A GRAPH-ORIENTED KNOWLEDGE REPRESENTATION AND UNIFICATION TECHNIQUE FOR AUTOMATICALLY SELECTING AND INVOKING SOFTWARE FUNCTIONS

William F. Kaemmerer and James A. Larson
Honeywell Computer Sciences Center
Artificial Intelligence Section
1000 Boone Avenue North
Golden Valley, Minnesota 55427

ABSTRACT

An interface to information systems that can automatically select, sequence, and invoke the sources needed to satisfy a user's request can have great practical value. It can spare the user from the need to know what information is available from each of the sources, and how to access them. We have developed and implemented a graph-oriented technique for representing software modules and databases, along with unification and search algorithms that enable an interface to perform this automatic programming function. The approach works for a large class of useful requests, in a tractable amount of run time. The approach permits the logical integration of pre-existing batch application programs and databases. It may also be used in other situations requiring automatic selection of software functions to obtain information specified in a declarative expression.

I INTRODUCTION

Users of computerized information systems often need to access multiple sources of information and multiple software programs in the course of performing a single, practical task. For example, a bank loan officer may need to access credit records, automobile book values, and amortization software to determine whether to grant a car loan. The use of multiple systems can burden users with the task of choosing which system to invoke to obtain each piece of desired information, and with the mechanical details of obtaining and combining intermediate results. A means is needed by which a person can access diverse information sources and software functions without being distracted by these details.

A way to meet this need is to provide a user-system interface that allows a person to access diverse information sources as if they were a single, virtual information system. We have developed and implemented an algorithm that automatically selects and sequences the "servers" needed to respond to a request for information stated in server-independent terms. (We use the term "server" to refer collectively to pre-existing batch application software as well as databases residing under database management systems.) The output consists of a series of expressions sufficient to invoke the servers and obtain the desired information.

II PROBLEM DEFINITION AND TERMINOLOGY

We use the term "server-unit" to refer to each retrievable unit of data (e.g., each type of tuple in a relational database) or each invokable function provided by an individual server. Our justification for applying the same term to data and functions is the observation that an invokable function of a server can also be considered a type of retrievable unit of data. It may be represented as a virtual relation between its input and output arguments. Each individual server may provide multiple server-units. For a relational database management system, a server-unit corresponds to each of the relations in the database. For an application program, a server-unit corresponds to each entry point of the program.

We view the functions and information available from a set of servers as collectively defining the "capability space" of a single, virtual server (Ryan and Larson, 1986). A representation of this space is derived by merging the representations of the server-units for each of the actual servers. Given a means of representing the semantics of the information collectively available from the server-units, the user may request information in server-independent terms by declaratively expressing the desired result in terms of the capability space. Satisfying the request is then a matter of finding and sequencing a set of server-units that is a procedural equivalent to the user's declarative expression.

The basic problem we have addressed is as follows: Given a set of servers and a user's request expressed in server-independent terms, how can server-units be automatically selected and invoked to satisfy the user's request? Solving this problem involves solving three subproblems:

a. The knowledge representation problem--Given a collection of servers, represent the data and functions supported by the servers. Essentially, the problem is to represent the semantics and relationships among entities in the capability space, and to define server-units in terms of that space.

b. The formulation problem--Given a request expressed in terms of the capability space, transform the request into an equivalent one expressed in terms of server-units.

c. The planning/execution problem--Given a request re-expressed in terms of server-units, determine a sequence in which to invoke those server-units that will obtain the information that satisfies the request.

The focus of this paper is a solution to the formulation problem. We discuss the knowledge representation approach to the extent necessary in this context. The planning/execution problem is one of finding a sequence for invocations that is sufficient to yield the proper result, then optimizing the sequence for execution efficiency. The optimization step is beyond the scope of this paper.

III RELATED WORK

Among major approaches to automatic programming (e.g., construction by theorem provers [Nilsson, 1980] knowledge-based program construction [Barstow, 1979], etc.) our work takes the "transformation" approach (e.g., [Burstall and Darlington, 1977]), in which an expression of a problem is successively transformed into a more specific form. Kim (1985) used a transformation approach to generate examples given a constraint formula expressed as a conjunction of predicates. In both Kim's work and ours, the goal is to return sets of variable bindings which satisfy the input expression. Kim's approach reduces the constraint formula to simpler terms for which known examples are stored, or from which variables' values can be found by algebraic solution. The stored examples for individual terms are tested to find those that satisfy the entire constraint expression; those that survive the test are combined to generate the desired result. This approach is not well suited to accessing database and application programs, however, since it is not feasible to generate results by successively testing each database record or potential application output for consistency with the input expression.

Gray and Moffat (1983) developed a method for transforming requests for information expressed as relational algebra queries into programs to access Codasyl databases. In their approach, multiple access paths are stored for each database relation, giving the alternative sequences in which the data items corresponding to the columns of the relation can be found. Combinations of access paths for the relations involved in a query are tested to find a combined path equivalent to the relational joins in the query. Our work is similar, in that we dynamically generate the necessary joins by finding the "overlap" of items involved in relations. However, Gray and Moffat assume that the user's request has specified the particular relations to be used; they then generate an efficient way to access them. In contrast, our work focuses on how to identify the particular database relations (and application programs) to be used, given a request expressed in server-independent terms.

In general, distributed database management systems (Ceri and Pelagatti, 1984) handle server-independent queries by replacing each object with its equivalent server-specific object, using a list of mappings from server-independent to server-dependent terms. This step, sometimes called "query modification" (Stonebraker, 1975) is used to solve the formulation problem for distributed databases. A problem with the query modification approach, however, is its requirement that a potentially large number of mappings be explicitly stored. The graph-oriented searching and unification technique we present avoids this problem, and provides a way to achieve the logical integration of application programs and databases. The technique may be useful in other situations that require the selection of primitive functions to solve problems stated at a higher level.

IV GRAPH REPRESENTATION OF SERVER-UNITS AND REQUESTS

A key to our approach is that the semantics of both user's requests and the information available from the server-units comprising the capability space can be expressed using graphs. We represent the semantic relationships among the information provided by server-units in structures, comparable to the conceptual graphs described by Sowa (1984). Several types of nodes exist in our graphs. "Concept nodes" are one-place predicates, C(x), denoting that entity x is a member of the class of entities C. "Role nodes" are two-place predicates, R(x,y), denoting that entity x bears relation R to the entity y (x and y are implicitly existentially quantified). The C and R predicates meaningful in the domain are derived from a hierarchically structured, slot-and-frame-based domain model (c.f. Brachman, 1983) which provides the definitions of the corresponding concepts and roles. "Selection nodes" are two-place predicates, S(x,c), serving to restrict the entities denoted by x to those for which the relation S between x and c holds. For example, the selection node EQUAL(name,"John") restricts name to be equal to "John". Finally, a "function node" is a multiple-place predicate specifying that the named functional relation holds among its arguments, e.g., SUM(x,y,sum). Currently, we restrict function nodes to simple arithmetic functions.

A connected graph is formed by a collection of nodes such that each node shares at least one argument with another node. Each predicate is a node in the graph. Each arc connects a argument that is common to two nodes. A connected graph represents an expression that is interpreted as the conjunction of the predicates that are the nodes of the graph. Thus, the graph:

```
person(x), birthday-of(x,d), date(d),
name-of(x,n), string(n), equal(n,"John")
```

denotes the set of x,d,n combinations such that x is a person, d is a date that is that person's birthday, n is a string that is that person's name, and n is equal to John.

A server-unit is represented as a single predicate with an (arbitrary) predicate name, and a list of formal arguments. The semantics of the server-unit are represented by asserting that the server-unit predicate is equivalent to the appropriate graph. For example, a relation in a relational database between a person's name and birthday is represented as follows:

```
birthday-relation(n,d) <=>
    person(x), birthday-of(x,d), date(d),
    name-of(x,n), string(n)
```

In general, a server-unit predicate for a database relation has as many arguments as there are columns in the relation. A program that computes w, the day of the week on which the date d falls, is represented as follows:

```
day-of-week(d,w) <=>
    date(d), weekday-of(d,w), string(w)
```

A server-unit predicate for an entry point of an application program has as many arguments as there are input and output arguments in that entry point.

Attached to the server-unit predicate are properties giving the owners of the server-unit, its mandatory inputs, and its available outputs. The "owners" property is the list of servers which provide the server-unit (several servers may provide the same information). The "mandatory inputs" property identifies which of the predicate's formal arguments must be bound or restricted before the server-unit may be meaningfully invoked. For database relations, this is nil, since no selection conditions need be specified, (e.g., when retrieving all tuples in the relation.) For a software function, the mandatory inputs property identifies the input arguments that must be supplied to the function. The "available outputs" property identifies which of the predicate's formal arguments are available as output from the invocation. For database relations, this is all columns of the relation. For a software function, the available outputs are the output arguments of the function. For example, the mandatory inputs property of the day-of-week program is the list (d), and the available outputs property is the list (w).

A user's request for information is encoded as an expression with a "head" and a "body." The head is a predicate with an arbitrary name, and an argument list specifying the arguments to be returned. The body of the request is a set of nodes forming a connected graph. For example, a user's request for the day of the week on which "John Doe" was born may be represented as follows:

```
answer(w) :-
    person(x), birthday-of(x,d), date(d),
    name-of(x,n),string(n),equal(n,"John")
    weekday-of(d,w), string(w)
```

If the argument list for the head predicate of the request is empty, the request is considered to be a "true/false" question. Otherwise, it is considered to be a request for all possible sets of variable bindings that can be found that satisfy the semantics of the body of the request. The variables that are the first arguments of selection nodes, if any, are called the "known" variables of the request.

Note that although this representation is similar to PROLOG, the body of the request is not processed by a software interpreter. Rather, the formulation problem is to find an expression in terms of server-units that is equivalent to the body of the request. From this expression, a sequence of server-unit invocations is readily generated. The representation allows the formulation problem to be solved by searching the set of server-unit graphs to find a set of graphs that collectively matches the body of the user's request, exclusive of the selection nodes. The selection nodes are then used to supply the input arguments to the server-units selected through this matching process.

V GRAPH UNIFICATION

Transforming a request expressed in terms of the capability space into an equivalent expression in terms of server-units is accomplished by unifying server-unit graphs and the body of the request until the request body is completely "covered." A node in the request body is considered covered when it is unified with a node from the graph for at least one server-unit. The request body is considered covered when all its non-selection nodes are covered. Once the body is covered, its non-selection nodes are replaced by the set of server-unit predicates that are equivalent to the server-unit graphs that have been unified with the request. The result is an expression, consisting of server-unit predicates and selection nodes, that is equivalent to the user's request.

When matching server-unit graphs to the body of a request, we do not require the entire server-unit graph to "unify" with the request body. Rather, we merge the server-unit graph with the request body only if at least one role or function node unifies with the request body. (This requirement prevents the inclusion of server-units that contribute nothing to coverage of relationships among variables in the request.) The additional nodes of the server-unit graph, not present in the request body, become available for unification with further server-unit graphs, as the algorithm proceeds in its search to cover the request graph. This allows the algorithm to find connections among server-units that are necessary to solve the user's problem, but which are not explicit in the original request. ("Hidden database joins" [Sowa, 1984] are one class of such necessary connections.)

We define the unification of a server-unit graph to the body of a request by giving the conditions under which a node from a server-unit graph may be unified to a node in the request body. Nodes may only be unified to nodes of the

same type (concept to concept, role to role, etc.) that have the same predicate name. (An extension of this approach, allowing unification of nodes to superordinate nodes in a type hierarchy, is beyond the scope of this paper).

Given a concept node C1(x1) from a server-unit graph and a concept node C2(y1) in the request graph, C1 can be unified to C2 if C1 and C2 are the same concept name. Unification takes place when x1 replaces all occurrences of y1 in the request graph. We require that x1 has not already been unified to any other concept in the server-unit graph in order to prevent "equivalencing" two distinct arguments in the request graph.

Given a role node R1(x1,x2) from a server-unit graph and a role node R2(y1,y2) in the request graph, R1 can be unified to R2 if R1 and R2 are the same role name, and for i=1,2, the concept node for xi in the server-unit graph may be unified to the concept node for yi in the request graph.

Given a functional node F1(x1,x2,...,xn) from a server-unit graph and a functional node F2(y1,y2,...,yn) in the request graph, F1 can be unified to F2 if F1 and F2 are the same functional node and for i=1,2,...n, the concept node for xi in the server-unit graph may be unified to the concept node for yi in the request graph.

Given a selection node S1(x1,c1) from a server-unit graph and a selection node S2(y1,d1) in the request graph, S1 can be unified to S2 if S2(x1,d1) implies S1(x1,c1) and the concept node for y1 in the server-unit graph may be unified to the concept node for x1 in the request graph. For example, LESS-THAN(x1,15) in a server-unit graph may be unified with LESS-THAN(y1,10) in a request graph, because the information requested is more restricted than the information available from the server.

Once all nodes in a server-unit graph that may be unified with a request body are identified, the unification is performed by substituting the arguments from the request body for the arguments in a copy of the server-unit graph. The resulting server-unit graph is merged with the request body by "superimposing" the two, and removing duplicate nodes. Note that there may be multiple ways to unify a server-unit graph and the request body, involving different nodes of the request body. This arises, for example, in requests whose solution involves joining a database relation to itself. In these cases, multiple copies of the server-unit graph are made and merged with the request. For example, a database relating a person's name, employee id, and the employee id of his/her manager would be used twice to cover a request for the name of some person's manager. The fact that the employee id's are used as the link between an employee and his/her manager emerges as a result of the multiple unifications.

VI THE FORMULATION ALGORITHM

The overall algorithm is as follows: The search space of server-units is first pruned by eliminating server-units whose graphs contain no role or function nodes with predicate names in common with those in the request body. These server-units can never be selected for merging with the request body. Conversely, the request may be immediately identified as one that cannot be handled by the server-units if it contains one or more nodes not found in any of them.

Next, a generate-and-test approach is used to perform a best-first search of the space of the power set of server-units to find a set that covers the request. Starting with the empty set, the generate portion takes the set most recently tested and found insufficient to cover the request, and generates its successors. It will have N successors, each created by adding to the set one of the server-units among the N not already in that set. The best set of server-units among all those already generated but not yet tested is selected as the candidate set for testing. The heuristic for "best" is to choose a set with the minimum score, defined as follows:

$$H = c1 * \text{card}(\{su\}) + c2 * \text{card}(\{\text{predicate-names}(r)\} - \{\text{predicate-names}(su)\})$$

where c1 and c2 are positive weighting coefficients, card is the cardinality function, {su} is the set of server-units being scored, and {predicate-names(r)} - {predicate-names(su)} is the set of nodes in the request for which nodes with identical names are not found in the server-units. This heuristic gives a better score to smaller sets of server-units, and sets leaving fewer nodes in the request body that have no potential covering in the set of server-units. If c2 is zero, the heuristic yields a breadth-first search of the power-set of server-units; if c1 is zero, the search is depth-first. Because the cardinality of the power set of server-units is finite, the search will always terminate. The heuristic improves the search by causing small sets with a greater likelihood of covering the request body to be examined first.

The test portion of the algorithm performs the unification of the candidate set with the request body, then tests whether all non-selection nodes of the request body are covered, and the server-units form a single, connected, acyclic dataflow graph that is sufficient to obtain the desired output. The latter condition is tested by forming a graph in which each server-unit predicate is a node. Nodes are connected by directed arcs from the available outputs of a server-unit to those server-units containing the same variables on their mandatory inputs list. Starting from the known variables and the output variables of server-units whose mandatory inputs are nil, a "mark" is propagated through the graph. The mark propagates from the mandatory input variables to the available output variables of a node, if and only if all mandatory input variables of the node

are marked. The mark propagates from node to node along the directed arcs of the graph. After all propagation is complete, the test succeeds if all mandatory input variables for each server-node are marked, and a mark has reached each argument found in the head of the request.

For example, the request to find the day of the week on which John was born, after unification with the set of server-units comprised of birthday-relation and day-of-week, appears as follows:

```
answer(w) :-
    birthday-relation(n,d),
    equal(n,"John"),day-of-week(d,w)
```

This formulation covers all non-selection nodes of the request. The mandatory inputs of birthday-relation are nil, and its available outputs are (n,d). The mandatory-inputs of day-of-week are (d), and its available outputs are (w). A mark starting from the known variable n, and the outputs of the server-units whose mandatory inputs are nil, propagates to all mandatory inputs of each server-unit, and the desired variable, w. Thus, this set of server-units succeeds as a formulation of the original request in terms of the capability space. From this, invocations to the servers can be readily generated and sequenced.

VII EXTENSIONS

As described above, the graph representation allows expression of requests for information that are simple conjunctions of predicates, equivalent to the "select/project/join" operations of relational algebra (Maier, 1983). It must be extended to enable representation of requests involving disjunctions, aggregations, and recursion. The key to handling these is to extend the graph representation of requests to include subgraphs. A subgraph is a request (head and body), whose head predicate appears in the body of another request. Disjunctions are represented by alternative subgraphs defining the same head predicate. Recursive requests are represented by alternative subgraphs, one of which has a body that directly or indirectly contains its own head predicate. An aggregation is represented by a node in the body of a request which contains the head predicate of a subgraph as one of its arguments.

Extension of the basic formulation algorithm to handle subgraphs involves performing the unification of server-units to each subgraph of the overall request body. A given graph is considered covered if and only if its body and all subgraphs referenced in its body are covered.

VIII IMPLEMENTATION

A prototype has been implemented on a Symbolics 3600 using Zetalisp. The implementation handles "select/project/join" types of requests, and requests involving aggregations. (Extensions for disjunctive and recursive requests are being designed.) The prototype accesses three separate databases and three application programs in the general domain of parts inventory and equipment maintenance. The databases and application programs run on two personal computers using commercially available software.

Table 1 shows the performance of the prototype for some typical requests, exclusive of the time taken by the personal computers in retrieving or computing answers. Out of a total of 17 server-units in the capability space for these databases and applications, more than half may be pruned from the search space for a typical request. The best-first search frequently yields the formulation of the request in server-unit terms with a search of a small fraction of this space. This is because the heuristic function readily discriminates between server-units based on the relevance of their server-graphs to nodes in the request graph. On the Symbolics 3600, the formulation of requests is accomplished in the order of 1 to 3 seconds of processing.

TABLE 1

Performance of the prototype
on several requests in a sample domain

Request (English equiv.)	Pruned search space (# of sets)	# of sets of server-units tested	CPU time used to formulate request
Is X a widget?	2**5	1	.345 sec.
What component is X a part of?	2**8	6	2.888 sec.
On what date was X replaced?	2**6	1	1.035 sec.
What was X last serviced?	2**6	1	.725 sec.
Where can more parts of type X be obtained?	2**5	2	1.544 sec.
On what day of the week was part X replaced?	2**7	2	1.928 sec.

IX EVALUATION

Our approach to the formulation problem has two major strengths. First, it is independent of the domain. It can be ported to another domain by creating the appropriate server-unit representations, and the software to create an invocation message to a server given its server-unit predicate. The second strength is the wide variety of requests that can be handled. Not only can a simple request be routed to the appropriate server, but a request can be automatically partitioned into subrequests, each processed by a different server.

Our current implementation has several shortcomings. First, the request language as implemented cannot yet deal with disjunctions and recursive requests (although it is possible to invoke server-units that are internally recursive). Second, rather than invoke a more efficient database query, the algorithm may choose to run an application program to respond to a true/false question, treating failure as negation. Finally, the input language is not useful as an end-user language; a more user-friendly interface is needed (Larson, et al., 1985).

X CONCLUSION

We have developed a graph-oriented technique for representing server capabilities. Used in conjunction with unification and heuristic search algorithms, it provides an approach to automatically selecting and invoking the appropriate servers to solve a user's problem. Our implementation shows that the approach can formulate a set of server invocations equivalent to a user's request, stated in server-independent, declarative terms, in a tractable amount of time. Included as a component of a larger user interface, this approach can yield the benefits of a uniform interface to multiple heterogeneous, pre-existing servers, hiding the existence of those servers by automatically generating sequences of server invocations to solve the user's problem.

REFERENCES

[1] Barstow, D. Knowledge-based Program Construction. North Holland, New York: Elsevier, 1979.

[2] Brachman, R. J., R. E. Fikes, and H. J. Levesque. "Krypton: A Functional Approach to Knowledge Representation." Computer, 16:10 (1983) 67-73.

[3] Ceri, S. and G. Pelagatti. Distributed Databases Principles and Systems. New York: McGraw-Hill, 1984.

[4] Burstall, R. M. and J. Darlington. "A Transformation System for Developing Recursive Programs." Journal of the Association for Computing Machinery, 24:1 (1977), 44-67.

[5] Gray, P. M. D. and D. Moffat. "Manipulating Descriptions of Programs for Database Access." Proc. IJCAI-83, Karlsruhe, W. Germany, August, 1983, pp. 21-24.

[6] Kim, M. W. EGS: "A Transformational Approach to Automatic Example Generation." Proc. IJCAI-85, Los Angeles, California, August, 1985, pp. 155-161.

[7] Larson, J. A., W. F. Kaemmerer, K. L. Ryan, J. Slagle, and W. T. Wood. "ATOZ: A Prototype Intelligent Interface to Multiple Information Systems." Proceedings of the IFIP Working Conference, the Future of Command Languages. Rome, Italy, September, 1985.

[8] Maier, D. The Theory of Relational Databases. Rockville, Maryland: Computer Science Press, 1983.

[9] Nilsson, N. J. Principles of Artificial Intelligence. Palo Alto, California: Tioga Publishing, 1980.

[10] Ryan, K. R. and J. A. Larson. "The use of E-R Data Models in Capability Schemas." Technical Report, Honeywell Computer Sciences Center, Golden Valley, Minnesota, March, 1986.

[11] Sowa, J. F. Conceptual Structures: Information Processing in Mind and Machine. Menlo Park, California: Addison-Wesley, 1984.

[12] Stonebraker, M. "Implementation of Integrity Constraints and Views by Query Modification," ACM/SIGMOD International Symposium on Management of Data, San Jose, California, May, 1975, pp. 65-78.