", AICHE Spring Nat. Meeting, Houston, TX; 1985; Paper 76.b; MicroF 37.

[Blah86]. Blaha, M.R., R.A. Peters, R.L. Motard, "A DATA MODEL FOR PROCESS FEASIBILITY STUDIES", AICHE Spring National Meeting, New Orleans, LA ; April 1986. Paper 43b

[Boer85]. Boerstra, M.L. Edtr., "ENGINEERING DATABASES", Report of the CIAD Project on Engineering Databases; Elsevier, 1985.

[Brad86]. Brady, Lynette I.; C. N.G. Dampney, "THE SEMANTICS OF RELATIONAL DATABASE FUNCTIONS", Proc. of the 5th Intl. Conf. on Entity-Relationship Approach.

[Brod80]. Brodie, M.L., "THE APPLICATION OF DATA TYPES TO DATABASE SEMANITC INTEGRITY", Information Systems, Vol. 5, No. 4, pp 287-296, 1980.

[Brod84]. Brodie, M.L., "ON THE DEVELOPMENT OF DATA MODELS", ON CONCEPTUAL MODELING, Michal L. Brodie and elta, Springer-Verlag, 1984, p19.

[Buch79]. Buchmann, A.P.; A.G. Dale, "EVALUATION CRITERIA FOR LOGICAL DATABASE DESIGN METHODOLOGIES", COMPUTER AIDED DESIGN, VOL.11, NO.3, MAY, 1979, P121.

[Buch84]. Buchmann, A.P., "CURRENT TRENDS IN CAD DATABASES", COMPUTER AIDED DESIGN, VOL. 16, NO. 3, MAY 1984, P123

[Bune79]. Buneman, O.P.; Clemons E.K., "EFFECIENT MONITORING RELATIONAL DATABASES", ACM, Trans. On Database Systems, Vol. 4, No. 3, Sept. 1979, pp368-382.

[Camp85]. Campbell, D.M.; Embley, D.W.; Czejdo, B., "A RELATIONALLY COMPLETE QUERY LANGUAGE FOR AN ENTITY-RELATIONSHIP MODEL", Proc. of the 4th Intl. Conf. on Entity-Relationship Approach, 1985.

[Canf85]. Canfield, F.B. , "IMPACT OF COMPUTER AIDED ENGINEERING IN THE PROCESS INDUSTRY(COMPUTER AIDED ENGINEERING IN THE PROCESS INDUSTRY)", AICHE Spring Nat. Meeting, Houston, TX; 1985; Paper 20c.

[Chen76]. Chen, Peter P.S., "THE ENTITIY-RELATIONSHIP MODEL-TOWARD A UNIFIED VIEW OF DATA", ACM Trans. On Database Systems; Vol. 1, No.1, March, 1976,p9.

[Chen77]. Chen, Peter P.S., "THE ENTITY RELATIONAL APPROACH TO LOGICAL DATABASE DESIGN", Q.E.D. MONOGRAPH SERIES, DATABASE MANAGEMENT, No. 6, 1977.

[Chen83a]. Chen, Peter P.S., "A PRELIMINARY FRAMEWORK FOR ENTITY-RELATIONSHIP MODELS", "ENTITY RELATIONSHIP APPROACH TO INFORM. MODELING AND ANALYSIS", P. Chen, Edtr; Elsevier Science Publ., 1983.

[Chen83b]. Chen, Peter P., Editor, "ENTITY-RELATIONSHIP APPROACH TO INFORMATION MODELING AND ANALYSIS", Proceeding of the 2nd Intl. conf. On Entity Relationship Approach, 1981, North Holland.

[Chen84]. Chen, Peter P.S., "AN ALGEBRA FOR A DIRECTIONAL BINARY ENTITY RELATIONSHIP MODEL", DATA ENGINEERING INTL. CONF., IEEE CH20313, p 37.

[Chen87]. Chen, H.P.; Chen, P.P., "THE SEMANTIC REGIONS OF ENTITY RELATIONSHIP GRPAH AND THE CONSTRAINS ON THE GLOBAL CONSISTENCY OF DATABASE", Working Paper, Departement of Computer Sci., Louisiana State Univ., 1987.

[Chol85]. Cholvy, Laurence; Jack Foisseau, "ENCODING REQUIREMENTS TO INCREASE MODELISATION ASSISTANCE", KNOW. ENG. IN COMPUTER AIDED DESIGN, J.S. Gero, Edtr, Elsevier Sci. Publ. B.V., North-Holland, IFIP 1985.

[Codd72]. Codd, E.F., "RELATIONAL COMPLETENESS OF DATABASE SUBLANGUAGES", DATABASE SYSTEMS; R. Rustin, Edtr; pp 65-98; Prentice-Hall; 1972.

[Crem83]. Cremers A.B. and Domanng, "AIM: AN INTEGRITY MONITOR FOR THE DATABASE SYSTEM INGRES", Proc. of 9th Intl. Conf. on VLDB, Oct.,1983.

[Darl82]. Darlington, J.; and elta, Editors, "FUNCTION PROGRAMMING AND ITS APPLICATIONS: AN ADVANCED COURSE", Cambridge University Press.

[Date81]. Date, C.J., "AN INTRODUCTION TO DATABASE SYSTEMS, VOL. I, 3rd. Edition", Addison-Wesley Publ. Comp., 1981.

[Date83]. Date, C.J., "AN INTRODUCTION TO DATABASE SYSTEMS, VOL. II.", Addison-Wesley Publ. Comp., 1983.

[Davi83]. Davis, C.G.; and elta, Editors, "ENTITY-RELATIONSHIP APPROACH TO SOFTWARE ENGINEERING", Proceedign of the 3rd Intl. Conf. on Entity Relationship Approach, 1983, North-Holland.

[Dehe76]. Deheneffe, C.; Hennebert, H., "NUL: A NAVIGATIONAL USER'S LANAGUAGE FOR A NETWORK STRUCTURE DATABASE", Proc. 1976 SIGMOD Conf.; pp135-142, ACM, New York, June, 1976.

[Ditt85]. Dittrich, K. R; Raymond A. Lorie, "OBJECT-ORIENTED DATABASE CONCEPTS FOR ENGINEERING APPLICATIONS", COMPINT, IEEE CH21360, 1985, P321.

[Ditt86]. Dittrich, K.R.; Kotz, A.M.; Mulle, J.A., "AN EVEN/TRIGGER MECHANISM TO ENFORCE COMPLEX CONSISTENCY CONSTRAINTS IN DESIGN DATABASE", SIGMOD Rec., Vol.15, No.3, 22-36,Sept.1986.

[Ditt85]. Dittrich, Klaus R. and elta, "A MULTILEVEL APPROACH TO DESIGN DATABASE SYSTEM AND ITS BASIC MECHANISM", COMPINT, Computer Aided Technologies, 1985, IEEE CH2136-0.

[Doga85]. Dogac, A.; Chen, P.P.; Erol, N., "THE DESIGN AND IMPLEMENTATION OF AN INTEGRITY SUBSYSTEM FOR THE RELATIONAL DBMS RAP", Proc. of The 4th Intl. Conf. on Entity Relationship Approach, October, 1985.

[Doga86]. Dogac, Asuman; F. Eyupoglu, "VERS- A VECTOR BASED ENTITY RELATIONSHIP DATABASE MANAGEMENT SYSTEM", Proc. of the 5th ER Conf., France, 1986.

[Dona85]. Donaldson, R.W. , "DATABASE MANGEMENT: THE REQUIREMENTS FOR PROCESS PLANT CAD", AICHE Spring Nat. Meeting, Houston, TX; 1985; Paper 14a; MicroF 13.

[East81]. Eastman, Charles M., "DATABASE FACILITIES FOR ENGINEERING DATABASE", PROC. OF THE IEEE, VOL. 69, NO. 10, OCT. 1981, P1249.

[East85]. Eastman, C. , Ali Kutay, "SPECIFICATION OF FORM:ULAE : A DISTRIBUTED ENGINEERING DATA MANAGEMENT SYSTEM", AICHE Spring Nat. Meeting, Houston, TX; 1985; Paper 20d; MicroF 8

[Ehri84]. Ehrich, H.D., H.W. Lipeck, M. Gogolla, "SPECIFICATION, SEMANTICS, AND ENFORCEMENT OF DYNAMIC DATABASE CONSTRAINTS", Proc. of 10th Intl. Conf. On VLDB, Singapore, Aug., 1984,pp301-308.

[Elma83]. Elmasri, R; Wiederhold, G., "GORDAS: A FORMAL HIGH-LEVEL QUERY LANGUAGE FOR THE ENTITY RELATIONSHIP MODEL", ENTITY RELATIONSHIP APPROACH TO INFORMATION MODELING AND ANALYSIS; P.Chen, Edtr; North-Holland, 1983.

[Eswa75]. Eswaran, Chamberlin, "FUNCTIONAL SPECIFICATION OF A SUBSYSTEM FOR DATA BASE INTEGRITY", Proc. of 1st Intl. Conf. On VLDB, Framingham, Sept, 1975.

[Fagi81]. Fagin, "A NORMAL FORM FOR RELATIONAL DATABASES THAT IS BASED ON DOMAINS AND KEYS", ACM, Trnas. On Database System, Vol. 6, No. 3, p387-415.

[Fenv85a]. Fenves, S. J., "REPRESENTATION AND PROCESSING OF ENGINEERING DESIGN CONSTRAINTS IN A RELATIONAL DATABASE", COMPINT, Computer Aided Technologies, 1985, IEEE CH2136-0.

[Fenv85b]. Fenves, S. J.; Rasdorf, W.J., "TREATMENT OF ENGINEERIGN DESIGN CONSTRAINTS IN A RELATIONAL DATABASE", Engineering with Computers, 1, no.1, 1985.

[Fern81]. Fernandez, E.B.;Summers, R.C.; Wood, C., "DATABASE SECURITY AND INTEGRITY", Addison-Weslsey, 1981.

[Flyn84]. Flynn, D.J., "AN ANALYSIS OF CERTAIN DATA MODELING WITH RESPECT TO THEIR HANDLING OF SELECTED INTEGRITY CONSTRAINTS", Proc. of 3rd British National Conf. on Database, L. Longstaff, Edtr, Cambridge University Press, 1984.

[Fode83]. Foderaro, J.K.; Sklower, K.L.; Layer, K., "THE FRANZ LISP MANUAL", University of California, Berkeley, 1983.

[Fros83]. Frost, R.A; Whittaker, S., "A STEP TOWARDS THE AUTOMATIC MAINTENANCE OF THE SEMANTIC INTEGRITY OF DATABASES", Computer Journal, Vol. 26, No.2, p124-133, May, 1983.

[Furt81]. Furtado, A.L., "DYNAMIC MODELING OF A SIMPLE EXISTENCE CONSTRAINT", Information Systems, Vol. 6, 1981, pp73-80.

[Gard79]. Gardarin, C.; Melkanoff M., "PROVING CONSISTENCY OF DATABASE TRANSACTIONS", Proc. of 5th Intl. Conf. On VLDB, Rio de Janeiro, Oct. 1979.

[Gray84a]. Gray, Mike, "DATABASES FOR COMPUTER AIDED DESIGN", NEW GENERATIONS OF DATABASE. ACADEMIC PRESS, 1984.

[Gray84b]. Gray, Peter, "LOGIC, ALGEBRA, AND DATABASES", Ellis Horword Ltd., 1984.

[Hamm76]. Hammer, M.; McLeod, D., "A FRAMEWORK FOR DATABASE SEMANTIC INTEGRITY", Proc. of 2nd Intl. Conf. On Software Engineering, San Francisco, 1976.

[Hamm78]. Hammer, M.; S. K. Sarin, "EFFICIENT MONITORING OF DATABASE ASSERTIONS", Proc. ACM-SIGMOD Conf., Austin,

June, 1978.

[Hard84]. Hardwick, Martin, "EXTENDING THE RELATIONAL DATABASE DATA MODEL FOR DESIGN APPLICATIONS", Proc. of 21st the Design Auto. Conf. , 1984.

[Hart85]. Hartzband, David J. ; Fred J. Maryanski, "ENHANCING KNOWLEDGE REPRESENTATION IN ENGINEERING DATABASES", IEEE, COMPUTER, SEPT. 1985, P39.

[Hask82]. Haskin, R.; R. Lorie, "ON EXTENDING THE FUNCTIONS OF A RELATIONAL DATABASE SYSTEM", Proc. ACM SIGMOD Intl. Conf. On the Management of Data, 1982, p207-212.

[Hatt85]. Hattori, F; and elta, "A COMPARISON OF LISP, PROLOG, AND ADA PROGRAMMING PRODUCTIVITY IN AI AREA", Software Engineering 8th Intl. Conf., 1985, p285.

[Heer83]. Heerema, F.J.; H. van Hedel, "AN ENGINEERING DATA MANAGEMENT SYSTEMS FOR COMPUTER AIDED DESIGN", ADV. ENG. SOFTWARE, 1983, VOL.5, NO. 2, P67.

[Horo84]. Horowitz, Ellis; Sahni, Sartaj, "FUNDAMENTALS OF DATA STRUCTURES IN PASCAL", Computer Science Press, 1984.

[Hsu,85]. Hsu, Arding; Imielinski, Tomasz, "INTEGRITY CHECKING FOR MULTIPLE UPDATES", Proc. of ACM-SIGMOD, 1985 , Proc. of International Conference On the Management of Data, 1985.

[Jone83]. Jones, J. Kevin, John A. Liles, "THE NEXT STEP IN IMPROVING PROCESS DESIGN PRODUCTIVITY", AICHE Symposium Series, S-231, p51, 1983.

[Jone86]. Jones, L.P., "A MODEL FOR THE OBJECT-ORIENTED SPECIFICATION OF DATABASE APPLICATIONS", Technical Report #86-010, Dept. of Comp. Science, Louisiana State Univ..

[Katz85]. Katz, R.H., "INFORMATION MANAGEMENT FOR ENGINEERING DESIGN", Springer-Verlag, 1985.

[Kim,85]. Kim, Won; Jay Banerjee, "SUPPORT OF ABSTRACT DATA TYPES IN A CAD DATABASE SYSTEM", COMPINT, Computer Aided Technologies, 1985, IEEE CH2136-0.

[Lafu79]. Lafue, G. , "AN APPROACH TO AUTOMATIC MAINTENANCE OF SEMANTIC INTEGRITY IN LARGE DESIGN DATABASES", National computer Conf., AFIPS, 1979.

[Lafu79]. Lafue, G.M.E., "INTEGRATING LANAGUAGE AND DATABASE FOR CAD APPLICATIONS", COMPUTER AIDED DESIGN, VOL. 11, NO. 3, MAY, 1979, P127.

[Lafu82]. Lafue, Gilles M.S., "SEMANTIC INTEGRITY DEPENDENCIES AND DELAYED INTEGRITY CHECKING", 8TH INTL. CONF. ON VERY LARGE DATA BASES, SEPT. 1982, P292.

[Lee,87a]. Lee, H.L.; Chen, P.P, "AN ER-BASED APPLICATIVE DATA LANGUAGE (ADL)", Technical Report 1988 (to be published), Department of Computer Science, Lousiana State University.

[Lee,87b]. Lee, H.L.; Chen, P.P., "A COMPUTATIONAL APPROACH TO SEMANTIC CONSTRAINT MODELING BY THE APPLICATIVE DATA LANGUAGE (ADL)", Technical Report 1988 (to be published), Department of Computer Science, Lousiana State University.

[Lee,87c]. Lee, H.L.; Chen, P.P., "RUN-TIME CONSTRAINT ENFORCEMENT STRATEGY FOR THE APPLICATIVE DATA LANGUAGE (ADL)", Technical Report 1988 (to be published), Department of Computer Science, Lousiana State University.

[Lili82]. Lilien, L.; Bhagara, B., "A SCHEME FOR VERIFICATION OF INTEGRITY ASSERTIONS IN A TRANSACTION PROCESSING SYSTEMS", Proc. of COMSAC '82, IEEE Comp. Soc. ,6th Intl. Computer Software and Appl. Conf. 139-148, 1982.

[Ling84]. Ling, T.W.; Rajagopalan, O., "A METHOD TO ELIMINATE AVOIDABLE CHECKING OF INTEGRITY CONSTRAINTS", Proc. of Trends and Applications, 1984, Making Database Work, 1984.

[Lori85]. Lorie, Raymond; and elta, "SUPPORTING COMPLEX OBJECTS IN A RELATIONAL SYSTEM FOR ENGINEERING DATABASES", OUERY PROCESSING IN DATABASE SYSTEMS, Won Kim, David S. Reiner, Don S. Batory, Springer-Verlag, 1985.

[MacL83]. MacLennan, Bruce J., "PRINCIPLES OF PROGRAMMING LANGUAGES: DESIGN, EVALUATION, AND IMPLEMENTATION", Holt, Rinehart and Winston, 1983.

[Maie84]. Maier, David, "THE THEORY OF RELATIONAL DATABASES", Pittman Publishing Ltd., 1984.

[Mait85]. Maitra, Partha P , "REQUIREMENT FOR A COMPUTER AIDED ENGINEERING SYSTEM FOR PROCESS PLANT DESIGN", AICHE Spring Nat. Meeting, Houston, TX; 1985; Paper 20b; MicroF 7.

[Malv81]. Malvestuto, F.M., "ON AN OPTIMAL POLICY FOR THE DATA BASE INTEGRITY CHECKING", IEEE, Proc. of a Symposium on Reliability and Distributed Software and Database Systems,p200-204, 1981.

[Mark83]. Markowitz, V.M.; Raz, Y., "ERROL: AN ENTITY-RELATIONSHIP ALGEBRA", ENTITY-RELATIONSHIP APPROACH TO SOFTWARE ENGINEERING; C. Davis and elta, Edtrs; North-Holland, 1983.

[Matu83]. Matusek, Norman A., "ENGINEERING PROJECT CONTROL DATABASE", AICHE Symposium Series, S-231, p69, 1983.

[Merr84]. Merrett, T.H., "RELATIONAL INFORMATION SYSTEMS", Reston Publ. Comp., Inc., 1984.

[Moor84]. Moore, Robert L.; and elta, "A REAL-TIME EXPERT SYSTEM FOR PROCESS CONTROL", ARTIFICIAL INTELLIGENCE APPLICATION, IEEE 21071, 1984.

[Moor86]. Moore, Robert L., "EXPERT SYSTEM IN PROCESS CONTROL: APPLICATIONS EXPERIENCE", Application of AI in Engineering Problems, 1st Intl. Conf., U.K., Springer-Verlage, 1986.

[Morg84]. Morgenstern, Mathew, "CONSTRAINT EQUATIONS: DEVLARATIVE EXPRESSION OF CONSTRAINTS WITH AUTOMATIC ENFORCEMENT", Proc. of 10th Intl. Conf. On VLDB, Singapore, 1984.

[Morg86]. Morgenstern, Matthew, "THE ROLE OF CONSTRAINTS IN DATABASES, EXPERT SYSTEMS, AND KNOWLEDGE REPRESENTATION", EXPERT DATABASE SYSTEMS, Larry Kerschberg, Benjamin/Cummings Publ. Company.

[Murt85]. Murthy, T. S.; J.S. Arora, "A SURVEY OF DATABASE MANAGEMENT IN ENGINEERING", ADVANCE ENGINEERING SOFTWARE, 1985 VOL. 7, NO.3, P126.

[Naka83]. Nakano, Ryohei, "INTEGRITY CHECKING IN A LOGIC ORIENTED ER MODEL", ENTITY RELATIONSHIP APPROACH TO SOFTWARE ENGINEERING, C.G. Davis, S. Jajodia, P.A. Yeh, North-Holland, 1983.

[Nava83]. Navathe, Shamkant B., "DATABASE MANAGEMENT OF COMPUTER AIDED DESIGN DATA", AICHE Symposium Series, S-231, p43, 1983.

[Neum82]. Neumann, Thomas; Christoph Hornung, "CONSISTENCY AND TRANSACTIONS IN CAD DATABASE", 8TH INTL. CONF. ON VERY LARGE DATA BASES, SEPT. 1982, P181.

[Nguy86a]. Nguyen, G.T., "SEMANTIC DATA ENGINEERING FOR GENERALIZED DATABASES", DATA ENGINEERING INTL. CONF., IEEE CH22616, 400.

[Nguy86b]. Nguyen, G.T., "OBJECT PROTOTYPES AND DATABASE SAMPLES FOR EXPERT DATABASE SYSTEMS", Proc. of the 1st Intl. Conf. On Expert Systems, Larry Kerschberg, April, 1986.

[Nico82]. Nicolas J.M., "LOGIC FOR IMPROVING INTEGRITY CHECKING IN RELATIONAL DATABASES", Acta Informatica, Vol. 18, July, 1982.

[Nico78]. Nicolas, J.M.; K. Yazdanian, "INTEGRITY CHECKING IN DEDUCTIVE DATA BASES", LOGIC AND DATABASE, Herve Gallaire, Jack Minker, Plenum Press, 1978.

[Niid77]. Niida, K.; H. Yagi; T. Umeda, "AN APPLICATION OF DATABASE MANAGEMENT SYSTEM (DBMS) TO PROCESS DESIGN", COMPUTER AND CHEMICAL ENGINEERING, VOL. 1, NO.1, P33, 1977.

[Pete83]. Peters, Richard A., "DEVELOPMENT OF A PROJECT ENGINEERING DATABASE : A SYSTEM APPROACH", AICHE Symposium Series, S-231, p59, 1983.

[Pete85]. Peters, Richard A. ; , "CHARACTERISTIC OF ENGINEERING DATA: IMPLICATION FOR DATABASE AND SYSTEMS DESIGN"", AICHE Spring Nat. Meeting, Houston, TX; 1985; Paper 20.a, MicorF #7. 1985.

[Poon80]. Poonen, G., "CLEAR: A CONCEPTUAL LANGUAGE FOR ENTITIES AND RELATIONSHIPS", CENTRALIZED AND DISTRIBUTED DATABASES; W. Chu and P.Chen, Edtrs; 1980.

[Roth84]. Roth, Mark A.; Korth, H.F.; Siberschatz, A., "EXTENDED ALGEBRA AND CALCULUS FOR NON-1NF RELATIONAL DATABASES", Technical Report, TR-84-36, Dec., 1984.

[Schr84]. Schrefl, M.; A.M. Tjoa; R.R. Wagner, "COMPARISON CRITERIA FOR SEMANTIC DATA MODELS", DATA ENGINEERING INTL. CONF., IEEE CH20313,P120.

[Shep84]. Shepherd, A. and Kerschberg, L., "PRISM: A KNOWLEDGE-BASED SYSTEM FOR SEMANTIC INTEGRITY SPECIFICATION AND

ENFORCEMENT IN DATABASE SYSTEM", Proc. ACM-SIGMOD Conf., Boston, 1984, pp307-315.

[Shep86]. Shepherd, Allan; Larry Kerschberg, "CONSTRAINT MANAGEMENT IN EXPERT DATABASE SYSTEMS", EXPERT DATABASE SYSTEMS, Larry Kerschberg, Benjamin/Cummings Publ. Company.

[Shos78]. Shoshani, A., "CABLE: A LANGUAGE BASED ON THE ENTITY RELATIONSHIP MODEL", Technical Paper, Comp.Sci. and Applied Math. Dept, Berkely, CA. 1978.

[Shos84]. Shoshani, Arie; Frank Olken; Harry K.T. Wong, "CHARACTERISTICS OF SCIENTIFIC DATABASES", PROC. OF THE 10TH INTL. CONF. ON VERY LARGE DATABASES,P147.

[Simo84]. Simon, Eric; Patrick Valduriez, "EFFICIENT ALGORITHMS FOR INTEGRITY CONTROL IN A DATABASE MACHINE", TRENDS AND APPLICATIONS: MAKING DATABASE WORK, IEEE CH20537, P53.

[Smit85]. Smith, Thomas R., "A DATA ARCHITECTURE FOR AN UNCERTAIN DESIGN AND MANUFACTURING ENVIRONMENT", 22ND DESIGN AUTOMATION CONF., 1985, P312.

[Spoo86]. Spooner, David L.; M. A. Milicia; D. B. Faatz, "MODELING MECHANICAL CAD DATA WITH DATA ABSTRACTION AND OBJECT-ORIENTED TECHNIQUES", DATA ENGINEERING INTL. CONF., IEEE CH22616, P416, 1986.

[Stal86]. Staley, Scott M.; David C. Anderson, "FUNCTIONAL SPECIFICATION FOR CAD DATABASES", COMPUTER AIDED DESIGN, 1986.

[Ston75]. Stonebraker M., "IMPLEMENTATION OF INTEGRITY CONSTRAINTS AND VIEWS BY QUERY MODIFICATION", Proc. ACM-SIGMOD Intl. Conf. On the Management of Data, 1975.

[Su,86]. Su, Stanley Y.W., "MODELING INTEGRATED MANUFACTURING DATA WITH SAM", IEEE, COMPUTER, Jan. 1986,p34.

[Subi86]. Subieta, K.; Missala, M., "SEMANTICS OF QUERY LANGUAGES FOR THE ENTITY RELATIONSHIP MODEL", Proc. of the 5th Intl. Conf. on Entity-Relationship Approach, 1985.

[Tabo83a]. Tabourier, Yves, "FURTHER DEVELOPMENT OF THE OCCURENCE STRUCTURE CONCEPT: THE EROS APPROACH", ENTITY RELATIONSHIP APPROACH TO SOFTWARE

ENGINEERING, C.G. Davis, S. Jajodia, P.A. Yeh, North-Holland, 1983.

[Tabo83b]. Tabourier, Yves; Dominique Nanci, "THE OCCURENCES STRUCTURE CONCEPT", ENTITY-RELATIONSHIP APPROACH TO INFORMATION MODELING AND ANALYSIS, P.P. Chen, North-Holland, 1983.

[Thee81]. Theerachetmongkol, A, "A DATA BASE SYSTEM WHICH STORE AND MAINTAINS MORE SEMANTICS", Ph.D. Dissertation, Dept. of Comp. Sci., Monash University, August, 1981.

[Thom86]. Thomas, Stan J.;Fischer, P.C., "NESTED RELATION STRUCTURES", Advances in Computing Research, Vol.3, Page 269-307.

[Tsic82]. Tsichritzis, D.C.; Lochovsky, F.H., "DATA MODELS", Prentice-Hall, Inc., 1982.

[Ullm82]. Ullman, J.D., "PRINCIPLE OF DATABASE SYSTEMS", Pitman Publishing Limited, 1982.

[Webe83]. Weber, W.;Stucky, W.;Karszt, J., "INTEGRITY CHECKING IN DATABASE SYSTEMS", Information Systems, Vol. 8, No.2, p125-136, 1983.

[Wile84]. Wilensky, Robert, "LISPCRAFT", W.W. NORTON & COMPANY, 1984.

[Wils80]. Wilson, Gerald A., "A CONCEPTUAL MODEL FOR SEMANTIC INTEGRITY CHECKING", Proc. of 6th Intl. Conf. On VLDB, 1980.

[Wins84]. Winston, P.H.; Horn, B.K.P.,"LISP, 2nd Ed.", Addison- Wesley Publ. Company, 1984.

[Wint85]. Winter, P. and C.J. Angus, "A RELATIONAL DATABASE FOR PROCESS ENGINEERING", AICHE Spring Nat. Meeting, Houston, TX; 1985; Paper 20e; MicroF 8.

[Wojc84]. Wojcik, Anthony S.;Joseph Kljaich; Nagendra Srinivas, "A FORMAL DESIGN VERIFICATION SYSTEM BASED ON AN AUTOMATIED REASONING SYSTEM", Proc. Of 21st Design Automation Conf., IEEE , 1984.

APPENDIX A. BNF OF APPLICATIVE DATA LANGUAGE (ADL)

Notation:

1. Non-terminals: Character strings with capital letters as the first characters. (Except those boldfaced operator keywords.)
2. Terminals: character strings with small letters as the first characters. (Including those boldfaced operator keywords.)
3. The following are grammar symbols. They are not part of the language.

[ ]# : The construct in the square brackets repeat n times with $1 \leq n \leq \infty$.

[ ]$: The construct in the square brackets repeat n times with $0 \leq n \leq \infty$.

[| |]: selections from the alternatives constructs in the square brackets.

! : The end of a grammar rule is marked by an exclamation mark.

4. Two start symbols of the grammar: QUERY, and CONSTRAINTS.

BNF of ADL:

```
QUERY::= Display ( Variable). |
         Statement_list, Display ( Variable). !
CONSTRAINT::= B_statement, Display (Variable ). |
        Statement_list,B_statement, Display (Variable ).!
Variable::= S_variable | D_variable | V_variable !
S_variable::= string !
D_variable::= string !
V_variable::= string !
B_variable::= string !
Statement_list::= Statement | Statement, Statement_list !
Statement::= S_statement | D_statement |
             V_statement | B_statement !
S_statement::= S_variable := S_expression !
D_statement::= D_variable := D_expression !
V_statement::= V_variable := V_expression !
B_statement::= B_variable := B_expression !
S_expression::= S_variable | (S_expression) |
                JoinS J_set (S_expression) |
                ProjectS PS_set (S_expression) |
                Group_byS G_set (S_expression) |
                SelectS Pformula1 (S_expression) |
                S_expression ∪ S_expression |
                S_expression ∩ S_expression |
                S_expression - S_expression |
                Join_list |
```

```
                (S_expression) Pformula1 !
D_expression::= D_variable | (D_expression) |
                SelectD Pformula2 (D_expression) |
                ProductD (D_expression, D_expression) |
                ProjectD PD_set (D_expression) |
                ExtendD Eq_set (D_expression) |
                Extract Ex_set (S_expression) !
V_expression::= V_variable | (V_expression) |
                CountS ( S_expression ) |
                DV_op (D_expression) |
                V_expression VV_op V_expression !
B_expression::= B_variable | (B_expression) |
                B_expression BB_op B_expression |
                ¬ B_expression |
                ST_pr (S_expression, S_expression) |
                DR_pr ( D_expression ) |
                VU_pr (V_expression, V_expreesion) !
DV_op::= CountD | SumD | AvgD | MaxD | MinD |ValueD !
VV_op::= + | - | * | / !
BB_op::= ∧ | ∨ !
ST_pr::= PS⊂ | PS⊃ | PS⊂ | PS⊃ | PS= !
DR_pr::= PD< | PD> | PD< | PD> | PD= !
VU_pr::= PV< | PV> | PV≤ | PV≥ | PV= | PV◇ !
SB_pr::= ⊆ | ⊇ | ⊂ | ⊃ ⊤ =s !
VB_pr::= < | > | ≤ | ≥ | = | ◇ !
J_set::= { Alist1 } !
PS_set::= { Alist1 } !
G_set::= { Alist2 } !
PD_set::= { Alist3 } !
Join_list::= S_expression | S_expression°Join_list !
Alist1::=  [string | string* | (string [x string]#)*] |
     [string | string* | (string [x string]#)*] , Alist1 !
Alist2::=  string | string, Alist2 !
Alist3::=[string | string*] | [string | string*], Alist3!

Pformula1::= Atom1 | (Pformula1) |
             Pformula1 BB_op Pformula1 | ¬ Pformula1 !
Pformula2::= Atom2 | (Pformula2) |
             Pformula2 BB_op Pformula2 |¬ Pformula2 !
Atom1::= string.string VB_pr Value_const |
         string.string VB_pr string.string |
```

```
        |[string^c|(string[xstring]#)^c]| VB_pr integer |
        [string^c|(string[xstring]#)^c] SB_pr Set_const !
Value_const::= integer | real | 'string' !
Set_const::= { Vector[,Vector]$} !
Vector::= <Value_const[,Value_const]$> !
Atom2::= string VB_pr Value_const | string VB_pr string !

Eq_set::= { Eq_list } !
Eq_list::= Equation | Equation, Eq_list !
Equation::= string:= Term !
Term::= Factor | (Term) | Term VV_op Term !
Factor::= Value_const | string | Bag_op (string^c) !
Bag_op::= SumB | CountB | AvgB | MaxB | MinB !

Ex_set::= { Ex_list } !
Ex_list::= [string|string*] <- [string|string*].string |
   [string|string*] <- [string|string*].string , Ex_list !
```

APPENDIX B. RELATIONAL COMPLETENESS OF ADL

The expressive power of earlier query languages for the relational data model are usually judged by their relational completeness as defined by [Codd72]. However, evolution in database query languages over the past ten to fifteen year already generate many high level query languages (e.g. SQL, QBE, QUEL) far exceed the expressive power of the original relational complete language [Date81, Ullm82, Gray84]. Relational completeness, thus, becomes an obsolete benchmark for new generation of database query languages.

Atzeni and Chen [Atze83] defined ER completeness and simplified ER completeness based on ER calculus. They claimed that earlier ERM based query languages CLEAR, CABLE did not satisfy either definition of completeness. Their definitions of completeness , however, did not handle queries comparing unconnected objects (equivalent to the θ-join in [Merr84]) in an ERM. Therefor, ER completeness is less powerful than relational completeness.

Campbell [Camp85] proposed another ER-completeness notion that is based on the relation algebra of the original Codd's proposal [Codd72]. Here, we also prove that ADL satisfies relational completeness that is equivalent to the relational algebra. We prove this mainly by using data space operators **ProductD**, **SelectD**, and **ProjectD** since in relational algebra we do not have object level operators like those in the entity space of ADL.

To prove the equivalence of ADL expressions and relational algebra expressions, we need some common data structure which both data manipulation languages can work on. We use the relational realization of ERM to prove the equivalence.

Theorem: ADL is a relationally complete query language based on the data structures and operators defined in this paper.

Proof: Assuming that each entity set and each relationship set corresponds to a relation in the relational data model. Attributes of entity relation E are Sch(E)= $\{EID,A_1,...,A_m\}$. Attributes of relationship relation R are Sch(R)= $\{EID_1,...,EID_n\}$. Each entity set and each relationship set must have a unique name. Therefore, all attributes in the database can be uniquely identified as $E_i.A_j$ or $R_h.A_k$.

The relational algebra that is relationally complete includes the following operators: $\cup_r$, $-_r$, $\mathbf{Project}_r$, $\mathbf{Select}_r$, $\mathbf{Product}_r$. We prove ADL is also relationally complete by showing that for each of the relational algebra operators there is an equivalent expression in ADL. Since relational algebra is based on data level computations, all operators in the equivalent ADL expressions are data space operators, **ProjectD**, **SelectD**, and **ProductD**. The

naming convention of these operators requires some modification of our initial proposal. Now, we propose the following naming convention:

1. The attributes of a d_relation as being extracted from an entity set or a relationship set should be identified as $E_i.A_j$ or $R_h.A_k$. Renaming is avoided.
2. When **ProductD** is applied on two d_relations, e.g. τ_3:= **ProductD**(τ_1, τ_2), with identical schemes, i.e., Sch(τ_1)=Sch(τ_2), we rename all the attributes of τ_2 by appending a numerical suffix at the end of each attribute names of τ_2. Therefore, all attributes in τ_3 are still uniquely identifiable.
3. The P_set in the **ProjectD** operation and the predicate formula in the **SelectD** operation use the same naming convention as 1.

Case 0. Every entity set or every relationship set now corresponds to a d_relation by extracting all the attributes from the base state implied by the entity set or the relationship set.

$$\tau_1 := \textbf{Extract}\ \{E1.A1 \leftarrow E1.A1, \ldots, E1.Am \leftarrow E1.Am\}\ (E),$$
$$\ldots$$
$$\tau_n := \textbf{Extract}\ \{Rn.A1 \leftarrow Rn.A1, \ldots, Rn.Ap \leftarrow Rn.Ap\}\ (R).$$

Case 1. $\textbf{Project}_r$ is equivalent to **ProjectD**.

A relation τ_1 derived from a series of relation algebra computations may form another relation τ_2 by using the $\textbf{Project}_r$ operator. τ_2:= $\textbf{Project}_r$ $_{P\_set}$ (τ_1). Based on the definition of **ProjectD** and the attribute naming convention we can easily see that τ_2:= **ProjectD** $_{P\_set}$ (τ_1) and **ProjectD** $_{P\_set}$ (τ_1) = $\textbf{Project}_r$ $_{P\_set}$ (τ_1). But there is a difference between **ProjectD** and $\textbf{Project}_r$. That is, **ProjectD** does not remove duplicates while $\textbf{Project}_r$ does. The existence of duplicates in a relation only creates inconvenience and occupies spaces. They do not cause any loss or gain of information during the relational operations.

Case 2. $\textbf{Product}_r$ is equivalent to **ProductD**.

Two relations τ_1 and τ_2 derived from a series of relation algebra computations may form a third relation τ_3 by the $\textbf{Product}_r$ operator. τ_3:= $\textbf{Product}_r$ (τ_1, τ_2). We can find the τ_3 by using **ProductD**, i.e. τ_3:= **ProductD**(τ_1, τ_2).

Case 3. $\textbf{Select}_r$ is equivalent to **SelectD**.

A relation τ_1 derived from a series of relation algebra computations may form another relation τ_2 by the $\textbf{Select}_r$ operator. τ_2:= $\textbf{Select}_r$ Pformula (τ_1). The same relation τ_2 can be derived from τ_2:= **SelectD** Pformula (τ_1). Pfomula is constructed from valid atoms as discussed in CHAPTER II.

Case 4. $\cup_r$ is equivalent to the combination of **ProjectD**, **SelectD**, and **ProductD**.

Since in the data space there is no direct equivalent operator to the relation union $\cup_r$, we need to prove that the combination of **ProjectD**, **SelectD**, and **ProductD** is equivalent to $\cup_r$. If there are two relations τ_1, τ_2 that are compatible for the union operation (all attributes are identical), we can find a third relation by taking the union of the two. $\tau_3 = \tau_1 \cup_r \tau_2$. Suppose that $Sch(\tau_1)=Sch(\tau_2)=\{E_1.A_1, E_2.A_2, \ldots, E_m.A_m\}$. We first create a relation that is the Cartesian product of all the attribute domains of τ_1 and τ_2.

τ_{31} := **Extract** {E1.A1<-E1.A1} (E_1),
τ_{32} := **Extract** {E2.A2<-E2.A2} (E_2),
...
τ_{3m} := **Extract** {Em.Am<-Em.Am} (E_m).

And τ_3' := **ProductD**(τ_{31},
ProductD(τ_{32},
.....,
ProductD($\tau_{3,m-1}$, $\tau_{3,m}$)...)).

Then we take the Cartesian product of τ_3' and τ_1, τ_2 to make τ_3". τ_3":= **ProductD**(**ProductD**(τ_3', τ_1), τ_2). Note that due to renaming convention of **ProductD** , the attributes of τ_3" are
$\{E_1.A_1, \ldots, E_m.A_m, E_{11}.A_{11}, \ldots E_{1m}.A_{1m}, E_{21}.A_{21}, \ldots E_{2m}.A_{2m}\}$.
Finally, we apply **SelectD** and **ProjectD** on τ_3" to obtain an equivalent τ_3 as that obtained by the relation union $\cup_r$.

τ_3 := **ProjectD** {E1.A1, ..., Em.Am} (
SelectD (E1.A1=E11.A11, ..., Em.Am=E1m.A1m) V
(E1.A1=E21.A21, ..., Em.Am=E2m.A2m) (τ_3''))).

Therefore, $\tau_3 = \tau_1 \cup_r \tau_2$. and $\cup_r$ is equivalent to the sequence of operations that consist of **ProductD**, **SelectD**, and **ProjectD**.

Case 5. $-_r$ is equivalent to the combination of **ProjectD**, **SelectD**, and **ProductD**. Again, in the data space there is no direct equivalent operator to the relation difference $-_r$, we need to prove that the combination of **ProjectD**, **SelectD**, and **ProductD** is equivalent to $-_r$. If τ_1 and τ_2 are two relations that are compatible for the difference operation (all attributes are identical), we can find a third relation by taking the difference of the two. $\tau_3 = \tau_1 -_r \tau_2$. Suppose that $Sch(\tau_1) = Sch(\tau_2) = \{E_1.A_1, E_2.A_2, \ldots, E_m.A_m\}$. We first create a relation that is the Cartesian product of all the attribute domains of τ_1 and τ_2.

τ_{31} := **Extract** {E1.A1<-E1.A1} (E_1),
τ_{32} := **Extract** {E2.A2<-E2.A2} (E_2),
...
τ_{3m} := **Extract** {Em.Am<-Em.Am} (E_m).

And τ_3' := **ProductD**(τ_{31},
ProductD(τ_{32},
.....,

ProductD($\tau_{3,m-1}$ $\tau_{3,m}$) . . .)) .

Then we take the Cartesian product of τ_3' and τ_2 to make τ_3''. τ_3'':= **ProductD** (τ_3',τ_2). Then, we apply **SelectD** and **ProjectD** on τ_3'' to obtain τ_3'''.

τ_3''':= **ProjectD**$_{\{E1.A1,...,Em.Am\}}$ (

SelectD $_{\neg(E1.A1=E21.A21,...,Em.Am=E2m.A2m)}$ (τ_3'')).

And τ_3'''':= **ProductD** (τ_3''', τ_1). Finally,

τ_3:= **ProjectD** $_{\{E1.A1,...,Em.Am\}}$ (

SelectD $_{E1.A1=E11.A11,...,Em.Am=E1m.A1m}$ (τ_3'''')).

Based on the definitions of **ProductD**, **SelectD**, **ProjectD**, we can see that τ_3 obtained is equal to τ_1 -$_r$ τ_2.

The reason that we did not include the union and difference operations in the data space is because such operations are done at the object level in the entity space instead of at the data element level in the data space. It is more effective, we believe, to perform such operations at the object level to reduce the computation space.

We have successfully proved that for each of the five relational algebra operators we can find an equivalent ADL expression to obtain the same result. Only a very small subset of operators in ADL is used to prove the relation completeness. Therefore, relation relational completeness can be achieved by a subset of all the operators in ADL.

APPENDIX C. APPLICATIVE DATA LANGUAGE (ADL) FOR QUERY FORMULATION.

Examples are adapted from [Subi86].

Example C. 1: (Figure C.1)
Query: Give Name and Job for employees making less than Smith.

ADL:
ς_1 := (*Employee*) Employee.name='Smith',
ς_2 := *Employee*,
τ_1 := **Extract** {Smith-salary <- Employee.salary} (ς_1),
τ_2 := **Extract** {Emp-name <-Employee.name,
Emp-job <-Employee.job,
Emp-salary <-Employee.salary} (ς_2),
τ_3 := **ProductD** (τ_2, τ_1),
τ_4 := **ProjectD** {Emp-name, Emp-job}
(**SelectD** Emp-salary < Smith-salary (τ_3)),
Display (τ_4).

Note in the following example how we can avoid the Cartesian Product operator in semantic constraint modeling by using aggregate values.

Example C. 2: (Figure C.1)
Constraint: Smith's salary is less than all the employee.

ADL:
ς_1 := (*Employee*) Employee.name='Smith',
ς_2 := *Employee* - ς_1,
τ_1 := **Extract** {Smith-salary<- Employee.salary} (ς_1),
τ_2 := **Extract** {Emp-salary <- Employee.salary} (ς_2),
μ_1 := **ValueD** (τ_1),
μ_2 := **MinD** (τ_2),
β := $PV_<$ (μ_1, μ_2),
Display (β).

{ $PV_<(\mu_1,\mu_2)$ is a value predicate [Lee 87b]. It is TRUE if $\mu_1 < \mu_2$; FALSE, otherwise. }

Here, we only use the minimum salary of all the employee except Smith to avoid the expensive Cartesian Product operator. If the constraint is stated as "greater than", we would have to attain the maximum salary for the comparison.

Example C. 3: (Figure C.1)
Query: Give the average salary.

ADL:
ς_1 := *Employee*,
τ_1 := **Extract** {Salary<-Employee.salary} (ς_1),
μ_1 := **AvgD** (τ_1),

Figure C.1 A Department, Employee Database

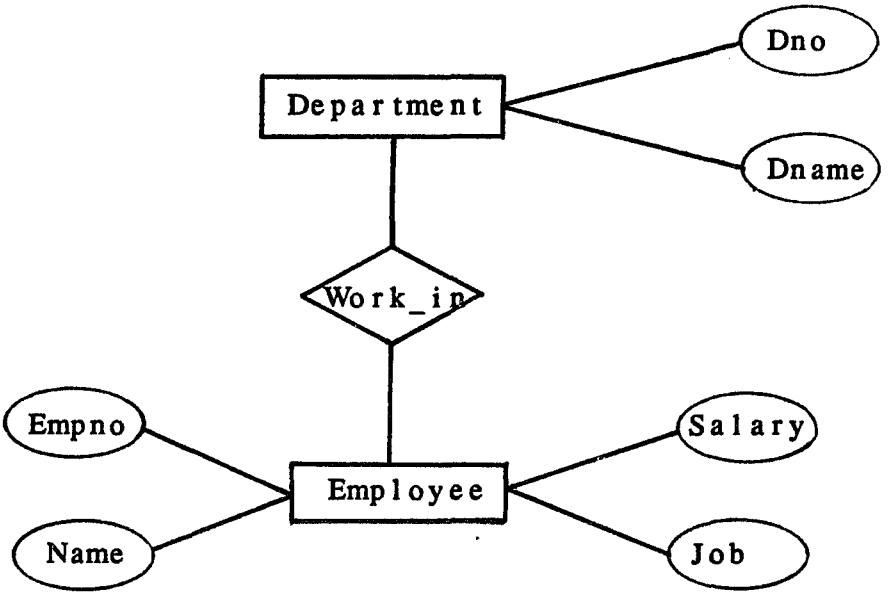
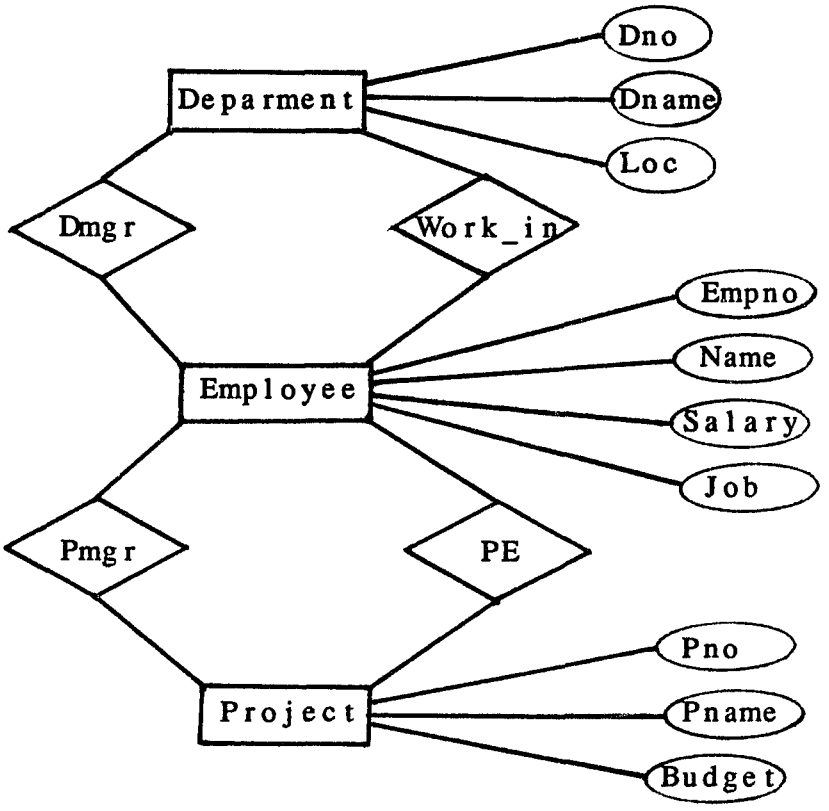


Figure C.2 A Department, Employee, Project Database



Display (μ_1).

Example C. 4: (Figure C.1)
Query: Give the name of the department and the average salary of the departments which have more than 50 employees.

ADL: ς_1 := *Department* ° *Works-in* ° *Employee*,
ς_2 := **Group_byS** {Department} (ς_1),
ς_3 := **SelectS** |Employee^c|>50 (ς_2),
τ_1 := **Extract** {Dept-name<-Department.dname,
Salary*<-Employee*.sal } (ς_3),
τ_2 := **ExtendD** {Avg-salary := **AvgB** (Salary^c)} (τ_1),
τ_3 := **ProjectD** {Dept-name,Avg-salary} (τ_2),
Display(τ_3).

Example C. 5: (Figure C.1)
Query: Give the average number of employee with respect to the number of department.

ADL: μ_1 := |*Employee*|,
μ_2 := |*Department*|,
μ_3 := μ_1/μ_2,
Display(μ_3).

Example C. 6: (Figure C.1)
Query: Give Empno, Name, Sal, Job, Dno, Dname for all employee.

ADL: ς_1 := *Department* °*Work_in* °*Employee*,
ς_2 := *Department* °*Dmgr* °*Employee*,
ς_3 := $\varsigma_1 \cup \varsigma_2$,
τ := **Extract** {Empno<-Employee.empno,
Name<-Employee.name,
Sal<-Employee.sal,
Job<-Employee.job,
Dno<-Employee.dno,
Dname<-Employee.dname} (ς_3),
Display (τ).

Example C. 7: (Figure C.1)
Query: For each department , give the maximal salary.

ADL: ς_1 := *Department* ° *Works-in* ° *Employee*,
ς_2 := **Group_byS** {Department} (ς_1),
τ_1 := **Extract** {Dept-name<-Department.dname,
Salary* <- Employee*.sal } (ς_2),

τ_2 := **ProjectD** {Dept-name, MaxSal} (
ExtendD {MaxSal := **MaxB**(Salary^c)} (τ_1)),
Display(τ_2).

Example C. 8: (Figure C.1)
Query: Give departments where some employees make less than 10,000.
ADL: ς_1 := *Department* ° *Works-in* ° *(Employee)*$_{\text{Employee.sal} < 10{,}000'}$,
ς_2 := **ProjectS** {Department} (ς_1),
τ_1 := **Extract** {Dept-name <- Department.dname} (ς_2),
Display (τ_1).

Example C. 9: (Figure C.1)
Constraint: Is it true that each department employs a programmer making more than 30,000 ?
ADL: ς_1 := *Department* ° *Works-in* ° *(Employee)*$_{\text{Employee.job='Progmr'} \wedge \text{Employee.sal} > 30{,}000'}$,
ς_2 := **ProjectS** {Department} (ς_1),
β := **PS**$_{=}$ (ς_2, Department),
Display (β).
{ **PS**$_{=}$(ς_1,ς_2) is a state predicate [Lee87b]. It is TRUE if ς_1 and ς_2 are equal; FALSE, otherwise. }

Example C.10: (Figure C.1)
Query: Give the average difference between a manager salary and the average salary of his employee in each department.
ADL: *Employee*$_1$:= *Employee*$_2$:= *Employee*,
ς_1 := *Department* ° *Works-in* ° *Employee*$_1$,
ς_2 := *Department* ° *Dmgr* ° *Employee*$_2$,
ς_{11} := **Group_byS** {Department} (ς_1),
ς_3 := **JoinS** {Department} (ς_{11}, ς_2),
τ_1 := **Extract** {Magr-salary <- Employee2.sal,
Emp-salary* <- Employee1*.sal} (ς_3),
τ_2 := **ExtendD** {Avg_Emp_Sal := **AvgB**(Emp-salary^c)} (τ_1),
τ_3 := **ProjectD** {Diff}
(**ExtendD** {Diff := Magr-salary - Avg_Emp_Sal} (τ_2)),
μ := **AvgD** (τ_3),
Display(μ).

Example C.11: (Figure C.2)
Query: Find the projects where the total salary is greater than the total budget of

the project.

ADL: ς_1 := *(Project ° PE ° Employee)* ∪ (Project ° Pmgr ° Employee),
ς_2 := **Group_byS** {Project} (ς_1),
τ_1 := **Extract** {Pname<-Project.pname,
Proj-budget<- Project.budget,
Emp-salary* <- Employee*.salary} (ς_2),
τ_2 := **ExtendD** {Total-sal := **SumB**(Emp-salary^c)} (τ_1),
τ_3 := **ProjectD** {Pname}
(**SelectD** Proj-budget < Total-sal (τ_2)),
Display(τ_3).

Example C.12: (Figure C.2)
Query: Give names of managers of departments with employees working for projects managed by Brown.
ADL: *Employee*$_1$:= *Employee*$_2$:= *Employee*$_3$:= *Employee*,
ς_1 := (*Employee*$_1$)Employee.name='Brown' °
Pmgr ° *Project* ° *PE* ° *Employee*$_2$ ° *Works-in* °
Department ° *Dmgr* ° *Employee*$_3$,
ς_2 := **ProjectS** {Employee3} (ς_1),
τ_1 := **Extract** {Mngr<-Employee3.name} (ς_2),
Display(τ_1).

Example C.13: (Figure C.2)
Query: Give employees working on all projects.
ADL: ς_1 := *Employee ° PE ° Project*,
ς_2 := **Group_byS** {Employee} (ς_1),
ς_3 := **Group_byS** * (Project),
ς_4 := **ProjectS** {Employee} (ς_3 ° ς_2),
τ_1 := **Extract** {Emp-name<-Employee.name} (ς_4),
Display(τ_1).

APPENDIX D. APPLICATIVE DATA LANGUAGE (ADL) FOR CONSTRAINT MODELING

Example D.1.1-D.1.14 are from [Tabo83a],[Tabo83b].
Example D.2.1-D.2.5 are from [Morg84],[Morg86].
Example D.3.1-D.3.5 are from [Hamm76].
Example D.4.1-D.4.7 are from [Nico82].
Example D.5.1-D.5.5 are from [Ston75].

Macro **GS** $_{\text{command1, command2}}$ (Operand) = **ProjectS** $_{\text{command1}}$ (**SelectS** $_{\text{command2}}$ (**Group_byS** $_{\text{command1}}$ (Operand))).

Example D.1.1 (Figure D.2. ERD for Relative Database).
Semantic: If a person x is child of a person y who is brother or sister of a person z, who is a parent of a person t, we expect that x, t are cousin.

$_{p}$: means the person plays the role of the parent,
$_{c}$: means the person plays the role of the child.

ADL: $Person1_{c} := Persone2_{p} := Person3_{p} := Person4_{c} := Person,$
$Person5 := Person6 := Person,$
$\varsigma_1 := Person1_{c} \circ Parent \circ Person2_{p} \circ Sibling \circ Person3_{p} \circ Parent \circ Person4_{c},$
$\varsigma_2 := Person5 \circ Cousin \circ Person6,$
$\beta := \mathbf{PS}_{\subseteq}(\mathbf{ProjectS}_{\{Person1c, Person4c\}}(\varsigma_1), \varsigma_2),$
Display (β).

Example D.1.2 (Figure D.1. ERD for the Car Registration Database).
Semantic: Any ownership is related to no more than one kind of owner (person, garage, or manufacture).

O:ownership, P:person, G:garage, M:manufacturer.

ADL: $\varsigma_1 := O \circ O\_is\_p \circ P,$ $\varsigma_{11} := \mathbf{ProjectS}_{\{O\}}(\varsigma_1),$
$\varsigma_2 := O \circ O\_is\_g \circ G,$ $\varsigma_{22} := \mathbf{ProjectS}_{\{O\}}(\varsigma_2),$
$\varsigma_3 := O \circ O\_is\_m \circ M,$ $\varsigma_{33} := \mathbf{ProjectS}_{\{O\}}(\varsigma_3),$
$\beta := \mathbf{PS}_{=}((\varsigma_{11} \cap \varsigma_{22}), \emptyset) \wedge \mathbf{PS}_{=}((\varsigma_{11} \cap \varsigma_{33}), \emptyset) \wedge \mathbf{PS}_{=}((\varsigma_{22} \cap \varsigma_{33}), \emptyset),$
Display (β).

Example D.1.3 (Figure D.1).
Semantics: A manufacturer may not be a transferee.

$O1_{er}$: transferer, $O2_{ee}$: transferee.

ADL: $O1_{er} := O2_{ee} := O,$
$\varsigma_1 := O1_{er} \circ Transfer \circ O2_{ee} \circ O\_is\_m \circ M,$

Figure D.1. A Car Registration Database

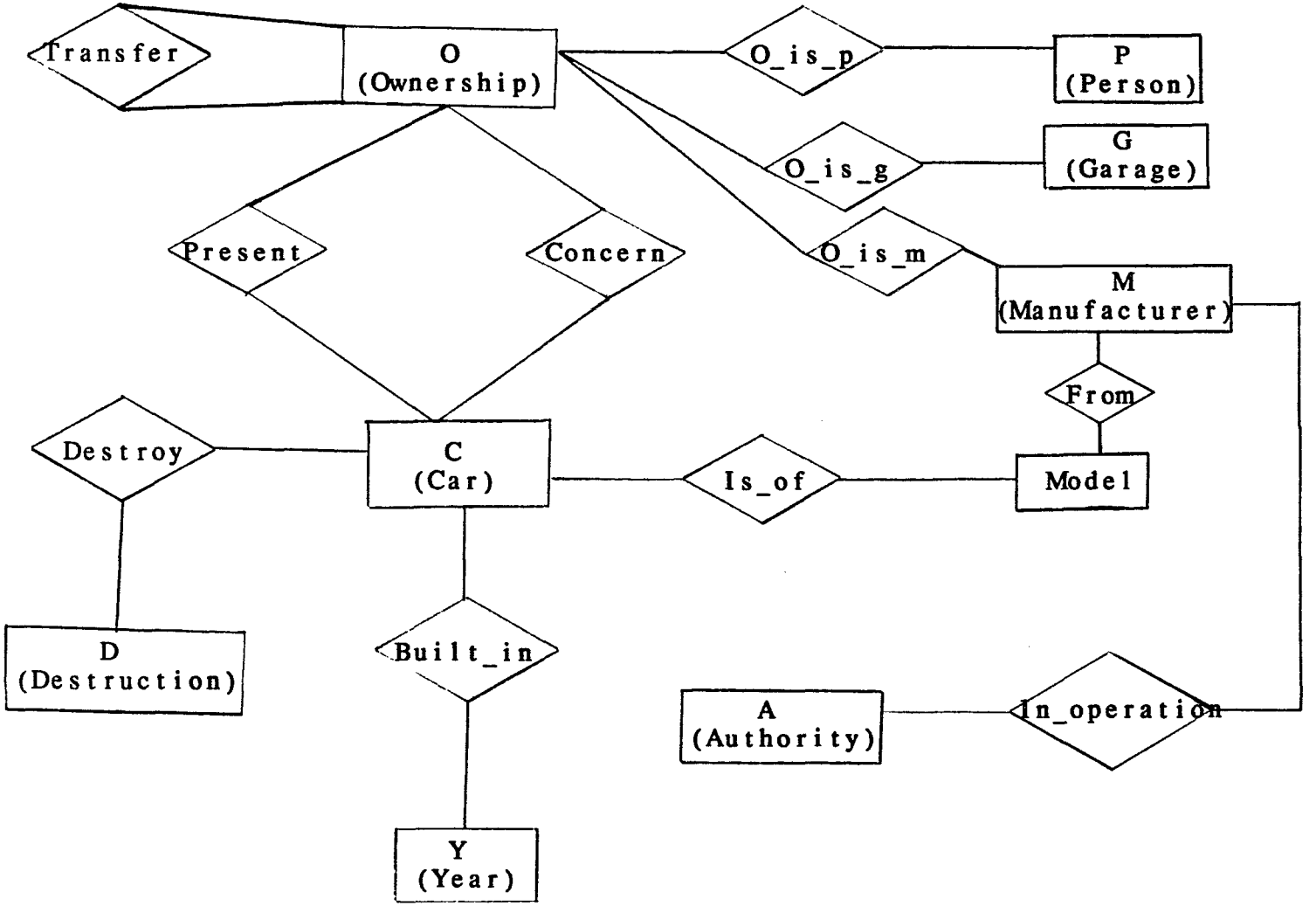


Figure D.2. A Relative Database

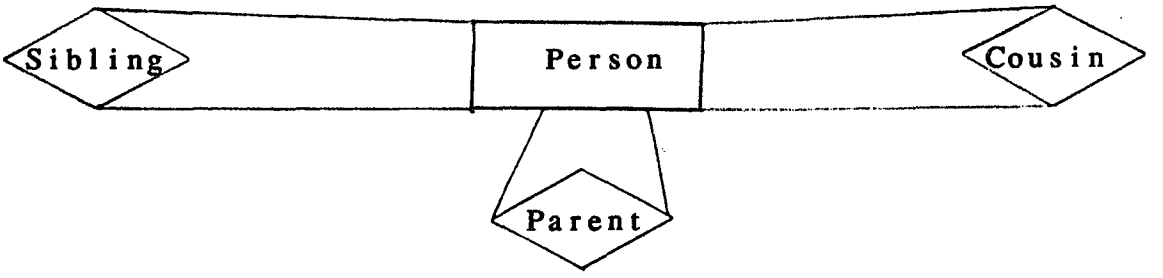


Figure D.3. An Office Database

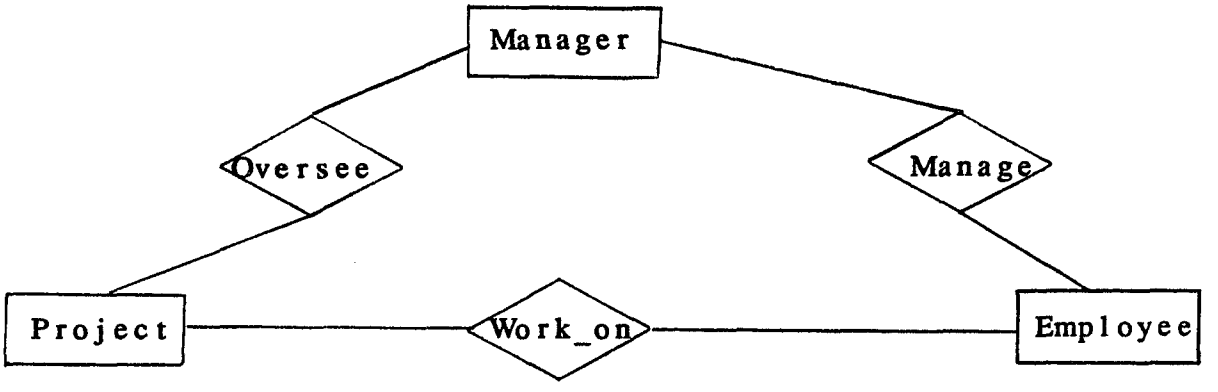


Figure D.4. A Department Database

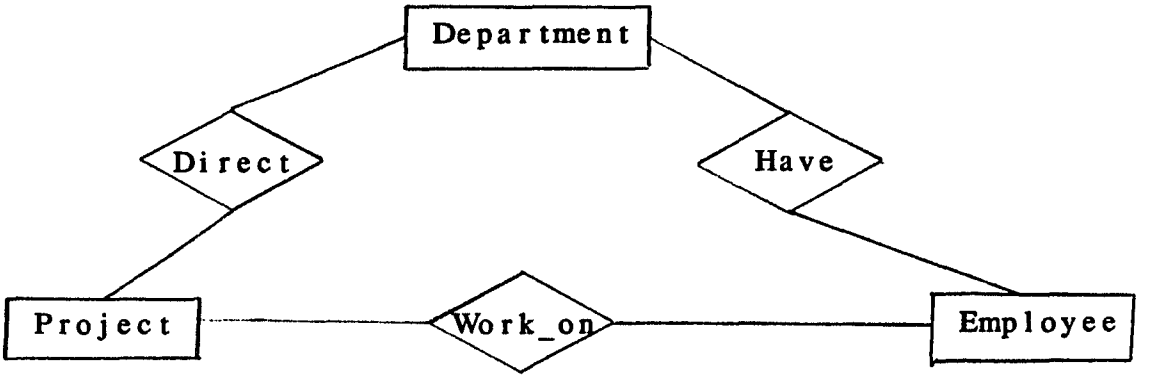


Figure D.5. A Company Database (1)

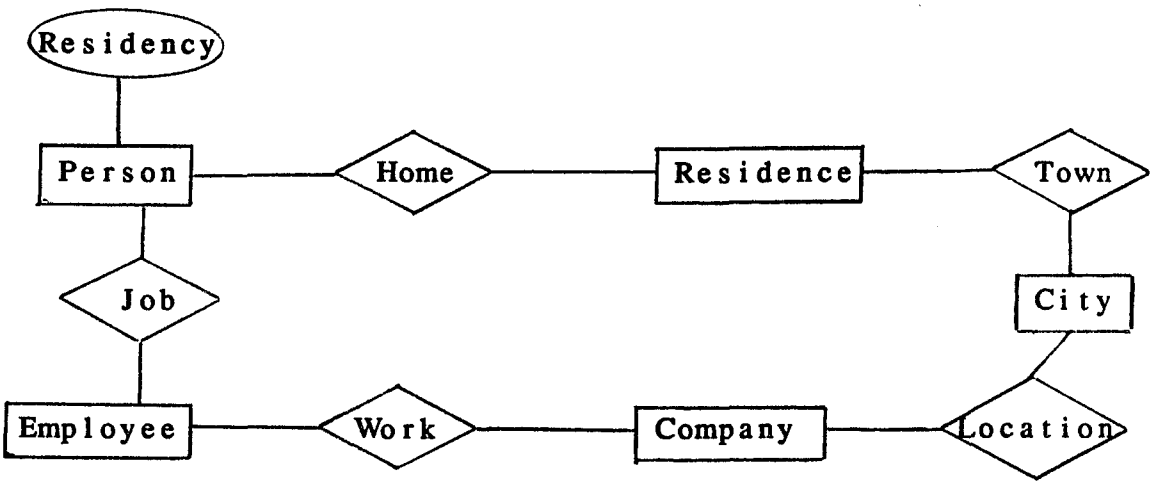


Figure D.6. A Department Store Database

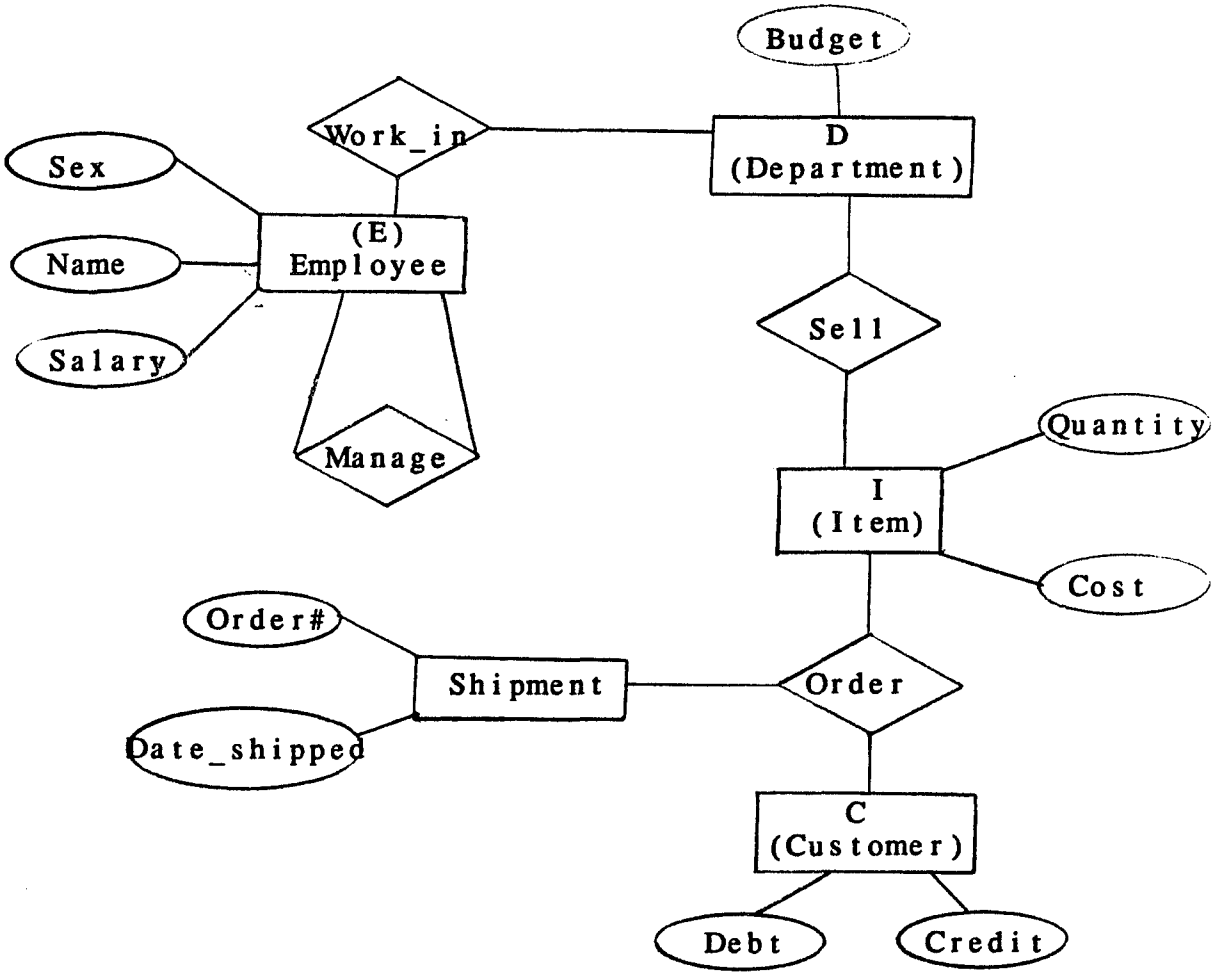


Figure D.7. A Company Database (2)

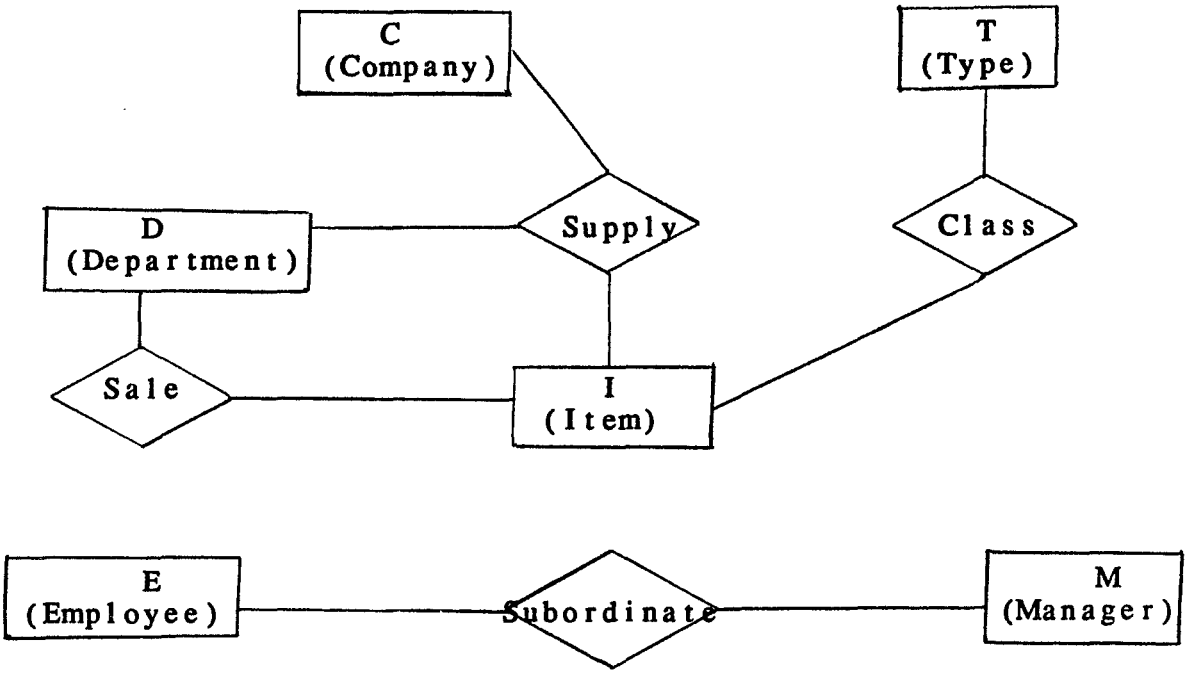
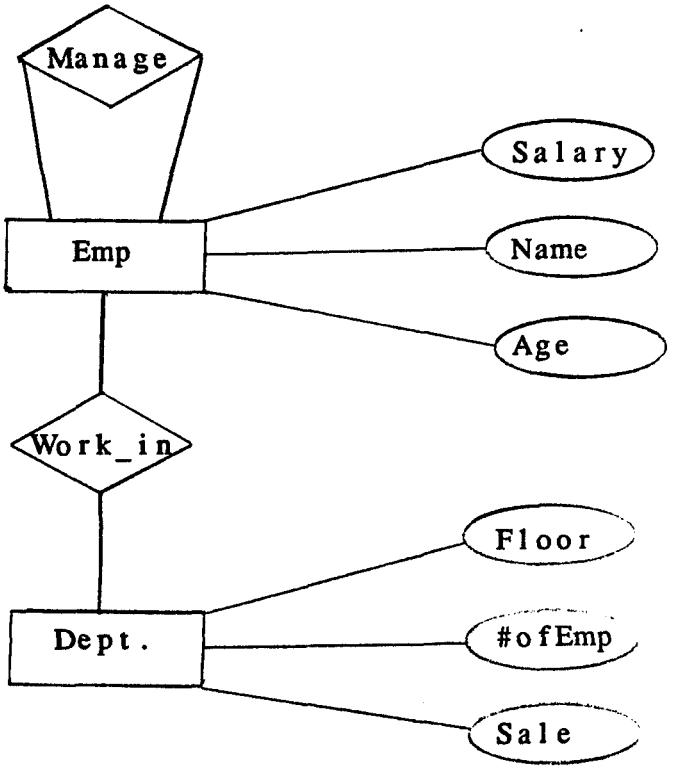


Figure D.8. A Department, Employee Database



β:= $\mathbf{PS}_{=}(\varsigma_1, \emptyset)$,
Display (β).

Example D.1.4 (Figure D.1).
Semantics: Ownership may not be transferred from a garage to another one.
ADL: $O1_{er} := O2_{ee} := O$, $G1 := G2 := G$,
$\varsigma_1 := G1 \circ O\_is\_g \circ O1_{er} \circ Transfer \circ O2_{ee} \circ O\_is\_g \circ G2$,
β:= $\mathbf{PS}_{=}(\varsigma_1, \emptyset)$,
Display (β).

Example D.1.5 (Figure D.1).
Semantics: Ownership may not be transferred from the manufacturer to persons.
ADL: $O1_{er} := O2_{ee} := O$,
$\varsigma_1 := M \circ O\_is\_m \circ O1_{er} \circ Transfer \circ O2_{ee} \circ O\_is\_p \circ P$,
β:= $\mathbf{PS}_{=}(\varsigma_1, \emptyset)$,
Display (β).

Example D.1.6 (Figure D.1).
Semantics: A present ownership may not be known as the "before" ownership in a transfer.
ADL: $O1_{er} := O2_{ee} := O$,
ς_1:= Present $\circ$ O1$_{er}$ $\circ$ Transfer $\circ$ O2$_{ee}$,
β:= $\mathbf{PS}_{=}(\varsigma_1, \emptyset)$,
Display (β).

Example D.1.7 (Figure D.1).
Semantics: An ownership which is the present ownership of a car, must concern this specific car.
C: car.
ADL: $\varsigma_1 := O \circ Present \circ C$,
$\varsigma_2 := O \circ Concern \circ C$,
β:= $\mathbf{PS}_{\subseteq}(\varsigma_1, \varsigma_2)$,
Display (β).

Example D.1.8 (Figure D.1).
Semantics: Both ownerships invoked in a transfer must concern the same car.
ADL: $O1 := O2 := O$,
$\varsigma_1 := O1 \circ Transfer \circ O2 \circ Concern \circ C$,
ς_{11}:= **ProjectS** $_{\{O1, C\}}$ (ς_1),
$\varsigma_2 := O \circ Concern \circ C$,
β:= $\mathbf{PS}_{\subseteq}(\varsigma_{11}, \varsigma_2)$,

Display (β).

Example D.1.9 (Figure D.1).
Semantics: If a manufacture is owning cars it must be in operation.

A: authority

ADL: ς_1:=$C^{\circ}Present^{\circ}O^{\circ}O\_is\_m^{\circ}M$, ς_{11}:= **ProjectS** $_{\{M\}}$ (ς_1),
ς_2:=$M^{\circ}In\_Op^{\circ}A$, ς_{22}:=**ProjectS** $_{\{M\}}$ (ς_{22}),
β:= **PS**$_{\subseteq}$(ς_{11}, ς_{22}),
Display (β).

Example D.1.10 (Figure D.1).
Semantics: A garage may not presently own cars built by more than 3 manufacturers.

ADL: ς_1:=$G^{\circ}O\_is\_g^{\circ}O^{\circ}Concern^{\circ}C^{\circ}is\_of^{\circ}M$,
ς_{11}:= **ProjectS** $_{\{G,M\}}$ (ς_1),
ς_{111}:= **GS** $_{\{G\}, |M^c| \geq 4}$ (ς_{11}),
β:= **PS**$_{=}$(ς_{111}, $\emptyset$),
Display (β).

Example D.1.11 (Figure D.1).
Semantics: Any ownership must invoke some person, garage, or manufacturer.

ADL: ς_1:=$O^{\circ}O\_is\_p^{\circ}P$, ς_{11}:= **ProjectS** $_{\{O\}}$ (ς_1),
ς_2:=$O^{\circ}O\_is\_m^{\circ}M$, ς_{22}:= **ProjectS** $_{\{O\}}$ (ς_2),
ς_3:=$O^{\circ}O\_is\_g^{\circ}G$, ς_{33}:= **ProjectS** $_{\{O\}}$ (ς_3),
ς_4:=O,
β:= **PS**$_{=}$(($\varsigma_{11} \cup \varsigma_{22} \cup \varsigma_{33}$), ς_4),
Display (β).

Example D.1.12 (Figure D.1).
Semantics: Any car must be presently owned or have been destroyed, not both.

ADL: ς_1:=$C^{\circ}Present^{\circ}O$, ς_{11}:= **ProjectS** $_{\{C\}}$ (ς_1),
ς_2:=$C^{\circ}Destroy^{\circ}D$, ς_{22}:= **ProjectS** $_{\{C\}}$ (ς_2),
ς_3:=C,
β:= **PS**$_{=}$(($\varsigma_{11} \cup \varsigma_{22}$), ς_3) $\wedge$ **PS**$_{=}$(($\varsigma_{11} \cap \varsigma_{22}$), $\emptyset$),
Display (β).

Example D.1.13 (Figure D.1).
Semantics: Any ownership of a car must be either related to a manufacturer, or due to a transfer, not both.

$O1_{er}$: transferer, $O2_{ee}$: transferee.

ADL: $O1_{er}$:=$O2_{ee}$:=O,

ς_1 := O ° O_is_M ° M, ς_{11} := **ProjectS** $_{\{O\}}$ (ς_1),
ς_2 := $O1_{er}$ ° $Transfer$ ° $O2_{ee}$, ς_{22} := **ProjectS** $_{\{O2ee\}}$ (ς_2),
ς_3 := O,
β := **PS**$_{=}$(($\varsigma_{11} \cap \varsigma_{22}$), $\emptyset$) $\wedge$ **PS**$_{=}$(($\varsigma_{11} \cup \varsigma_{22}$), ς_3),
Display (β).

Example D.1.14 (Figure D.1).
Semantics: If a car is owned, or has been owned, by a manufacturer, it must be one of his own models.

ADL: ς_1 := C ° $Concern$ ° O ° O_is_M ° M, ς_{11} := **ProjectS** $_{\{C,M\}}$ (ς_1),
ς_2 := C ° is_of ° $Model$ ° $From$ ° M, ς_{22} := **ProjectS** $_{\{C,M\}}$ (ς_2),
β := **PS**$_{=}$(ς_{11}, ς_{22}),
Display (β).

Example D.2.1 (Figure D.3. ERD for Office Database).
Semantics: A manager oversees those projects which his employee works.

ADL: ς_1 := M ° $Manage$ ° E ° $Work\_on$ ° P, ς_{11} := **ProjectS** $_{\{M,P\}}$ (ς_1),
ς_2 := M ° $Oversee$ ° P,
β := **PS**$_{=}$(ς_2, ς_{11}),
Display (β).

Example D.2.2 (Figure D.4. ERD for Department Database).
Semantics: The projects directed by a department are those worked on by all the employees in the department.

ADL: ς_1 := D ° $Have$ ° E,
ς_{11} := **Group_byS** $_{\{D\}}$ (ς_1),
ς_2 := D ° $Have$ ° E ° $Work\_On$ ° P,
ς_{22} := **Group_byS** $_{\{D,P\}}$ (ς_2),
ς_3 := **ProjectS** $_{\{D,P\}}$ (**JoinS** $_{\{D,E^*\}}$ (ς_{11}, ς_{22})),
ς_4 := D ° $Direct$ ° P,
β := **PS**$_{=}$(ς_3, ς_4),
Display (β).

Example D.2.3 (Figure D.4).
Semantics: Department direct those projects that have at least n, at most m employees work on.

ADL: ς_1 := D ° $Have$ ° E ° $Work\_on$ ° P,
ς_{11} := **ProjectS** $_{\{D,P\}}$
(**SelectS** $_{n \leq |E^c| \leq m}$ (**Group_byS** $_{\{D,P\}}$ (ς_1))),
ς_2 := D ° $Direct$ ° P,
β := **PS**$_{=}$(ς_{11}, ς_2),

Display (β).

Example D.2.4 (Figure D.5. ERD for Company Database).
Semantics: A person who is a local resident must live in a town where the company he works for is located. (One person only works for one company.)
ADL: $Ci1 := Ci2 := Ci$,

$\varsigma_1 := (P)_{\text{P.Residency='local'}}$,
$\varsigma_2 := \mathbf{SelectS}_{\text{ci2=ci1}}\ (Ci1\,^{\circ}Town\,^{\circ}R\,^{\circ}Home\,^{\circ}P\,^{\circ}Job\,^{\circ}E\,^{\circ}Work\,^{\circ}CO\,^{\circ}Locate\,^{\circ}Ci2)$,
$\varsigma_{22} := \mathbf{ProjectS}_{\{P\}}\ (\varsigma_2)$,
$\beta := \mathbf{PS}_{=}(\varsigma_1, \varsigma_{22})$,
Display (β).

Example D.2.5 (Figure D.5).
Semantics: The locations of all the companies for which a person works for must be in the same city in order for the person to be qualified as a local resident.
ADL: $Ci1 := Ci2 := Ci$,

$\varsigma_1 := (P)_{\text{Residency='local'}}$,
$\varsigma_2 := \mathbf{SelectS}_{\text{Ci2=Ci1}}\ (Ci1\,^{\circ}Town\,^{\circ}R\,^{\circ}Home\,^{\circ}P\,^{\circ}Job\,^{\circ}E\,^{\circ}Work\,^{\circ}CO\,^{\circ}Locate\,^{\circ}Ci2)$,
$\varsigma_{22} := \mathbf{ProjectS}_{\{P,CO\}}\ (\varsigma_2)$,
$\varsigma_{222} := \mathbf{Group\_byS}_{\{P\}}\ (\varsigma_{22})$,
$\varsigma_3 := P\,^{\circ}Job\,^{\circ}E\,^{\circ}Work\,^{\circ}CO$,
$\varsigma_{33} := \mathbf{Group\_byS}_{\{P\}}\ (\mathbf{ProjectS}_{\{P,CO\}}\ (\varsigma_3))$,
$\varsigma_4 := \mathbf{ProjectS}_{\{P\}}\ (\mathbf{JoinS}_{\{P,CO^*\}}\ (\varsigma_{33}, \varsigma_{222}))$,
$\beta := \mathbf{PS}_{=}(\varsigma_4, \varsigma_1)$,
Display (β).

Example D.3.1 (Figure D.6. ERD for Department Store Database).
Semantics: The salary of each employee in the sales department is less than that of his manager.
ADL: $E1 := E2 := E$,

$\varsigma_1 := E1\,^{\circ}Manage\,^{\circ}E2\,^{\circ}Work\_in\,^{\circ}(D)_{\text{D.name='sales'}}$,
$\tau_1 := \mathbf{Extract}\ \{\text{Man\_Sal<-E1.salary}, \text{Emp\_Sal<-E2.salary}\}\ (\varsigma_1)$,
$\beta := \mathbf{PD}_{>}\ (\tau_1)$,
Display (β).

Example D.3.2 (Figure D.6).
Semantics: The salary of every employee is less than $50,000.
ADL: $\varsigma_1 := E$,

τ_1 := **Extract** {Emp_sal<- E.salary} (ς_1),
τ_2 := **ExtendD** {Limit:=50000} (τ_1),
β:= **PD**$_<$ (τ_2),
Display (β).

Example D.3.3 (Figure D.6).
Semantics: The average salary of all employee salary is at least equal to the salary of Robert Jones.
ADL: ς_1 := E, τ_1 := **Extract** {Emp_Sal<-E.salary} (ς_1),
ς_2 := (E) E.name='Robert Jones',
τ_2 := **Extract** {J_Sal<- E.salary} (ς_2),
Avg_sal := **AvgD**(τ_1), Jones_Sal := **ValueD**(τ_2),
β:= **PV**$_>$(Avg_sal, Jones_Sal),
Display (β).

Example D.3.4 (Figure D.6).
Semantics: Each department has at most two employees with a salary of more than $50,000.
ADL: ς_1 := **GS** $^{\{D\},\ |E^c|>2}$ (**ProjectS** $^{\{E,D\}}$ ((E) $_{salary>50000}$ °Work_In°D))),
β:= **PS**$_=$(ς_1, $\emptyset$),
Display (β).

Example D.3.5 (Figure D.6).
Semantics: The number of female employees is at least 40% the total number of employees.
ADL: ς_1 :=E,
ς_2 :=(E) E.sex='female',
β:= **PV**$_\leq$((0.40 * $|\varsigma_1|$), $|\varsigma_2|$),
Display (β).

Example D.4.1 (Figure D.7. ERD for company A database).
Semantics: When a department sells an item then there is a company which supplies it with this item.
ADL: ς_1 :=D °$Supply$ °I,
ς_2 :=D °$Sale$ °I,
β:= **PS**$_=$(ς_1, ς_2),
Display (β).

Example D.4.2 (Figure D.7).

Semantics: No other companies than company C supplies type T4 items.

ADL: $\varsigma_1 := C \circ Supply \circ I \circ Class \circ Type,$
$\varsigma_{11} :=_{\mathbf{GS}} \{C\}, \{<T4>\} \subseteq I^c (\varsigma_1),$
$\beta := \mathbf{PS}_{=}(\varsigma_{11}, \{<C>\}),$
Display (β).

Example D.4.3 (Figure D.7).

Semantics: Any company that supplies guns also supplies bullets.

ADL: $\varsigma_1 := C \circ Supply \circ (I)_{I='gun'},$
$\varsigma_{11} :=_{\mathbf{ProjectS}_{\{C\}}} (\varsigma_1),$
$\varsigma_2 := C \circ Supply \circ (I)_{I='bullet'},$
$\varsigma_{22} :=_{\mathbf{ProjectS}_{\{C\}}} (\varsigma_2),$
$\beta := \mathbf{PS}_{\subseteq}(\varsigma_{11}, \varsigma_{22}),$
Display (β).

Example: 4.4 (Figure D.7).

Semantics: Any company that supplies type T1 items also supplies type T2 items.

ADL: $\varsigma_1 := C \circ Supply \circ I \circ Class \circ T,\ \varsigma_{11} :=_{\mathbf{ProjectS}_{\{C,T\}}} (\varsigma_1),$
$\varsigma_{111} :=_{\mathbf{GS}} \{C\}, \{<T1>\} \subseteq \mathrm{TYPE}^c (\varsigma_{11}),$
$\varsigma_{112} :=_{\mathbf{GS}} \{C\}, \{<T2>\} \subseteq \mathrm{TYPE}^c (\varsigma_{11}),$
$\beta := \mathbf{PS}_{\subseteq}(\varsigma_{111}, \varsigma_{112}),$
Display (β).

Example: 4.5 (Figure D.7).

Semantics: No company must supply two different departments with item I.

ADL: $\varsigma_1 := (C \circ Supply \circ (I)_{I='I'}) \circ D$
$\varsigma_{11} := \mathbf{ProjectS}_{\{C,D\}} (\varsigma_1),$
$\varsigma_{111} := \mathbf{GS}_{\{C\}, |D^c|=2} (\varsigma_{11}),$
$\beta := \mathbf{PS}_{=}(\varsigma_{111}, \emptyset),$
Display (β).

Example: 4.6 (Figure D.7).

Semantics: Whenever an employee is a subordinate of another employee who himself is a subordinate of a third one, then the first one is a subordinate of the third one.

*This one requires managers and employees be treated as one entity type; but play different roles.

ADL: $E1 := E2 := E3 := E,$
$\varsigma_1 := E1 \circ Subord \circ E2 \circ Subord \circ E3,$
$\varsigma_{11} := \mathbf{ProjectS}_{\{E1,E3\}} (\varsigma_1),$
$\varsigma_2 := E4 \circ Subord \circ E5,$

β:= $\mathbf{PS}_{\subseteq}(\varsigma_{11}, \varsigma_2)$,
Display (β).

Example D.4.7 (Figure D.7).
Semantics: There is at least one type T3 item which is supplied by every company.
ADL: ς_1:= $C^{\circ}Supply^{\circ}I^{\circ}Class^{\circ}(T)_{T='T3'}$,
ς_{11}:= **ProjectS** $_{\{C,I\}}$ (ς_1),
ς_{111}:= **Group_byS** $_{\{I\}}$ (ς_{11}),
ς_2:= **Group_byS** $_{*}$ (C),
ς_3:= **JoinS** $_{\{C^*\}}$ (ς_{111}, ς_2),
β:= **CountS**(ς_3) $\geq$ 1,
Display (β).

Example D.5.1 (Figure D.8. ERD for Department A)
Semantics: Employee salaries must be positive,
ADL: τ_1:= **Extract** $_{\{Emp\_sal<-Emp.salary\}}$ (E),
τ_{11}:= **ExtendD** $_{\{Limit:=0\}}$ (τ_1),
β:= $\mathbf{PD}_{>}$ (τ_1),
Display (β).
(Note: This constraint is a domain constraint which is best modeled at the data definition level).

Example D.5.2 (Figure D.8)
Semantics: Everyone in the toy department must make more than $8000
ADL: ς_1:= $Emp^{\circ}Work\_in^{\circ}(Dept)_{Dept.name='toy'}$,
τ_1:= **Extract** $_{\{Emp\_Sal<-\ Emp.salary\}}$ (ς_1),
τ_{11}:= **ExtendD** $_{\{Limit:=8000\}}$ (τ_1),
β:= $\mathbf{PD}_{>}$ (τ_{11}),
Display (β).

Example D.5.3 (Figure D.8)
Semantics: Employee must earn less than ten times the sales volume of their department if their department has a positive sales.
ADL: ς_1:= $Emp^{\circ}Work\_in^{\circ}(Dept)_{Dept.sale>0}$,
τ_1:= **Extract** $_{\{Emp\_Sal<-Emp.salary,\ Dept\_Sale<-Dept.sale\}}$ (ς_1),
τ_{11}:= **ProjectD** $_{\{Emp\_sal,Rate\}}$ (**Extract** $_{\{Rate:=10*Dept\_Sale\}}$ (τ_1)),
β:= $\mathbf{PD}_{<}$ (τ_{11}),
Display (β).

Example D.5.4 (Figure D.8)

Semantics: No employee can make more than his manager.

$Emp1_m$: manager, $Emp2_w$: worker.

ADL: $Emp1_m := Emp2_w := Emp$,

$\varsigma_1 := Emp1_m \,^{\circ}Manage\,^{\circ} Emp2_w$,

$\tau_1 :=$ **Extract** {Man_Sal<-Emp1.salary, Emp_Sal<-Emp2.salary} (ς_1),

$\beta :=$ **PD**$_{>}$ (τ_1),

Display (β).

Example D.5.5 (Figure D.8)

Semantics: Harding must make more than twice the average employee salary.

ADL: $\varsigma_1 := (E)_{\text{E.name='Harding'}}$,

$\tau_1 :=$ **Extract** {H_Sal<- Emp.salary} (ς_1),

$\varsigma_2 := E$, $\tau_2 :=$ **Extract** {Emp_Sal<- Emp.salary} (ς_2),

$\mu_1 :=$ **ValueD** (τ_1), $\mu_2 :=$ **AvgD** (τ_2),

$\beta :=$ **PV**$_{>}(\mu_1, (2.0 * \mu_2))$,

Display (β).

APPENDIX E. EQUIVALENT EXPRESSION IN ADL

How to reformulate an ADL specification depends on the following factors: 1. the physical implementation of the database, 2. the current database state, and 3. the system hardware and software configurations. However, if it is apparent in most cases that Exp1 is more expensive to evaluate than Exp2 in terms of space and time required, we express the desirable replacement with Exp1 -> Exp2 in the "Optimization Direction" entry. On the other hand, if more information is needed to judge which expression is more efficient, we express them as Exp1 ? Exp2. Some of the optimization rules are adapted from [Ullm83], [Gray84], [Maie84].

Many of the equivalent expressions have preconditions that must be met. These preconditions involve the schemes of predicates (used in **SelectS**, **SelectD**) and formulas (used in **ExtendD**). We define them as follows. #Definition E.1: **Sch(P), Sch(f).** If P is a predicate formula used in the **SelectS** operator, **Sch(P)** is the set of symbols of the involving entity set in the predicate formula. If P is a predicate formula used in the **SelectD** operator, **Sch(P)** is the set of symbols of attributes in the predicate formula. Assume f is a function in the **ExtendD** operator. **Sch(f)** is the set of attribute symbols of the formula in the operand d_relation. ###

#Example E.1: Sch(P), Sch(f).

ς_2:= **SelectS** $_{E1.A < 100 \wedge E4.B \geq 23}$ (ς_1).
Sch($(E_1.A < 100 \wedge E_4.B \geq 23)$) = { E_1, E_4}.
τ_2:= **SelectD** $_{A < 20 \wedge B \leq 44}$ (τ_1).
Sch($A < 20 \wedge B \leq 44$)) = { A, B}.
τ_2:= **ExtendD** $_{\{ANEW := A1 + A2\}}$ (τ_1).
Sch({ANEW:= $A_1 + A_2$}) = { A_1 , A_2 }. ###

A. EQUIVALENT EXPRESSION IN THE Entity space

Precondition- None.
Optimization Direction- Exp1 ? Exp2.
Rule 1: $\varsigma_1 \cap \varsigma_2 \equiv \varsigma_2 \cap \varsigma_1$.
Rule 2: $\varsigma_1 \cup \varsigma_2 \equiv \varsigma_2 \cup \varsigma_1$.
Rule 3: $\varsigma_1 \circ \varsigma_2 \equiv \varsigma_2 \circ \varsigma_1$.
Rule 4: $\varsigma_1 \cap (\varsigma_2 \cap \varsigma_3) \equiv (\varsigma_1 \cap \varsigma_2) \cap \varsigma_3$.
Rule 5: $\varsigma_1 \cup (\varsigma_2 \cup \varsigma_3) \equiv (\varsigma_1 \cup \varsigma_2) \cup \varsigma_3$.
Rule 6: $\varsigma_1 \circ (\varsigma_2 \circ \varsigma_3) \equiv (\varsigma_1 \circ \varsigma_2) \circ \varsigma_3$.

Precondition- None.
Optimization Direction- Exp1 -> Exp2.

Rule 7: **ProjectS** $_{P\_set1}$ (**ProjectS** $_{P\_set2}$ (ς)) $\equiv$ **ProjectS** $_{P\_set1 \cap P\_set2}$ (ς).

Precondition- None.
Optimization Direction- Exp1 -> Exp2.
Rule 8: **SelectS** $_{P1}$ (**SelectS** $_{P2}$ (ς)) $\equiv$ **SelectS** $_{P1 \wedge P2}$ (ς).

Precondition- Sch(P) $\subseteq$ P_set.
Optimization Direction- Exp1 ? Exp2.
Rule 9: **ProjectS** $_{P\_set}$ (**SelectS** $_{P}$ (ς)) $\equiv$ **SelectS** $_{P}$ (**ProjectS** $_{P\_set}$ (ς)).

Precondition- Sch(P) $\subseteq$ Sch(ς1).
Optimization Direction- Exp1 -> Exp2.
Rule 10: **SelectS** $_{P}$ (ς_1 $^\circ$ ς_2) $\equiv$ (**SelectS** $_{P}$ (ς_1)) $^\circ$ ς_2.

Precondition- Sch(P_1) $\subseteq$ Sch(ς_1), Sch(P_2) $\subseteq$ Sch(ς_2)EOptimization Direction- Exp1 -> Exp2.
Rule 11: **SelectS** $_{P1 \wedge P2}$ (ς_1 $^\circ$ ς_2) $\equiv$ **SelectS** $_{P1}$ (ς_1) $^\circ$ **SelectS** $_{P2}$ (ς_2).

Precondition- Sch(P_1) $\subseteq$ Sch(ς_1), Sch(P_2) $\subseteq$ Sch(ς_2),
Sch(P_3) $\subseteq$ Sch(ς_1) $\cup$ Sch(ς_2).
Optimization Direction- Exp1 -> Exp2.
Rule 12: **SelectS** $_{P1 \wedge P2 \wedge P3}$ (ς_1 $^\circ$ ς_2) $\equiv$ **SelectS** $_{P3}$ ((**SelectS** $_{P1}$ (ς_1)) $^\circ$ (**SelectS** $_{P2}$ (ς_2)))

Precondition- Sch(P) $\subseteq$ G_set.
Optimization Direction- Exp1 -> Exp2.
Rule 13: **SelectS** $_{P}$ (**Group_byS** $_{G\_set}$ (ς)) $\equiv$ **Group_byS** $_{G\_set}$ (**SelectS** $_{P}$ (ς)).

Precondition- None.
Optimization Direction- Exp1 -> Exp2.
Rule 14: **SelectS** $_{P}$ ($\varsigma_1 \cup \varsigma_2$) $\equiv$ ((**SelectS** $_{P}$ (ς_1)) $\cup$ (**SelectS** $_{P}$ (ς_2))).
Rule 15: **SelectS** $_{P}$ ($\varsigma_1 \cap \varsigma_2$) $\equiv$ ((**SelectS** $_{P}$ (ς_1)) $\cap$ (**SelectS** $_{P}$ (ς_2))).

Precondition- None.
Optimization Direction- Exp1 -> Exp2.
Rule 16: **SelectS** $_{P1}$ (ς) $\cup$ **SelectS** $_{P2}$ (ς) $\equiv$ **SelectS** $_{P1 \vee P2}$ (ς).
Rule 17: **SelectS** $_{P1}$ (ς) $\cap$ **SelectS** $_{P2}$ (ς) $\equiv$ **SelectS** $_{P1 \wedge P2}$ (ς).
Rule 18: **SelectS** $_{P1}$ (ς) $^\circ$ **SelectS** $_{P2}$ (ς) $\equiv$ **SelectS** $_{P1 \wedge P2}$ (ς).

Precondition- None.
Optimization Direction- Exp1 -> Exp2.
Rule 19: **ProjectS** $_{P\_set}$ ($\varsigma_1 \cup \varsigma_2$) $\equiv$ **ProjectS** $_{P\_set}$ (ς_1) $\cup$ **ProjectS** $_{P\_set}$ (ς_2).

Precondition- P_set1 = P_set $\cap$ Sch(ς_1),
P_set2 = P_set $\cap$ Sch(ς_2).
Optimization Direction- Exp1 -> Exp2.
Rule 20: **ProjectS** $_{P\_set}$ (ς_1 $\circ$ ς_2) $\equiv$ **ProjectS** $_{P\_set}$ ((**ProjectS** $_{P\_set1 \cup J\_set}$ (ς_1) $\circ$ (**ProjectS** $_{P\_set2 \cup J\_set}$ (ς_2))).

Precondition- If P, then Q.
Optimization Direction- Exp1 -> Exp2.
Rule 21: **SelectS** $_{P \wedge Q}$ (ς) $\equiv$ **SelectS** $_{P}$ (ς).
Rule 22: **SelectS** $_{P \vee Q}$ (ς) $\equiv$ **SelectS** $_{Q}$ (ς).

B. EQUIVALENT EXPRESSION IN THE DATA SPACE

Precondition- None.
Optimization Direction- Exp1 ? Exp2.
Rule 23: **ProductD** (ς_1, ς_2) $\equiv$ **ProductD** (ς_2, ς_1).
Rule 24: **ProductD** (ς_1, **ProductD** (ς_2, ς_3)) $\equiv$ **ProductD** (**ProductD** (ς_1, ς_2) , ς_3)).

Precondition- None.
Optimization Direction- Exp1 -> Exp2.
Rule 25: **ProjectD** $_{P\_set1}$ (**ProjectD** $_{P\_set2}$ (ς)) $\equiv$ **ProjectD** $_{Pset1 \cap Pset2}$ (ς).

Precondition- None.
Optimization Direction- Exp1 -> Exp2.
Rule 26: **SelectD** $_{P1}$ (**SelectD** $_{P2}$ (ς)) $\equiv$ **SelectD** $_{P1 \wedge P2}$ (ς).

Precondition- Sch(f_2) $\subseteq$ Sch(τ).
Optimization Direction- Exp1 -> Exp2.
Rule 27: **ExtendD** {ANEW2:= f2(A1,...,An)} (**ExtendD** {ANEW1:= f1(A1,...,Am)} (τ)) $\equiv$ **ExtendD** {ANEW1:=f1(A1,...,An),ANEW2:=f2(A1,...,Am)} (τ).

Precondition- Sch(P) $\subseteq$ Sch(ς_1).
Optimization Direction- Exp1 -> Exp2.
Rule 28: **SelectD** $_{P}$ (**ProductD** (ς_1, ς_2)) $\equiv$ **ProductD** ((**SelectD** $_{P}$ (ς_1)), ς_2).

Precondition- Sch(P_1) $\subseteq$ Sch(ς_1), Sch(P_2) $\subseteq$ Sch(ς_2).
Optimization Direction- Exp1 -> Exp2.
Rule 29: **SelectD** $_{P1 \wedge P2}$ (**ProductD** (ς_1 , ς_2)) $\equiv$ **ProductD** (**SelectD** $_{P1}$ (ς_1) , **SelectD** $_{P2}$ (ς_2)).

Precondition- Sch(P_1) $\subseteq$ Sch(ς_1), Sch(P_2) $\subseteq$ Sch(ς_2),

$Sch(P_3) \subseteq Sch(\varsigma_1) \cup Sch(\varsigma_2)$.

Optimization Direction- Exp_1 -> Exp_2.

Rule 30: **SelectD** $_{P1 \wedge P2 \wedge P3}$ (**ProductD** $(\varsigma_1 , \varsigma_2)) \equiv$ **SelectD** $_{P3}$(**ProductD** (**SelectD** $_{P1}(\varsigma_1)$,**SelectD** $_{P2}(\varsigma_2)$)).

Precondition- $Sch(P) \subseteq Sch(\tau)$.

Optimization Direction- Exp1 -> Exp2.

Rule 31: **SelectD** $_{P}$ (**ExtendD** $_{\{ANEW:=f(A1,...,An)\}}$ $(\tau)) \equiv$ **ExtendD** $_{\{ANEW:=f(A1,...,An)\}}$(**SelectD** $_{P}$ (τ)).

Precondition- $P\_set1 \subseteq Sch(\tau_1)$, $P\_set1 \cap Sch(\tau_2) = \emptyset$;
$P\_set2 \subseteq Sch(\tau_2)$, $P\_set2 \cap Sch(\tau_1) = \emptyset$.

Optimization Direction- Exp1 -> Exp2.

Rule 32: **ProjectD** $_{P\_set1 \cup P\_set2}$ (**ProductD** $(\tau_1, \tau_2)) \equiv$ **ProductD** (**ProjectD** $_{P\_set1}$ (τ_1), **ProjectD** $_{P\_set2}$ (τ_2)).

APPENDIX F. OPERATORS OF INCREMENTAL COMPUTATION

We discuss here how each incremental operator is computed in terms of the following primitive operators: Normal Entity space Functions [Lee87a]:

$\cup$, $\cap$, -, **JoinS**, **ProjectS**, **SelectS**, **Group_byS**,

CountS.

Normal Entity space Predicates [Lee87b]:

$\mathbf{PS}_{\subset}$, $\mathbf{PS}_{\supset}$, $\mathbf{PS}_{\subset}$, $\mathbf{PS}_{\supset}$, $\mathbf{PS}_{=}$.

Normal Data Space Functions [Lee87a]:

SumD, **CountD**, **AvgD**, **MaxD**, **MinD**.

Normal Data Space Predicates [Lee87a]:

$\mathbf{PD}_{\geq}$, $\mathbf{PD}_{\leq}$, $\mathbf{PD}_{>}$, $\mathbf{PD}_{<}$, $\mathbf{PD}_{=}$.

Operators for Signed Data Structures:

α, **Trace_bk**, **Sign**, **Assign_plus**, **Assign_minus**,

Unsign, **Plus_part**, **Minus_part**, **Sign_change**.

Therefore, incremental operators defined hereafter are actually procedures or subroutines built from the more primitive operators. Incremental data structures are the inputs to these procedures. For operators that are target incremental operators, values and booleans are their outputs. For those that are not, incremental data structures are the outputs. The incremental operators defined here are:

1. Entity space Functions and Data Space Functions
   - 1.A. Incremental Entity space Operators
     - (1). -' (2). $\cap$' (3). $\cup$' (4.) **SelectS'**
     - (5). **JoinS'** (6). **ProjectS'** (7). **Group_byS'**.
   - 1.B. Transformation of Incremental States to Incremental D_relations.
     - **Extract'**.
   - 1.C. Incremental Data Space Operators
     - (1). **SelectD'** (2). **ProjectD'** (3). **ExtendD'**.
2. Aggregate Functions and Predicates
   - 2.A. Incremental Aggregate Functions and Predicate In the Entity space
     - (1). **CountS'** (2).$\mathbf{PD}_{\subset}$' (3). $\mathbf{PD}_{\supseteq}$' (4). $\mathbf{PD}_{\subset}$'
     - (5). $\mathbf{PD}_{\supset}$' (6). $\mathbf{PD}_{=}$'
   - 2.B. Incremental Aggregate Functions and Predicate In the Data Space
     - (1). **SumD'** (2). **CountD'** (3). **AvgD'** (4). **MaxD'**
     - (5). **MinD'** (6). **ValueD'**
     - (7). $\mathbf{PD}_{\geq}$', $\mathbf{PD}_{\leq}$', $\mathbf{PD}_{>}$', $\mathbf{PD}_{<}$', $\mathbf{PD}_{=}$'.

Note that the prime symbol," ' ", appended at the end differentiates an

incremental operator from a normal operator.

1. INCREMENTAL COMPUTATION- Entity space FUNCTIONS AND DATA SPACE FUNCTIONS.

1.A. Incremental Entity space Operators

The incremental computation of entity space operators can be divided into cases according to the following factors:
(a). THE OPERAND IS Ø OR NOT,
(b). LEFT OPERAND OR RIGHT OPERAND.

(1) Incremental -', ς_3':= ς_1' -' ς_2'.
(a) Both operands are Ø.
ς_1'=ς_2'=Ø.
ς_3':=Ø.
or ς_3':= ς_1' -' ς_2' := Ø.

(b) Left operand is not Ø, right operand is.
That is, ς_2'=Ø,
and let ς_4:= **Unsign** (ς_1'),
ς_5:= **Trace_bk** $_{\varsigma 4}$ (ς_2~),
ς_6':= **Sign** $_{\varsigma 1}$, (ς_5),
ς_3':= ς_1' - ς_6'.

(c) Right operand is not Ø, left operand is.
That is ς_1'=Ø,
and let ς_4:= **Unsign** (ς_2'),
ς_5:= **Trace_bk** $_{\varsigma 4}$ (ς_1~),
ς_6':= **Sign** $_{\varsigma 2}$, (ς_5),
ς_7':=ς_6' $\cap$ ς_2',
ς_3':= **Sign_change** (ς_7').
Note that if one of the operand is Ø, it only need to do partial evaluation only once.

(d) Both operands are not Ø.
(i) Resolve self images conflict in ς_1', ς_2'(to be explained),
For $\forall$ v that v $\in$ **Unsign**(ς_1') and also v $\in$ **Unsign**(ς_2'), use the following calculation formula:
ς_4:= **Unsign**(ς_1') $\cap$ **Unsign**(ς_2'),
ς_{11}':= **Sign** $_{\varsigma 1}$, (ς_4),

ς_{22}':= **Sign** $_{\varsigma 2'}$ (ς_4),

ς_{11}'	ς_{22}'	ς_4'
$\oplus$ v	$\oplus$ v	nil
$\oplus$ v	$\ominus$ v	$\oplus$ v
$\ominus$ v	$\oplus$ v	$\ominus$ v
$\ominus$ v	$\ominus$ v	nil

ς_4':= ς_{11}' -' ς_{22}' using the above formula.

(ii) Remove self images,

ς_{111}':=ς_1'- ς_{11}',

ς_{222}':=ς_2'- ς_{22}',

(iii) Check ς_{111}'

ς_5:=**Unsign**(ς_{111}'),

ς_6:=**Trace_bk** $_{\varsigma 5}$ (ς_2~),

Therefore, ς_5':=**Sign** $_{\varsigma 1'}$(ς_6),

ς_6':=ς_{111}'- ς_5'.

(iv) Check ς_{222}'

ς_7:=**Unsign** (ς_{222}'),

ς_8:=**Trace_bk** $_{\varsigma 7}$ (ς_1~),

Therefore, ς_7':=**Sign** $_{\varsigma 222'}$ (ς_8),

ς_8':=ς_7' $\cap$ ς_{222}',

ς_9':=**Sign_change** (ς_8'),

(v) Put all together

ς_3':= ς_4' $\cup$ ς_6' $\cup$ ς_9'

All four cases (a)-(d) can be combined into one case i.e. case (d), if we assume **Trace_bk** $_{\varnothing}$ (ς~):= $\varnothing$.

(e) Both operands are not $\varnothing$ - using partial realization.

We can simplify the above computation logic by using partial realization, but it requires more extensive partial evaluation.

ς_{10}:= **Unsign**(ς_1') $\cup$ **Unsign**(ς_2'),

ς_{11}~:= **Trace_bk** $_{\varsigma 10}$ (ς_1~),

ς_{22}~:= **Trace_bk** $_{\varsigma 10}$ (ς_2~),

ς_{33}~:= ς_{11}~ - ς_{22}~,

ς_{11}:= ς_{11}~ α ς_1',

ς_{22}:= ς_{22}~ α ς_2',

ς_{33}:= ς_{11} - ς_{22},

ς_3':= (**Assign_plus** (ς_{33} - ς_{33}~)) $\cup$
(**Assign_minus** (ς_{33}~ - ς_{33})).

It is important to note the difference of (d) and (e). They both give the same result but at different expenses. Computation in (d) takes more consideration of the nature of the operator and the semantic meaning of the signed data

structure. It minimizes partial evaluation whenever possible. But, computation in (e) simplifies the computation logic at the expense of a larger partial evaluation (ς_{10}). If $|\varsigma_{10}| >> |\varsigma_5|$ and $|\varsigma_{10}| >> |\varsigma_7|$, computation in (d) is preferred. If, however, $|\varsigma_{10}|$, $|\varsigma_5|$, and $|\varsigma_7|$ are about the same sizes, (e) is preferred.

(f) Examples.

Assuming the following case for the examples given:

$\varsigma_1\tilde{}$:={ <a1>, <a2>, <a3>, <a4>},

$\varsigma_2\tilde{}$:={ <a2>, <a3>, <a5>},

$\varsigma_3\tilde{}$:= $\varsigma_1\tilde{}$ - $\varsigma_2\tilde{}$:= {<a1>,<a4>},

ς_3':= ς_1' -' ς_2' in all the following examples.

#Example F. 1.

ς_1':= { $\oplus$<a5> },

ς_2':= { $\ominus$<a5> },

If ς_3':= ς_1' -' ς_2',following (d).(i) we get ς_3':={$\oplus$<a5>}.

If, however, we don't have (d).(i) to resolve the self image conflict problem and just to use (d).(iii) and (d).(iv) for the computation, we would end up with:

Using ς_1' and (d).(iii) we get ς_6':= $\emptyset$,

Using ς_2' and (d).(iv) we get ς_9':= $\emptyset$,

ς_3':= ς_6' $\cup$ ς_9' := $\emptyset$. This is in contrary to reality. Therefore, we have to have (d).(i) to resolve the self image problem. ###

#Example F. 2.

ς_1':={ $\oplus$<a5> }, ς_2':=$\emptyset$,

According to (b), ς_4:= **Unsign** (ς_1') := {<a5>},

ς_5:= **Trace_bk** $_{\varsigma 4}$ ($\varsigma 2\tilde{}$) := {<a5>},

ς_6' := **Sign** $_{\varsigma 1}$, (ς_5) := {$\oplus$<a5>},

ς_3' := ς_1' - ς_6' := $\emptyset$. ###

#Example F. 3.

ς_1':=$\emptyset$, ς2':= { $\ominus$<a2> }.

According to (c), ς_4:= **Unsign** (ς_2'):= {<a2>},

ς_5:= **Trace_bk** $_{\varsigma 4}$ ($\varsigma_1\tilde{}$):={<a2>},

ς_6' := **Sign** $_{\varsigma 2}$, (ς_5):= {$\ominus$<a2>},

ς_7' :=ς_6' $\cap$ ς_2' := {$\ominus$<a2>},

ς_3' := **Sign_change** (ς_7'):= {$\oplus$<a2>}. ###

#Example F. 4.

ς_1':={ $\oplus$<a5> }, ς_2':={ $\ominus$<a2> }.

According to (d),

(i) ς_4':= Ø,
(ii) ς_{111}':= ς_1'-ς_{11}':=ς_1', ς_{222}':= ς_2'-ς_{22}':=ς_2',
(iii) ς_6':=Ø, (Computation is similar to Example F. 2)
(iv) ς_9':={⊕<a2>}, (Computation is similar to Example F.3)
(v) ς_3':= ς_4' ∪ ς_6' ∪ ς_9':= { ⊕<a2>}. ###

(2) Incremental ∩', ς_3':= ς_1' ∩' ς_2'.
(a) Both operands are Ø.
ς_1' = Ø and ς_2' = Ø.
ς_3':=Ø.

(b) One operand is Ø, the other is not Ø.
Assuming ς_2'=Ø. Let ς_4:= **Unsign**(ς_1'),
ς_5:= **Trace_bk** $_{\varsigma 4}$ (ς_2~),
ς_6':= **Sign** $_{\varsigma 1}$, (ς_5),
ς_3':= ς_6' ∩ ς_1'.

(c) Both operands are not Ø.
(i) resolve self image conflict
ς_4:= **Unsign**(ς_1') ∩ **Unsign**(ς_2'),
ς_{11}':= **Sign** $_{\varsigma 1}$, (ς_4),
ς_{22}':= **Sign** $_{\varsigma 2}$, (ς_4),
Resolve the self image conflict by the following formula:

ς_{11}'	ς_{22}'	ς_4'
⊕ v	⊕ v	⊕ v
⊕ v	⊖ v	nil
⊖ v	⊕ v	nil
⊖ v	⊖ v	⊖ v

ς_4':= ς_{11}' ∩' ς_{22}' using the above formula.
(ii) Remove self images
ς_{111}':=ς_1' - ς_{11}',
ς_{222}':=ς_2' - ς_{22}',
(iii) check ς_{111}'
ς_5:= **Unsign**(ς_{111}'),
ς_6:= **Trace_bk** $_{\varsigma 5}$ (ς_2~),
Therefore, ς_5':= **Sign** $_{\varsigma 1}$,(ς_6),
ς_6':= ς_{111}' ∩ ς_5'.
(iv) check ς_{222}'
ς_7:=**Unsign** (ς_{222}'),
ς_8:=**Trace_bk** $_{\varsigma 7}$ (ς_1~),
Therefore, ς_7':= **Sign** $_{\varsigma 222}$,(ς_8),

ς_8':=ς_7' $\cap$ ς_{222}',
(v) put all together
ς_3':=ς_4' $\cup$ ς_6' $\cup$ ς_8'

(d) Both operands are not $\emptyset$ - using partial realization.

An alternative way of doing the same computation is as follows:

ς_{10}:= **Unsign**(ς_1') $\cup$ **Unsign**(ς_2'),
$\varsigma_{11}\tilde{}$:= **Trace_bk** $_{\varsigma 10}$ ($\varsigma_1\tilde{}$),
$\varsigma_{22}\tilde{}$:= **Trace_bk** $_{\varsigma 10}$ ($\varsigma_2\tilde{}$),
$\varsigma_{33}\tilde{}$:= $\varsigma_{11}\tilde{}$ $\cap$ $\varsigma_{22}\tilde{}$,
ς_{11}:= $\varsigma_{11}\tilde{}$ α ς_1',
ς_{22}:= $\varsigma_{22}\tilde{}$ α ς_2',
ς_{33}:= ς_{11} $\cap$ ς_{22},
ς_3':= (**Assign_plus** (ς_{33} - $\varsigma_{33}\tilde{}$)) $\cup$ (**Assign_minus** ($\varsigma_{33}\tilde{}$ - ς_{33})).

(e) Examples

The examples are based on the following case:

$\varsigma_1\tilde{}$:= {<a1>,<a2>,<a3>,<a4>},
$\varsigma_2\tilde{}$:= {<a2>,<a3>,<a5>},
$\varsigma_3\tilde{}$:= $\varsigma_1\tilde{}$ $\cap$ $\varsigma_2\tilde{}$:= {<a2>,<a3>},
ς_3':= ς_1' $\cap$' ς_2' in all the following examples.

#Example F. 5.
ς_1':={$\oplus$<a5>}, ς_2':={$\ominus$<a5>},
According to (c), ς_3':=$\emptyset$.

If without resolving the self image conflict, we would end up with: ς_6':={$\oplus$<a5>}, ς_8':=$\emptyset$, then ς_3':=ς_6' $\cup$ ς_8':= {$\oplus$<a5>} which is not correct.
###

#Example F. 6.
ς_1':={$\oplus$<a5>}, ς_2':=$\emptyset$. According to (b), ς_3':={$\oplus$<a5>}.
###

#Example F. 7.
ς_1':={$\oplus$<a5>,$\ominus$<a2>},
ς_2':={$\oplus$<a1>,$\ominus$<a3>},
According to (c), ς_3':={$\oplus$<a5>,$\ominus$<a2>,$\oplus$<a1>,$\ominus$<a3>}. ###

(3) Incremental $\cup$', ς_3':= ς_1' $\cup$' ς_2'.

(a) Both operands are $\emptyset$.
ς_1'=ς_2'=$\emptyset$,

ς_3':=Ø.

(b) One operand is not Ø, the other is Ø.
Assume ς_2'=Ø. Let ς_4:=**Unsign**(ς_1'),
ς_5:=**Trace_bk** $_{\varsigma 4}$ (ς_2~),
ς_6':= **Sign** $_{\varsigma 1}$, (ς_5),
ς_3':= ς_1' - ς_6'.

(c) Both operands are not Ø.
(i) Resolve self image conflict
ς_4:= **Unsign**(ς_1') ∩ **Unsign**(ς_2'),
ς_{11}':= **Sign** $_{\varsigma 1}$, (ς_4),
ς_{22}':= **Sign** $_{\varsigma 2}$, (ς_4),
Resolve the self image conflict by the following formula:

ς_{11}'	ς_{22}'	ς_4'
⊕ v	⊕ v	⊕ v
⊕ v	⊖ v	nil
⊖ v	⊕ v	nil
⊖ v	⊖ v	⊖ v

ς_4':= ς_{11}' ∪' ς_{22}' using the above formula.
(ii) Remove mirror images
ς_{111}':= ς_1'- ς_{11}',
ς_{222}':= ς_2'- ς_{22}',
(iii) check ς111'
ς_5:= **Unsign**(ς_{111}'),
ς_6:= **Trace_bk** $_{\varsigma 5}$ (ς_2~),
Therefore, ς_5':= **Sign** $_{\varsigma 1}$,(ς_6),
ς_6':= ς_{111}' - ς_5'.
(iv) check ς_{222}'
ς_7:= **Unsign**(ς_{222}'),
ς_8:= **Trace_bk** $_{\varsigma 7}$ (ς_1~),
Therefore, ς_7':= **Sign** $_{\varsigma 222}$,(ς_8),
ς_8':=ς_{222}' - ς_7'.
(v) put all together
ς_3':= ς_4' ∪ ς_6' ∪ ς_8'

(d) Both operands are not Ø - using partial realization.
An alternative way of doing the same computation is as follows:
ς_{10}:= **Unsign**(ς_1') ∪ **Unsign**(ς_2'),
ς_{11}~:= **Trace_bk** $_{\varsigma 10}$ (ς_1~),

$\varsigma_{22}^{\sim}$:= **Trace_bk** $_{\varsigma 10}$ ($\varsigma_2^{\sim}$),
$\varsigma_{33}^{\sim}$:= $\varsigma_{11}^{\sim} \cup \varsigma_{22}^{\sim}$,
ς_{11}:= $\varsigma_{11}^{\sim} \alpha \varsigma_1$',
ς_{22}:= $\varsigma_{22}^{\sim} \alpha \varsigma_2$',
ς_{33}:= $\varsigma_{11} \cup \varsigma_{22}$,
ς_3':= (**Assign_plus** ($\varsigma_{33} - \varsigma_{33}^{\sim}$)) ∪
(**Assign_minus** ($\varsigma_{33}^{\sim} - \varsigma_{33}$)).

(e) Examples.
The examples are based on the following case:
$\varsigma_1^{\sim}$:={<a1>,<a2>,<a3>}, $\varsigma_2^{\sim}$:={<a2>,<a3>,<a4>},
$\varsigma_3^{\sim}$:=$\varsigma_1^{\sim} \cup \varsigma_2^{\sim}$:={<a1>,<a2>,<a3>,<a4>},
ς_3':= ς_1' ∪' ς_2' in the following examples.

#Example F. 8.
ς_1':={⊖<a2>}, ς_2':={⊖<a2>},
Then according to (c), ς_3':={⊖<a2>} ###

#Example F. 9.
ς_1':={⊕<a5>}, ς_2':={ ⊖<a3> },
According to (c), ς_3':={⊕<a5>}. ###

#Example F. 10.
ς_1':={⊖<a2>}, ς_2':={⊕<a1>},
According to (c), ς_3':=Ø. ###

(4) Incremental SelectS', ς_2':= SelectS' Pformula (ς_1').
(a) The operand is Ø.
If ς_1'=Ø, ς_2':=Ø.

(b) The operand is not Ø.
The definition of **SelectS'** is the same as **SelectS** defined in [Lee87a] only that **SelectS'** now works on signed data structure.

(c) Example.
#Example F. 11.
$\varsigma_1^{\sim}$:= {<a1,b2>, <a2,b2>, <a3,b3>},
$\varsigma_2^{\sim}$:= **SelectS** $_{A \geq \text{'a2'}}$ ($\varsigma_1^{\sim}$),
$\varsigma_2^{\sim}$:= {<a2,b2>,<a3,b3>},
If ς_1':= {⊕<a1,b3>,⊖<a2,b2>} and ς_2':= SelectS' $_{A \geq \text{'a2'}}$ (ς_1'), according to (b), ς_2':= {⊖<a2,b2>}. ###

The **SelectS'** operator is the most straight forward implementation of all the entity space operators. It provides the most effective way to filter out undesirable vectors.

(5) Incremental JoinS', ς_3':= JoinS' $_{J\_set}$ (ς_1',ς_2').

Note that J_set equals to the intersection of schemes of ς_1 and ς_2. It is, therefore, sometimes omitted.

(a) Both operands are Ø.

ς_2'=ς_1'=Ø,

ς_3':=Ø.

(b) One operand is Ø, the other is not Ø.

Let ς_2':=Ø,

ς_4:=**Unsign**(ς_1'),

ς_5:=**ProjectS** $_{J\_set}$ (ς_4),

ς_6:=**Trace_bk** $_{\varsigma 5}$ (ς_2~),

ς_3':=**Sign** $_{\varsigma 1}$, (**JoinS** J_set (ς_4,ς_6)).

(c) Both operands are not Ø.

ς_7:= **ProjectS** $_{J\_set}$ (**Unsign** (ς_1')) ∪ **ProjectS** $_{J\_set}$ (**Unsign**(ς_2')),

ς_{11}~:= **Trace_bk** $_{\varsigma 7}$ (ς_1~),

ς_{22}~:= **Trace_bk** $_{\varsigma 7}$ (ς_2~),

ς_{33}~:= **JoinS** (ς_{11}~, ς_{22}~),

ς_{11}:= ς_{11}~ α ς_1',

ς_{22}:= ς_{22}~ α ς_2',

ς_{33}:= JoinS (ς_{11}, ς_{22}),

ς_3':= (**Assign_plus** (ς_{33}-ς_{33}~) ∪ **Assign_minus** (ς_{33}~-ς_{33})).

(d) Examples.

ς_1~ ⊆ A x B,

ς_2~ ⊆ B x C,

ς_1~:= { <a1,b1> <a2,b1> <a2,b2> <a2,b3> },

ς_2~:= { <b1,c1> <b1,c3> <b4,c5> <b3,c1> },

ς_3~:= **JoinS** $_{\{B\}}$ (ς_1~, ς_2~)

:={<a1,b1,c1>,<a1,b1,c3>,<a2,b1,c1>,<a2,b1,c3>,<a2,b3,c1>}

ς_3':= **JoinS'** (ς_1', ς_2') in the following examples.

#Example F. 12.

ς_1':= {⊕<a3,b1>}, ς_2':=Ø,

From (b), we get ς_3' := {⊕<a3,b1,c1>, ⊕<a3,b1,c3>}. ###

#Example F. 13.

ς_1':= {$\ominus$<a1,b1>}, ς_2':= {$\ominus$<b1,c1>}.

From (c), ς_7:= {<b1>},

$\varsigma_{11}^{\sim}$:={<a1,b1>,<a2,b1>}, $\varsigma_{22}^{\sim}$:={<b1,c1><b1,c3>},

ς_{11}:= {<a2,b1>}, ς22:={<b1,c3>},

$\varsigma_{33}^{\sim}$:={<a1,b1,c1>,<a1,b1,c3>,<a2,b1,c1>,<a2,b1,c3>}

ς_{33}:= {<a2,b1,c3>},

ς_3':={ $\ominus$<a1,b1,c1>,$\ominus$<a1,b1,c3>,$\ominus$<a2,b1,c1>} ###

#Example F. 14.

ς_1':= {$\oplus$<a3,b1>}, ς_2':={$\ominus$<b1,c1>}.

From (c), ς_7:= {<b1>},

$\varsigma_{11}^{\sim}$:={<a1,b1>,<a2,b1>}, $\varsigma_{22}^{\sim}$:={<b1,c1><b1,c3>},

ς_{11}:= {<a1,b1>,<a2,b1>,<a3,b1>}, ς_{22}:={<b1,c3>},

$\varsigma_{33}^{\sim}$:={<a1,b1,c1>,<a1,b1,c3>,<a2,b1,c1>,<a2,b1,c3>}

ς_{33}:={<a1,b1,c3>,<a2,b1,c3>,<a3,b1,c3>},

ς_3':={ $\oplus$<a3,b1,c3>,$\ominus$<a1,b1,c1>,$\ominus$<a2,b1,c1>} ###

(6) Incremental ProjectS', ς_2':= ProjectS' $_{P\_set}$ (ς_1').

(a) The operand is Ø.

ς_1'= Ø,

ς_2':=Ø.

(b) The operand is not Ø.

ς_4:= **Unsign**(ς_1'),

ς_5:= **ProjectS** $_{P\_set}$ (ς_4),

$\varsigma_{11}^{\sim}$:= **Trace_bk** $_{\varsigma 5}$ ($\varsigma_1^{\sim}$),

$\varsigma_{22}^{\sim}$:= **ProjectS** $_{P\_set}$ ($\varsigma_{11}^{\sim}$),

P_set is the set of symbols of the projected entity sets in **ProjectS'** or **ProjectS**.

ς_{11}:= $\varsigma_{11}^{\sim}$ α ς_1',

ς_{22}:= **ProjectS** $_{P\_set}$ (ς_{11}),

ς_2':= **Assign_plus**(ς_{22} - $\varsigma_{22}^{\sim}$) $\cup$ **Assign_minus**($\varsigma_{22}^{\sim}$ - ς_{22}).

(c) Example.

#Example F. 15.

$\varsigma_1^{\sim}$:= {<a1,b1>,<a1,b2>,<a2,b2>},

$\varsigma_2^{\sim}$:= **ProjectS** $_{\{A\}}$ ($\varsigma_1^{\sim}$) := {<a1>,<a2>},

ς_1':= {$\ominus$<a1,b1>}, ς_2':= **ProjectS'** $_{\{A\}}$ (ς_1')

According to (b) ς_2':= Ø.

(7) Incremental Group_ByS, ς_2':= Group_byS' $_{G\_set}$ (ς_1').

Note that G_set is the set of symbols of classifying entity sets.

(a) The operand is Ø.

ς_1'=Ø,

ς_2':=Ø.

(b) The operand is not Ø.

ς_4:= **Unsign** (ς_1'),

ς_5:= **ProjectS** $_{G\_set}$ (ς_4),

ς_{11}~:=**Trace_bk** $_{\varsigma 5}$ (ς_1~),

ς_{22}~:=**Group_byS** $_{G\_set}$ (ς_{11}~),

ς_9':=**Assign_minus** (ς_{22}~),

ς_{11}:= ς_{11}~ α ς_1',

ς_{22}:=**Group_byS** $_{G\_set}$ (ς_{11}),

ς_{10}':=**Assign_plus** (ς_{22}),

ς_2':= ς_9' $\cup$ ς_{10}'.

(c) Example.

#Example F. 16.

ς_1~:= {<a1,b1>,<a1,b2>,<a1,b3>,<a2,b3>}

ς_2~ := **Group_byS** $_{\{A\}}$ (ς_1~)

:= { <a1,{<b1>,<b2>,<b3>}>, <a2,{<b3>}> }.

If ς_1':= {$\ominus$<a1,b1>, $\oplus$<a2,b1>} and

ς_2':= **Group_byS'** $_{\{A\}}$ (ς_1'). According to (b),

ς_2':= { $\ominus$<a1,{<b1>,<b2>,<b3>}>, $\ominus$<a2,{<b3>}>,

$\oplus$<a1,{<b2>,<b3>} > ., $\oplus$<a2, {<b1>,<b3>}> }.

1.B. Transformation of Incremental States to Incremental D_relations

Incremental Extract', τ' := **Extract'** $_{\{F1<\text{-}A.f1,...\}}$ (ς')

The definition of **Extract'** is the same as **Extract** defined in [Lee87a] except that **Extract'** now works on signed data structures.

#Example F. 17: **Extract'**

ς':= {$\oplus$<a1,b3>, $\ominus$<a2,b3>} and Sch(ς'):={A,B}.

a1:=(1,2), a2:=(3,4), Sch(A):={f1,f2}.

b3:=("k","j"), Sch(B):={f3,f4}.

τ':= **Extract'** $_{\{F1<\text{-}A.f2,\ F2<\text{-}B.f3\}}$($\varsigma$1'), Sch($\tau$'):={F1,F2}.

Then, according to the above computation rule:

τ':= {(2, "k"), (4, "k")}.

Notice that the data tuples (2, "k"),(4, "k") involve data elements instead of objects.

1.C. Incremental Data Space Operators

The incremental computation is much more simplified for the data space operators than for the entity space operators. For one reason, we don't have to deal with **JoinS**, **Group_byS**, and set operators in the data space. All the incremental data space operators are similar to their corresponding data space operators only that incremental operators are now applied to signed data structures and the results are also signed data structures.

(1) Incremental SelectD', τ_2':= SelectD' $_{\text{Pformula}}$ **(τ_1').**
The definition of **SelectD'** is the same as **SelectD** defined in [Lee87a] except that **SelectD'** now works on signed data structures.

#Example F. 18: **SelectD'**
τ_1':={$\oplus$(2, "l"), $\ominus$(4, "a")}, Sch(τ'):={F1,F2}.
τ_2':= **SelectD'** $_{F1 > 3}$ (τ_1'),
So τ_2':= {$\ominus$(4, "a")},

(2) Incremental ProjectD', τ_2':= ProjectD' $_{\text{P\_set}}$ **(τ_1').**
The definition of **ProjectD'** is the same as **ProjectD** defined in [Lee87a] only that **ProjectD'** now works on signed data structures. Note that since the **ProjectD** does not remove duplicate tuples, neither does the **ProjectD'**.

#Example F. 19: **ProjectD'**
τ_1':={$\oplus$(2, "l"), $\ominus$(4, "a")}, Sch(τ_1'):={F1,F2}.
τ_2':= **ProjectD'** $_{\{F1\}}$ (τ1'),
So τ_2':= {$\oplus$(2),$\ominus$(4)},

(3)Incremental ExtendD', τ_2':= ExtendD' $_{\text{Eq\_list}}$ **(τ_1').**
The definition of **ExtendD'** is the same as **ExtendD** defined in [Lee87a] only that **ExtendD'** now works on signed data structures.

#Example F. 20: **ExtendD'**
τ_1':={$\oplus$(30, "l"), $\ominus$(90, "a")}, Sch(τ1'):={F1,F2}. τ_2':= **ExtendD'** $_{\{Fnew:=F1/3\}}$ (τ1'), Sch(τ_2'):={F1,F2,Fnew}. So τ_2':= {$\oplus$(30, "l", 10), $\ominus$(90, "a", 30)}.

2. INCREMENTAL COMPUTATION- AGGREGATE FUNCTIONS AND PREDICATES

Note that all the aggregate functions and predicates are the target incremental functions and predicates. [Lee87c]

2.A. Incremental Aggregate Functions and Predicates in the Entity Space

(1). Incremental CountS', μ := CountS'(ς').

There is only one incremental aggregate function in the entity space, i.e. **CountS'**.

$$\mu := \textbf{CountS'}\ (\varsigma'),$$

CountS' is calculated according to the following formula:

$$\mu.\text{new\_value} := \mu.\text{last\_value} + |\textbf{Plus\_part}(\varsigma')| - |\textbf{Minus\_part}(\varsigma')|$$

μ.last_value is an attribute of the μ node in the incremental transition digraph. The last_value attribute is updated upon each evaluation of μ.

(2). Incremental $PS_{\subseteq}$', β:= $PS_{\subseteq}$'(ς_1', ς_2').

The computation of $\overline{PS}_{\subseteq}$' is as follows:

(a). If $\varsigma_1'=\varsigma_2'=\emptyset$, β.new_value:= β.last_value.

The Update has no effect on the new value of β.

(b). If β.last_value:= TRUE then

ς_3:= **Unsign**(ς_1') $\cup$ **Unsign**(ς_2'),

$\varsigma_{1s}^{\sim}$:=**Trace_bk** $_{\varsigma 3}$ ($\varsigma_1^{\sim}$),

$\varsigma_{2s}^{\sim}$:=**Trace_bk** $_{\varsigma 3}$ ($\varsigma_2^{\sim}$),

ς_{1s}:=$\varsigma_{1s}^{\sim}$ α ς_1',

ς_{2s}:=$\varsigma_{2s}^{\sim}$ α ς_2',

Finally, β.new_value:= $PS_{\subseteq}$ (ς_{1s}, ς_{2s}),

if β.new_value:=FALSE then

begin

β.fail1:= ς_{1s} - ς_{2s}, β.fail2:=$\emptyset$.

end.

(c). If β.last_value:= FALSE then

ς_3:= **Unsign**(ς_1') $\cup$ **Unsign**(ς_2'),

$\varsigma_{1s}^{\sim}$:=**Trace_bk** $_{\varsigma 3}$ ($\varsigma_1^{\sim}$),

$\varsigma_{2s}^{\sim}$:=**Trace_bk** $_{\varsigma 3}$ ($\varsigma_2^{\sim}$),

ς_{1s}:=$\varsigma_{1s}^{\sim}$ α ς_1',

ς_{2s}:=$\varsigma_{2s}^{\sim}$ α ς_2',

ς_{1f}':= **Sign** $_{\varsigma 1}$,(**Unsign** (ς_1') $\cap$ β.fail1)

fail1s:= β.fail1 α ς_{1f}',

ς_{1sf}:= fail1s $\cup$ ς_{1s}.

Finally, β.new_value:= $PS_{\subseteq}$ (ς_{1sf}, ς_{2s}),

if β.new_value:=FALSE then

begin
β.fail1 := ς_{1sf} - ς_{2s}, β.fail2:=$\emptyset$.
end
else
begin
β.fail1:=β.fail2:=$\emptyset$
end.

Note that there are still some rooms for further optimization in the above algorithm. However, more logic reasoning will be added to differentiate various cases. Here, we only give a general framework for the evaluation of incremental predicates.

(3). Incremental $PS_{\supseteq}$', β:= $PS_{\supseteq}$'(ς_1', ς_2').

The same as above only that the notations for ς_1 and ς_2 switch with each other.

(4). Incremental $PS_{\subset}$', β:= $PS_{\subset}$'(ς_1', ς_2').

The computation of $PS_{\subset}$' is as follows:

(a). If ς_1'=ς_2'=$\emptyset$, β.new_value:= β.last_value.
The Update has no effect on the new value of β.

(b). If β.last_value:= TRUE then
ς_3:= **Unsign**(ς_1') $\cup$ **Unsign**(ς_2'),
ς_{1s}~:=**Trace_bk** $_{\varsigma 3}$ (ς_1~),
ς_{2s}~:=**Trace_bk** $_{\varsigma 3}$ (ς_2~),
ς_{1s}:=ς_{1s}~ α ς_1',
ς_{2s}:=ς_{2s}~ α ς_2',
Finally, β.new_value:= $PS_{\subset}$(ς_{1s}, ς_{2s}),
if β.new_value:=FALSE then
begin
β.fail1 := (ς_{1s} - ς_{2s}) $\cup$ (ς_{1s} $\cap$ ς_{2s}),
β.fail2:= (ς_{1s} $\cap$ ς_{2s})
end.

(c). If β.last_value:= FALSE then
ς_3:= **Unsign**(ς_1') $\cup$ **Unsign**(ς_2'),
ς_{1s}~:= **Trace_bk** $_{\varsigma 3}$ (ς_1~),
ς_{2s}~:= **Trace_bk** $_{\varsigma 3}$ (ς_2~),
ς_{1s}:=ς_{1s}~ α ς_1',
ς_{2s}:=ς_{2s}~ α ς_2',
ς_{1f}':= **Sign** $_{\varsigma 1}$,(**Unsign** (ς_1') $\cap$ β.fail1)
fail1s:= β.fail1 α ς_{1f}',
ς_{1sf}:= fail1s $\cup$ ς_{1s}.
ς_{2f}':= **Sign** $_{\varsigma 2}$,(**Unsign** (ς_2') $\cap$ β.fail2)

fail2s:= β.fail2 α ς_{2f}',
ς_{2sf}:= fail2s $\cup$ ς_{2s}.
Finally, β.new_value:= $\mathbf{PS}_{\subset}(\varsigma_{1sf}, \varsigma_{2sf})$,
if β.new_value:=FALSE then
begin
β.fail1 := $(\varsigma_{1sf} - \varsigma_{2sf})$,
β.fail2 := $(\varsigma_{2sf} - \varsigma_{1sf})$.
end
else
begin
β.fail1:=β.fail2:=$\emptyset$
end.

(5). Incremental $\mathbf{PS}_{\supset}$', β:= $\mathbf{PS}_{\supset}$'(ς_1', ς_2').
The same as above only that the notation for "ς_1" and "ς_2" switch with each other.

(6). Incremental $\mathbf{PS}_{=\!=}$', β:= $\mathbf{PS}_{=\!=}$'(ς_1', ς_2').
The computation of $\mathbf{PS}_{=\!=}$' is as follows:
(a). If ς_1'=ς_2'=$\emptyset$, β.new_value:= β.last_value.
The Update has no effect on the new value of β.
(b). If β.last_value:= TRUE then
ς_3:= **Unsign**(ς_1') $\cup$ **Unsign**(ς_2'),
$\varsigma_{1s}\tilde{}$:=**Trace_bk** $_{\varsigma 3}$ ($\varsigma_1\tilde{}$),
$\varsigma_{2s}\tilde{}$:=**Trace_bk** $_{\varsigma 3}$ ($\varsigma_2\tilde{}$),
ς_{1s}:=$\varsigma_{1s}\tilde{}$ α ς_1',
ς_{2s}:=$\varsigma_{2s}\tilde{}$ α ς_2',
Finally, β.new_value:= $\mathbf{PS}_{=\!=}(\varsigma_{1s}, \varsigma_{2s})$,
if β.new_value:=FALSE then
begin
β.fail1 := $\varsigma_{1s} - \varsigma_{2s}$, β.fail2:=$\varsigma_{2s} - \varsigma_{1s}$,
end.
(c). If β.last_value:= FALSE then
ς_3:= **Unsign**(ς_1') $\cup$ **Unsign**(ς_2'),
$\varsigma_{1s}\tilde{}$:=**Trace_bk** $_{\varsigma 3}$ ($\varsigma_1\tilde{}$),
$\varsigma_{2s}\tilde{}$:=**Trace_bk** $_{\varsigma 3}$ ($\varsigma_2\tilde{}$),
ς_{1s}:=$\varsigma_{1s}\tilde{}$ α ς_1',
ς_{2s}:=$\varsigma_{2s}\tilde{}$ α ς_2',
ς_{1f}':= **Sign** ς_1'(**Unsign** (ς_1') $\cap$ β.fail1)
fail1s:= β.fail1 α ς_{1f}',
ς_{1sf}:= fail1s $\cup$ ς_{1s}.
ς_{2f}':= **Sign** $_{\varsigma 2'}$(**Unsign** (ς_2') $\cap$ β.fail2)

fail2s:= β.fail2 α ς_{2f}',
ς_{2sf}:= fail2s $\cup$ ς_{2s}.
Finally, β.new_value:= $\mathbf{PS}_{=}(\varsigma_{1sf}, \varsigma_{2sf})$,
if β.new_value:=FALSE then
    begin
    β.fail1 := ς_{1sf} - ς_{2sf}, β.fail2:= ς_{2sf} - ς_{1sf}.
    end
else
    begin
    β.fail1 := β.fail2 := $\emptyset$,
    end.

In all the above procedures, if one of the ς_1, ς_2 is a constant, it can be treated as follows:

(a) If ς_1=constant, ς_1':=$\emptyset$.

(b) If trace back is necessary, we can use the following equivalence:

$$\mathbf{Trace\_bk}_{\varsigma 2}(\varsigma_1) = \mathbf{Trace\_bk}_{\varsigma 2}(\{\text{Constants}\}) = \varsigma_2 \cap \{\text{Constant}\}.$$

2.B. Incremental Aggregate Functions and Predicates in the Data Space

Remember in the data space computation, we are talking about tuples and d_relations, not vectors and states any more. Arguments in the tuples are individual data instead of entities.

Again, the system needs to maintain a historic record of the last evaluation of the aggregate functions. When we apply **SumD**, **CountD**, **AvgD**, **MaxD**, or **MinD** to a d_relation ,τ, it must be a single attribute d_relation. If it is not, the above functions are undefined. The same cases are for incremental functions, **SumD'**, **CountD'**, **AvgD'**, **MaxD'**, and **MinD'**.

(1). Incremental SumD', μ:= SumD'(τ')

μ := **SumD'**(τ') can be computed as follows:

μ.new_value:=μ.last_value+(**SumD**(**Unsign**(**Plus_part**(τ'))))
- (**SumD** (**Unsign** (**Minus_part**(τ')))).

(2). Incremental CountD', μ:= CountD'(τ')

μ := **CountD'** (τ') can be computed as follows:

μ.new_value:= μ.last_value + |**Plus_part**(τ')|
- |**Minus_part**(τ')|

(3). Incremental AvgD', μ:= AvgD'(τ')

μ := **AvgD'** (τ') can be computed as follows:

The μ keeps two historic records of the last values of **CountD**, and **SumD**. So
μ.new_SumD:=μ.last_SumD+(**SumD (Unsign (Plus_part**(τ'))))
-(**SumD (Unsign (Minus_part**(τ')))),
μ.new_CountD:=μ.last_CountD + |**Plus_part**(τ')|
- |**Minus_part**(τ')|,
μ.new_value:= μ.new_SumD / μ.new_CountD.

(4). Incremental MaxD', μ:= MaxD'(τ').
μ:= **MaxD'**(τ') can be computed as follows:
Let τ_plus:= **Unsign(Plus_part** (τ')),
τ_minus:= **Unsign(Minus_part** (τ')),
If μ.last_value < **MaxD**(τ_plus) then
μ.new_value:= **MaxD** (τ_plus)
else μ.new_value:= μ.last_value,
If μ.new_value = **MaxD**(τ_minus) then require full evaluation of $\tau\tilde{}$.
{Notice that the last if-statement is testing μ.new_value.}

(5). Incremental MinD', μ:= MinD'(τ').
μ:= **MinD'**(τ') can be computed as follows:
Let τ_plus:= **Unsign(Plus_part** (τ')),
τ_minus:= **Unsign(Minus_part** (τ')),
If μ.last_value > **MinD** (τ_plus) then
μ.new_value:= **MinD** (τ_plus)
else μ.new_value:= μ.last_value,
If μ.new_value = **MinD** (τ_minus) then require full evaluation of $\tau\tilde{}$.
{Notice that the last if-statement is testing μ.new_value.}

The last statements in both MaxD' and MinD', where full evaluations of τ are required, is in place because we are deleting the previous minimum or maximum from τ. To find out the next minimum or the next maximum requires full knowledge of τ. This is actually the only place in our incremental computation that requires full evaluation of a particular d_relation to get a new value for a node. But, we should not worry too much about this since the rarity of this would happen (we just happen to delete the last maximun or minimum without adding a new one to replace it).

(6). Incremental ValueD', μ:= ValueD'(τ').

The **ValueD** operator is restricted to the following type of d_relations in the data space computation:

μ:= **ValueD** (τ), τ:= {(d1)} and μ:=d1.

Therefore, there are only two possibilities for τ'. One is τ':=$\emptyset$. The other is τ':={$\ominus$(d1),$\oplus$(d11)}. The second possibility is subjected to the following

constraint:

μ.last_value = **ValueD(Unsign (Minus_part** (τ'))).

Now, let's look at the formula for **ValueD'**.

μ:= **ValueD'** (τ').

(a). If τ'= $\emptyset$, then

μ.new_value:= μ.last_value

(b). If τ <> $\emptyset$, then

μ.new_value := **ValueD(Unsign (Plus_part** (τ'))).

(7). Incremental $PD_{<}$',$PD_{>}$',$PD_{\leq}$',$PD_{\geq}$',$PD_{=}$'.

Since they are all handled in the same way, we only discuss the last case. When we apply these d_relation predicates, we must make sure that the d_relations are relations with exactly two attributes of the same data type that can be compared. The same rule should be followed for the incremental predicates and the incremental d_relations.

β := **$PD_{=}$'**(τ') can be computed as follows:

(a). If τ'=$\emptyset$ then β.new_value:=β.last_value,

(b). If β.last_value:= TRUE then

begin

τ_plus:= **Unsign(Plus_part** (τ')),

β.new_value:= **$PD_{=}$** (τ_plus),

if β.new_value:= FALSE then

β.fail:= {tuples that are false in τ_plus}

end,

(c). If β.last_value:= FALSE then

begin

τ_plus:= **Unsign(Plus_part** (τ')),

τ_minus:= **Unsign(Minus_part** (τ')),

τ_plusf:= τ_plus $\cup$ (β.fail - τ_minus);

β.new_value:= **$PD_{=}$** (τ_plusf),

if β.new_value:= FALSE then

β.fail:= {tuples that are false in τ_plusf}

else

β.fail:=$\emptyset$

end.

APPENDIX G. THE PARTIAL_EVAL ALGORITHM FOR DYNAMIC PARTIAL EVALUATION

We will look at how to carry out dynamic partial evaluation by modifying evaluation strategies. We assume there is at most one level of nesting in the hyper entity space.

#Definition G.1: **UngroupS**
The un-group operator is the reverse of **Group_byS** operator. It is a mapping from a hyper entity space to the primary entity space (see definitions in [Lee87a]).

ς_2:= **UngroupS** (ς_1)
$\varsigma_1 \subseteq \Psi_1$, Ψ_1=AxBx..xHxG*, $\varsigma_2 \subseteq \Psi_2$, Ψ_2=AxBx..xHxG.
For $<a_i,b_j,....,h_m,\{<g_{i1}>,...,<g_{in}>\}> \in \varsigma_1$,
$<===>$ $<a_i,b_j,...,h_m,g_{i1}>,...,<a_i,b_j,...,h_m,g_{in}> \in \varsigma_2$.
Therefore,ς_1:=**Group_byS** {A,B,..,H} (ς_2), ς_2:=**UngroupS**(ς_1).

#Example G.1: **UngroupS**

ς_1:={ <{<a1>,<a2>}> },
ς_2:=**UngroupS**(ς_1) :={ <a1> , <a2> }.
ς_3:={ <a1,{<b1>,<b2>} >, <a2,{<b2>}> }
ς_4:=**UngroupS**(ς_3) :={ <a1,b1>, <a1,b2>, <a2,b2> } . ###

Global C(N,E):= **Subgraph**(ς_1); {The algorithm will optimize the **Subgraph**(ς_1) to construct the **Subgraph**(ς_2) }

```
Procedure Partial_eval (ς, Subgraph(ς1));
  begin
  sink_node:= Find_sink_node(C(N,E));
  Push(ς, sind_node)
  end. {Partial_eval, the main program.}

procedure Push(ς,node);
  begin
  if node.pre=∅ then Insert_node(ς,node) { if it is a
    base node then insert a new filter node above it }
  else if node.pre_op={Group_byS} then
                Insert_at_group_by(ς,node)
            {if the previous operation is Group_byS
              go to procedure Insert_at_group_by }
  else if node.pre_op={JoinS} then Insert_at_join(ς,node)
                {if the previous operation is JoinS
                  go to procedure Insert_at_join }
```

```
  else
     for each pre_node in node.pre do Push(ς,pre_node);
  {recursively push the ς down to the leaves}
  end; {Push}

procedure Insert_node(ς,node);
  begin
  Create_node(new_node);
  new_node.label:=Next_label(C(N,E));
  new_node.type:= 'state';
  new_node.pre:={node.label};
  new_node.post:=node.post;
  new_node.pre_op:= {(SelectSS ς)};
  For each next_node in node.post Do
    begin
    next_node.pre:=next_node.pre - {node.label}
                    ∪ {new_node.label};
    E:=E-{<node, next_node>}∪{<new_node,next_node>}
    end;
  node.post:={new_node.label};
  N:=N ∪ {New_node};
  E:=E ∪ {<node, new_node>}
  end; {Insert_node}

procedure Insert_at_group_by (ς,node);
  begin
  pre_node:= get_node(node.pre); {Note because it is a
      Group_byS operation, node_pre only has one node.}
  if(Sch(ς) ⊆ node.E_state) then push(ς, pre_node)
  else
    begin
    Insert_node(ς, node);
    ςς:= UngroupS (ς);
    Push(ςς, pre_node)
    end
  end; {Insert_at_group_by}

procedure Insert_at_join (ς, node);
  begin
  for the two nodes in node.pre do
    begin
```

```
    pre_node1:= first node in node.pre;
    pre_node2:= second node in node.pre
    end;
  if (SchE(ς) ⊆ Sch(pre_node1) ) and
    (SchE(ς) ∩ Sch(pre_node2) =∅ )   then
            Push(ς, pre_node1);
  else if (SchE(ς) ⊆ Sch(pre_node2) ) and
       (SchE(ς) ∩ Sch(pre_node1) =∅ )  then
            Push(ς, pre_node2);
  else
    begin
    Insert_node(ς,node);
    ςς1:= ProjectS pre_node1.E_state (ς);
    ςς2:= ProjectS pre_node2.E_state (ς);
    Push( ςς1, pre_node1);
    Push( ςς2, pre_node2)
    end
  end; {Insert_at_join}.
```

Note:

(A). Some of the procedures, functions used in the above algorithm require further explanation.

(B). **Create_node**(new_node): create an empty node data structure and assign it to the variable new_node.

(C). **Get_node**({node.label}): return a node data structure represented by the node label. The argument is a set of only one node label.

(D). **Find_sink_node**(C(N,E)): find the sink_node of a graph.

(E). **Next_label**(C(N,E)): find the next symbol for representing a node in the digraph.

#Example G.2: **Trace_bk**

If ς_1 is defined through the following computation procedure:

ς_1:=**Group_byS** $_{\{A\}}$ (**ProjectS** $_{\{A,B\}}$ $(A\ ^{\circ}R_{AB}\ ^{\circ}B\ ^{\circ}R_{BC}\ ^{\circ}C)$),

Subgraph(ς_1) is presented in Figure G.1. If we use the dynamic partial evaluation strategy, we can see:

ς_2:= **Trace_bk** $_{\varsigma}$ (ς_1). and $\varsigma \subseteq \Psi$, Ψ=AxB*.

So, **Subgraph**(ς_2) according to the **Partial_eval** algorithm in APPENDIX G is shown in Figure G.2. ###

Figure G.1. Transition Digraph For Subgraph(ς_1)

ς_1:=**Group_byS** $_{\{A\}}$ (**ProjectS** $_{\{A,B\}}$ $(A * R_{AB} * B * R_{BC} * C)$),

π: **ProjectS**,
Σ: **Group_byS**,
$\bowtie$: **JoinS**.

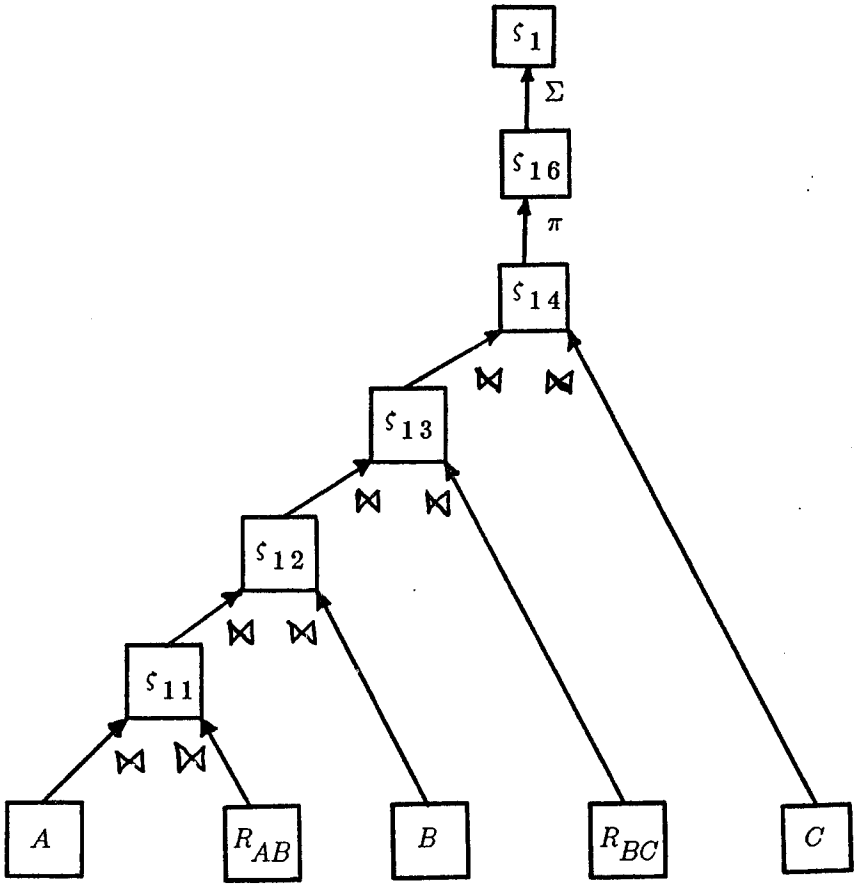
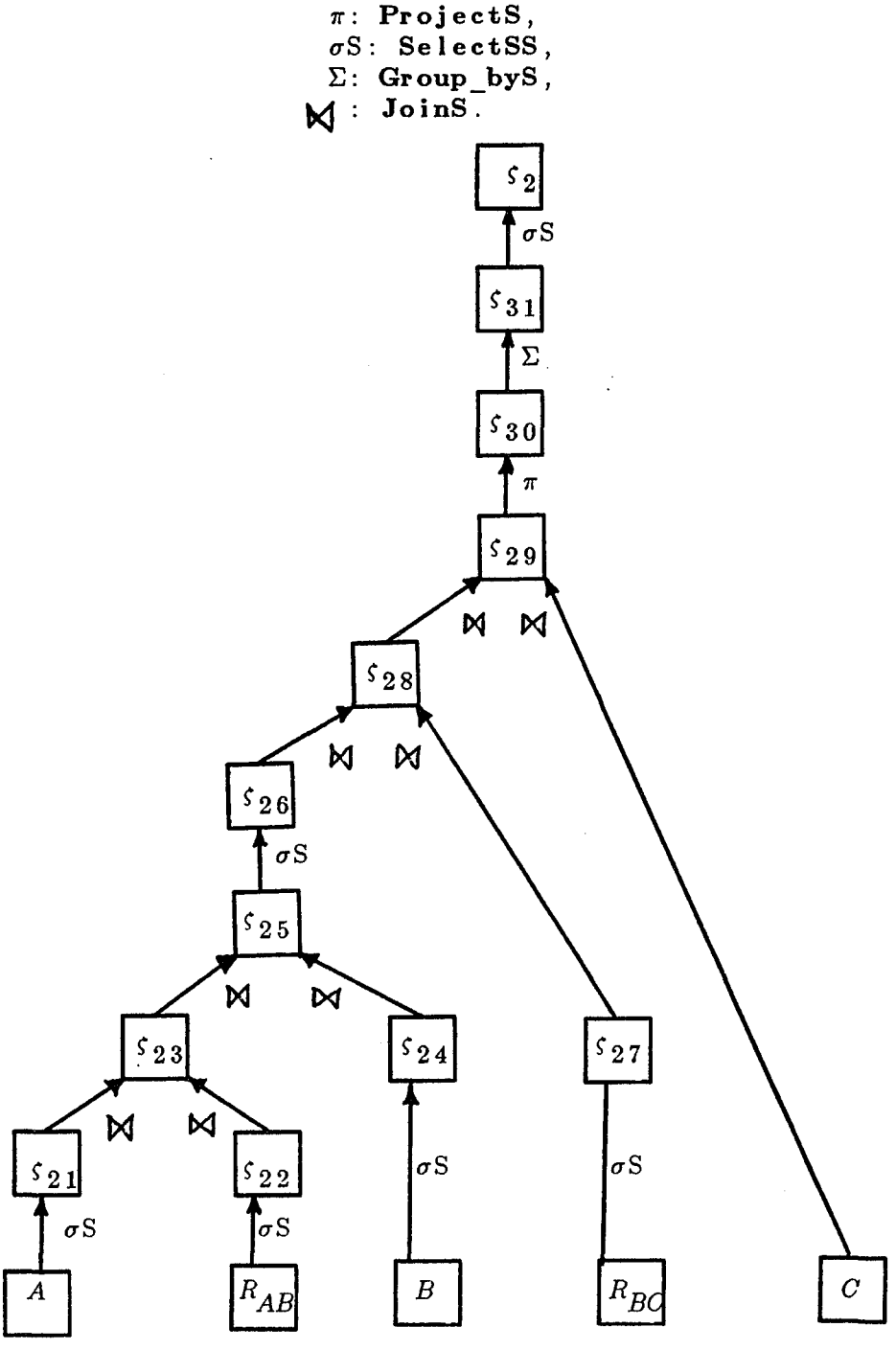


Figure G.2. Transition Digraph For Subgraph(ς2)

ς_1:=**Group_byS** $_{\{A\}}$ (**ProjectS** $_{\{A,B\}}$ $(A \circ R_{AB} \circ B \circ R_{BC} \circ C)$),

Subgraph(ς_2):= **Partial_eval**(ς, **Subgraph**(ς_1)).



APPENDIX H. ASSUMPTIONS IN THE PHYSICAL IMPLEMENTATION FOR PERFORMANCE EVALUATION.

Performance criteria of run-time computation strategy is based on the block access cost. We assume that block reads and writes are at a premium.

1. GENERAL NOTATIONS

(A). r_1, r_2 ,.., r_i are the total number of records for file f_1,f_2, ...,f_i.
Each file cooresponds to either an entity state, a d_relation , an entity set, or a relationship set. We still use the ADL symbols to represent them, i.e. ς, τ, E, R. Note that $\not\models r_i$, $\not\models r_j$.
(B). b_i is the number of bytes it takes for a single record in file f_i.
(C). B_i is the number of blocks for file f_i. Bx_i is the number of blocks for an index file of an entity set file.
(D). Total number of bytes per block is N bytes. N=256 bytes/block for all the examples.
(E). Assume B_i is derivable from r_i and b_i and N. The procedure is called **Block**(r_i,b_i,N).

```
Procedure Block(r_i,b_i,N)
    begin
    Let n:= r_i / Int(N/b_i),
    if n= integer, then B_i:=n,
                else B_i:=n+1
    end.
```

(F). Once we get value or booleans in the transition digraph, they are kept in the main memory. Further computations will simply treat them as variables in the main memory, no secondary storage is necessary.

2. FILE STRUCTURES

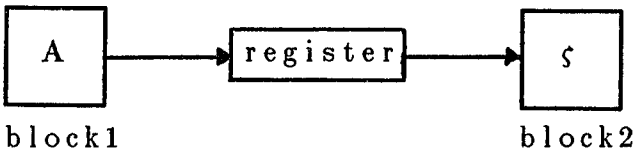
(A). Each entity set file is a random file of data records. Attribute values are stored as record fields in the file. A separate dense index file is kept for each entity set. Each index has eight bytes; four bytes for the key, four bytes for the physical location pointer.
(B). Each relationship set file is a file of data records. Each record contains keys from associated entity sets. All the keys have four bytes.
(C). Any entity state in the intermediate step of constraint computation will be stored in a random file of entity vector records. Each record consists of keys from the entity sets.
(D). D_relations derived in the computation will be stored in a random file of data

records each of that consists of data values extracted from related entity records.

3. IMPLEMENTATION OF ENTITY SPACE OPERATORS AND PREDICATES (those related to the Examples 3.1, 3.2, and 3.3.)

(A). **SelectS**: **SelectS** P (A) --> ς.

Main memory use two blocks and one register for processing the **SelectS** operator.



*Once block2 is full, it is written to the disk storage.

---------- Assumption ----------

A: r1, b1, B1.

ς: r2, b2, B2.

The portion of records that is selected is f.

r2:=r1 * f.

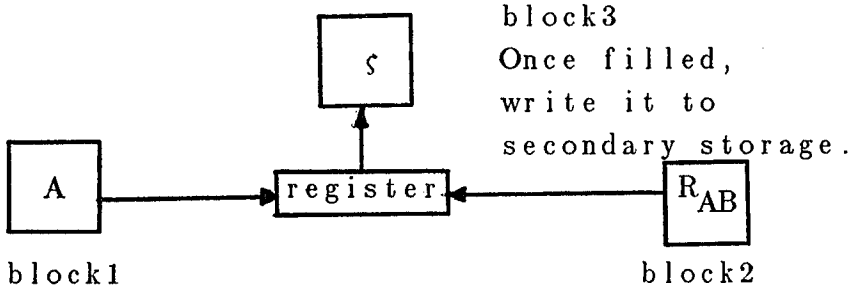
---

read := B1.

write:= B2.

(B). **JoinS** :

(i). ς:=**JoinS**(A,R_{AB}). A is an entity set. R_{AB} is a relationship set.



---------- Assumption ----------

A : r1, b1, B1, Bx1. Bx1 is the number of blocks in the index file of A.

R_{AB} : r2, b2, B2.

ς : r3, b3, B3.

Let the portion of R_{AB} that can be joined be f.

r3:= r2 * f.

We can first merge sort A's index file.

read: (Log_2 r1) x Bx1.

write:(Log_2 r1) x Bx1.

Then, merge sort R_{AB}.

read: (Log_2 r2) x B2.

write:(Log_2 r2) x B2.

Finally, we can compare the two sorted file to join them together.

read: Bx1 + B2.

write:= B3.

Total read : (1+Log_2 r1) x Bx1 + (1+Log_2 r2) x B2.

Total write: (Log_2 r1) x Bx1 + (Log_2 r2) x B2 + B3.

Note that if the join is to check the referential dependency, it does not require any write.

(ii). ς_3:= **JoinS** (ς_2, ς_3).

---------- Assumption ----------

ς_1 : r1, b1, B1.

ς_2 : r2, b2, B2.

ς_3 : r3, b3, B3.

Again, we first merge sort the two file ς_1, ς_2.

Then, we join them together in one pass of each file.

Total read : (1+Log_2 r1) x B1 + (1+Log_2 r2) x B2.

Total write: (Log_2 r1) x B1 + (Log_2 r2) x B2 + B3.

(C). **ProjectS**: ς_2:= **ProjectS** $_{P\_set}$ (ς_1).

---------- Assumption ----------

ς_1 : r1, b1, B1.

ς_2 : r2, b2, B2.

We can first merge sort ς1.

read: (Log_2 r1) x B1.

write:(Log_2 r1) x B1.

Then, we project and remove duplicate in one pass.

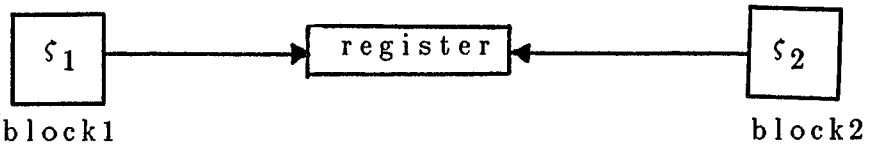
read: B1.

write:B2.

Total read: (1+ Log_2 r1) x B1.

Total write:(Log_2 r1) x B1 + B2.

(D). Predicate: $\mathbf{PS}_{\subseteq}(\varsigma_1,\varsigma_2)$.



---------- Assumption ----------

ς_1 : r1, b1, B1.

ς_2 : r2, b2, B2.

First, merge sort ς1.

read: $(\text{Log}_2\ r1) \times B1$.

write:$(\text{Log}_2\ r1) \times B1$.

Second, merge sort ς2.

read: $(\text{Log}_2\ r2) \times B2$.

write:$(\text{Log}_2\ r2) \times B2$.

Finally, run through both file once to check inclusion.

read: B1+B2.

write: 0.

VITA

The author, Herman Hai-Lou Lee, is the son of Wei-Chi and Shan-Fei Wu Lee. He was born on December 26, 1953 in Taipei, Taiwan.

He received his B.S. in Chemical Engineering from National Taiwan University in 1976.

In August, 1980, he was graduated from the West Virginia University with a M.S. in Chemical Engineering.

From 1980 to 1984, he worked for Vulcan Material Company as a Research and Development Process Engineer and is a Licenced Engineer in the State of Kansas since 1984.

He entered the Graduate School of Louisiana State University in August 1984. He is presently a candidate for the degree of Doctor of Philosophy in Computer Science.

He married Betty Wen-Bey Lee in 1981, and they have one son, Michael.

DOCTORAL EXAMINATION AND DISSERTATION REPORT

Candidate: HERMAN HAI-LOU LEE

Major Field: COMPUTER SCIENCE

Title of Dissertation: SEMANTIC CONSTRAINT MODELING IN DATABASE USING THE APPLICATIVE DATA LANGUAGE

Approved:

Peter P. Chen
Major Professor and Chairman

William Boyle
Dean of the Graduate School

EXAMINING COMMITTEE:

John A. Dawson III

J. Buck Jones

Leslie P. Jones

Odd K. Kf

George S. Tracy

Date of Examination:

DECEMBER 11, 1987