*Information Systems* Vol. 15, No. 1, pp. 61–72, 1990
Printed in Great Britain. All rights reserved

0306-4379/90 $3.00 + 0.00
Copyright © 1990 Pergamon Press plc

# PARTITIONING AND COMPOSING KNOWLEDGE

GIO WIEDERHOLD, PETER RATHMANN, THIERRY BARSALOU, BYUNG SUK LEE and DALLAS QUASS
Stanford University, Stanford, CA 94305-2171, U.S.A.

(*Received for publication* 16 *November* 1989)

**Abstract**—This paper argues for an approach which places the management of large knowledge bases into a comprehensive, engineering-oriented framework, and reports on an initial demonstration of these concepts. The underlying concepts are well-recognized as being effective in many areas of science:

1. Partitioning of the knowledge into manageable segments.
2. Rules for the composition of these segments.
3. A language to provide access to these segments, control their composition, and provide the power of the system in a flexible and clear way.

The motivation for this research is to deal with problems that are beginning to occur in large knowledge-based systems. As current developments of such systems lead to further growth, we foresee that their management needs will exceed the capabilities of the existing system infrastructure. In particular, we find that in the past issues related to knowledge maintenance have been ignored. Maintenance of knowledge-bases is critical if the systems are to persist.

## 1. INTRODUCTION

Problems of knowledge maintenance in large knowledge-based systems motivate our research. Today these problems are evident in only some instances, but will become more prevalent as knowledge-based systems grow in scope and depth, and last beyond the lifetime of a Ph.D. Thesis. Some researchers from the AI community have looked towards database technology to help in dealing with issues of size and update management [1]. Database systems have focused on simple structuring and normalization to deal with large bodies of information, and do not deal well with the complexities of structures needed to represent knowledge.

We are using concepts from database research here as well, but must be very careful in intermingling database and knowledge-base representations. We need to avoid creating a combination with the weaknesses of the two fields, rather than the strengths. Future information systems will benefit from distributed knowledge sources and distributed computation. An architecture to deal with future systems must consider the technological opportunities that are becoming available. We see these systems supporting decision-makers through a two-phase process:

1. Locating and selecting relevant factual data and aggregating it according to the decision alternatives.
2. Processing and reducing the data so that the number of alternative choices to be decided among is small, and the parameters for each choice are aggregated to a high conceptual level.

Today most of these support tasks are carried out by human experts who mediate between the database and the decision maker. For many tasks in medicine, warfare, emergency relief and other areas requiring rapid actions, dependence on human intermediaries introduces an intolerable delay. Future information systems will increasingly need to use automatic mediators to speed up these support processes [2].

The databases, the mediators and the applications will all reside on nodes of powerful networks. The end-users will always have computers available to serve their specific tasks. We refer to those machines as application workstations, although they may at times be large and powerful processors.

### *1.1. Large knowledge bases*

We expect that future information systems will contain quantities of knowledge in order to support high-level decision-making tasks [3, 4]. A few large systems of this type exist today [5] and more are being planned, some of extremely large size [6]. In the process of building these systems and endowing them with great deductive power, the issue of long-term maintenance is underemphasized. This issue is recognized by the people actually using large knowledge bases [7].

The lack of emphasis on maintenance in early systems is easy to understand. At first, knowledge seems to be a static resource to be acquired, represented and utilized. However, the world changes, and both the underlying data and the knowledge we derive from this data change, albeit at different rates. Large and long-lived systems need a clear approach on how changes to data and knowledge are to be managed.

In database design, update has always been a concern and has affected the storage representation and hence, the methods of retrieval that are feasible. Methods for representation of knowledge which seem best for retrieval may become inadequate when updates to knowledge become a concern. In turn, a representation suitable for maintenance will require adaptation of the methods used to exploit the stored knowledge.

This paper focuses on the specifics of knowledge management. We will need to deal with knowledge update and retrieval. We have argued earlier for a distinction between data (that portion of information which can be mechanically maintained) and knowledge (the portion requiring expertise for its maintenance) [8]. Expertise is required for knowledge maintenance because changes can have wide implications. The distinction between knowledge and data is less sharp in utilization, since here integration is essential.

In addition to distinguishing knowledge and data, our approach further partitions knowledge along two dimensions: horizontally and vertically. Before describing the partitioning, however, we present some background to justify our criteria for *horizontal* partitioning.

### 1.2. *Overview of the paper*

This paper deals with a specialization of the mediator concept elucidated in [2]. The partitions we will define are the SoDs† introduced in that paper.

Our partitioning involves data and two categories of knowledge-based processing. Access to data was surveyed in [2]. In the next section we elaborate a conceptual distinction within knowledge-based systems as pragmatic vs formal approaches. This distinction defines a boundary we use for an engineered partitioning of large knowledge bases. We will assign pragmatic processing predominantly to the application layer and formal processing predominantly to the SoD layer.

Subsequently we discuss how knowledge may be partitioned into manageable units, and in Section 4 we present the approaches available for their synthesis. We follow a traditional engineering principle here: analysis of a problem into solvable subcomponents, followed by a synthesis phase into a product. Section 5 of the paper presents a simple demonstration. A mapping of the conceptual architecture into modern, distributed hardware follows. Finally, we list some hard topics yet to be addressed. In the conclusions, we discuss some generalizations now foreseen, but in our work best delayed until we have gained experience with the concepts presented here.

†The term SoD is a new term to correspond with the new concept described here. We found that all other words we could think of already had excessive semantic baggage.

## 2. PARADIGMS OF ARTIFICIAL INTELLIGENCE

The problems we are addressing are not novel, but are related to what we view as the source of some controversy in artificial intelligence research. We find two equally valid paradigms in artificial intelligence: the *pragmatic* paradigm, and the *formal* paradigm. We use these two terms simply as convenient labels, and include in the formal paradigm the logic-based approaches, which seek a formal, typically mathematical, grounding, and in the pragmatic paradigm those that focus on the cognitive aspects of human knowledge.

The knowledge-base partitioning we propose recognizes their differences and is intended to support and profit from both of them. We will briefly discuss some salient features of each.

### 2.1. *The pragmatic paradigm*

Much knowledge exists in the minds of experts. It is obtained from education and experience, and forms the most powerful tool we have for solving problems [9]. One of the great powers of such knowledge is that an expert, when confronted with a new set of facts, can use extrapolations and analogies to predict and evaluate the future effects of actions. The internal models in the experts' minds are undoubtedly quite deep and extremely difficult to extract. However, the rules by which these experts operate can be extracted, at least in part.

The acquisition of knowledge from experts has led to a large and successful activity starting from MYCIN [10] and documented in [3]. In general, only surface knowledge needs to be obtained to have effective systems focusing on advice-giving on one specific topic. Modest numbers of rules, often fewer than 100, have provided effective encodings of some experts' domain knowledge. For many domains, however, more rules are needed.

More depth in the knowledge base is needed when expert systems are to encompass knowledge covering more than one topic, i.e. knowledge from more than one expert. Due to their interaction, the number of rules for problems covering multiple topics increase faster than their sum.

Even more serious is the issue of mutual consistency, when disparate topics are joined. We cannot expect the surface extraction of the internal models of two experts, covering dissimilar but overlapping topics, to match. More depth, i.e. the explicit representation of internal causal events and the logic which leads to their external expression, is likely to be needed. Mismatches of terms used to describe internal phenomena makes the results hard to validate. The issue of mismatch in databases has been addressed by a recent thesis [11]; in expert systems the problem is harder.

User interfaces and explanation facilities further greatly increase the size of systems. When the set of rules becomes large, problems of performance, validation and knowledge maintenance become critical.

*2.1.1. The formal paradigm.* The alternative paradigm is the formal paradigm, which has received a major impetus since logic programming languages have appeared on the scene and made experimentation in this direction effective [12]. Here we often see a direct exploitation of underlying data resources, and wide variety of schemes to make data access effective [13].

The formal paradigm derives all its answers from well-founded base rules and their composition. Heuristics are mainly used to improve the performance of the systems, typically by focusing search. Most accepted heuristics can be shown to have no effect on the result values [alpha beta]; others have a small risk of missing some potentially useful results [maximal objects].

The formality of the approach provides much confidence in the results, but also leads to some obvious weaknesses. We perceive as the fundamental weakness that any provable scheme is restricted to deal with the past up to the present. Any extrapolation of results into the future can never be *proven*, since unpredictable events can always occur. Unfortunately, the beneficial use of information by decision-makers is always due to a prediction of the future.

### *2.2. Combining the two paradigms*

We need both the power of the formal approach, to make large systems predictable and manageable, and the power of expert abstraction and extrapolation. The interaction of the rules in an expert system is such that the user cannot predict the result—and that is of the essence of the service which is provided. In multi-expert systems, the roles of experts and users are intertwined. As these systems grow, a point is reached where an expert can no longer predict the outcomes. Formal structures will help with the managing the knowledge, but the complexity of interacting bodies of knowledge is such that truly large systems need a partitioning.

The right combination will let us build future systems which are both reliable and non-trivial. Combining concepts from these two paradigms is not novel; we see it everywhere in today's practice, wherever systems are effectively used. However, today's tools do not promote any partitioning of the two types of knowledge, it is even hard to separate deductive rules and ground facts.

When we analyze practical systems today, we find a mixture of both paradigms, but often a dominance of one over the other, according to the application and the taste of the designer. In a recent paper we survey a number of projects, tools and approaches that provide a knowledge-based layer for dealing with data [2]. We used the term *mediator* to capture the general concept of a knowledge layer between the user and the data. Mediators may be programs, written by an expert, in which heuristic knowledge is fully integrated with the formal techniques.

### *2.3. Heuristics*

In our discussion we often focused on the issue of heuristics. We found that use of heuristics does not in itself provide a discrimination of the pragmatic and formal paradigms. Heuristics are nearly always used to deal with computational complexity. Most knowledge processing paradigms would not be feasible in practice without their use, and optimization strategies used heuristics based on parameters such as expected domain sizes, user needs, etc. to develop practical solutions. The results obtained by such strategies are typically correct but not necessarily optimal.

In pragmatic systems we see a further exploitation of heuristics. Here application knowledge may provide heuristics about adequate approximate solutions. These may have errors in terms of set membership or rankings, but without taking such risks then no answers would be obtained. The pragmatic systems, in that sense, model with an unfortunate accuracy the situations faced by decision makers in practice.

### *2.4. Large systems*

We have stated earlier that we are primarily concerned with *large systems*. Unfortunately, there are no simple criteria for the size of knowledge-based systems. A simple count of rules is a deceptive measurement. Some apparently large systems may use a substantial number of rules to store ground facts or static data. The expert knowledge may still consist of only a few hundred deductive rules.

Some other expert systems that do embody much knowledge use fairly simple knowledge representations; for instance, AI/RHEUM [14] uses an interaction matrix of symptoms and diagnosis. Expanding such a simple representation, however, (for instance, to include issues such as time dependencies [15]) has been difficult.

## 3. PARTITIONING

There are two dimensions to the partitioning of the information systems we foresee. Horizontal partitioning divides the architecture into three main layers, as summarized in the following Table 1.

Table 1

| Layer | Type of information | Deductions supported | Implemented with |
|---|---|---|---|
| **H3** | Broad application knowledge | Pragmatic reasoning | Expert applications |
| **H2** | Formal domain knowledge | Logical inference | SoDs |
| **H1** | Factual knowledge or data | Relational algebra | Relational database |

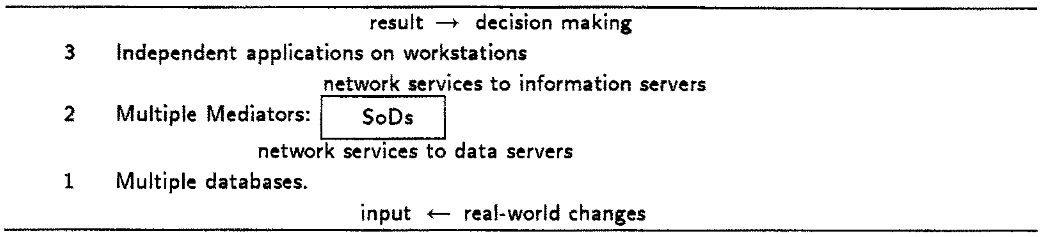
result → decision making
3 Independent applications on workstations
network services to information servers
2 Multiple Mediators: SoDs
network services to data servers
1 Multiple databases.
input ← real-world changes

Fig. 1. Interfaces for three horizontal layers of this architecture.

These layers have been sketched already in [2]. The need for distinguishing updates to factual data and knowledge (for instance constraint rules) is reiterated in [16].

There is another dimension of partitioning in our model. Layers **H1, H2** and **H3** cannot be monolithic entities, and each will be vertically partitioned. The bottom layer, **H1**, may contain multiple autonomous databases and **H2** will contain many SoDs. These SoDs may have some limited interaction as peers, but more importantly, they will share the underlying databases by means of views. At the top level (**H3**), multiple applications will exist, sharing and combining knowledge from the SoDs of layer **H2.**

This paper focuses on the central issue of knowledge partitioning in the relatively formal layer **H2**, but must, of course, also deal with the interfaces to the supporting data layer **H1** and the supported application layer **H3**.

### *3.1. Desiderata for the partitioning of knowledge into SoDs*

Recall that our objective is to make knowledge manageable. Two prerequisties have to be fulfilled to achieve this goal:

1. The knowledge can be formally Structured.
2. The knowledge is limited to manageable Domain of discourse.

Since eventually, we also wish to support automatable combinations of the results of the SoDs, we also must be concerned that the SoDs use mergable representations for their knowledge. It is hence not adequate to have arbitrary dissimilar SoDs, whose results are presented on, say, distinct windows of a terminal. This approach forces the end-user to perform all integration visually or manually by cut-and-paste methods. While having multiple terminal windows provides an advance over a desk piled high with multiple pieces of paper and terminals, it does not cover our vision of the future.

The principle for the vertical partitioning into SoDs is based on *Domains of Knowledge*. These are seen to correspond simultaneously in multiple dimensions:

1. They are limited in scope so that *one expert* can cover them and recognize inconsistencies.
2. They each have *one consistent* structure imposed on them so that changes in knowledge (or the underlying facts which led to a piece of knowledge) can be accommodated with secondary changes of limited and predictable scope.
3. They deal with *one constrained* set of *base data* so that updates to the underlying data and data structure required by new knowledge can be handled effectively and unambiguously.
4. They produce *one range of results*, understandable in terms of scope and depth by the end-user applications.

An essential hypothesis for our research is that the partitions for these four dimensions can be made congruent. Complementary to this partitioning hypothesis will be a combination hypothesis, to follow in Section 4.

From these criteria we can derive some more specific observations on the natural structures we expect to find in the domains of a SoD. We plan to exploit these structures whenever possible.

### *3.2. The structure and related semantics of SoDs*

The structure of a SoD expresses the structure of its semantics. We hope to have many structurally similar SoDs, since we expect that that common structures will appear in many different domains, although their labels and cardinalities may differ greatly. Our goal is as well to decompose knowledge-bases so that the structure of many SoDs will be simple. We prefer hierarchical structures, but realize that some SoDs will need to use sets, DAGs or more complex representations.

*3.2.1. Hierarchical structures.* In any specific domain there is strong tendency to impose a hierarchy on the knowledge structures, which often corresponds with organizational requirements of the organizations dealing with the information. Hierarchies are instantiations of the divide-and-conquer paradigm we are trying to exploit also within the SoDs. When manipulating data through a hierarchy, we have a predefined generalization–specialization structure. Processing in such a structure is much easier to manage than in arbitrarily connected networks of knowledge—many important problems which are intractable for general graphs have $\mathcal{O}(n \log n)$ solutions for hierarchies.

The hierarchical structure is beneficial in operations of grouping and aggregating base data into higher level abstractions, searching for specific information defined by predicates which describe such abstractions, and disambiguating updates.

While predicates can specify abstraction levels directly (department in a personnel hierarchy) quantitative goals may be satisfied by finding the right level of the hierarchy. If we need more programmers than we can find in our department, a move up to the division level may satisfy that request [17].

*3.2.2. Closed worlds.* We expect our SoDs to be self-describing and inspectable, and an important part of a SoD is a description of what kinds of assumptions we can make about the domain the SoD represents. Some of the SoDs will be able to support the closed-world assumption (CWA) [18]. This assumption is commonly made when dealing with databases, but is risky for general expert systems. If the maintaining expert's confidence and the intrinsic definition of a domain is such that the CWA holds, then operations requiring universal quantification and negation can be supported in SoD, otherwise they should not be supported.

In a SoD dealing with corporate personnel and maintained by an expert attached to the personnel department, the CWA is likely to be valid. A SoD dealing with database consultants may be able to locate many instances of consultants, but is unlikely to be able to locate all of them until an ACADEMY OF DATABASE CONSULTING is established, and all non-members are disbarred.

Since SoDs are not restricted to relational data we will eventually need to support more flexible formalisms, such as circumscription [19].

*3.2.3. Closure.* We will often look for a SoD to provide all instances satisfying some precisely stated criteria of relatedness. For example we may want to find all the descendants of a given person, or find all the papers which are "similar" to a given example paper. Such queries look for some sort closure in the domain, and for SoDs with a hierarchical structure, these queries are often expressed most naturally in terms of transitive closure. At other times, like in the "similar" papers example, we will to use distance-based concepts which are not transitive. While for simple structures, such closure-based queries can be dealt with by simple extensions to database query languages, we will need to provide some fairly complex computations to answer such queries over more general SoDs. This issue intertwines closely with the ideas of closed-world SoDs expressed above. Whether or not the closed world assumption holds in a SoD may affect the implementation of a closure-based query, but more importantly, drastically affects the interpretation and confidence to be attached to the results of the query.

### *3.3. Evaluation functions*

For decision-making process we often need only the *n best* alternatives according to some ranking. The rank is obtained by an evaluation-function; such functions may be simple (say, the highest paid programmers) or complex (say, the best programmers). The SoDs for these two queries may be distinct, although the database views needed for the evaluation may be overlapping. The highest paid programmers are obtained from the personnel SoD; the challenge here is to find an efficient algorithm that can avoid unnecessary database accesses. The SoD to find the best programmer will be complex and will depend both on some experts' insights and on complex database access functions in order to collect all the correlative data. In fact, there may be more than one SoD available to answer the best-programmer query, say the Brooks-best-programmer SoD and the Orr-best-programmer SoD.

*3.3.1. Inspectability.* We now arrive at a new criterion for SoDs. We wish them to be inspectable. Whereas simple formal systems may hide lower level information in order to maintain application independence, we cannot see doing this for SoDs because the application user should have the capability of determining whether the Brooks-SoD or the Orr-SoD is best for the current objective. Such an inspection may be mediated by an inspector SoD, and may not support copying of the SoD or direct access to the base data.

*3.3.2. Declarative approaches.* To support inspectability it is desirable that, as much as possible, the processes within a SoD be driven by declarations and formal parameters. We would hope to capture the differences of Brook's evaluation and Orr's evaluation by parameter settings and that the same processing routine can be employed by both SoDs.

### *3.4. From data to SoDs*

Because we propose a partitioned architecture for future information systems, an important issue is the interface between the supporting data layer **H1** and the SoDs of layer **H2**. Although the SoDs are most naturally implemented with object structures (as discussed in Section 5), we use relational databases as the storage scheme for factual information of layer **H1**.

Storing information in the form of complex objects can seriously inhibit sharing—different groups of users will need to assign different object boundaries to the same information [20]. However, object-oriented presentations of information can be clearer and more concise than long tables of voluminous text. A desirable compromise is to provide an object-oriented interface to relational data, combining many of the better features of each representation [2]. Such an interface serves as an effective mapping from databases to SoDs, translating **H1**'s relational tuples into **H2**'s object instances. An active area of our research has been directed toward this goal.

We introduce an object-based interface on top of a relational database system. This architecture does not call for storing objects explicitly in the database, but rather for generating and manipulating temporary object instances by binding data from base relations to predefined object templates. The three components of the object interface are:

1. The *object generator* maps relations into *object templates*; each of which can be a complex combination of join and projection operations on the base relations. In addition, an *object network* groups together related templates, thereby identifying different object views of the same database. The set of object networks constructed over a given database form an *object schema*, which, like the data schema for a relational database, represents the domain-specific information needed to gain access to the objects. The whole process is knowledge-driven, using the semantics of the database structure.
2. The *object instantiator* provides non-procedural access to the actual object instances. A declarative query specifies the template of interest. Combining the database-access function (stored in the template), and the specific selection criteria, the system automatically generates the relational query and transmits it to the DBMS, which in turn transmits back the set of matching relational tuples. In addition to performing the database-access function, the object template specifies the structure and linkage of the data elements within the object. This information is necessary for the tuples to be correctly assembled into the desired instances.
3. The *object decomposer* implements the inverse function; that is, it maps the object instances back to the base relations. This component is invoked when changes to some object instances need to be made persistent at the database level. An object instance is generated by collapsing (potentially) many tuples from several relations. By the same token, one update operation on an object may result in a number of update operations that need to be performed on the base relations. We plan to apply here results of research in the KBMS project, which deals with updating through relational views [22].

An object template therefore represents a view of the database. Instantiation selects, retrieves and aggregates relevant data into object instances that can now be manipulated by a SoD. In addition, the SoDs can share factual information by sharing the object templates and their access functions. The same object, say a person, can be instantiated by more than one SoD, let's say in one SoD as a faculty member and in another SoD as a database consultant.

The formal design for this approach is domain-independent. It is then our belief that ideas, principles and programs developed in this process will be applicable to other knowledge-based interface approaches.

## 4. COMPOSITION

A single SoD has a power which is comparable to that of a simple expert system with access to a database or, in the database paradigm, of an advanced database query processor. Such systems are typically limited to one domain and implemented using one type of structure. Simple hierarchical systems can be effective for some tasks, as classification, ranking of alternatives, etc. but are rarely adequate for multi-objective assessments and decision-making support [23]. We do not wish to make our SoDs more complex, lest we lose maintainability.

Instead we wish to make them composable. Successful composition is the second hypothesis in this research.

One way in which SoDs can cooperate is as peers, working like a team of expert advisors to a top executive, to solve a common problem. In order to cooperate, they will need a way to exchange information. To facilitate this, the high-level language, by which applications query and command SoDs should have good algebraic properties. We discuss the basic language features in this section and will return to present further work needed in Section 7.

There is another kind of composition that must be supported, and this composition relates to the internal structure of a SoD. A SoD is a complex entity, containing data, inference techniques, knowledge and abstractions. All these subunits should be sharable, and in fact must be shared wherever possible. This is the exactly same reason that databases are normalized or software is built of reusable modules—duplicated structure leads to inefficiency and update anomalies.

### *4.1. An access language for SoDs*

We envisage SoDs to be used by high-level, heuristic applications. Flexibility of access requires that the interface be non-rigid, and the intent to be able to deal with multiple SoDs in an application requires composability as further addressed in the next section.

We hence specify an access language SAL which provides access to information produced by the SoDs. This information is seen to have the form of instantiated complex objects, similar to the nested-relation tuples described by [24]. The mappings from data resources to these objects is hidden within SoDs. We do need, however, some additional functionality.

This language is not fully specified, but it must support primitives to specify:

1. Selection of subsets of objects satisfying user defined criteria.
2. Transitive closure
3. Constraints on the cardinality $r$ of answer sets.
4. A *best* predicate to select from a ranking.
5. Computation over temporal data.

We will not discuss this last feature in this paper, although it is obvious that to project results into the future, some temporal processing is needed, as shown in some of our earlier research [25, 26].

It is important to note that distinct SoDs may support the extended primitives (2–4 above) in different ways, dependent on the structure of their domains. The *best* predicate is especially likely to be interpreted in a domain-sensitive manner. Without a defined *best* predicate a SoD can just return the $r$ first object instances when a cardinality constraint is imposed.

Note that this language is intended to provide a smooth and sensible transition between the traditional database and PROLOG styles of data retrieval. The database style, exemplified by DATALOG, retrieves all instances, i.e. implies a cardinality constraint $r = \infty$. The Prolog style retrieves initially the first instance found, implying $r = 1$. Having the cardinality specified explicitly also addresses a vexing problem in the database-to-programming-language interface. Most programming languages deal only with fixed length structures, or at best with variable length structures up to a certain maximum size. (As our current demonstration is implemented in Lisp, this issue does not now arise.)

Today, without the knowledge encoded in SoDs, the methods for retrieving the *best* information are explicitly specified by the user. It is likely to require distinct methods for multiple domains. Both in database and Prolog access styles, these specifications require knowledge of each the underlying domains and their structure. In today's database languages a sensible specification is likely impossible to state, so that all the data has to be retrieved into memory, and then processed and reduced by application programs.

The application at layer **H3** takes the information provided by the SoDs, composes it, and reduces it as desired by intersecting results of distinct SoDs with each other. It also presents the information in the most appropriate forms to the user. To service the **H3** layer we are looking for a language similar in style to a relational algebra, rather than to a language such as SQL which attempts to provide a user-friendly interface as well as programmed access, and fails at both.

Having a language interface simplifies the tasks of the SoDs at layer **H2** since direct external presentation issues are ignored. Enough corresponding meta-data must be made available to layer **H3** so that smart formatting and pleasant presentation is feasible [27]. We have not addressed this issue yet. We are experimenting with a smart menu system, using such knowledge.

### *4.2. A SoD result language*

In order to make SoDs comparable, one SoD must be able to act on the results of another. We therefore define a SoD result language SOREL by which this kind of communication can take place. SOREL will be extremely simple and limited, especially in comparison to the SoD access language. One way of looking at this is to realize that SAL is in some sense a union of capabilities, since it must be powerful enough to express anything we would ask of a SoD, while the SoD result language is more like an intersection, since it should be understood by all SoDs.

The exact form of SOREL will depend on the SoDs and the needs of the application, but for most applications, we expect the SoDs to return only ground data, i.e. tuples, relations and object identifiers.

The answers given by a SoD in SOREL will be returned to the application at **H3**. The application may then use results directly or as input to another SoD.

*4.2.1. Identifying shared objects.* One of the first problems that must be handled for SoDs to work together cooperatively is to get them to agree on a common ground for communication. Human experts often disagree as to the meanings of words or of concepts, and this will be a problem for SoDs as well. In one common case, the architecture and its support for definitional composition can help greatly in identifying shared objects. Because our SoDs can be built from simpler, shared components, it will often happen that two SoDs will be using an object created at a lower level. In this case, it is easy for the SoDs to recognize that they are sharing the same object—they are both looking at the same object identifier.

In the more general case, identification is not this easy. If the SoDs are using objects created on different computer systems, or if the objects are created at a higher level, we can easily have two computer "objects" (with distinct identifiers) that nevertheless denote the same abstract object in the real world. When this happens, we will have to compare the objects, relying on key values and matching heuristics. Perhaps we can invoke a SoD to help us with the merging tasks. If the domains of the attributes to be merged are actually mismatched, then we certainly need intelligent processing, and we may need rankings based on the *best* match [11].

### *4.3. Power of the combined system*

By partitioning the knowledge base, we gain the ability to use and combine special-purpose SoDs and their knowledge representations without having to build one super-interpreter which understands all knowledge representations (and all the combinations of the knowledge representations). This partitioning then makes maintenance much more tractable. However, in partitioning the data into SoDs, and allowing them to communicate only via the restricted SoD result language, we lose some of the arbitrary connectiveness associated with knowledge representations, such as semantic nets.

This loss of connectivity may reduce the expressive power of the system. For example, let's say that when designing a wing, an aircraft designer looks at two SoDs, one of which can evaluate and optimize a design for aerodynamic performance, and another SoD which looks at mechanical strength and weight. Since the wing should have good performance according to the criteria of both SoDs, the designer is faced with an iterative (or even trial-and-error) process, of checking designs though both SoDs and looking for a global optimum.

This iterative process might have been avoidable, if the two SoDs were unified into one super-SoD able to find a global optimum for aerodynamics, strength and weight. What this example tells us is that the design process which splits knowledge into SoDs is quite critical. A given partitioning may gain us a great deal in terms of implementation, maintenance and assignment of responsibility, but may also incur a significant cost in expressive power.

## 5. A DEMONSTRATION

To demonstrate the concepts, the students on the KSYS project have chosen the task of assigning reviewers for journal papers submission. Four SoDs serve the task:

1. Relevance—we need reviewers with a background relevant to the submitted paper. This task is performed by matching in a keyword classification hierarchy.
2. Quality—we prefer the most qualified reviewers. For this task we rank potential reviewers based on their published output in books, journals, etc.
3. Conflict avoidance—we cannot assign reviewers to friends or colleagues. Here we match people based on institutional affiliation in overlapping intervals.
4. Responsiveness—the reviewers must produce their reviews in time. Here we can look at a log of electronic-mail interactions.

We can use this example to elucidate the difference of the AI paradigms allocated to level **H3** and **H2**. The tasks in the SoDs at layer **H2** can all be defined quite formally. At the top layer **H3** some unwarranted pragmatic heuristics are used to implement the reviewer selection task. For instance:

1. Having written high-quality publications in a topic area does not assure one that the candidate does equally well as reviewer. It is the best guess that our application task can make, but we all know some excellent critics who do not write much. The mapping of qualified_writer → qualified_reviewer is pragmatic. The establishment of a set of qualified_writers is adequately formal to justify its allocation to a SoD.
2. Having worked together does not make one a friend, and being a friend does not imply favoritism. But we do not need to weed out risky matches—in fact, due to prior publications the most likely *best* match is the submittor of the paper.
3. Electronic mail responsiveness is probably only weakly correlated with fast reviewing—there are people who respond instantly to email and never respond to review requests.

The language currently used between layers **H1** and **H2** is LISP because it supports the extensibility essential to rapid research progress.

The data accessed by the first three SoDs are distinct views of an extensive bibliography of knowledge and database references, collected over about 18 yr, with about 6000 entries. Information kept includes type of publication (for the Quality-SoD), authors (the principal identifiers), author's location (for the Conflict-avoidance-SoD), publication details and sequence with dates (for the Conflict-avoidance-SoD), title, abstract and classification (the last three are used by the Relevance-SoD).

Two of the SoDs are currently implemented—relevance and conflict avoidance. They are implemented as Lisp programs which have access to the object system and a commercial relational database. As we gain more experience, we intend to replace the Lisp code with a more declarative representation.

At the simplest level, the Relevance-SoD takes a keyword (or list of keywords) describing the subject of the paper to be reviewed, and looks in the database for authors who have written papers on the keyword(s). If enough authors are returned by this database query, this is all that happens. If however, the database query does not find enough authors, or if the application asks for more candidate reviewers later, the Relevance-SoD will replace the original query by a more general one, in order to increase the cardinality of the result.

This capability is an example of query generalization [17]. It is possible because the SoD makes use of some of the semantics of the keywords. The keywords are arranged in a hierarchy, in which the parent is the more general keyword, and the children the more specific. If a query does not return sufficiently many results, a concept of semantic distance in the hierarchy is used to suggest alternate keywords to try.

The data structures used by the SoD are designed to efficiently support this kind of iterated query style efficiently. A set of authors can be found by a succession of related queries can be answered with about the same total effort as would be needed to find that same set of authors with one more general query.

The application interface is simply a set of Lisp functions which the application can use. As our system evolves, we intend to build a higher-level interface. In our design, an application which is looking for reviewers would submit a query of the form:

```
select best 3 reviewer
from relevance-SoD
where relevant='knowledge-base'
  and reviewer not in
    (select all friend
    from conflict-avoidance-SoD
    where author='Gio Wiederhold')
```

Note that this query refers to two different SoDs. Since a particular SoD can only answer queries about its own domain, this query is translated into a slightly lower level form, which specifies the individual queries to the SoDs, and the information flow between them.

```
a:=select all friend
    from conflict-avoidance-SoD
    where author='Gio Wiederhold'
b:=select best 3 reviewer
    from relevance-SoD
    where relevant='knowledge-base'
    and not in a
RETURN b
```

Note that even this second query is fairly high level. It refers to such abstractions as relevant, which are implemented by the SoDs.

## 6. THE IMPLEMENTATION ARCHITECTURE

The demonstration is implemented in fairly straightforward way, but a short description will illustrate some issues better than an abstract discussion can.

### 6.1. SoD implementation

The criteria we have listed encourages an implementation which supports object-oriented type definitions. We building a simple Lisp-frame structure to support SoDs. Low level frames correspond to database schema entries and support retrieval from databases; data that is retrieved must be bound into the object-type structures and represent object instances as discussed in the previous section. Concepts such as trackers [13] deal with effective handling of partially- or fully-instantiated sets of data.

Functions and predicates from the object type definitions are inherited by the object instances. Default values are overridden by any actual data retrieved from the database.

Common methods, as selection and transitive closure, will be shared by multiple SoDs, especially when their general structure is similar. Sharing should be possible even if their object types and instances differ. General parameters, as the CWA, can cause alternate variants of methods to be invoked. Note again that SoDs are not distinguished by their program structure or algorithms, but rather by their structure and domain knowledge.

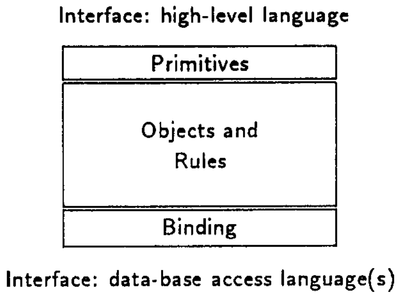
Interface: high-level language
Primitives
Objects and Rules
Binding
Interface: data-base access language(s)

Fig. 2. The components of a SoD.

### 6.2. Elements of a SoD

The specific structure of a SoD is shown in Fig. 2. Each SoD contains a simple hierarchy of objects pertaining to its domain. The views provided by the objects compose the entire view of the SoD over the database. Rules embedded in the object structures are used to control the instantiation of objects and compute dynamic slot values. Each SoD provides primitive operations it can support and accepts the parameters it needs from applications. Binding interfaces the SoD objects with the underlying databases by retrieving object instances generated from the databases. For efficiency, the instances of the SoD objects are bound into memory as early as possible.

In terms of their external interface we expect SoDs to be free-standing units, accessible on the high-speed communication networks now in the planning stage. For efficient execution, SoDs can be replicated on other computing nodes where the data (**H1**) or the applications (**H3**) reside.

### 6.3. Structure of an object

We indicated earlier that we are implementing the SoDs using frames, similar to those seen in the UNITS system [28] and its successors such as KEE and RX [25]. This means that an object is implemented as a frame in a Lisp structure.

A frame is composed of a number of slots. Each slot is labeled, and contains the following elements:

1. An indication of its SoD membership.
2. A domain definition, to constrain its values.
3. A value.

Values may be constants or references to other objects. Constants occur mainly in frames that have been instantiated from the database.

An object frame inherits its slots from the SoD that it is a member of. A frame in our system that belongs to a single SoD differs but little from frames seen in other systems. The differences arise when an object becomes a member of multiple SoDs.

### 6.4. Structural support for composition

An object may be a member of multiple SoDs. It is the task of the binding layer to recognize that information for an object already exists and performs the binding for the two overlapping instances.

The joint object will inherit the slots from all the SoDs it belongs to. We see here a departure from the common schemes used when multiple inheritance is needed: the information is not intermingled according to local rules. Since we mainly use information from the database, it is not likely that there will be rules to cover the variety of interactions that can be realized among objects from distinct SoDs. The disjoint inheritance is also imposed on the values in the object slots:

- An inherited slot value is inherited from only one specified SoD.

We hence provide for multiple inheritance into objects, but not into the same slots of objects. This rule eliminates the multiple inheritance problem for which no general solution is likely to be found. We find solutions that have been proposed too specific for a general system, but recognize that multiple inheritance is a valid and useful concept.

An example will clarify our approach. Say that the Personnel-SoD has retrieved an individual (John) with location, job-classification, salary, etc. information from a PEOPLE database. John is also being retrieved by a Skills-SoD as possessing the skills and a willingness to do weekly consulting on some topic. The slots identifying John are identical and shared, not requiring inheritance. The salary slot belongs to the Personnel-SoD, and may either be explicitly retrieved or filled in by inheritance for all employees of that classification. The fee slot belongs to the Skills-SoD, and may be estimated by averaging known fees of similar individuals. There is less likely to be a well-established hierarchy here.

For some decision-making process at layer **H3** we may actually need an income estimate. The application can obtain the distinct components and combine them as it pleases.

If the task of estimating incomes is frequent and consistency is desired, then it should be formalized. This means we assign a new expert to the task and let her define a SoD for income estimation. An income slot, inherited from that SoD may be adjoined to the object for John and income is then computable on the basis of salary, fee, alimony, and any other financial reward slots that other SoDs may instantiate in this object. The values in this slot will not be subject to inheritance, only the formula is inherited.

It is clear why inspectability of SoDs is needed. The questions of composability are so complex that it is often desirable to determine how a value as income is computed. Still, we wish to delegate the actual computation to a SoD, in which we normally place some trust. The confidence in the SoD emulates confidence we have in the reports and summaries provided by specialists from our Personnel department, the Skills specialists and in our assistants who compose the information. Only if we need to question the result do we inquire into their methods.

## 7. SUBPROBLEMS TO BE ADDRESSED

The task of managing large knowledge-bases, which undergo growth and change is daunting. While we have sketched those aspects of our approach that seem clear to us, there are many tasks which require expansion and generalization.

We will list some here. For some of these we have some ideas on how to address them, other problems are quite open.

### *7.1. Object identification*

Correct object identification is critical for the matching operations at layers **H3**. While objects instantiated with SoDs at layer **H2** have a simple linkage with the underlying database, we can use database keys or derived surrogates from layer **H1** to identify objects.

When derived objects are created within SoDs such identifiers may become difficult to link. The fact that SoDs will share computational processes can help, but probably not guarantee correct matching when information follows different processing paths.

### *7.2. Dynamic slot generation*

Dynamic slot values are derived using knowledge about the data in the databases. This may take the form of a default values when the base data are unpopulated, procedural functions over the base data or declarative rule sets.

| Label | SoD | domain | value |
|---|---|---|---|
| ID | — | identifier | internal |
| name | — | identifier | John |
| job_class | Personnel | code | G21 |
| salary | Personnel | dollars | 35000 |
| deductions | Payroll | count | 3 |
| skill | Consult | code | 2324, 2386, 3756 |
| fee | Consult | dollars | 1000 |
| willingness | Consult | +scale | 4 |
| income | Estimator | formula | salary + alimony*12 + fee*52 |

Fig. 3. Frame with SoD labeled slots.

The issues in this area involve deciding at what point to compute the derived value and determining how to recompute this value when the base data changes. It may even be that some derived values are stored in the database for efficiency. In this case we may need trigger mechanisms to update the values when the base data changes.

At a higher level of abstraction we must consider how objects acquire new slots. In our example a slot was acquired by merging selected objects with the Estimator-SoD. How such a procedure can be generalized has not yet been defined. A follow-on phase could have the application at **H3** define a private SoD, or its equivalent, so that private computations can be attached to materialized objects in layer **H2.** We do not foresee dynamic generation of data accessing slots.

### *7.3. Language optimization*

Choosing an algebraic language SAL for communicating with the SoD should enable optimization. Currently, we process the SoDs in the order mentioned in the task definition, but other sequences are likely to provide better performance. While we understand issues of join ordering [29], we now have new operations that will require new optimization rules.

This SAL language operates on larger granules of primitives than current 4GL languages. Semantically similar primitives of the language will be executed differently in the various SoDs. To perform global optimization the SoDs have to be able to provide abstractions or evaluation functions of their methods to the global optimizer.

Note that SoDs interact at the language interface level in at least two ways:

1. The output from one SoD may help another SoD reduce its search.
2. The output from one SoD may necessitate a previously-executed SoD to be reexecuted.

For example, when searching for "three competent and responsive reviewers," the list of competent reviewers could help reduce the search for responsive reviewers, but if only one of the competent reviewers turns out to be responsive, then perhaps the "competency" test should be relaxed and reexecuted in order to return the requested three reviewers. In neither of two cases will it be necessary to ship large volumes of data for resolution of the intersection result to the computer used for the application.

### *7.4. Object instantiation*

In the system design adopted for KSYS, a binding module interfaces between the frame system layer and database layer. It provides object instances generated from databases data into the frame system.

The instances of frames used by SoDs are generated from relational databases. Each frame prototype for a SoD defines a view of the database for selecting a subset of the database as frame instances. When frame instances are needed, its view is translated into a relational query and delivered to the database. The query results are stored in main memory and processed. We expect that for many complex queries delivered to the database we cannot achieve reasonable performance by simply delivering the queries to the database.

We are thus developing a *binding* strategy for minimizing accesses to secondary storage databases. The binding strategy is to cache the multiple query results in a nested, prejoined form for compact storage for retrieval of frame instances. Queries delivered to the database are modified as needed whenever the binding module detects that relevant reusable query results have already been bound into the main memory.

### *7.5. Interacting SoDs*

At present the top application layer is the executive responsible for the integration of knowledge obtained from SoDs. An extension of this architecture we must investigate is the hierarchical composition of a SoD from subSoDs. In this way the parent SoD would perform the task of integrating knowledge from subSoDs, and itself might be a subSoD of another SoD. For this to be possible, the interface exported from a SoD (i.e. the query language supported) must provide a superset of the functionality used by a SoD.

This direction moves us closer to the interacting ACTORS paradigm [30]. We do, however, still expect to impose constraints on their composition, and in that sense are closer to concepts of the ORG approach [31].

## 8. CONCLUSION

We have presented an approach to deal with the management of large knowledge-based systems. The approach is based on a domain and structure-sensitive partitioning of the data and knowledge to be managed, and careful and limited interactions among the partitions. A simple demonstration illustrates our approach.

We define the criteria for SoDs, our principal unit for the partitioning, and discuss the effects of the criteria. With the benefits of partitioning a loss of power is induced; we can no longer navigate in seemingly arbitrary ways throughout the knowledge base. It is difficult to assess the cost–benefit ratio of this tradeoff. We are optimistic that it is high; analogies can be found in human organizations as well as in other large computer systems.

In our current demonstration the efficiency cannot be measured. We know that acceptance of new technology requires both conceptual benefits as well as reasonable efficiency and we hope to gain efficiency with our binding approaches. These will benefit from the structure information that SoDs provide.

Automation of techniques of knowledge management will be essential in a wide range of future applications. We hope and expect that the principles we have laid out will contribute to an orderly and productive growth of the field.

*Acknowledgements*—This paper presents research results developed primarily with the KBMS project on management of large knowledge bases, supported by DARPA under Contract N39-84-C-211. Useful insights were gathered by interaction with researchers at DEC (project title "Reasoning about R1ME") and the RX knowledge acquisition project, NCHSR/DHHS HS04389, NLM LM-4334, by NIH RR HD-12327, RR-0785 and F32 GM08092. We would like to thank Professor Arthur Keller, Surajit Chaudhuri and Keith Hall for their help and careful reading of earlier drafts of this paper.

## REFERENCES

[1] L. Kerschberg (Ed.). *Expert Database Systems*. Benjamin-Cummins, Reading, Mass. (1985).

[2] G. Wiederhold. The architecture of future information systems. (Abstract) *Proc. Int. Symp. on Database Systems for Advanced Applications, KISS and IPSJ*, Seoul (1989).

[3] E. Feigenbaum, P. Nii and P. McCorduck. *The Rise of the Expert Company: How Visionary Companies are Using Artificial Intelligence to Achieve Higher Productivity and Profits*. Times Books (1988).

[4] L. B. Methlie and R. H. Sprague. Knowledge representation for decision support systems (1985).

[5] J. Bachant and J. McDermott. R1 revisited: four years in the trenches. *AI Mag.* **5**(3), 21–32 (1984).

[6] D. Lenat, M. Prakash and M. Shepherd. Cyc: using common sense knowledge to overcome brittleness and knowledge acquisition bottlenecks. *AI Mag.* **6**(4), 65–85 (1986).

[7] V. E. Barker and D. E. O'Connor. Expert systems for configuration at Digital: XCON and beyond. *Commun. ACM* **32** (3), 298–318 (1989).

[8] G. Wiederhold. Knowledge versus data. *On Knowledge Base Management Systems* (M. L. Brodie and J. Mylopoulos, Eds), pp. 77–82. Springer-Verlag, New York (1968).

[9] D. Lenat and E. Feigenbaum. On the thresholds of knowledge. *IJCAI 87*, Milan (1987).

[10] E. H. Shortliffe. *Computer-Based Medical Consultations: MYCIN*. Elsevier, New York (1976).

[11] L. De Michiel. Performing operations over mismatched domains. *IEEE Data Engng 5*, Los Angeles, Feb (1989).

[12] H. Gallaire, J. Minker and J.-M. Nicolas. Logic and databases: a deductive approach. *ACM Comput. Surv.* **16,** 153–185 (1984).

[13] S. Ceri, G. Gottlob and G. Wiederhold. Interfacing relational databases and PROLOG efficiently. *IEEE Trans. Software Engng* Feb, 153–164 (1989).

[14] L. C. Kingsland, D. A. B. Lindberg and G. C. Sharp. AI/RHEUM: a consultant system for rheumatology. *J. Med. Systems* **7,** 221–227 (1983).

[15] A. Bolour, T. Anderson, L. Dekeyser and H. Wong. The role of time in information processing: a survey. *ACM SIGART Newslett.* **80,** 28–48 (1982).

[16] H. Katsuno and A. O. Mendelzon. A unified view of propositional knowledge base updates. Report Univ. of Toronto (1989).

[17] S. Chaudhuri. Generalization and a frameword for query generalization. *IEEE Data Engng 6*, Los Angeles, Feb (1990).

[18] R. Reiter. On closed world data bases. *Logic and Data Bases* (H. Gallaire and J. Minker, Eds), pp. 119–140. Plenum, New York (1978).

[19] J. M. McCarthy. Circumscription—a form of non-monotonic reasoning. *Artificial Intell.* **13,** 27–39 (1980).

[20] G. Wiederhold. Views, objects and databases. *IEEE Comput.* **19**(12), 37–44 (1986).

[21] T. Barsalou. An object-based architecture for bio-medical expert database systems. *Proc. Twelfth Symp. Comput. Applications in Medical Care*, pp. 572–578. IEEE Computer Society (1988).

[22] A. M. Keller. The role of semantics in translating view updates. *IEEE Comput.* **19**(1), 63–73 (1986).

[23] J. R. Miller. *Professional Decision Making—A Procedure for Evaluating Complex Alternatives*. Praeger, New York (1970).

[24] M. A. Roth, H. F. Korth and A. Silberschatz. Extended algebra and calculus for nested relational databases. *ACM TODS* **13** (2), 389–417 (1988).

[25] R. L. Blum. Discovery and representation of causal relationships from a large time-oriented clinical database. the RX project. *Lecture Notes in Medical Informatics*. Springer-Verlag, New York (1982).

[26] I. deZegher-Geets, A. Freeman, M. Walker, R. Blum and G. Wiederhold. Summarization and display of on-line medical records. *MD Comput.* **5**(3), 38–45 (1988).

[27] J. Mackinlay and M. Genesereth. Expressiveness and language choice data. *Knowledge Engng* **1**(1), 17–29 (1985).

[28] M. Stefik. An examination of a frame-structured representation system. *IJCAI* **79,** Tokyo (1979).

[29] A. Swami and A. Gupta. Optimization of large join queries. *Proc. ACM-SIGMOD Int. Conf. on Management of Data* (1988).

[30] C. Hewitt, P. Bishop and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. *IJCAI 3, SRI*, Aug, pp. 235–245 (1973).

[31] T. W. Malone, K. R. Grant, F. A. Turbaks, S. A. Brobst and M. D. Cohen. Intelligent information-sharing systems. *CACM* **30**(5), 390–402 (1987).