FORMALIZATION OF DATABASE SYSTEMS - AND A FORMAL DEFINITION OF IMS

DINES BJØRNER & HANS HENRIK LØVENGREEN

DEPARTMENT OF COMPUTER SCIENCE, TECHNICAL UNIVERSITY OF DENMARK
DK-2800 LYNGBY, DENMARK

ABSTRACT: *Drawing upon an analogy between Programming Language Systems and Database Systems we outline the requirements that architectural specifications of database systems must fulfill, and argue that only formal, mathematical definitions may satisfy these. Then we illustrate some aspects and touch upon some uses of formal definitions of data models and database management systems. A formal model of* IMS *will carry this discussion. Finally we survey some of the existing literature on formal definitions of database systems. The emphasis will be on constructive definitions in the denotational semantics style of the* VDM: Vienna Development Method. *The role of formal definitions in international standardization efforts is briefly mentioned.*

Computing Review CR82 Classification & Keywords

F.3.2 Semantics of Database Systems
D.3.1 Database System Formal Definitions & Theory
D.2.1 Specification Languages, Methodologies, Tools
D.2.0 Database Standards
H.2.1 Data Models
H.2.3 Data Description, Manipulation & Query Langs.
F.3.1 Specifying, Reasoning about & Verifying DBMSs

Abstraction Techniques, Functional & Logic Programming, Pre/Post conditions, Datastructure Invariants

CONTENTS

0. INTRODUCTION

There is a growing and increasingly more widespread interest in mathematical semantics definitions of various aspects of database systems. There is also a growing acceptance of such definitions in both international standardization work, in architectural design, as well as as a basis for development of correct implementations, and verification of correctness of uses of databases.

The aim of this invited paper is to make a larger fraction of the database community of both researchers and practitioners aware of these facts, of what it takes to construct and use such definitions, and what their aims are.

This introduction serves to delineate and otherwise relate the subject.

We start with an analogy. It compares notions from the field of (conventional) programming languages with notions from the field of databases.

(1) In the former we speak of programming languages. In the latter of data systems, i.e. of data models, and data definition, manipulation, & query languages (DDL, DML, resp. QL). We claim that the two sides relate: the data model (DM) aspects of Pascal-like languages, are the data type aspects: data structures (values) and their primitive operations. The data definition aspects of these programming languages are the data type and variable definitions, respectively declarations. Finally, the data manipulation & query aspects are the statement & expression constructs.

(2) This was the first step of our analogy. In our next step we relate programming language processors to database management systems, DBMS. By a language processor we understand either a compiler + runtime system, or an interpreter. The function of a language processor is to execute (input) programs in the presence of (input) data (incl. files, etc.). The purpose of a DBMS is to execute (input) commands (of DDL/DML/QLs) in the presence of stored (i.e. database) data.

(3) In our final step we relate programs + input data to input commands + databases. Programs correspond to commands, input data (primary input & input from files, etc.) correspond to databases.

(1) We shall now exploit the analogy. Previously, it was considered of utmost importance to secure the correctness of language processors for the following reason: all programs, including programs implementing DBMSs, on their way from conception to execution, would pass through a compiler or interpreter. The programs might have been believed to be correct, but if e.g. the compiler was wrong, then all would be wrong. From considerations of both program correctness and compiler/interpreter correctness arose the desire to construct precise definitions of programming languages. From these, it was expected, one could eventually prove cor-

rectness both of programs and of compilers. Now our dependence & reliance on database systems is such that we must require the same stringent requirements as to their precise description. Just like we define programming languages precisely, we must now require similar definitions of data models and the related (possibly abstract) DDLs, DMLs & QLs.

(2) In programming language processor, i.e. in interpreter, compiler and run-time system development we use the formal definition of the programming language. Thus we systematically derive language processors from language definitions. Correspondingly, we argue that DBMSs be rigorously derived from formal definitions of data models & their DDLs, DMLs & QLs.

(3) In program development we use the formal definition of the programming language to verify its correctness. Similarly we can use the formal definition of a data model & its DDL, DML & QL, to verify properties of any particular database and/or any particulare use, through a DML, DDL or QL command, of such a database.

We observe the following distinction: to specify a data model and its DDL, DML & QL (or: Programming Language) is one thing. To specify a database (resp.: program) is quite another thing. In this paper we focus on the former, but the techniques for the latter may be the same. Common to all specifications is their language, the meta-language. The implementation language for a DBMS (as derived from a data model + DDL/DML/QL specification) may be Pascal, whereas the implementation language for a database usually is a DDL/DML/QL.

The specification language of this paper is the *VDM* notation language; it is called *META-IV*, and is described very briefly in an appendix.

1. DEFINITION OBJECTIVES & REQUIREMENTS

We outline the requirements that users must put on database system definitions. We distinguish between requirements expected to be fulfilled by some software and the architectural definitions of the software functions: concepts & facilities. The latter fulfill the former. Requirements definition speak of the software, whereas definition requirements speak, at a meta-level, of how we express definitions including the way in which definitions express desired properties.

The objective of a definition of a data model & its DDL/DML/QL is that it specifies something for both the intended users & developers of the implied family of DBMSs. The definition is a legal contract between users & developers. The objective of a definition is also that it expresses the requirements put on the data model, i.e. the expectations laid down in uses thereof. We shall not further touch upon this relationship to requirements analysis & definition.

The objectives result in certain requirements. For it to formally fulfill the rôle of "legal contract" a definition must be consistent & complete, and uses of what the definition defines must be provably correct w.r.t. the specification. By a consistent definition we mean one which ascribes an un-ambiguous, precise, non-trivial semantics. By a complete definition we mean one which ascribes semantics to all applicable constructs. These properties can both be easily verified to hold (or not hold) for denotational, e.g. VDM-based, definitions.

For it to usefully fulfill this rôle a definition should furthermore not be unduly long, i.e. be reasonably short, and it should be comprehensible by the two parties concerned: those who write reference & introductory manuals for uses of what is defined, and those who develop implementations of what is defined. One could list, as a definition objective that the object which has been defined, in casu: the Data System is well-conceived: free from mis-conceptions, conceptually clean and with an optimum of notions; and that the defined data system has properties which are either transparently defined or relatively easy to ascertain from the definition.

A definition must be accessible to reasonably skilled software engineers, i.e. they should be able to effectively find their way into & "around" the internals of a definition. In addition to the definition being (de facto) correct, definition users must also believe it to be correct. "Correctness" of a definition, in addition to consistency and completeness, etc., also means that it is permissible. Thus desired non-determinism, quasi-parallelism, concurrency and resource usage optimization must be expressed by the definition.

It should not, per se, be a requirement or an objective that a definition be formal, let alone executable.

To summarize, the objectives & requirements of a definition can be itemized:

(i) legal contract between user & developer,

(ii) consistent & complete,

(iii) comprehensible & concise (short & precise),

(iv) accessible & referencable,

(v) correct & believed correct,

(vi) permissive - where appropriate,

(vii) suitable for user manual technical writers, implementors & validators

2. FORMAL DEFINITIONS & THEIR USES

We now argue that only formal definitions will satisfy the above-stated objectives & requirements.

By formal definition we mean a definition about which questions can be answered objectively. Mathematics then is our tool to achieve this. Several definitional styles are possible. They fall in two groups: the constructive models of either denotational or operational semantics, and the implicit definitions of either axiomatic or algebraic semantics.

Given a definition of the latter kind one is usually required to show that there exists a non-trivial model for there to exist any sensible realization. Current implicit definition styles achieve this model requirement at the expense of forcing rather low-level abstractions resulting in rather voluminous definitions. A drawback of algebraic definition styles, as contrasted to denotational ones is their apparent inability to cope with higher order functional abstractions.

Given suitably high-level abstractions denotational semantics today appear to satisfy most of the requirements put to definitions of sequential, or deterministic systems. The more permissive aspects can be expressed in either axiomatic or structured operational semantics [Gordon 81a].

We shall focus on the VDM approach to formalization of database Systems. VDM, for Vienna Development Methodology, is a denotational semantics based software development method, which, however, permits carefully controlled naïve set theoretic predicates to achieve a high degree of expressability. VDM is extensively documented in the literature: currently four books [Bjørner 78a, Jones 80a, Bjørner 80a, Bjørner 82b] in addition to many papers. For database-oriented introductions to VDM you may consult [Bjørner 80b].

The discussion which follows will not be a tutorial on how to use VDM for purposes of e.g. formal definitions, nor on how to read such VDM definitions. Instead we shall outline some of the methodological steps in constructing such definitions, or models as we shall henceforth call them. The discussion will be carried relative to the Formal Model of IMS given as an appendix.

Semantic Domains

— The Database Structure:

In architecting a new Data Model (or variant thereof) the most important thing to first decide upon is the semantic objects (or states): "that which we wish to speak about". In the case of IMS it was given by others, and our formalization is hence a recording of the results of a semantic analysis. We found that IMS states could be abstracted as shown, first in (1), subsequently, decomposing these into constituent objects, defined in (2), (9), and (14).

The object defined by formulae (1, 2, 9, 14) are representationally abstracted mathematical Domains. That is: we have abstracted away any implementation concerns to focus on what the user sees. The Domain equations define the class of all objects (usually too many, but see below).

Thus we do not believe in defining such object domains by displaying one or two "snapshot" pictures of "typical" such objects in the form of drawings with boxes and pointers, etc. Our major misgiving about such drawings is that they are only snapshots, too few to extrapolate the entire allowed class. Our subsidiary misgivings about drawings or tables of example database constellations is that one is not told exactly the semantics of the primitives used in the drawings, let alone their combination.

— The Database Invariants

But the domain definitions (1, 2, 9, 14) sometimes define too much. It is, in general, not possible to express, in the form of domain equations, all the internal consistency constraints which usually must hold within and between sub-parts of objects. These data model integrity constraints are instead expressed in the form of predicate functions called (data type or data structure) invariants. Invariants apply to objects of defined domains and are defined to hold only for those objects which are well-formed. The IMS invariants are defined in (4, 5, 6, 10, 11, 23).

— Auxiliary Database Notions

Finally, before dealing with data manipulation, we deal with a variety of auxiliary notions, usually functions of various kind — notions which are either used in establishing invariants, or, subsequently, in establishing the meaning of query/update commands.

These ancillary functions are illustrated by formulae (3, 7, 8, 12, 13, 15-22). The reader is encouraged to study these from this point of view.

o o o o

We find that the process of establishing what "the whole thing is all about", i.e. the semantic domains, their invariants and auxiliary notions, is the most crucial — and most rewarding, hence the most important. We find, in contrast, that many database treatments all too often begin by explaining conrete syntax, trying to cover the domain of applicable commands as completely as possible, and failing, invariably, to explain, to any acceptable depth, the semantic domains. Their structure only transpires indirectly from explications of command semantics.

We find that very many database proposals are conceptually constrained by unnecessary implementation concerns and by a lack of a suitable abstraction medium. We find that denotational modelling of data models very quickly brings one into interesting generalizations. The abstraction language of e.g. VDM relieves one from many unwanted clerical details, and thereby enables one to better exploit ones' mental capability. In short: A good, concise and well-founded notation is very important, also for architectural design work.

Syntactic Domains

— The Database Commands

In establishing the semantic domain the architect naturally had certain data manipulation and query capabilities in mind. Basic aspects of these are normally reflected in some of the auxiliary functions -- e.g. (8, 15-22). Thus the foundations upon which design and explication of the database commands have already been laid. And we can proceed to architecting these commands (24, 25, 27, 31, 39, 45, 50, 53).

— The Context Condition Constraints of Commands

The commands designate database objects by naming segment types and fields (25.0, 25.2, 25.3)*. In an actual database some of these names may not be, or yet have been, defined. Moreover, the commands may contain data values (24.4, 24.6, 25.2-3). These may not be of the kind described in the appropriate parts of the Catalogue (2.2, 2.7). As part of command elaboration one therefore have to check that only appropriate names and values are used (26, 28, 29, 30, 40, 46, 51, 54).

Semantic Functions

Finally we are ready to make precise the specific semantics of commands. Formulae (32-38, 44, 48-49) can be considered auxiliary functions used in explicating the denotations of several commands; with (41-43, 47, 52, 55) being the actual, main semantic functions.

o o o o

We have completed the denotational modelling of a non-trivial data model, the IMS. Granted, it is an abstraction of IMS, but the definitions illustrate what it takes to construct such a definition and what one obtains: an abstract "realization" - an easy and not too costly way of "playing around with and exercising" an entire spectrum of IMS facilities. Instead of a lot of national and natural language words one has something far more tractable: something that can be objectively scrutinized, something which can serve as a very precise departure point for estimates of, and actual, implementation, and for educational and training purposes.

The illustrated IMS definition is of a nature which can be implemented very quickly if one omits considerations of resource efficiency. A number of executable versions of the shown definition can be thought of. Either one has, once and for all, an interpreter for the meta-language used; or one transliterates the definition into for example ML (of LCF [Gordon 79a]) or SETL ([Schwartz 73a]). Such "proto-typing" is by many considered useful. We should like to point out, however, the important clarifying rôle the construction and the existence of a paper definition. In fact, we should like to urge that architectural definitions such as the one of the appendix, but for new concepts, new "inventions", be communicated, circulated, read and discussed for years before serious, constly attempts be made to implement them.

* NN.M refers to M'th line of formula NN, 0-origin.

3. RELATED WORK — An Annotated Bibliography

In this section we first survey some of our own work, in section 3.0, and then, in sections 3.1-3.7, that of others, all, however, restricted to the VDM viewpoint.

3.0 Bjørner et al.

The following references outline the general principles for applying denotational, specifically VDM modelling techniques to databases: [Bjørner 80b, Bjørner 80d, Bjørner 82cd]. The first two represent early versions of the latter two. [Bjørner 82c] describes relational, hierarchical and network data models. Both relational algebra (procedural) and predicate calculus (DSL-alpha, SEQUEL, SQL) query languages are modelled in detail. The latter, basically borrowed from [Nilsson 76a], formed the basis for [Hansen 80a]. The hierarchical data model is carefully formalized, first rather generally, then with a bias towards System 2000. The network data model formalization is oriented towards Bachmans ([Bachman 70a]) Data Structure Diagram, and is very general. [Bjørner 80b] "smoothtalks" the reader into understanding basics of VDM by first modelling very simple file system ideas; thus that paper may serve as a sufficient entry into being able to read VDM definitions. In [Bjørner 82c] it is not obvious how the abstractions relate to (i.e. injects into) database management systems for respecture data models. This relationship, except for the case of the relational model, is dealt with in [Bjørner 82d]. For the hierarchical model is shown the socalled hierarchical sequential access method (hsam), hierarchical direct access methods (hdam) with so-called "Child-twin" pointers respectively entire file pointers. For the network model only a single step of development is shown: one towards 'chained pointer' realizations, eventually leading to the 'current-of' and 'area' notions of the CODASYL/DBTG report [CODASYL 71a].

3.1 A. Hansal: A Model of PRTV [Hansal 76a]

This is the earliest known application of VDM to database specification. The actual work was carried out in the summer of 1974 when A. Hansal was an undergraduate student. The report gives a fairly complete and faithful model of the "Peterlee Relational Test Vehicle" (PRTV) — otherwise known as IS/1. PRTV is a relational algebra based DBMS actually implemented by the IBM UK Scientific Centre in the early 1970s.

3.2 J.F.Nilsson: Formalization & Realization of Database Systems [Nilsson 76a]

In this Ph.D.Thesis VDM modelling techniques are applied as a tool in both semantic analysis of im-known and novel relational database concepts, and in the synthesis of a relational data model proposal.

3.3 J.Lindenau: "A Descriptive Query-Language for the Network Data Model — with a Formal Semantics Definition in META-IV" [Lindenau 81a]

This M.Sc. Thesis is in German. In its first part (85 carefully annotated pages) is given a formal VDM-style model of the semantics of the CODASYL/DBTG proposed DML for the network data model. This definition faithfully models crucial data model aspects such as the currency notions, and constraints among data model components (such as record types, set types and currency pointers). It then models the semantics of 8 variants of the 'find' command, and the 'get' (2 variants), 'store' 'erase' 'modify' (5 variants), 'connect', 'disconnect' and 'move' commands. It is to be recommended that this entire part of the thesis be published in English.

In its second part a query language extension featuring relational algebraic terms is super-imposed on the system of part 1. The semantics of the new constructs are given in terms of their translation into "ordinary" DML constructs. Part 3 then documents an implemented translator.

3.4 Ths.Olnhoff: "Functional Semantics Description of Query Operations in a 3-level Data Model" [Olnhoff 81a]

This Ph.D.Thesis is in German. The 200 page work carefully analyzes the so-called ANSI/SPARC proposal for a 3-level external/conceptual/internal schema view of data bases. The specific model studied is the relational. The approach is that of using VDM/META-IV as a tool in this investigation. Thus the 'relations' between the external and the conceptual, and between the conceptual and the internal schemas are studied by synthesizing and studying the injection and abstraction (or retrieval) functions for both the data objects and the command operations.

3.5 E.Neuhold et al.

The work of section 3.4 also had a background in work "pre-dating" VDM-influence: [Biller 74a, 75a, 76a]. The joint papers with Olnhoff [Neuhold 80a, 81a] summarizes and extends the work mentioned in section 3.4. Our own work in applying VDM to data bases was partly prompted by a desire to simplify the treatments of the Biller et al. papers.

3.6 W.Lamersdorf et al.: Pascal/R

It was J.W.Schmidt of the Hamburg University who first suggested the idea of formalizing his very elegant relational extensions to Pascal. Initial work on this was done by W.Lamersdorf as part of a M.Sc. course. Two reports, and Lamersdorf's German Ph.D.Thesis, now documents this effort [Lamersdorf 80ab]. In one report the ideas of Pascal/R are carefully introduced, hand-in-hand with their formalization. The other report gives the complete formal definition. This work seems to have inspired the proposal that a possible ANSI 'Relational Data Model' standard also be formalized, in fact [X3/ANSI/SPARC 81a] exemplifies such a formal (VDM) definition.

3.7 B.S.Hansen & S.U.Palm: System/R — A Model of the PL/I Users Interface [Hansen 80a]

This is an internal, IBM Confidential, report of the IBM San Jose Research Laboratory, California, and of the Computer Science Department of the Technical University of Denmark. It represents about 4 man-months of relatively in-experienced undergraduate student work spread over half a year. Although based only on a single reference manual and no access to System/R designers nor a System/R itself, it remarkably accurately describes the multiple user, recovery based version of System/R as implemented and operated by the IBM San Jose Research Laboratory. Thus the model, besides defining SQL with views etc., employs a meta-process notion to capture, as abstractly and implementation-independent as possible, the notions of 'shared access' to data as well as 'transaction commits' and 'backouts'.

This is not the place to relate the very many, very interesting experiences obtained in the process of constructing this model, but only to say that in our opinion IBM would do the entire database community a most valuable service in making a ("correct") update of this definition publicly available.

It appears that the IBM Scientific Center in Heidelberg, W.Germany, is currently pursuing very interesting end-user database system architectural design work based on the above work.

4. CONCLUSION

We have presented three lines related to formalization of database systems: (1) rationales for doing so; (2) an actual 'realistic' model of crucial aspects of IMS, the IBM Information Management System, by far the most wide-spread DBMS of the 1970s; and (3) an annotated survey of related formalization work.

The work of e.g. Olnhoff & Neuhold point to a next "generation" of database formalization efforts: those of capturing the more recent proposals of the data base architects: researchers & designers, and the works of most of the researchers mentioned in section 3 bears promise to the hope that database proposals in the 1980s will be carried by mathematically clean formalizations.

REFERENCES & BIBLIOGRAPHY

[Biller 74a] H.Biller & E.J.Neuhold: *Formal View on Schema-Subschema Correspondence*, in: IFIP'74, N-H, 1974.

[Biller 75a] H.Biller & G.Glatthaar: *On the Semantics of Data Definition Languages*, GI-5, Jahrestagung, LNCS 34, 1975.

[Biller 76a] H.Biller: *On the Semantics of Data Bases: The Semantics of Data Manipulation Languages*, in: 'Modelling in Data Base Management Systems' (ed.G.M.Nijssen), IFIP TC-2 Working Conf., N-H, 1976.

[Bjørner 78a] D.Bjørner & C.B. Jones (eds.): *The Vienna Development Method: The Meta-Language*, LNCS 61, 1978.

[Bjørner 78b] D.Bjørner: *Programming in the Meta-Language -- A Tutorial* in: [Bjørner 78a].

[Bjørner 80a] D.Bjørner (ed.): *Abstract Software Specifications* LNCS 86, 1980.

[Bjørner 80b] D.Bjørner: *Formalization of Data Base Models* in: [Bjørner 80a], pp. 144-215.

[Bjørner 80c] D.Bjørner: *Formal Description of Programming Concepts: a Software Engineering Viewpoint*, MFCS '80, LNCS 88, pp. 1-21, 1980.

[Bjørner 80d] D.Bjørner: *Application of Formal Models*, in: 'Data Bases', INFOTECH Proceedings, Oct. 1980.

[Bjørner 81a] D.Bjørner: *The VDM Principles of Software Specification and Program Design*, in: 'Formalization of Programming Concepts', LNCS 107, pp. 44-74, 1981.

[Bjørner 82a] D.Bjørner & S.Prehn: *Software Engineering Aspects of VDM* Int'l.Seminar: 'Software Factory Experiences 2', Capri, Italy, N-H, 1982.

[Bjørner 82b] D.Bjørner & C.B.Jones: *Formal Specification and Software Development* Prentice/Hall Int'l., London 1982.

[Bjørner 82c] D.Bjørner & H.H.Løvengreen: *Formalization of Data Models* chapter 12 of [Bjørner 82b].

[Bjørner 82d] D.Bjørner: *Realization of Database Management Systems* chapter 13 of [Bjørner 82b].

[Bjørner 83*] D.Bjørner: *Software Architectures and Programming Systems Design*, approx. 1000 pages lecture notes, TUD, 1983.

[CODASYL 71a] *Data Base Task Group (DBTG), CODASYL 1971 Report* ACM, 1971.

[Gordon 79a] M.Gordon, R.Milner & C.Wadsworth: *Edinburgh LCF* LNCS 78, 1979.

[Hansal 76a] A.Hansal: *A Formal Definition of a Relational Data Base System*, IBM (Peterlee,UK) UKSC0080, June 1976.

[Hansen 80a] B.S.Hansen & S.U.Palm: *A Formal Model of System R*, TUD, Aug. 1980 (IBM Confidential).

[Hardgrave 72a] W.T.Hardgrave: *A Retrieval Language for TreeStructured Data Base Systems* in:'Informations Systems' COINS IV,ed.J.Tou, Plenum Press, pp.137-160, 1972.

[IBM a] IBM Corporation: *IMS VS version 1, Application Programmers Reference Manual* SH20-9026.

[Jones 77a] C.B.Jones: *Program Specification and Formal Development*, in: Int'l.Comp.Symp., ICS'77, North-Holland, pp 537-554, 1977.

[Jones 79a] C.B.Jones: *Constructing a Theory of a Data Structure as an aid to Program Development*, Acta Informatica 11, pp. 119-137, 1979.

[Jones 80a] C.B.Jones: *Software Development: A Rigorous Approach*, Prentice-Hall International, London 1980.

[Lamersdorf 80ab] W.Lamersdorf & J.W.Schmidt: *Semantic Definition of Pascal R* Reports 73-74, Inst. of Informatics, Univ. of Hamburg, July 1980.

[Lindenau 81a] J.Lindenau: *Eine Deskriptive Anfragesprache für das Netzwerk-Datenmodell mit formaler Definition der Semantik in Meta-IV*, M.Sc. Thesis (in German), Kiel Univ., Inst. für Informatik, March 1981, 175 pages.

[Louis 82a] G.Louis & A.Pirotte: *A Denotational Definition of the Semantics of DRC, A Domain Relational Calculus* VLDB Conf. 1982, these proceedings.

[Neuhold 80a] E. Neuhold & Ths. Olnhoff: *The Vienna Development Method (VDM) and its Use for the Specification of a Relational Data Base System*, IFIP'80 (ed.S.Lavington), N-H, 1980.

[Neuhold 81a] E.Neuhold & Th.Olnhoff: *Building Data Base Management Systems Through Formal Specifications*, LNCS 107, pp 169-209, 1981.

[Nilsson 76a] J.F.Nilsson: *Relational Data Base Systems: Formalization and Realization*, Ph.D. Thesis, TUD, ID 641, Sept.1976.

[Olnhoff 81a] Ths. Olnhoff: *Funktionale Semantikbeschreibung von Anfrageoperationen in einem dreischictigen relationaler Datenbanksystem*, Ph.D. Thesis, Stuttgart/ Hamburg Univ.,Inst.f.Informatik, May 1981,210 pgs.

[Owlett 77a] J.Owlett: *Deferring and Defining in Databases* in: 'Architecture & Models in DBMS's', Proc.IFIP Work. Conf. on Modelling in DBMS, N-H, 1977.

[Pirotte 82a] A.Pirotte: *A Precise Definition of Basic Relational Notions and of the Relational Algebra* ACM SIGMOD Record, to appear, 1982.

[Plotkin 81a] G.D.Plotkin: *Structured Operational Semantics* Lecture notes, Aarhus University Computer Science Department, Denmark, 1981.

[Schwartz 73a] J.T.Schwartz: *On Programming: The SETL Language* Courant Institute of Mathematics, New York University, N.Y., 1973.

[X3/ANSI/SPARC 81a] (eds.M.L.Brodie & J.W.Schmidt) *Relational Database Task Group Final Report* Doc. SPARC 81-690, 1981.

Reference abbreviations:

N-H:	North-Holland Publ., Amsterdam, The Netherlands.
LNCS:	Lecture Notes in Computer Science, Springer-Verlag, Heidelberg, W.Germany.
TUD:	Technical University of Denmark, Computer Science Department.

APPENDIX I: A Formal Model of IMS

A.1 Introduction

This appendix contains an abstract model which is intended to describe the most fundamental concepts of IBM's hierarchical database system, IMS (Information Management System).

The model covers only the most fundamental constructs of the data manipulation language DL/1. First of all, the catalogue and the data part is modelled, and then the semantics is given for the six fundamental commands: Get Unique, Get Next, Get Next within Parent, Insert, Delete, and Replace. However, only the most simple forms of these are treated.

The model is based primarily on the user manual of IMS [IBM a]. Readers familiar with IMS should be aware that the model is an abstraction and that the following features are not modelled:

- Command Codes
- "Advanced Features": Multiple Positioning,etc.
- Logical Databases
- Access Dependent Features
- Positioning in wrong calls
- "Wrap around" of position
  (start of database = end of database)
- Get-Hold calls
- Insert Rules other than FIRST

It is believed, anyway, that the model is a reasonable description of "elementary" IMS.

A.2 The Database Structure: Semantic Domains

The IMS Hierarchical Database is considered to consist of a Catalogue (Scheme), a Data Part, and current Positions:

```
HDB            ::  CTLG  DP  POS                       (1)
```

A.2.1 Catalogue

```
CTLG           ::  (SegmTp ⇸ SegmDescr) Ord            (2)
SegmDescr      ::  RecDescr  CTLG
RecDescr       ::  (FieldId ⇸ TYPE)  Seqinf
                   SearchFields
Seqinf         =   Unique | Multiple | NON-SEQ
Unique         ::  FieldId
Multiple       ::  FieldId
SearchFields   ::  FieldId-set
Ord            =   ...
TYPE           =   INT | ...
SegmTp         =   TOKEN
FieldId        =   TOKEN
```

The Catalogue defines a hierarchy of Segment Types and their associated Record Descriptors. Sibling Segment Types are ordered by the Order Component with which we assume a relation << indexed by the ordering:

```
type: <<   :  Ord → (SegmTp  SegmTp ⇸ BOOL)           (3)
```

The records of a segment type are described by the required fields and their types, whether the records must be ordered by a key field or not, and which fields may be used for selection.

No Segment Type may appear more than once in a hierarchy. We therefore have the following invariant on the catalogue:

```
inv-CTLG(mk-CTLG(sdm,ord)) ≜                            (4)
 (∀mk-SegmDescr(recd,ctlg') ∈ rngsdm)
    (inv-RecDescr(recd) ∧ inv-CTLG(ctlg'))
∧(∀stp₁,stp₂ ∈ domsdm)
    (stp₁≠stp₂ ⊃
        (let stps₁ = stypes(s-CTLG(sdm(stp₁)) in
         let stps₂ = stypes(s-CTLG(sdm(stp₂)) in
         stps₁∩stps₂ = {} ∧
         (stps1∪stps₂)∩domsdm = {}) )
∧inv-Ord(ord,domsdm)

inv-RecDescr(mk-RecDescr(tpm,seq,searchfs)) ≜       (5)
  searchfs ⊆ domtpm
∧ cases seq: (mk-Multiple(fid) → fid∈domtpm,
              mk-Unique(fid)   → fid∈domtpm,
              T                → true)
inv-Ord(,) ≜ ...                                       (6)

type: inv-CTLG:       CTLG               →  BOOL    (4)
type: inv-RecDescr:   RecDescr           →  BOOL    (5)
type: inv-Ord:        Ord SegmTp-set     →  BOOL    (6)
```

All the segment types of a (sub-)catalogue are found by:

```
stypes(mk-CTLG(sdm,)) ≜                                 (7)
  domsdm ∪
    union{stypes(s-CTLG(sdm(stp))) | stp∈domsdm}
type:  CTLG → SegmTp-set
```

Since the segment types are unique within the catalogue, it is often convenient to have direct access the associated record description:

```
get-RecDescr(stp,mk-CTLG(sdm,)) ≜                         (8)
   if stp ∈ domsdm
      then s-RecDescr(sdm(stp))
      else (let stp' ∈ domsdm be s.t.
                       stp∈stypes(s-CTLG(sdm(stp'))) in
            get-RecDescr(stp,s-CTLG(sdm(stp')))
type:  SegmTp  CTLG ⥲ RecDescr
pre:   inv-CTLG(ctlg) ∧ stp∈stypes(ctlg)
```

A.2.2 Data Part

```
   DP        =  SegmTp →m SEGMENT*                        (9)
   SEGMENT   :: Record  DP
   Record    =  FieldId →m VAL
   VAL       =  Intg | ...
```

The Data Part associates a list of segments to each segment-type. Each segment consists of a Record and an associated sub-data-part. Thereby the Records are arranged in a hierarchy where the records of a segment type under a given parent are ordered by the list. A record just associates a value to each Field.

A Data Part of an HDB must follow the structure prescribed by the catalogue:

```
inv-DP(dp)mk-CTLG(sdm,) ≜                                 (10)
   domdp = domsdm
  ∧(∀stp∈domdp)
      (let segml = dp(stp) in
       let mk-SegmDescr(rd,ctlg') = sdm(stp) in
       let mk-RecDescr(tpm,seq,)  = rd        in
        (∀mk-SEGMENT(rec,dp') ∈ elemssegml)
           (inv-Record(rec,tpm) ∧ inv-DP(dp')ctlg')
       ∧check-sequence(segml,seq))
type: DP → (CTLG ⥲ BOOL)
pre:  inv-CTLG(ctlg)
```

```
inv-Record(rec,tpm) ≜                                     (11)
   domrec=domtpm ∧
   (∀fid∈domrec)(type-of(rec(fid))=tpm(fid))
type: Record (FieldId →m TYPE) → BOOL
```

```
type-of(val) ≜ (is-Intg(val) → INT,...)                   (12)
type: VAL → TYPE
```

The following function will check that the records of a segment list are ordered according to their key field. In IMS multiple records with the same key may be allowed.

```
check-sequence(segml,seq) ≜                               (13)
  seq=NON-SEQ
 ∨(∀i,j∈indsegml)
    (let keyf = s-FieldId(seq) in
     let ikey = s-Record(segml[i])(keyf),
         jkey = s-Record(segml[j])(keyf) in
    ((i<j) ⊃ cases seq:
                (mk-Unique()   → ikey<jkey,
                 mk-Multiple() → ikey≤jkey)))
```

A.2.3 Positioning in the Data Base

```
 POS  = (s-current:[Path]  s-parent:[Path])
 Path = (SegmTp  N1)*                                     (14)
```

The IMS system maintains two positions in the database, each one identifying a unique segment. The current position (roughly speaking) identifies the record last accessed, and the parent position identifies the segment under which the Get Next Within Parent command may search.

A segment and its associated record is uniquely identified by a so-called Path, which gives the segment type and an index in the associated list of segments for each level on the way down from the root data part to the segment. An empty Path may be thought of as denoting an imaginary "system segment" which cannot be accessed.

If the current Path is missing *(nil)* the current position is at "the end of the Data Base". If the Parent Path is missing, no records can be selected by "Get Next Within Parent".

Now there follow some useful Path operations to be used later in the definitions of the commands. Firstly, all the possible paths in a data part.

```
all-paths(dp) ≜                                           (15)
   if dp=[]
     then {<>}
     else {<(stp,i)>^p |
                   stp∈domdp ∧ i∈inddp(stp) ∧
                   p∈all-paths(s-DP(dp(stp)[i]))}
type: DP → Path-set
```

Retrieval of the subdatapart, and the record denoted by a path:

```
get-DP(path,dp) ≜                                         (16)
  cases path:
    (<>                → dp,
     <(stp,i)>^path' →
                   get-DP(path',s-DP(dp(stp)[i])))
type: Path DP ⥲ DP
pre:  path∈all-paths(dp)
```

```
get-Record(path,dp) ≜                                     (17)
  (let path'^<(stp,i)>   = path              in
   let dp'               = get-DP(stp,dp)    in
   s-Record(dp'(stp)[i]))
type: Path DP ⥲ Record
pre:  path∈all-paths(dp)\{<>}
```

Note that the *get-DP* extends to the "system segment" *(<>)*. The following function tests whether a path denotes a segment below the segment denoted by a parent path. If the parent path is missing no segment can be below it.

```
beneath(path,parent) ≜                                    (18)
   parent≠nil ∧
   is-prefix(parent,path) ∧ lenpath>lenparent
type: Path  [Path] → BOOL
```

A.2.4 Sequencing in the Data Base

All the records of the data base are ordered in a so-called hierarchical sequence corresponding to a parent first, left-to-right traversal of the data part tree. In our model this ordering is reflected by an ordering on the unique path identifications of the records. In this way, the ordering becomes a lexicographical ordering on the paths. Since the ordering depends on the ordering of segment types at each level, the catalogue is needed in the following function, determining the ordering:

```
precedes(p1,p2)mk-CTLG(sdm,ord) ≜                              (19)
  (p2=<> → false,
   p1=<> → true,
   T →(let <(stp1,i)>^p1' = p1 in
       let <(stp2,j)>^p2' = p2 in
       (stp1≠stp2 → stp1 <<ord stp2,
        i≠j       → i<j,
        T         →
               precedes(p1',p2')s-CTLG(sdm(stp1)))))
type: Path  Path → (CTLG ⥲ BOOL)
pre:  (∃dp∈DP)(inv-DP(dp)ctlg) ∧
      {p1,p2} ⊆ all-paths(dp)
```

For later use, we define a few operations using the sequence concept.

```
first-path(ps)ctlg ≜                                           (20)
   if ps={}
      then nil
      else (Δp∈ps)(∀p'∈ps\{p})(precedes(p,p')ctlg)
type: Path-set ⥲ (CTLG ⥲ [Path])
```

```
follows(p1,p2)ctlg ≜                                           (21)
  (p2=nil → false,T → precedes(p2,p1)ctlg)
type: Path  [Path] ⥲ (CTLG ⥲ BOOL)
```

```
next-path(p)(ctlg,dp) ≜                                        (22)
   first-path({p' | p'∈all-paths(dp) ∧
                    follows(p',p)ctlg})
type: [Path] ⥲ (CTLG  DP ⥲ [Path])
```

A.2.5 Well-formedness of the Data Base

At the end of our treatment of the Data Base Structure we formulate the global conditions to be met by the Data Base:

```
inv-HDB(ctlg,dp,pos) ≜                                         (23)
   inv-CTLG(ctlg)
  ∧inv-DP(dp) ctlg
  ∧(let (current,parent) = pos in
   (current=nil ∨ current∈all-paths(dp))
  ∧(parent=nil  ∨ parent∈all-paths(dp)\{<>}))
type:  HDB → BOOL
```

A.3 Data Manipulation

In this section we define the semantics of the six fundamental commands of IMS: Get Unique, Get Next, Get Next within Parent, Insert, Delete, & Replace. The full abstract syntax and distribution functions are given below:

A.3.1 Syntactic Domains

```
Cmd    =  Gu | Gn | Gnp | Isrt | Dkt | Repl                    (24)
Gu     :: [Ssa+]
Gn     :: [Ssa+]
Gnp    :: [Ssa+]
Isrt   :: Ssa+  Record
Dict   :: ()
Repl   :: Record

Ssa    =  (SegmTp  [Qual])                                     (25)
Qual   =  Eq | ...
Eq     :: FieldId  VAL
Record =  FieldId ⥲m VAL
```

Precondition:

```
pre-Cmd[cmd]hdb ≜                                              (26)
  cases cmd:
    (mk-Gu(°) → pre-Gu(cmd)hdb, ... → ... )
```

The definition of the semantics of the commands is divided into three parts. First we discuss the common concept of Sequential Search Arguments (ssa's). We then treat the data retrieval commands (the Get's), and finally the data modifying commands. For each command we define the conditions in relation to the Data Base to be satisfied before application of the command, and the interpretation function defining the semantics of the command.

Note that the special Get-Hold commands are not modelled as they are almost logically equivalent to the Get calls except that they indicate that the succeding command may be a replacement or deletion.

A.3.2 Segment Search Arguments

```
Ssa-list  =  [Ssa+]                                            (27)
Ssa       =  (SegmTp  [Qual])
Qual      =  Eq | ...
Eq        :: FieldId  VAL
```

The purpose of the *Ssa-list* is to determine a set of segments of a given type by giving qualifications which must be satisfied either by the segments of the type themselves, or by their ancestors. The qualification for a segment type is given by a Sequential Search Argument (Ssa). In IMS, however, not all levels on the path to the desired segment type need to be given in the list. In this case the system will assume so-called implied Ssa's (see below). Also, the list may be totally left out, in which case all segments of the data base are selected. The qualification may have many forms in IMS. Since these are not our primary concern we only cover the case where a field in a record is required to have a certain value. The qualification may be left out, in which case all segments of the type apply.

For an *Ssa-list* to be valid in the context of a HDB, the Ssa's of the list, taken from left to right, must "lie" on a path from the root to the desired segment type, given by the last Ssa:

```
pre-Ssa-list(ssal)mk-HDB(ctlg,,) ≜                          (28)
  (ssal=nil  → true
   T         → pre-Ssa-list'(ssal)ctlg)
type: [Ssa+] → (HDB ⥲ BOOL)
```

```
pre-Ssa-list'(ssal)mk-CTLG(sdm,) ≜                          (29)
 cases ssal:
 (<>                       →  true,
  <(stp,qual)>⁀ssal' →
    if stp∈domsdm
     then
      (let mk-SegmDescr(recd,ctlg') = sdm(stp) in
       (qual=nil ∨ pre-Qual(qual)recd) ∧
        pre-Ssa-list'(ssal')ctlg')
     else
      (let stps = {stp' | stp'∈domsdm ∧
                  stp∈stypes(s-CTLG(Sdm(stp'))} in
       cases stps:
       ({}      → false,
        {stp'} →
            pre-Ssa-list'(ssal)s-CTLG(sdm(stp'))))
type: Ssa* → (CTLG ⥲ BOOL)
```

```
pre-Qual(qual)mk-RecDescr(tpm,,srchfs) ≜                    (30)
   cases qual:
   (mk-Eq(fid,val) → fid∈srchfs ∧
                     tpm(fid)=type-of(val),
    ... )
type: Qual → (Recdescr ⥲ BOOL)
```

The function *type-of: VAL → TYPE* is assumed to be given.

For an *Ssa-list* to be used, the missing levels must be supplied by the system. Such *Ssa's* are qualified by a special qualification indicating that the level was implied. We therefore introduce completed *Ssa-lists* as lists of:

```
   Ssa' = (SegmTp [Qual|IMPL])                               (31)
```

Now the *Ssa-list* may be completed by:

```
complete(ssal)mk-CTLG(sdm,) ≜                               (32)
cases ssal:
(<>                 →  <>,
 <(stp,q)>⁀ssal' →
   if stp∈domsdm
    then <(stp,q)>⁀complete(ssal')s-CTLG(sdm(stp))
    else (let stp' be s.t.
                stp∈stypes(s-CTLG(sdm(stp'))) in
          <(stp',IMPL)>⁀
                complete(ssal)s-CTLG(sdm(stp'))))
type: Ssa* ⥲ (CTLG ⥲ Ssa'*)
```

An *Ssa-list* may be evaluated in two ways. We give the simple one first:

```
eval-Ssa-list(ssal)(ctlg,dp) ≜                              (33)
   if ssal=nil
    then all-paths(dp)
    else (let ssal' = complete(ssal)ctlg in
          search(ssal',dp))
type: [Ssa+] ⥲ (CTLG  DP ⥲ Path-set)
```

```
search(ssal,dp) ≜                                           (34)
  cases Ssal:
  (<>                    →  {<>},
   <(stp,q)>⁀ssal' →
       {<(stp,i)>⁀p |
          i∈inddp(stp) ∧
          (let mk-SEGMENT(rec,dp') = dp(stp)[i] in
          (q=nil ∨ q=IMPL ∨ satisfies(rec,q)) ∧
           p∈search(ssal',dp'))})
type: Ssal'*  DP ⥲ Path-set
```

```
satisfies(rec,q) ≜
   cases q:  (mk-Eq(fid,val) → rec(fid)=val,    ... )
type: Record  Qual ⥲ BOOL
```

We see that in this simple case, implied *Ssa's* are treated like unqualified *Ssa's*.

In the *Get Unique* and *Insert Commands*, the implied *Ssa's* must be treated according to some special rules which we quote from [IBM a]:

"(1) If the prior call established position on a segment type that the current call is using as an implied segment type, an SSA qualified with current position is assumed at that level. (This is true even if the segment has nonunique keys.) If a parent level qualified SSA is provided for other than the parent's current position, an unqualified SSA is assumed by DL/I for all missing levels below that parent.

(2) If the prior call did not establish position on any segment type implied in the current call, then DL/I assumes an unqualified SSA at that level."

It does not seem quite clear how to interpret these rules. One idea would be to follow the current path "as long as possible", but it appears to be too restrictive. Instead, the interpretations formalized below seem to conform better to the rules.

The idea is to try how far the current path can be used still fulfilling the qualifications of the completed *Ssa-list* treating implied *Ssa's* as u nqualified. This will result in a prefix of the current path. From this prefix we take the part down to, and including, the last implied level, and use this part as the first part of the selected path where the rest is found by the usual search given above.

```
eval-Ssa-list-P(ssal,cur)(ctlg,dp) ≜                        (36)
  if ssal = nil
   then all-paths(dp)
   else (let ssal' = complete(ssal)ctlg in
          if cur = nil
           then search(ssal',dp)
           else search-along-path(ssal',cur,dp))
type: [Ssa+] [Path] ⥲ (CTLG  DP ⥲  Path-set)
```

```
search-along-path(ssal,cur,dp) ≜                              (37)
 (let j = try-path(ssal,cur,dp) in
  let k = max({ k'| 1≤k'≤j ∧
                       ssal[k']=(,IMPL)}∪{0} in
  let cur' = <cur[i]  | iεindcur  ∧ i≤k>    in
  let ssal'= <ssal[i] | iεindssal ∧ i>k>    in
  let dp'  = get-DP(cur',dp)                in
  { cur' ^ p | pεsearch(ssal',dp') } )
type: Ssa+  Path  DP  →  Path-set
```

```
try-path(ssal,cur,dp) ≜                                       (38)
 if ssal=<> ∨ cur=<>
  then 0
  else (let <(stp,q)>^ssal'      = ssal          in
        let <(stp',i)>^cur'      = cur           in
        let mk-SEGMENT(rec,dp')  =  dp(stp')[i]  in
        if stp=stp'∧(q=nil∨q=IMPL∨satisfies(rec,q))
         then 1 + try-path(ssal',cur',dp')
         else 0)
type: Ssa* Path DP ⥲ N0
```

Try-path will return the length of the longest prefix of the current path that satisfies the qualifications in the *Ssa-list*.

A.3.3 Retrieval Commands

IMS has three commands for data retrieval:

```
Gu   :: [Ssa+]                                                (39)
Gn   :: [Ssa+]
Gnp  :: [Ssa+]
```

Get Unique retrieves the first record in the hierarchical sequence which is selected by the *Ssa-list*. *Get Next* retrieves the first record when starting at the current position which is selected by the *Ssa-list*. *Get Next within Parent* works as *Get Next* except that the record must be a descendant of the record denoted by the actual parent position.

All *Get* commands will set the current position to the record retrieved, but only *Get Unique*, and *Get Next* will change the parent position.

Precondition:

For all *Get* commands, the *Ssa-list* must be valid:

```
pre-"GET"(mk-"GET"(ssal))hdb ≜                                (40)
   pre-Ssa-list(ssal)hdb
type: "GET" ⥲ (HDB ⥲ (HDB ⥲ BOOL)
"GET": Gu,Gn,Gnp
```

Interpretation:

```
int-Gu(mk-Gu(ssal))mk-HDB(dp,ctlg,pos) ≜                      (41)
 (let (cur,) = pos in
! let paths = eval-Ssa-list-P(ssal,cur)(ctlg,dp) in
  let p     = first-path(paths)ctlg             in
  let (pos',res) =
              (p=nil → (pos,NOT-FOUND),
               T     → ((p,p),get-Record(p,dp))  in
  (mk-HDB(dp,ctlg,pos'),res))
```

```
int-Gn(mk-Gn(ssal))mk-HDB(dp,ctlg,pos) ≜                      (42)
 (let (cur,) = pos in
  let paths  = eval-Ssa-list(ssal)(dp,ctlg)          in
  let paths' = {p | pεpaths ∧
!                   follows(p,cur)ctlg}              in
  let p      = first-path(paths') ctlg               in
  let (pos',res) =
              (p=nil → (pos,NOT-FOUND),
               T     → ((p,p),get-Record(p,dp)) in
  (mk-HDB(dp,ctlg,pos'),res))
```

```
int-Gnp(mk-Gnp(ssal))mk-HDB(dp,ctlg,pos) ≜                    (43)
 (let (cur,parent) = pos in
  let paths = eval-Ssa-list(ssal)(dp,ctlg)           in
  let paths'= {p | pεpaths ∧
!                  follows(p,cur)ctlg ∧
!                  beneath(p,parent)}                in
  let p     = first-path(paths')ctlg                 in
  let (pos',res)   =
          (p=nil → (pos,NOT-FOUND)
!          T     → (p,parent),get-Record(p,dp))) in
  (mk-HDB(dp,ctlg,pos),res))

type: "GET" → (HDB ⥲ HDB  (Record | NOT-FOUND))
```

The exclamation marks indicate the points where the definitions differ. Note that many textbooks let you have the impression that *Get Unique* uses the simple form of *Ssa* evaluation; this is not the case!

A.3.4 Modification Commands

IMS has three commands which may modify the data part of the data base. This section starts with the definition of a common function, then follows each of the commands separately, and at the end we give some auxiliary functions.

Modification:

Common to the modification commands is that they may change part of the Data Part of the Data Base. The following function performs the task of inserting a new Sub-Data-Part at a given position.

```
modify(parent,subdp)dp ≜                                      (44)
 cases parent
 (<>                  → subdp,
  <(stp,i)>^parent' →
     (let segml                = dp(stp)  in
      let mk-SEGMENT(rec,dp') = segml[i] in
      let dp" = modify(parent',subdp)dp' in
      dp + [stp → segml+[i↦mk-SEGMENT(rec,dp")]]))
type: Path  DP ⥲ (DP ⥲ DP)
```

Insertion:

```
   Isrt  :: Ssa+  Record                                      (45)
```

The *Insert* command inserts a record at the position indicated by the *Ssa-list*. The last element of the list indicates the type of the record, and must be unqualified:

```
pre-Isrt(mk-Isrt(ssal,rec))mk-HDB(ctlg,dp,)≜         (46)
   pre-Ssa-list'(ssal)ctlg
  ∧(let ssal'⌢<(stp,q)> = ssal in
    q = nil ∧
    inv-Record(rec,get-RecDecr(stp,ctlg)))
type: Isrt → (HDB ⥲ BOOL)
```

```
                                                      (47)
int-Isrt(mk-Isrt(ssal,rec))mk-HDB(ctlg,dp,pos)≜
 (let ssal'⌢<stp,nil>  = ssal in
  let (cur,parent)     = pos  in
  let paths  = eval-Ssal-P(ssal,cur)(dp,ctlg)      in
  let inspos = first-path(paths)                   in
  trap exit  with  (mk-HDM(dp,ctlg,pos),FAILED)    in
  if inspos=nil
   then exit
   else (let subdp       = get-DP(inspos,dp)       in
         let (subdp',i) =
                    insert-rec(stp,rec,ctlg)(subdp) in
         let dp'      = modify(inspos,subdp')dp    in
         let current'= inspos⌢<(stp,i)>            in
         let parent' =
            adjust-insertion(inspos,stp,i)parent   in
         let pos'     = (cur',parent')             in
         (mk-HDB(ctlg,dp',pos'),SUCCEEDED)))
type: Isrt ⥲ (HDB ⥲ (HDB  (FAILED|SUCCEEDED)))
```

The *Int-Isrt* function first identifies the subdata part in which the record is to be inserted. Then the record is inserted in this sub data part which is again inserted in the whole data part. Finally, the positions are modified to reflect the insertion.

```
find-position(segml,rec,fid) ≜                        (48)
  if segml=<> ∨ s-Record(hdsegml)(fid)≥rec(fid)
     then 1
     else 1 + find-position(tlsegml,rec,fid)
type: SEGMENT*  Rec  Fid  →  N_1
```

```
insert-rec(stp,rec,ctlg)dp ≜                          (49)
 (let mk-RecDescr(,seq,) = get-RecDescr(stp,ctlg)in
  let segml              = dp(stp)                in
  let i =
    cases seqinf:
    (NON-SEQ         → 1,
     Unique(fid)    →
         if (∃j∈indsegml)
              (s-Record(segml[j](fid)=rec(fid))
            then exit
            else findposition(segml,rec,fid),
     Multiple(fid) →
                 find-position(segml,rec,fid)  ) in
  let segml' = insert-elem(mk-SEGMENT(rec,[]),i) in
  (dp + [stp → segml'],i))
type: SegmTp  Record  CTLG ⥲ (DP ⥲ DP)
```

The *insert-rec* function inserts the record in the subdata part under the given segment type and at the right position according to its key value.

Delete:

```
    Dlet  ::  ()                                      (50)
```

The *Delete* command removes the record denoted by the current position. The "system record" can, of course, not be deleted.

```
pre-Dlet(mk-Dlet())mk-HDB(,,(cur,)) ≜                 (51)
   cur≠nil ∧ cur≠<>
type:Dlet → (HDB ⥲ BOOL)
```

```
int-Dlet(mk-Dlet())mk-HDB(ctlg,dp,pos) ≜              (52)
 (let (cur,parent) = pos in
  let del-parent⌢<(stp,i)>= current             in
  let subdp    = get-DP(parent,dp)              in
  let segml    = subdp(stp)                     in
  let subdp'   = subdp +
                 [stp → remove-elem(segml,i)]   in
  let dp'      = modify(parent,subdp')dp        in
  let current'= next-path(cur)(ctlg,dp)         in
  let parent' =
      adjust-removal(del-parent,stp,i)parent in
 (mk-HDB(ctlg,dp',(current',parent')))
type: Dlet ⥲ (HDB ⥲ HDB)
```

First, the subdata part in which the current record is situated is found, the record deleted and the data part updated. Finally, the positions are adjusted.

Replace:

```
    Repl  ::  Record                                  (53)
```

The *Replace* command updates some fields of the current record. The record provided must be sub-record with values of right type, and the key field must not be changed:

```
pre-Repl(mk-Repl(rec))mk-HDB(ctlg,dp,(cur,))≜  (54)
  cur≠ nil ∧ cur ≠ <>
 ∧(let cur'⌢<(stp,)>            = cur in
   let mk-RecDescr(tpm,seq,) =
                           get-RecDescr(stp,ctlg) in
   domrec ⊆ domtpm
  ∧(∀fid∈domrec)(type-of(rec(fid)) = tpm(fid))
  ∧cases seq:
     (mk-Unique(fid)   → fid∉domrec,
      mk-Multiple(fid) → fid∉domrec,
      T                → true          ))
```

```
int-Repl(mk-Repl(rec))mk-HDB(ctlg,dp,pos) ≜      (55)
 (let (rpos⌢<(stp,i)>,) = pos    in
  let subdp = get-dp(rpos,dp)     in
  let segml = subdp(stp)          in
  let mk-SEGMENT(rec',dp") = segml[i]                  in
  let segm   = mk-SEGMENT(rec' + rec,dp")              in
  let subdp'= subdp + [stp → segml + [i→ segm ]] in
  let dp'    = modify(parent,subdp)dp                  in
  mk-HDB(ctlg,dp',pos))
type:Repl ⥲ (HDB ⥲ HDB)
```

A.3.5 Adjustment of Parent Path

After insertion or deletion of segments the current and parent paths may no longer be valid. The current path is changed explicitly as shown in the formulas, whereas the parent path must be adjusted if it passes through the modified subdata-part.

```
adjust-insertion(inspos,stp,i)parent ≜                    (56)
  (parent=nil → nil,
   ¬beneath(parent,inspos)
              → parent,
    T         →
      (let rest =
           <parent[j] | leninspos<j≤lenparent> in
       let <(stp',j)>^rest' = rest                 in
       (stp≠stp' → parent,
        j<i      → parent,
        T        → ins-pos^<(stp',j+1)>^rest'))
type: Path  SegmTp  N1 ⥲ ([Path] ⥲ [Path])
```

```
adjust-deletion(delpos,stp,i)parent ≜                     (57)
  (parent=nil → nil,
   ¬beneath(parent,delpos)
              → parent
   T          →
      (let rest =
           <parent[j] | lendelpos<j≤lenparent> in
       let <(stp',j)>^rest' = rest                 in
       (stp≠stp' → parent,
        j=i      → nil,
        j<i      → parent,
        j>i      → del-pos^<(stp',j-i)>^rest'))
type: Path SegmTp N1 ⥲ ([Path] ⥲ [Path])
```

Note that if a segment of the parent path is deleted *(j=i)* the parent path is set to undefined *(nil)*.

A.3.6 Auxiliary List Functions

Insertion of an element just before the i'th position of a list:

```
insert-elem(a,al,i) ≜                                     (58)
  if i=1
    then <a>^al
    else hdal^insert-elem(a,i-1,tlal)
type: A  A*  N1  ⥲  A+
pre : i∈{1:lenal+1}
```

Removal of the i'th element of a list:

```
remove-elem(al,i) ≜                                       (59)
  if i=1
    then tlal
    else hdal^remove-elem(i-1,tlal)
type: A+  N1  ⥲  A*
pre : i∈indal
```

Test whether a list is a prefix of another:

```
is-prefix(l1,l2) ≜ (∃l'∈A*)(l2=l1^l')                     (60)
type: A*  A* → BOOL
```

APPENDIX II: Meta-Language *(META-IV)* Survey

DATA TYPES

INTG, N_0, N_1 Integers and Natural Numbers (larger than or equal to *0* respectiuvely *1*) with the usual operators: $+$, $-$, $\times$, $=$, $\neq$, $<$, $\leq$, etc.

BOOL Boolean truth values: *true* and *false* with the usual operators: $\wedge, \vee, \neg, \supset$. Predicate expressions involve use of existensial, $\exists$, unique existensial, $\exists!$, and universal, $\forall$, quantification.

QUOT Quotations are enumerated atomic objects, e.g. FAILED, SUCCEEDED, NOT-FOUND, denoting themselves. Only operators are: = and $\neq$.

TOKEN Non-enumerated atomic objects denoting themselves. Only operators are: = and $\neq$.

Sets Finite sets of "finite" (e.g. not functional) objects: *-set* (suffix) applied to domain *A* yields domain of finite subsets of domain *A*. Set forming expressions are: $\{ a_1,a_2, \ldots ,a_n \}$ and $\{ a \mid P(a) \}$: the set of elements satisfying *P*. Usual operators: $\cup, \cap, \subset, \subseteq, \in, \setminus$, *card*, etc. Empty set $\{\}$.

Tuples Finite length sequences, or lists, of objects: * and $^+$ (suffix) applied to domain *A* yields domain of tuples whose elements are *A* objects, * also generates the empty, <>, tuple, $^+$ does not. Tuple forming expressions are: $<a_1,a_2, \ldots,a_n>$ and $<f(i)|P(i)>$: the tuple of all those $f(i)$ elements, ordered as by ordering of *i*, satisfying *P*. List operators: *hd*, *tl*, *len*, $[\circ]$, ^, *elems*, *inds* denote "head", "tail", length, selection, concatenation, elements and indices. Also (in)equality: $(\neq)$ =.

Maps Finite domain functions (from *A* into *B*) denoted by: $A \underset{m}{\rightarrow} B$, with *A* elements being simple, i.e.non-functional finite objects. Map forming expressions are: $[a_1 \to b_1, a_2 \to b_2, \ldots ,a_n \to b_n]$ and $[d(x) \to r(y) \mid P(x,y)]$: the map from a_i to b_i, respectively $d(x)$ to $r(y)$ for all $P(x,y)$. Operators: *dom*, *rng*, $\cup, +, \setminus, ()$, $^\circ$ denote: domain, co-domain (range), extension, redefinition/override, restriction, application and composition. Also (in)-equality: $(\neq)$ =. Empty map: [].

Functions Lambda-functions defined either as $f(a) \triangleq$ *expression* or $\lambda a.$*expression*. Domains of total, or partial *A* to *B* functions written: $A \to B$, respectively $A \tilde{\to} B$. Only operation is application: ( ).

Trees Domain of *A*-named, resp. anonymous, trees of B_i objects defined by: $A :: B_1\ B_2 \ldots B_N$, & $(B_1\ B_2 \ldots B_N)$; which defines named, resp. anonymous, tree constructor (decomposer) functions: *mk-A*(...), resp. (...), and selector functions: *s-*B_1, *s-*B_2, ... *s-*B_N.

ABSTRACT SYNTAX

(or Domain Equations) are of either the *A=D-expression* or *A::D-expression* form, where *D-expression* may involve the set, tuple, map, function, or tree domain constructing operators: *-set*, *, $^+$, $\underset{m}{\rightarrow}$, $\to$, $\tilde{\to}$ respectively *::* and (...). Additional domain operators are: | (non-discriminated union), and [...] (optional domain).

LANGUAGE CONSTRUCTS

META-IV in addition to conventional, applicative constructs (*if then else* etc.) makes heavy use of the following:

(let id = expr in body)

Landin sugar for *(λid.body)(expr)*: *id* bound to all free occurrences in *body* and denoting substitution of *expr* for all these. Multiple, potentially mutually recursive *let* definitions are common.

(let mk-A(x,y, ... ,z) = atree in body)

decomposes *atree* into its constituent components. Otherwise as above.

$(b_1 \rightarrow e_1, b_2 \rightarrow e_2, \ldots, b_n \rightarrow e_n)$

McCarthy (LISP) conditional, b_n may be T denoting *true*.

cases $e: (v_1 \rightarrow e_1, v_2 \rightarrow e_2, \ldots\ldots, v_n \rightarrow e_n)$

Ordinary *cases* construct. Often used with v_i being $mk\text{-}B_i(\ldots)$ and then denoting decomposition of e-tree with ..... identifiers being bound in e_i.

$(\iota id \in S)(P(id))$

Unique descriptor expression: the unique object, in *S*, which satisfies *P*, undefined if not unique.

f(mk-A(x,y, ... ,z)) $\underline{\Delta}$ *body*

Function definition equivalent to:

f(atree)$\underline{\Delta}$
  (let mk-A(x,y,...,z)=atree in
   body)

ACKNOWLEDGEMENT

We gratefully acknowledge the work done by Bo Stig Hansen, Jan Storbank Pedersen and Lennart Schulz on earlier versions of the IMS model.