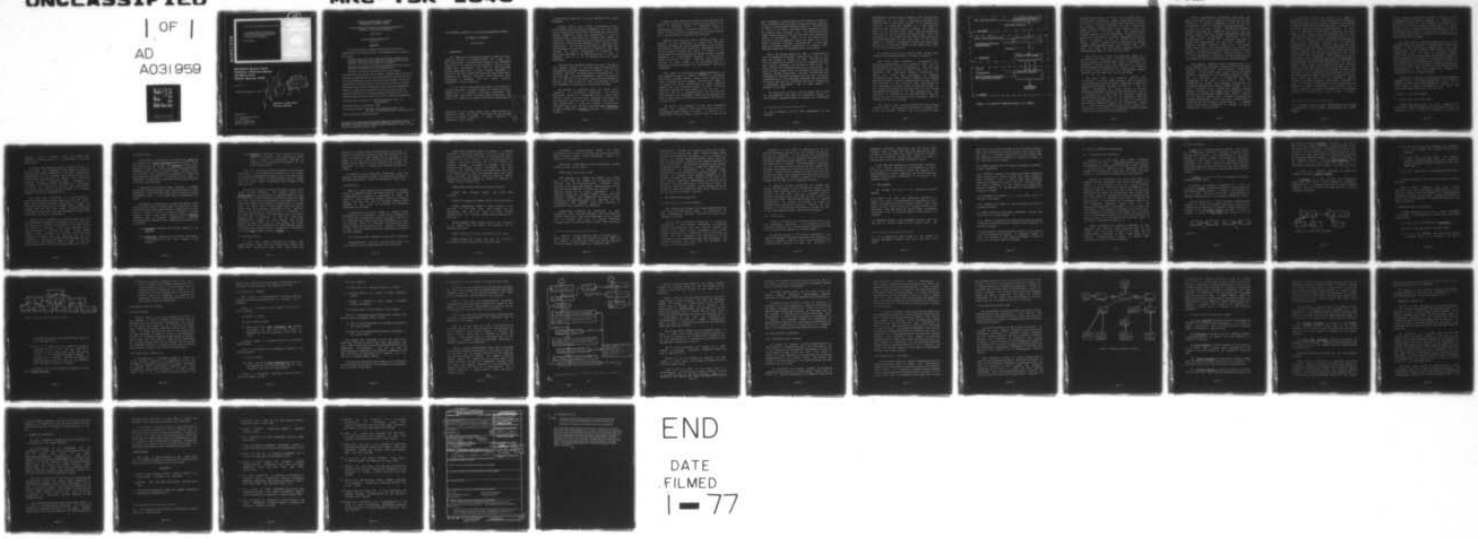
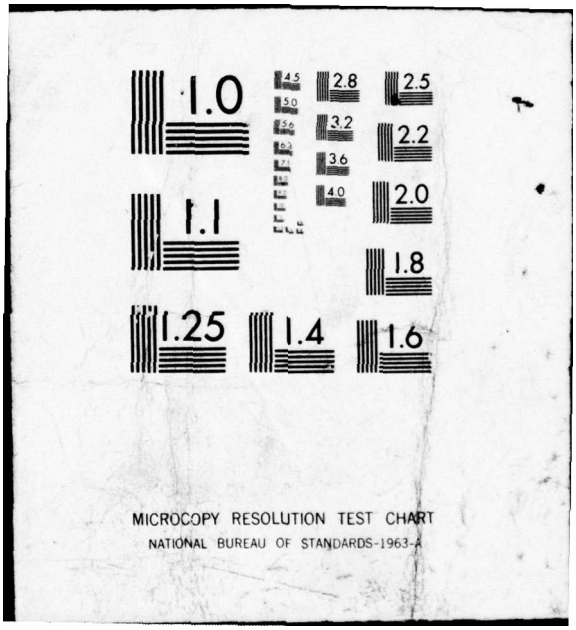
AD-A031 959 WISCONSIN UNIV MADISON MATHEMATICS RESEARCH CENTER F/G 5/8
ON PERFORMANCE MODELLING OF DATA BASE MANAGEMENT SYSTEMS - AN I--ETC(U)
JUL 76 A REITER DAAG29-75-C-0024
UNCLASSIFIED MRC-TSR-1648 NL

| OF |
AD
A031959



END

DATE
FILMED
1-77


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

MRC Technical Summary Report #1648

ON PERFORMANCE MODELLING OF DATA BASE MANAGEMENT SYSTEMS - AN INDUCTIVE APPROACH

Allen Reiter

Mathematics Research Center
University of Wisconsin—Madison
610 Walnut Street
Madison, Wisconsin 53706

July 1976

(Received November 18, 1975)

Approved for public release
Distribution unlimited

Sponsored by

U. S. Army Research Office
P. O. Box 12211
Research Triangle Park
North Carolina 27709

UNIVERSITY OF WISCONSIN - MADISON
MATHEMATICS RESEARCH CENTER

ON PERFORMANCE MODELLING OF DATA BASE MANAGEMENT SYSTEMS - AN INDUCTIVE APPROACH

Allen Reiter

Technical Summary Report #1648
July 1976

ABSTRACT

This paper describes a simulation model in which user jobs are synthesized from basic building blocks. The modelling process consists of three stages:

translation from a user view of data and processes dependent on the data base management system (DMS) into a standard form consisting of explicit access paths over logical data structures;

translation of the logical structures and operations into block-oriented structures and operations on a virtual machine; and

execution of a number of concurrent jobs on a real machine.

This paper deals only with the last two stages. Standard forms for the logical structures and operations and for the virtual machine are described; they are as free as possible from the data view of the particular DMS. We describe a generalized modelling framework, which becomes a model of a particular DMS when various "plug-in" modules are added. The data representation features of a DMS enter as parameters for the second stage, while the resource management tactics are the parameters for the last stage. The proposed model structure is intended as a basis for DMS design experiments.

Computing Reviews category: 4.6 (Software Evaluation, Tests, and Measurement).

AMS (MOS) Subject Classification: 68A50

Key Words and Phrases: data base, data management system, data structures, performance evaluation, simulation model.

Work Unit Number 8 (Computer Science)

Sponsored by the United States Army under Contract No. DAAG29-75-C-0024.
The author is on leave from the Computer Science Department, Technion - Israel Institute of Technology, Haifa, Israel.

ON PERFORMANCE MODELLING OF DATA BASE MANAGEMENT SYSTEMS - AN INDUCTIVE APPROACH

Allen Reiter

1. INTRODUCTION

The modelling of computing systems is hardly a science today; at its best, it is perhaps nearer to a black art. The difficulties of modelling are caused by the tremendous complexity of the systems and the myriads of details which affect their performance. Not only is each individual serially-executing task a highly complex subsystem in itself, but we must contend with multiple tasks concurrently competing for numerous resources; these resources, while functioning in parallel, are not (quite) mutually independent in their operation.

At the heart of the modelling effort is the desire to obtain quantitative information about the effects of various changes (e.g., in hardware, scheduling algorithms, data organization, task load) on overall system performance. The reason for modelling is simply that actual experimentation

Sponsored by the United States Army under Contract No. DAAG 29-75-C-0024. The author is on leave from the Computer Science Department, Technion - Israel Institute of Technology, Haifa, Israel.

is prohibitively expensive (if at all feasible) and rarely employed.

Analytic models which attempt to describe an entire system suffer from poor accuracy because of the very high level of abstraction and because of necessary but not always tenable assumptions about the distribution in time of relevant events. Such models are perhaps most useful in indicating gross performance and in studying the individual behavior of various system components while ignoring their interactions. The information obtained via such models can yield valuable insight when taken in conjunction with data obtained via actual measurements or simulation. In particular, knowledge of the theoretical norm is vital if departure from it is to be detected and the reason understood.

Discrete-event simulation models have more flexibility than analytic models in that any amount of detail may be incorporated, and model complexity is limited only by the resources available to develop and run the model [20]. On the other hand, environment-dependent input distributions must be supplied to the model; these are often suspect in how well they reflect reality. Also, the simulator is faced with problems in validating and calibrating the model.

One approach to modelling which has gained recent popularity is trace-driven simulation. An existing operational system is instrumented with a probe which collects actual run-time data. These data are abstracted and utilized to form the input to the model. The model is then parametrized to allow for changes in those system variables under study. We shall call such models deductive, as they are based on a posteriori analysis of experimental evidence.

Among the many advantages claimed for this approach are credibility, accuracy, low-level resolution, and ease of use [17]. A deductive model is undeniably useful in "fine-tuning" a system. Its obvious limitation is that it cannot be used to predict performance of a proposed system; i.e. one for which no trace data are available.

Somewhat less obviously, even in existing systems, the decision on what constitutes an "event" for the probe greatly influences the type of changes which can be studied. For example, a probe which records all input-output activity may be useless in studying the effect of changes to the underlying data structures (such as changing from an indexed sequential to a hash-coded directory); in this case the probe is at too low a level. Moreover, many high-level "events" are often hidden in the processing logic of a program and cannot be detected by conventional monitoring techniques.

This paper is concerned with <u>inductive</u> modelling. In this approach a model of a computer system is synthesized from basic building blocks. These blocks determine the level of abstraction of the model. Building such a model is a process not unlike constructing an actual computing complex with its operating system and then writing an application program for each element in the job mix. To keep the task at a manageable level, much of the detail clearly needs to be suppressed. Before we can describe this task, we must narrow our universe of discourse and establish a point of view.

We restrict our attention to data base management systems, i.e. computer systems whose primary function is manipulating large amounts of data. In addition to a conventional operating system, several layers of generalized

data management software (DMS) may be present; we shall in part be concerned with analysis of some basic DMS components so that we can synthesize different types of architectures of DMS's in the model. One simplifying assumption is that such systems are I/O-bound, i.e. the central-processor usage characteristics of the job mix do not have significant impact on performance. While we do not ignore the CPU entirely, we do not propose to equip the model with tools which facilitate CPU-usage representation. On the other hand, we do desire careful description of features which directly impact the I/O characteristics of the tasks.

There are various aspects of a DMS. It incorporates a view of information structures, imposing this view on the user who wishes to access the data. The DMS may have built-in mechanisms by which the data are accessed; the user must specify processes over his data in those terms. {*} Since the DMS must ultimately execute on a real machine, it also provides a mapping from its information structures onto physical storage structures. Finally, the DMS must have algorithms for executing processes over the real structures corresponding to the conceptual processes over the logical structures, and also for allocating and sharing resources in a time-shared environment. Each of these aspects is relevant to our task.

Our basic point of view is that the model shall be used for experimenting with DMS and application design. It should enable a system designer to evaluate the performance

---

{*} This is termed a lack of data independence in the programs.

implications of various design tradeoffs under various usage assumptions, or to make a comparative evaluation of different DMS's in the general context of an application. Typically, a prospective customer for a DMS knows the expected usage only approximately; moreover, the usage is likely to be changing with time. Relative performance figures may be just as valuable as absolute ones. Thus the usefulness of the model does not hinge on the accuracy of the usage assumptions. At the same time, if the actual system is suitably instrumented and collects appropriate statistics, it should be possible to feed them to the model for fine-tuning the operation.

The general layout of an inductive model, showing three stages of the modelling process, is shown in Figure 1.

The first stage translates a user-supplied task description specified at a high conceptual level into an intermediate-level system representation. The user task description is in terms of the logical data view of some particular DMS (relational, network, hierarchical, or other). The system task description (in Standard Intermediate Data Base Language - SIDBL) is in a standard form and includes a concrete representation for each access path employed by this particular DMS. Data however are still addressed at a logical level; no reference is made to any physical attributes of the data items or of the storage representations employed by the DMS implementation. The translator is dependent on the particular DMS in question, and may be similar in structure to the translator used in implementing the actual system.

The second stage involves modelling the data storage representation. The logical data elements are mapped onto blocks of secondary storage, and on this basis the data

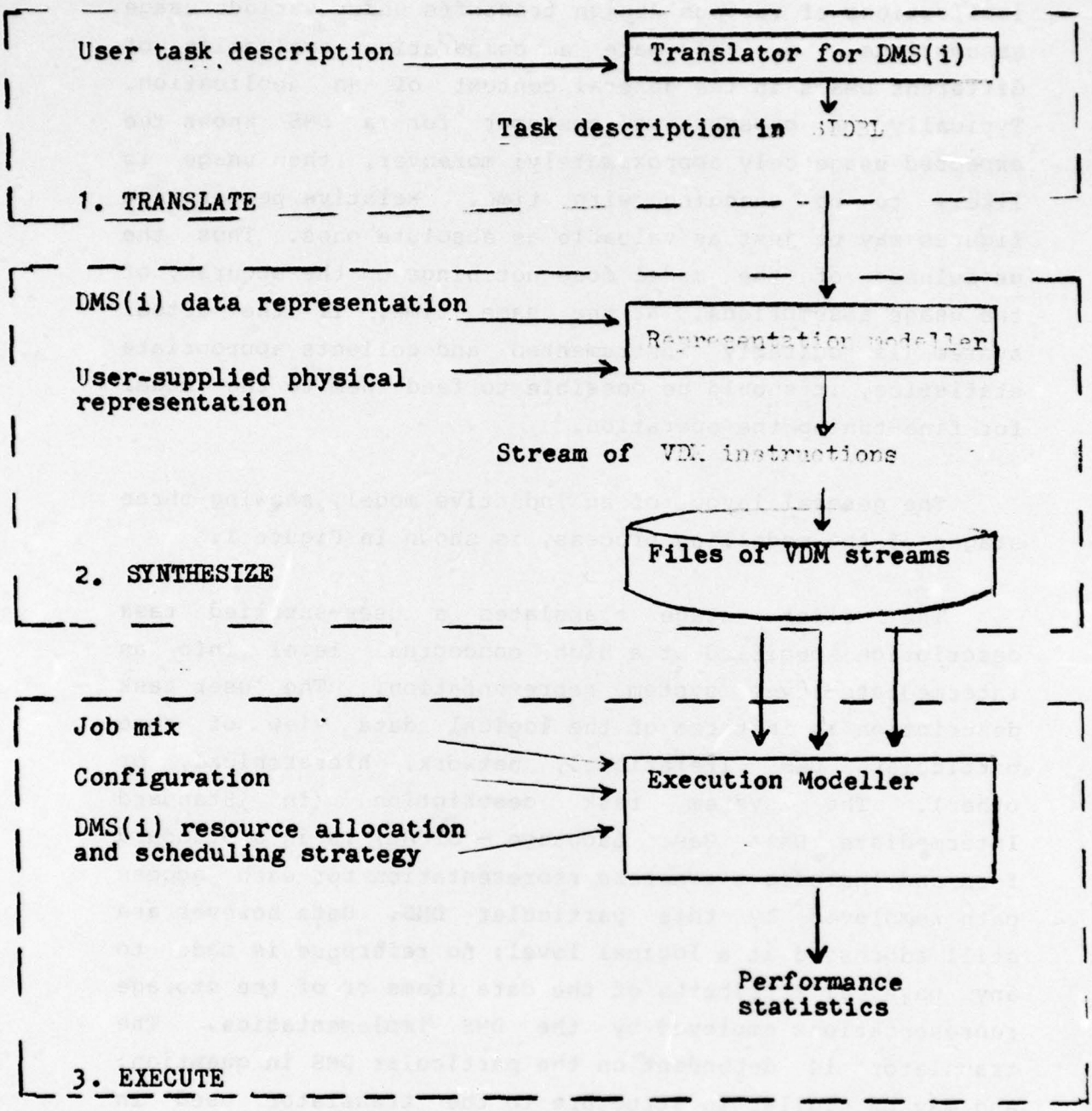


Figure 1. An inductive simulation model, in 3 stages.

manipulation operations of SIDBL are translated into a block-oriented instruction stream. These instructions are in terms of a virtual data machine (VDM); questions dealing with dynamic resource allocation as well as some other issues (system configuration, file extents, etc.) are postponed until the final stage. The representation modeller consists of data description mechanisms and a generalized translation algorithm independent of specific DMS's. In addition, it must be provided with various (small) "plug-in" modules which characterize the storage representation features used by each specific DMS. When these are added, the generalized framework becomes a representation model for this particular DMS.

The output of the first two modelling stages is a synthesized job description in terms of low-level events. One can conceive of existing DMS's being instrumented for informing a monitor when such events occur; a trace of a program execution collected by the monitor would resemble the event stream generated by our representation modeller and would play the same role for the next stage.

The third stage, the execution modeller EXM, incorporates all of the system dynamics. EXM is a model of a multiprogrammed operating system, resembling somewhat such commercially-available packages as SAM [1]. EXM is, however, parametrized for various choices of resource management strategies (relevant to DMS performance), and it is intended for a wide class of operating system and hardware architectures. EXM is not explicitly concerned with data representations, but is used to translate the process model obtained in Stage 2 into performance figures for a given system architecture and job mix. Again, for a fixed choice of strategies our generalized framework becomes an execution model for one particular DMS.

Storage representation differences among DMS's may result in different event streams for the same user process. Differences in resource management tactics, job mix, and configuration, result in different execution statistics for a given set of event streams. The VDM thus provides a convenient interface for separating the static from the dynamic issues. Moreover, storage representation modelling is often subject to stochastic variability, as the representation in storage of a particular data base is usually not deterministic but may vary depending on insertion sequence, deletion history, etc. In studying issues which impact on the system dynamics, it is useful to be able to insulate the results from the effects of random changes in storage representation. This we are able to do by first generating a (fixed) set of VDM streams and then using these for various experiments with system dynamics.

The problem of modelling data representations at a level suitable for microscopic modelling was also addressed by Senko et al ([14], [15]). Senko proposes a four-tiered approach. At the highest level, a user's view of processes and data is expressed in a non-procedural language (FORAL). This is translated into an access-path view which corresponds to the input to our stage 2: data are represented as strings and processes are described in a procedural language which closely resembles SIDBL. At the third (encoding) level, strings are assigned concrete physical attributes and mapped onto contiguous virtual storage units called basic encoding units (BEU's). Finally, at the physical device level, the BEU's are mapped onto actual storage devices. Sibley [18] uses a similar model in discussing the issue of data transferability. A somewhat similar four-level architecture is proposed in [13] for implementing a relational DMS.

Our approach differs from Senko's in a number of significant ways. First, we are concerned with developing general mechanisms which are used for modelling the dynamics of DMS execution, not only with description mechanisms for the static aspects of the data representations. {*} Thus, e.g. in modelling a B-tree "insert" operation ([8]) which causes a block to split, it is not sufficient for us to have a representation of the "before" and "after" states of the data base; we want to be able to describe the intermediate steps in the splitting algorithm of a particular implementation of B-trees. Second, Senko views directory decoding as a low-level function related to physical device "access methods"; we view it as a high-level function and model directories using the same mechanisms as for other higher-level data structures. This enables us to take a more uniform view of physical access which is independent of the underlying hardware or DMS implementation. We can for example obtain static descriptions of the block reference patterns for the same process using each of ISAM ([6]), SIS ([3]) and HISAM ([7]) organizations, whereas for Senko these differences would appear only in the execution statistics. Our representation modeller maps data onto physical blocks, which, while not having yet been assigned a specific location in secondary storage, nevertheless correspond to units of physical storage and will each eventually be the subject of a single well-defined I/O operation (not an "access method"). Our block is therefore at a considerably lower level than Senko's BEU, being in fact a primitive

---

{*} Although Senko's earlier FOREM system [16] included the simulation of disk accesses, the mechanisms were largely *ad hoc*.

concept uniformly defined for all DMS's. Finally, our VDM is equipped with a procedural language; no counterpart is described by Senko for the encoding or physical device levels. This is because Senko's orientation at the lowest modelling level is directed toward the hardware; our VDM is an operating-system-level interface which enables us to study (among other things) the effects of a multiprogramming resource-sharing environment.

This paper does not deal with translation into a SIDBL representation; this issue will be addressed in future work. However, a data description language which, with some modifications, may be a suitable "front end" for our model, is described in [19]. We sketch in successive sections SIDBL, VDM, and the representation and execution modellers. The descriptions are not meant to be exhaustive; indeed, any such system must be open-ended at all levels in order to be able to accomodate new developments in data base technology. The author first dealt with the problem of modelling data representations in [9]; the present work contains a substantial revision of that material.

A simulation model which utilizes the general framework discussed here is currently operational. It is written in FORTRAN IV (which is the host language for SIDBL). It is being used to conduct experiments in DMS design; some results have been described in [10] and [11].

2. SIDBL - A DATA BASE LANGUAGE

2.1 Storage paths and access paths.

Before describing SIDBL, we must establish the distinction between storage and access paths for items in a data base. For our purposes we consider a single-level

(possibly virtual) secondary store, and assume that redundancy is not utilized, i.e. an item appears in storage exactly once.

A storage path determines where in storage the item is to be placed. The placement of some items in storage is arbitrary; for others, depending on the DMS, hierarchical relationships may be utilized when placing occurences of various items in proximity to one another. Thus, a file may "contain" records, a repeating group - instances of occurences, a composite field - subfields, etc. Since there is exactly one storage path for each item, relationships more complex than a tree are rarely utilized in a storage strategy, and are not considered here.

An access path corresponds to a particular way of processing collections of data items, and (for many DMS's) there may be several different access paths to an item. An access path may coincide with a storage path; indeed, the storage path may have been chosen in order to optimize performance along this access path (presumably, because this access path is used very frequently.)

When an item is accessed in a real data base, many of the structural parameters associated with the item (e.g. the number of its storage path neighbors in the same block) are implicitly available. This is true regardless of the access path used in reaching the atom. In our model, however, these parameters are obtained iteratively when traversing a storage path (the algorithms for this are discussed in Section 4) and may not be available via other access paths. The reader is asked to keep in mind the distinction between storage and access paths in the sequel.

2.2 View of data.

A data base is a collection of (instances of) atoms and of (instances of) access connections among the atoms. Atoms and connections are called the elements of a data base. Examples of atoms are: a file; a record; a compound field; a character string. While an element usually contains data, and therefore will eventually be assigned a "length" and other physical attributes, it is possible to have atoms or connections which serve a conceptual function only and have no physical manifestation.

A connection from atom1 to atom2 connotes an ordered processing relationship among them, allowing access to atom2 immediately after accessing atom1 without needing to access intermediate atoms. In this sense atom2 is addressable from atom1.

Motivated by considerations in modelling storage paths and by the fact that all access paths are directed, we think of connections as being hierarchical: a connection is either from a "father" to a "son", or from a "brother" to a "next brother". Access paths among brothers define sequences (ordered lists) of elements. Specifically, we identify three different connection types, dispensing with others for conciseness.

2.2.1 Successor connection for the next element in the sequence.

2.2.2 Oldest-son connection from a father atom("owner" of the sequence) to the first element of the sequence.

2.2.3 Keyed-son connection from a father atom to any element of a sequence. Such connections always imply a system of directories, discussed in section 4. The directory is transparent at the SIDBL command level.

Note the distinction between oldest-son and keyed-son connections. If the elements of a sequence are not keyed, the second and subsequent elements are not connected (along this path) to the father; the second element can be reached from the father only by reaching the first element of the sequence, etc.

Access path connections other than along storage paths always utilize pointers. The pointer itself is a pseudo-element of the data base: it occupies space and must be physically accessed before the access from atom1 to atom2 can take place, but is transparent to the SIDBL user. The access from atom1 to the pointer is always along a storage path. For DMS's which utilize simple structures such as linked lists for access path representation (e.g. IDMS [4]) access to the pointer is trivial, as it is necessarily in the same block as atom1. For other DMS's (e.g. if pointer arrays, a type of directory, are used for access path representation) access to the pointer may involve significant work on the part of the system. We stress however that this is transparent for SIDBL commands: access is directly from atom1 to atom2. Storage-path connections are called local; others are called global.

2.3 Data description.

A data base schema specifies the element (and pseudo-element) types and the hierarchical storage and access path relationships among them. Each atom type

appears as a uniquely named item description node (IDE). If there is a father-son connection from some occurence of this atom to a sequence of other atoms, this is reflected in the schema by a link from this IDE to the IDE for the first element of the sequence, with an indication of whether the connection is local or global; the brothers are similarly linked.

In addition to the access path connections, each IDE contains user-specified values for attributes relevant to the modelling process. These are discussed in Section 4.5.

2.4 Operations.

SIDBL is a simple node-at-a-time navigational language similar to CODASYL's DML[2] and DIAM's RDL[14]. Processing is represented as a sequence of operations on trees. We traverse a tree, inspecting, adding, changing, or deleting nodes. A CPU clock may also be advanced between SIDBL commands to take care of processing not otherwise represented in the model.

A process has an arbitrary number of (named) stacks associated with its execution. A stack can be considered to be a data base variable which takes, as values, pointers to a current data base location and its antecedents in a tree. (It contains values of attributes associated with these nodes.) Stacks play the same role as 'currency indicators' in DML; they are however explicitly specified as arguments by SIDBL commands.

The following SIDBL commands are currently implemented.

STINIT(atomtype) - sets up a new stack and points it to (an occurence of) the specified atom in the data base.

LOCATE DOWN (<sequence number of element> or RANDOM) "executes" a father-son connection. From the current father atom occurence we access the oldest son occurence and then a succession of next brothers, until the specified atom is reached. If the sons are keyed, we do not traverse the sequence; instead, a directory is decoded. The newly accessed atom becomes current. Optionally, the atom preceding the indicated atom in the access path (which may be the father atom, in case the indicated atom is the oldest son) is accessed. This mode is usually followed by a MODIFY command and is used for example in adding or removing an element to/from a linked list.

LOCATE NEXT accesses the next brother occurence.

LOCATE SAME accesses (again) the current atom occurence.

LOCATE UP accesses the father atom in this access path.

INSERT SEQUENTIALLY (DOWN or NEXT) creates a new occurence of the target atom. This command has the implication that the new atom is being created "sequentially", for instance during initial creation of a data base.

INSERT RANDOMLY (DOWN or NEXT) creates a new occurence of the target atom, and models its insertion into an existing data base.

DELETE removes the current atom from the data base.

MODIFY changes the content (but not the structural attributes) of the current atom in the data base.

LOCK/UNLOCK prevents/allows access by other simultaneous processes to the current block or current area. (More elaborate access control mechanisms require an extension of our model.)

OPEN/CLOSE invoke models of the corresponding run-time system routines for this area.

STEND erases the current stack.

Some commands (e.g. INSERT and DELETE) are allowed only if the node was reached via a storage path, as we otherwise lack the storage-structure attribute values required by the modelling process. SIDBL requires that the user explicitly establish a storage path even when the modelled system has no similar requirement. {*} We therefore provide for a parameter CONTROL implicit for SIDBL traversal commands: when CONTROL=ON, the parameter values required by the modelling process are generated, but the corresponding VDM commands which would otherwise be generated by the translator are supressed.

Higher-order operations are represented as short subroutines using the above commands. One such subroutine is TRAV, which visits every local descendant of the current atom in preorder (with exits for possible additional processing when we enter and leave a node.) TRAV is entirely

---

{*} Explicit storage paths may need to be established in other contexts. In IDMS, when modelling the FIND OWNER OF SET command, a storage path to the owner must be specified if the set member was reached via a different path.

schema-driven. To model the sequential traversal of a file, we execute the sequence OPEN - TRAV - CLOSE; to model random access of one record and examination of its contents we execute the sequence LOCATE DOWN RANDOM - TRAV. Note that this very gross view of sequential and random processing is often sufficiently accurate when investigating the effects of changes in data structures, scheduling algorithms, or configuration on performance. In modelling IDMS, subroutines for such functions as FIND PRIOR RECORD IN SET execute the required SIDBL commands depending on the presence of "prior" and "owner" pointers for the set. For this purpose, additional DMS-dependent information may be included in the data description; in the case of IDMS, two additional flags were required. Similar subroutines can be written to model set intersection and union operations on pointer arrays or to model a file sort.

3. THE VIRTUAL DATA MACHINE VDM

3.1 The need for a "virtual" machine.

The VDM provides a means for process specification one level below that provided by SIDBL. It incorporates the data representation features specified by the user and/or inherent in the DMS.

We can think of two people cooperating in running an installation: a data manager, responsible for selecting software and determining the data structure design, but knowing little about resource management; and a system manager, responsible for configuring and fine-tuning the system and for scheduling the execution of various processes, but not knowledgeable about data management. The former communicates with the latter by describing each process as a VDM command stream.

Depending on the level of sophistication of the resource-management strategy, the VDM process description may be too elaborate. In this case we will be expending unnecessary overhead first in generating the description and then while simulating process execution. Thus, we allow for the possibility of elaborate buffer-management strategies and for the sharing of data among concurrent (independent) processes; when modelling a DMS which does not employ such tactics, a simpler process description might have sufficed. On the other hand, having generated such descriptions we are able to investigate the potential payoffs in introducing changes into the DMS resource management tactics.

A VDM-level process description can provide some insight into the amount of work required by a process. Two different data organizations (e.g. with and without "owner" pointers for some IDMS sets) will in general result in different VDM descriptions, even when the user's view of the process is the same. The VDM descriptions can be compared qualitatively, for example by counting the number of different blocks accessed by each one. However, the main function of a VDM process description is to form the input to the final modelling stage, the execution modeller.

3.2 View of data.

A block is the basic unit of data storage and the unit for input/output operations. It is accessed by a single primitive I/O action via a unique address (translated by EXM into a unit-oriented address via extents tables.)

To make the concept of "I/O action" independent of the underlying hardware architecture we adopt the convention for rotating secondary memories that an action consists of an arm-positioning maneuver, at most one revolution to

accomplish rotational positioning, and the actual data transfer. Thus, in ISAM, where the track index must be read in order to find the block, the 'READ' operation comprises two or more distinct actions even though the work is accomplished within the hardware and channel logic.

At the VDM level commands are not explicitly concerned with I/O. However, the object of each command is a block. At this level it is convenient to treat the address as a three-dimensional entity consisting of "area" of physical storage, "type" (logical subdivision of a file){*}, and "sequence number".

3.3 VDM commands.

The argument for each of the following is a block address.

3.3.1 FETCH is the basic data addressing operation: it causes the block to be positioned for processing in core. FETCH is executed every time a process needs a data item within a block. We do not rely on a previous FETCH for the same block, since EXM allows for various buffer replacement strategies, some of which may cause an active block to be overlaid.

3.3.2 RELEASE signals to core management routines that the task has no further need of the current block. The

---

{*} For an example of using "type" in the context of storage management tactics, see [11]. See also 3.3.9 below.

semantics of the operation depend on the allocation strategy used. From the point of view of the task, this means that the block is no longer "active" and (if its contents have been modified) should be recorded in secondary storage at the convenience of the system.

3.3.3 STORE requests that the block be recorded in secondary storage immediately.

3.3.4 PREFETCH requests the system to attempt scheduling input of one or more blocks beyond that block currently being processed. This is usually used when modelling sequential processing of a chain (see 4.2). Only the "area" and "type" components of the address are significant; prefetch modelling is accomplished by a scan-ahead of the instruction stream.

3.3.5 ALLOCATE is a request for a core buffer prior to creating a new block.

3.3.6 ALBLOCK is a request for the allocation of a block of secondary storage.

3.3.7 FREE informs the disk space management routine that the block is no longer needed.

3.3.8 MODIFY is identical with FETCH except that the buffer is marked as modified and will eventually need recording in secondary storage.

3.3.9 LOCK/UNLOCK prevents/permits access to all or part of an area by concurrent processes. If "sequence number" is null, all blocks of this type are intended; if "type" is also null, the entire area is intended.

4. THE DATA REPRESENTATION MODELLER

4.1 Representations and models.

Elements of a data base have various attributes associated with them. A representation of a data base is the assignment of values to the attributes of each element. A representation modeller is an algorithm which can enumerate the elements of a data base and their attribute values. In this section we describe one such modeller.

Instead of enumerating the entire data base, our modeller operates iteratively: when going from element A to element B, an environment for B (i.e. various attribute values) is created. One of the goals of the modeller is to determine when B is in the same block as A; for local connections, we assume that B would have been placed into the same block as A provided that there was enough room to do so. Many of the attributes are used in making the determination of "required" and "available" space for B. The modelling algorithm makes use of the previously-established environment of A to obtain B's environment. Little memory is utilized: if B is a son of A, A's environment is preserved in the stack; if B is a brother of A, B's environment replaces A's; if B is the father of A, then we return to B's environment (which had been saved in the stack), discarding the one for A.

Before sketching the modelling algorithm, we must introduce some auxiliary data structures basic to the mapping from the logical constructs of SIDBL onto the block-oriented constructs of VDM. These enable us to describe representational features of specific DMS's conveniently within a general framework.

4.2 Data structures.

A chain is an ordered sequence of blocks. The blocks may be physically adjacent in storage, or explicit chain pointers may be present. The significance of the chain ordering is usually related to the concept of "sequential processing" (and possibly to the storage allocation philosophy) of the DMS. Pointers to other blocks may also be present associated with some of the the data elements within the chain, but these are distinguished from the chain pointers.

A cluster is part (or all) of a sequence of brothers contained in one block.

A sequence of brothers represented as a chain of clusters is proper if the chain ordering is consistent with the ordering of the sequence. In Figure 2, the sequence in (a) is proper, the one in (b) is not. We assume that all sequence representations in the modelled system are proper.

We also assume that a proper sequence representation contains a cluster B from which every other cluster of the sequence can be reached by a sequence of chain connectors or pointers. B is the initial cluster; the chain in which B appears is called the main chain; every cluster not in the

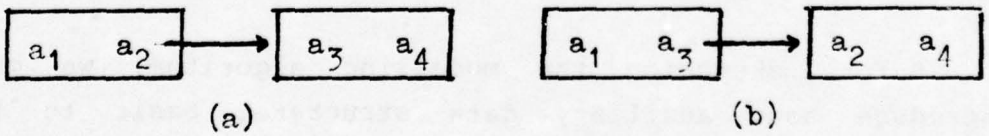


Figure 2.

main chain is called an offshoot. In Figure 3, B1 and B4 form the main chain (B1 being the initial cluster), while B2, B3, and B5 are offshoots. Intuitively, B1 and B4 were allocated for the sequence from a prime storage area, while B2, B3, and B5 were added later as a result of the insertion of additional sequence elements. p1 and p2 are pointers to the offshoot clusters. They are pseudo-elements: they do not exist from the user's point of view, but can conveniently be treated as elements by the modelling system.

A subset of a sequence consisting of consecutive elements is called a sequence segment.

A directory is a data structure for representing connections from atom A to each element of a sequence of keyed sons of A. It consists of a tree of proper sequence segments, the root being in the same block as A, with the following properties:

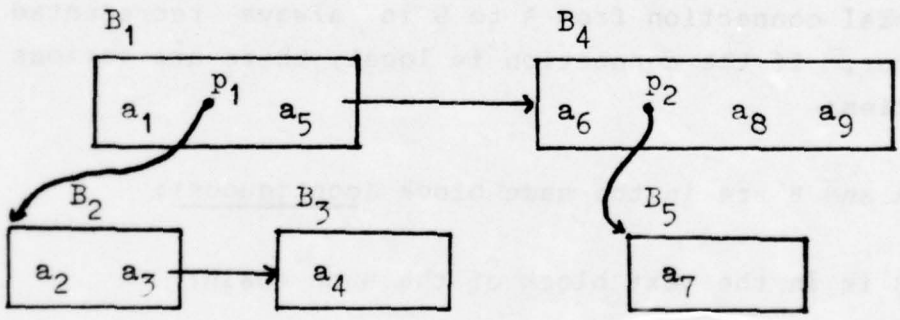


Figure 3. A chain with offshoots.

(a) At each level, all of the elements are brothers, i.e. the sequences are segments of one (long) sequence.

(b) At each level but the lowest, the elements (actually pseudo-elements) are each pointers to initial clusters of distinct sequences at the next lower level;

(c) At the lowest level, the elements are the sons of A;

A directory normally has at least two levels, including the root.

Figure 4 shows a directory with three levels. {a11,a12,...b23} are a sequence of keyed sons of A. {a,b} and {a1,...a5,b1,b2} are two sequences of pseudo-elements (directory indices at two levels). Although not shown, we do not exclude the possibility of offshoots appearing at any level of the tree.

4.3 Representation of connections.

A global connection from A to B is always represented by a pointer. If the connection is local, there are various possibilities:

(a) A and B are in the same block (<u>contiguous</u>);

(b) B is in the next block of the same chain;

(c) B is in an offshoot, the pointer to which is contiguous with A (in principle, this may be

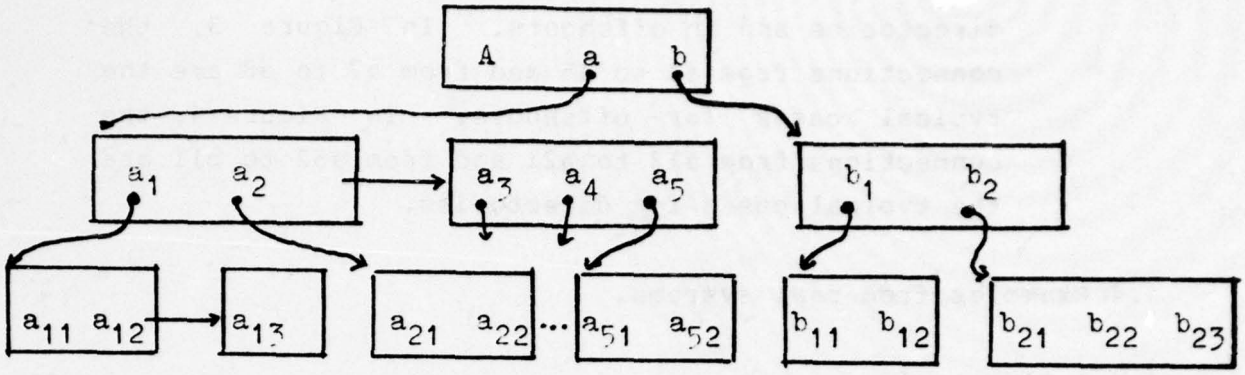


Figure 4. A directory with three levels.

iterated: the pointer to the offshoot may itself be in an offshoot, etc.);

(d) If B is a keyed son of A, there is a directory connecting B to A. To go from one level of a directory to the next lower level, the initial cluster of the indicated sequence segment is accessed via a pointer, and then the segment is traversed using (a), (b), and (c) as necessary until the desired element is reached.

In addition, for a local successor connection, we may have a broken connector:

(e) There is no explicit connector from A to B, but B can be reached via an element encountered earlier in the storage path to A. Such connectors arise in directories and in offshoots. In Figure 3, the connections from a4 to a5 and from a7 to a8 are the typical cases for offshoots. In Figure 4, the connections from a13 to a21 and from a52 to b11 are the typical cases for directories.

4.4 Examples from real systems.

4.4.1 Directories.

ISAM [6], HISAM [7] (for "root segments"), and SIS [3] are examples of similar directory organizations; for our purposes, they can be partially characterized as follows. In SIS (which uses B-trees [8]), all chains are one cluster long. ISAM may have longer (overflow) chains at the lowest level. In HISAM, offshoots may occur at the lowest level. For ISAM and HISAM, the number of levels is user-specified; for SIS, it is computed depending on the number of elements, block size, and other factors. IDMS [4] uses hash-coding for its CALC records, which for our purposes can be regarded as a two-level directory; identifiers which hash to the same block (initial cluster) give rise to overflow chains.

4.4.2 Other local connections.

In the storage of "secondary segments", HISAM uses chains (without offshoots) in a linearization of the tree. In MUMPS [5], oldest-son connections are always represented by pointers, while next-brother connections are either contiguity or next-in-chain connectors. In IDMS, records stored "via" a set are brothers and the set owner is the

father; both oldest-son and next-brother connections can be contiguity or next-in-chain, and offshoots may occur.

4.5 Attributes of elements.

Each element or pseudo-element of a data base has the following attributes associated with it. We group them by function.

Group 1 -- Attributes of the element as the root of a tree or subtree:

(a) Length, in bytes;

(b) Total number of sons;

(c) Total size of the local descendant set (defined recursively as the sum of the lengths and descendant set sizes of all sons, if the sons are local);

(d) Total number of directory levels (if a directory index element).

Group 2 -- Attributes of the element as a member of an ordered sequence:

(e) Ordinal number;

(f) Total size of the local subsequent set (the sum of the lengths and the local descendant sets of local brothers to the right of this element).

Group 3 -- Attributes associated with the block in which the element resides:

(g) Block address;

(h) Block size, i.e. maximum capacity (in bytes);

(i) Loading density, i.e. amount of space currently unused;

(j) Number of elements in this cluster, including possible offshoots;

(k) Ordinal number of the element in this cluster.

Group 4 -- Attributes associated with the tactics for placing sons and brothers within a block:

(l) Size of local descendant set actually contained in the current block;

(m) Size of local subsequent set actually contained in the current block.

The values for attributes (a)-(c), (h), and (i) are supplied by the user as part of the data description (see 2.2). These values may be specified deterministically or as probability distributions. (d) and (j) may be user-specified or computed by the modeller, depending on the DMS. The others are generated by the modeller, (g), (l), and (m) in a DMS-dependent fashion.

The attribute values for each element and its ancestors in the storage path describe the environment of the element, and are stored in the stack.

4.6 Operation of the representation-modelling algorithm.

To illustrate the operation of the algorithm, we sketch the logic for translating the LOCATE DOWN and LOCATE NEXT commands (Figure 5). In the ensuing discussion, the logic is not DMS-dependent unless explicitly so indicated.

Steps 1, 1n, and 2 are self-explanatory: the element attributes are generated and/or updated and stored in the stack. In translating LOCATE DOWN to a keyed son, we are now descending to the next level of the directory.

Step 3: The oldest-son and next-brother connections may be local or global independently; the connection type is specified in the schema.

Step 4: We get here any time a new block must be accessed, whether for a global connection, next-in-chain, or for an offshoot. If an explicit pointer is involved, the prior (current) block must be accessed to obtain the pointer. If the current block is in the same chain as the successor, a RELEASE command is generated. Optionally, a PREFETCH command is also generated.

Step 5: A plug-in DMS-dependent module must be provided for the block address determination. The module's identity is specified via the element descriptor in the schema and may vary from element to element. For example, different modules compute the address for the cylinder index, track index, and prime, cylinder and independent overflow areas for ISAM. The address computation for an offshoot will generally differ from that for a next-in-chain. For a global connection, a random address within the area is generally used.

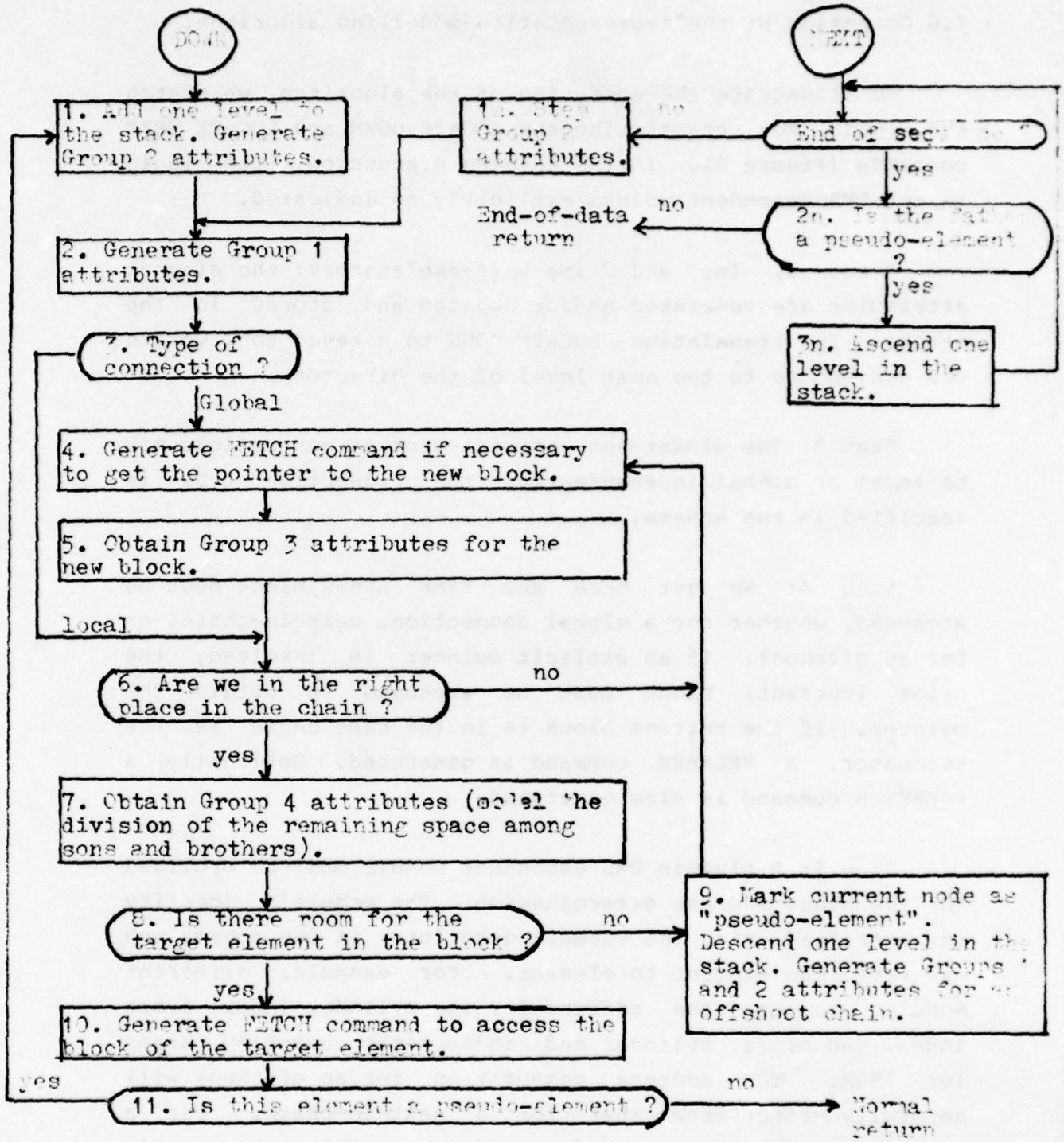


Figure 5. Translation of LOCATE DOWN and LOCATE NEXT commands.

Step 6: If the sequence number of the target element exceeds the sequence number of the last element in the current cluster, the next cluster in the chain must be accessed.

Steps 7 and 8: We have found the block in which the element would normally belong. We must now determine whether the element is actually there; it may have been placed into an offshoot chain due to overflow. The determination is DMS-dependent, as attributes (l) and (m) need to be computed. For a system such as OSAM, which places the entire descendant set before the successor set in a linearization of the tree, the computation is straightforward. For others (e.g. IDMS) a probabilistic model is used to determine the division of the remaining space among sons and brothers.

Step 9: The target element is in an offshoot chain. A level is added to the stack, and we continue with step 4. The number of elements in the offshoot chain (attribute (b) for the pseudo-element) is available as a byproduct of the DMS-dependent computation of step 7.

Step 10: We now know where the element is; a FETCH command is generated. Optionally, a PREFETCH command may also be generated here.

Step 11: If we are decoding a directory and have reached an index level, the process is repeated until we reach the lowest level.

Steps 2n and 3n of NEXT: If the father node is a pseudo-element, we have reached a broken connector: the end of an offshoot chain, or the end of sequence segment in a directory. We ascend one level, and repeat the process.

Generally, the previously-current block is not the same as the block to which we are now returning; a RELEASE command for the previous block is generated.

The INSERT commands mostly use the same logic, but require several additional DMS-dependent modules. Translation of the other SIDBL commands is straightforward.

The above logic is oriented toward the modelling of connectors for local connections. Hence the usefulness of our model is directly related to the extent to which the modelled DMS utilizes such concepts. As an extreme case, we may consider a DMS which has no local connections. (Such organizations, in which every access path is represented by a pointer, are sometimes adopted for convenience of implementation, especially if performance considerations are not vital.) In modelling this system within our framework, most of the mechanisms described above would not be used.

4.7 Some problems in modelling.

4.7.1 Dependence on past history.

In almost all systems, the data representation at a given moment is a function not only of the data content and of the hardware- and user-specified parameters, but also of the processing history of the data base. Thus, adding a record and then deleting it will generally not return the representation to its original state. The greater the degree of self-organization in the modelled system, the more difficult it is to model.

To illustrate one problem, consider the following possible implementation of a tree-handling system such as MUMPS [4]. Data are represented in storage as binary trees

(left pointer to oldest-son, right to next-brother). A block always contains one binary subtree, with some of the branches pointing to descendant subtrees in other blocks. When a node is added it is placed in the same block as its predecessor (father or left brother). If a block overflows, the subtree in it is traversed in order to find a branch (left or right) which most equally splits the subtree in two; the half not containing the original root is moved to a different block and the cut branch is replaced by a pointer to the new block.

A little reflection should convince the reader that the resulting structure for a large data base depends on the order in which the various nodes were added to the system. The number of possible valid representations grows combinatorially with the data base size. Specifically at stake is the model algorithm for computing the attributes (l) and (m) of 4.5. The algorithm described in [9] produces one possible representation which incorporates the rules of formation; it is not parametrized for the data base creation history. The underlying hope (based on heuristic considerations) is that the possible variations balance out with respect to performance of the processing programs; validating this assumption appears to be a difficult and poorly defined task requiring a vast amount of experimental evidence from actual systems.

4.7.2 Local memory of the model.

The attribute values are computed (deterministically or probabilistically) based on global data descriptions. However, data modification operations cause local changes in these values. As an example, an INSERT operation changes the previously-computed "available space" attribute of the block. The model can remember in the stack only a few of

the implied changes, and only for as long as the block stays current. This limits the resolution power of the model for a complex sequence of interrelated operations. In general, we expect that an update task is factored into a series of "small" subtasks, each of which can be accurately modelled within the memory constraints, and which are mutually independent in the sense that local changes in attribute values need not be remembered from one to the next.

5. THE EXECUTION MODELLER (EXM)

EXM accomplishes the binding between the static process descriptions and their execution on a real machine. The implementation is described elsewhere ([12]); here we give a brief description, focusing on parametrization for different run-time environments.

A functional schematic of EXM is shown in Figure 6. It includes the major hardware components which affect DMS performance: processors, I/O resources (disks, channels), and main storage buffers, and their respective (software) managers: task scheduler, I/O scheduler, and storage allocator. As was the case for the representation modeller, our approach is to provide the general program framework and the description mechanisms independently of the DMS being modelled, and to augment this by DMS-dependent "plug-in" modules.

Task execution consists of reading a VDM stream and issuing the corresponding command. Each block reference is sent to the "contents supervisor" to determine whether the block is currently in core. Between two successive references by a task to a block, the block might be overwritten. This is the general mechanism; a specific DMS need of course not make use of such tactics. Similarly, it

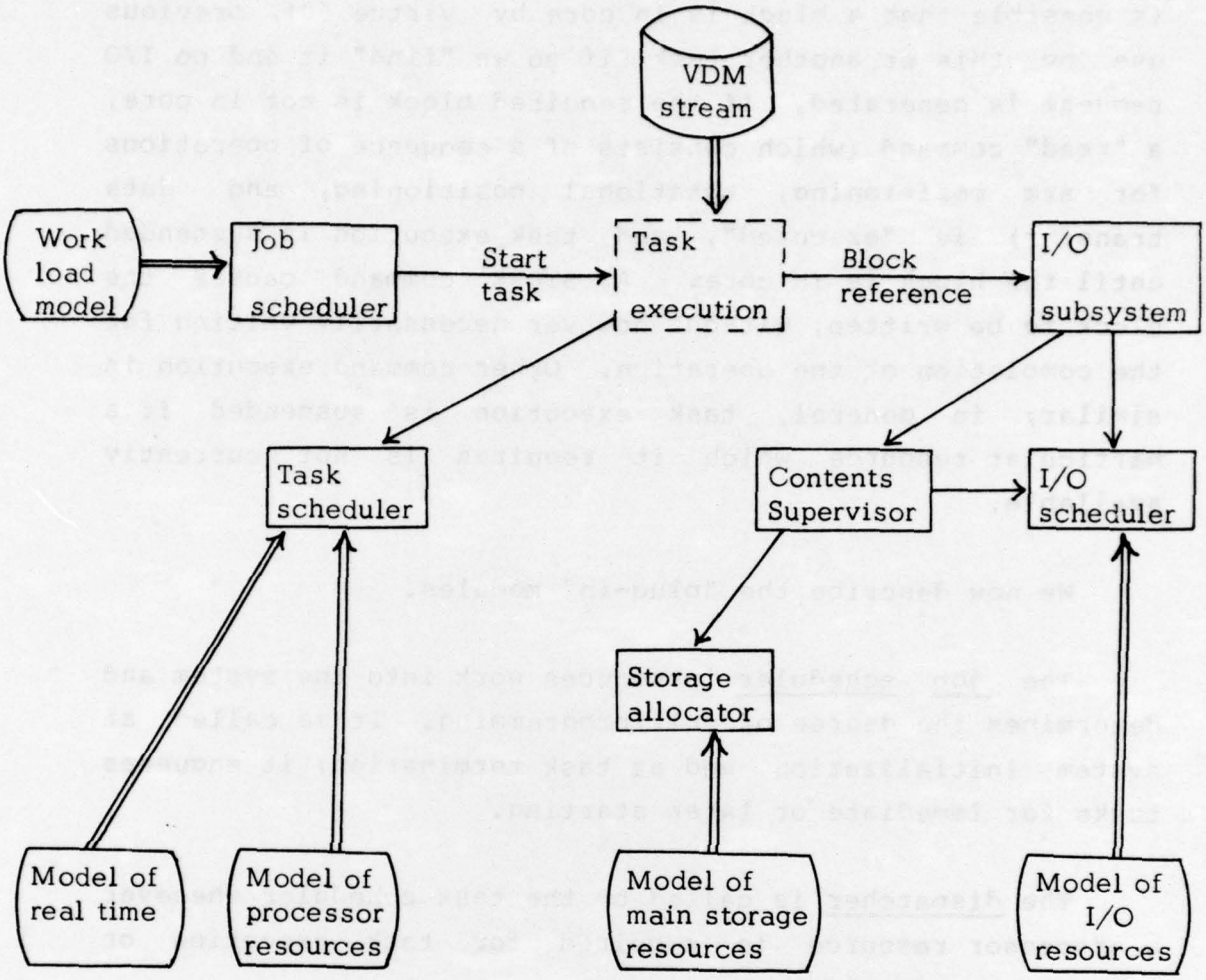


Figure 6. Functional framework of the EXM.

is possible that a block is in core by virtue of previous use by this or another task; if so we "find" it and no I/O request is generated. If the required block is not in core, a "read" command (which consists of a sequence of operations for arm positioning, rotational positioning, and data transfer) is "executed", and task execution is suspended until the block is in core. A STORE command causes the block to be written, without however necessarily waiting for the completion of the operation. Other command execution is similar; in general, task execution is suspended if a particular resource which it requires is not currently available.

We now describe the "plug-in" modules.

The job scheduler introduces work into the system and determines the degree of multiprogramming. It is called at system initialization and at task termination; it enqueues tasks for immediate or later starting.

The dispatcher is called by the task scheduler whenever a processor resource is required for task execution or interrupt processing.

The disk scheduler is called whenever an I/O request is to be placed onto a device queue. The position of the request in the queue depends on the scheduling discipline used.

The channel scheduler is called when a channel becomes free and there are several disks waiting for a channel. It determines which one of the disks will be serviced next.

The storage allocator is called to allocate one buffer from a buffer pool. Many possible management strategies are

supported. Buffer pools may be associated with each task or shared among several concurrent tasks. Allocation policies may overlay a buffer which is in active use. A policy may take into account buffer status, number of current users, and the time the contents were last referenced, all of which are maintained by the system. If the former contents of a buffer have been modified, the allocator must first write the block to the disk.

The allocator is also called for the RELEASE command. The buffer is identified by its contents only. The semantics of the RELEASE command depend on the allocation policy for the pool; there may be other concurrent users of this buffer, or its contents may already have been overlaid.

The prefetch scheduler is called for the PREFETCH command. Depending on the current state of the system resources it may request buffer allocation and initiate the input of one or more blocks in advance of the current location.

The disk space management routine is called for the ALBLOCK and FREE commands. A garbage collection module may be invoked by the job scheduler when the latter detects "slack" activity.

OPEN/CLOSE routines are called via the corresponding commands.

These modules characterize a particular set of DMS strategies. Hopefully, the task of writing such modules for a particular system is considerably easier given the general framework. In the current system, the "plug-in" modules account for only some 10-15% of the total FORTRAN code. It is likely that two quite different DMS's may use similar

tactics in managing some resources, so that this part of the model would not need to be rewritten.

In addition to the "plug-in" modules, the hardware parameters need to be specified. The mechanisms for this are quite conventional and are described in [12].

6. MODELLING A GENERAL MIX

How do we model performance in a mixed environment, i.e. one in which non-data-base jobs are also present? We attempt to give a partial answer to this question. As we have not addressed the problem of modelling other types of computational processes, we can, at best, hope to obtain information about the data management part of the mix. To do this, we need to make some assumptions with regard to the contention for the same resources by the data base tasks and other jobs.

For simplicity, let us assume that the other jobs either consume a negligible amount of CPU time or else operate at a lower dispatching priority (this assumption is often valid in practice); then there is no interference to the data base tasks in CPU usage. Also, we assume that core storage is partitioned in a fixed fashion between data-base and other tasks, so that there is no contention for core. The other source of interference is in I/O activity on the same units. It may be possible to model this interference by representing the other activity in terms of its data needs, i.e. as a data base task.

The most common source of interference is system "spooling": the use of disks as buffers for printing. The operation of a spooler can, for our purposes, be modelled as the sequential traversal of a simply-structured file, using

the tools already developed. The size of this file, and the frequency of invocation of this job, will contain built-in assumptions about the print-output characteristics of the mix.

7. SUMMARY AND CONCLUSION

We have attempted to sketch the total architecture of an inductive data base system modeller.

Task description at an intermediate level is accomplished in SIDBL. The level is sufficiently low to be free of logical data views, but is above implementation-dependent considerations. SIDBL is not capable of task description independently of the DMS being modelled: in general different DMS's will have different access paths (and hence will produce different SIDBL descriptions) for a given task over the same data base. However, SIDBL does provide a clear separation between the task description and DMS implementation issues.

The major contribution of this work is in factoring DMS implementation issues into various small modules and in providing the framework within which experiments can be made. Two distinct models - static data representation and dynamic resource allocation - are involved. Parametrization of the former required defining new data structures appropriate for describing relevant features of DMS implementations.

The interface between the static and dynamic models is in terms of block-oriented operations. It is not clear whether this restricts the scope of the model. Somewhat surprisingly this has not proven to be a major handicap in

modelling IBM's ISAM, which in some ways is track- and cylinder- rather than block-oriented in its philosophy.

A microscopic simulation model is rather expensive to run, and is really suitable for a very detailed look at system operation within a narrow time window. {*} Also, to be useful for data base application design, a modelling system must be equipped with tools not discussed here, such as convenient data specification languages. Nonetheless, our model in its present state is proving itself useful for conducting experiments on DMS implementation tradeoffs.

ACKNOWLEDGEMENT

The author is deeply grateful to Prof. Edgar Sibley for his insightful comments and editorial pen which led to a major revision of an earlier draft of this paper.

REFERENCES

1. Applied Data Research. System Analysis Machine user guide (P302S). Princeton, N. J. May 1975.

2. CODASYL. Data Base Task Group report. ACM, New York, 1971.

3. Control Data Corporation. 6000 Scope Indexed Sequential programmer's reference guide.

{*} We typically run our model for 30 seconds to several minutes of simulated time.

4. Cullinane Corp. IDMS DDL and DML reference guides. Wellesley, Mass. April 1975.

5. Digital Equipment Corporation. MUMPS-15 Language reference guide.

6. IBM Corporation. OS data management services guide (GC26-3746).

7. -----. Information Management System/360 version 2 system and application design guide (SH20-0910-3). 1972.

8. Knuth, D.E. The art of computer programming, vol. 3: Sorting and searching. Addison-Wesley, 1973.

9. Reiter, A. Data models for secondary storage representations. Proceedings, Very Large Data Bases conference, 22-24 Sept. 1975 ACM (New York, 1975) 87-119.

10. ----. Some experiments in directory organization - a simulation study. Proceedings, Computer Performance Modeling, Measurement, and Evaluation Symposium, Boston, 29-31 March 1976. ACM (New York, 1976), 1-9.

11. ---- A study of buffer management policies for data management systems. TSR #1619, Mathematics Research Center, University of Wisconsin - Madison, March 1976.

12. ----, and Finkel, B. Simulating a virtual machine. TSR #1626, Mathematics Research Center, University of Wisconsin - Madison, May 1976.

13. Schmid, H.A., and Bernstein, P.A. A multilevel architecture for relational data base systems. Proceedings, Very Large Data Bases conference, 22-24 Sept. 1975, ACM (New York, 1975) 202-226.

14. Senko, M.E., Altman, E.B., Astrahan, M.M., and Fehder, P.L. Data structures and accessing in data base systems. IBM Systems Journal 12(1), 1973, 30-93.

15. Senko, M.E. The DDL in the context of a multilevel structured description: DIAM II with FORAL. Data Base Description (Douque and Nijssen, eds.) North Holland Publ. Co., 1975, 239-258.

16. ---- et al. File Design Handbook. Final Report, AF 30602-69-C-0100, IBM Research (san Jose, 1969).

17. Sherman, S.W., and Browne, J.C. Trace-driven modelling: review and and overview. Proceedings, Symposium on the Simulation of Computer Systems, 19-20 June 1973. ACM, 201-207.

18. Siblev, E.H. The data base future: System continuity and networking. University of Maryland ISM T.R. No. 1, Dec. 1974.

19. Siblev, E.H., and Taylor R.W. A data definition and mapping language. Communications of the ACM 16,12, December 1973, 750-759.

20. Teorev, T.J., and Merten, A.G. Considerations on the level of detail in simulation. Proceedings, Symposium on the Simulation of Computer Systems, 19-20 June 1973. ACM, 137-143

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: 1648

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle): ON PERFORMANCE MODELLING OF DATA BASE MANAGEMENT SYSTEMS - AN INDUCTIVE APPROACH

5. TYPE OF REPORT & PERIOD COVERED: Technical Summary Report - no specific reporting period

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s): Allen Reiter

8. CONTRACT OR GRANT NUMBER(s): DAAG29-75-C-0024

9. PERFORMING ORGANIZATION NAME AND ADDRESS: Mathematics Research Center, University of Wisconsin, 610 Walnut Street, Madison, Wisconsin 53706

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS: U. S. Army Research Office, P.O. Box 12211, Research Triangle Park, North Carolina 27709

12. REPORT DATE: July 1976

13. NUMBER OF PAGES: 42

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): MRC-TSR-1648

15. SECURITY CLASS. (of this report): UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

data base
data management system
data structures
performance evaluation
simulation model

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

This paper describes a simulation model in which user jobs are synthesized from basic building blocks. The modelling process consists of three stages:

translation from a user view of data and processes dependent on the data base management system (DMS) into a standard form consisting of explicit access paths over logical data structures; → next page

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

221200

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ABSTRACT (Cont'd)

cont. →

translation of the logical structures and operations into block-oriented structures and operations on a virtual machine; and

execution of a number of concurrent jobs on a real machine.

This paper deals only with the last two stages. Standard forms for the logical structures and operations and for the virtual machine are described; they are as free as possible from the data view of the particular DMS. We describe a generalized modelling framework, which becomes a model of a particular DMS when various "plug-in" modules are added. The data representation features of a DMS enter as parameters for the second stage, while the resource management tactics are the parameters for the last stage. The proposed model structure is intended as a basis for DMS design experiments.