A KNOWLEDGE-BASED APPROACH TO DATA MANAGEMENT FOR INTELLIGENT USER INTERFACES

Carol A. Broverman, W. Bruce Croft

Department of Computer and Information Science
University of Massachusetts Amherst, Massachusetts. 01003

ABSTRACT

An intelligent user interface (POISE) is described that provides facilities for defining and supporting higher-level user tasks. Although an object-based data model forms an important part of the POISE system, other types of knowledge such as task descriptions and tool descriptions are required. The management of instantiations of the task and object descriptions is a complex process because POISE both predicts user actions and allows multiple, competing interpretations of user actions. In this paper, we describe how the knowledge base (including the object data model) is defined and used by the intelligent interface. We also describe an implementation of the knowledge base in a frame-based representation language.

I INTRODUCTION

The data models used in database systems provide languages for describing objects, relationships between objects, constraints, and actions that are to be performed on the objects [TSIC82]. A database management system manages the schemas defined with the data model and the instantiations of the schemas that result from dynamic processing (i.e. the database). The object-oriented information that is captured with a data model forms an important part of the knowledge required for an intelligent user interface, but additional mechanisms are required to describe and manage other types of knowledge that are essential for this application. In this paper, we shall show how data models and other types of knowledge can be combined and managed to support an intelligent user interface.

Certain types of information systems can be characterized as consisting of a set of tools that support user tasks in a particular environment. Office information systems are a good example of this type of system and they have been used as the major testbed for the intelligent interface described in this paper. The tools in current office systems are designed to carry out simple tasks that are common to most offices. For example, tasks such as communication, time management and document production are supported by the electronic mail, calendar and text editor tools. A more effective system would support higher-level tasks that are directly related to the goals or functions of the office. This type of task often involves decision-making, complex sequences of actions, and interaction with a number of other people. The intelligent interface described in this paper consists of formalisms used to describe tasks and mechanisms for managing instantiations of these tasks.

The main types of knowledge required for the intelligent user interface consist of task descriptions, object descriptions and tool descriptions. These descriptions are intimately related because tasks manipulate objects through the use of tools. For example, the task of processing orders in a particular company could be described in terms of actions of form and mail tools on various form objects, together with actions and decision points that have no corresponding tools. On the basis of this task description, the intelligent interface would, in effect, provide a "virtual" tool that could support the order processing task. By separating task descriptions from tool descriptions, the addition of new tools will affect only the way in which a task is supported, rather than the description of a task. This division between task and tool descriptions is analogous to the separation of logical and physical levels of description in data models.

Task descriptions in the intelligent user interface play a similar role to application programs in typical database systems in that they refer to objects defined in the knowledge base "schema". However, in contrast to the very structured nature of the algorithms specified in application programs, task descriptions often have steps that rely on the problem-solving abilities of the person(s) using the system [FIKE80,BARB83]. Task descriptions are constantly subject to change, both at an organizational level and by individuals. Task descriptions also represent only a typical way of carrying out a task and many exceptions are possible. The type of support provided by the intelligent interface depends on the amount of structure in the task involved. Generally, the interface can automate the more structured parts of a task and provide assistance to users for the less structured parts [CROF84].

In the next section, we give an overview of the POISE intelligent interface and its capabilities for task support. The third section contains a more detailed discussion of the types of knowledge used in POISE and the relationships between them. In particular, we discuss the role of a data model in the overall knowledge base and how constraints can be specified in both the object and task descriptions. During the operation of POISE a variety of task and object instantiations are created to record the current and predicted states of user activities. The management of these instantiations is the subject of the fourth section. Finally, we show how the knowledge base, including the data model, can be implemented using a frame-based representation language.

II OVERVIEW OF POISE

The POISE system provides task support on the basis of hierarchies of task descriptions. The task descriptions specify the typical steps involved in the task, the objects that are affected by the task, and the goals of the task steps. The ability to combine recognition of user actions and planning using the descriptions and goals gives POISE great flexibility in the type of task support it can provide.

A simplified diagram of the main POISE components is shown in Figure 1. The *knowledge base*, which is the main subject of this paper, consists of two main parts. The first part is the relatively static description of the tasks, objects and tools in a particular environment. This part of the knowledge base also contains the dictionaries and other information used by the natural language analysis and generation components of the system.

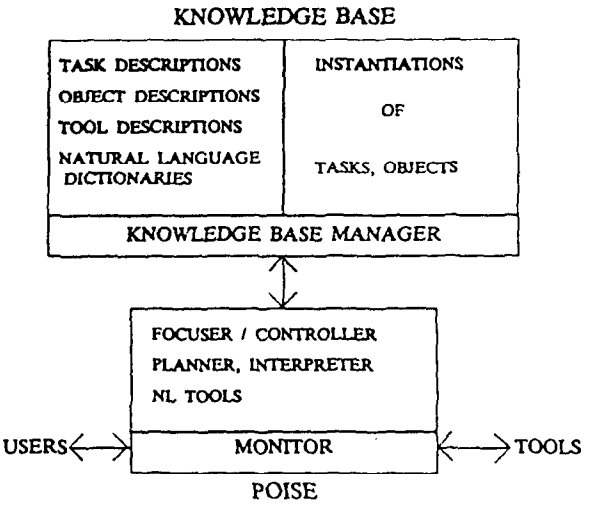

Figure 1: The POISE Intelligent Interface.

The second part of the knowledge base contains the instantiations which describe the dynamic state of the system. For example, in the static part of the knowledge base there might be a task description for filling out a purchase order form. There would also be a description of this form as an object type and its relationship to other form objects used by the system. After a user had started to fill in a particular purchase order form, the dynamic part of the knowledge base would contain a partial instantiation of the "Fill_out_purchase_order_form" task with values derived from the actual values filled in by the user. There would also be an instantiation of the database object that represents the actual purchase order form.

The instantiations in the knowledge base are generated and used by other POISE components such as the interpreter and planner. The knowledge base manager controls access to the knowledge base and provides the operations needed to manipulate the knowledge.

The *monitor* provides the interface between the user, the tools and POISE. It is designed to allow the user to interact directly with "off-the-shelf" tools or to interact with tools through an interface specified within the monitor. This design avoids simulating within POISE the sophisticated interfaces of some tools, but enables the system to understand user actions. The tool descriptions, which are used by the monitor, define how tasks are implemented with the tools.

The other major components of POISE are the *focuser, interpreter* and *planner*. The interpreter is responsible for interpreting user actions in the context of the task descriptions. Since there may be multiple, concurrent and competing interpretations of actions, the focuser is used to choose the most likely interpretations and to control the system's actions [CARV84]. The focuser must also provide a mechanism for backtracking should a user action or user error result in incorrect interpretations.

The planner is similar to the focuser in that it manages interpretations of user activities. However, in contrast to the focuser's emphasis on the recognition of user actions, the planner takes stated task goals and directs the user through sequences of actions designed to achieve those goals. The focuser, interpreter and planner must work in close cooperation for the system to be able to make predictions when attempting to recognize user actions, or to interpret user actions during the planner-directed execution of a task.

Both the interpreter and planner are aided by the constraint propagation which occurs during the execution of a task. Specific user actions apply constraints to the general task descriptions. Constraints also hold between steps specified in task descriptions and this information is used to propagate constraining parameters throughout the executing task.

As stated previously, the main emphasis of the discussions in this paper is on the knowledge base and how it is used by POISE components such as the focuser, interpreter and planner.

II THE KNOWLEDGE BASE

Tasks, objects and tools are represented in the knowledge base using different formalisms. These formalisms capture different, though related, information that is used by the intelligent interface. The following sections contain a discussion of each formalism and the type of information represented.

A. Task Descriptions

In order to represent the possible sequences of concurrent actions in a task, we are using a modified version of an Event Description Language [BATE84]. An example task description is presented in Figure 2. The algorithmic syntax of the procedure is specified by the IS clause, modified by the COND clause, and has its parameters defined by the WITH clause. The conditions required by a task in order to begin are specified by the PRECONDITION clause while the goals satisfied by a task are contained in the SATISFACTION clause.

The IS clause of the task definition provides a precise way of describing the standard algorithm for accomplishing a task in terms of other tasks and primitive operations (tool invocations). The sequence of constituent tasks is specified using regular expression operator, for example, Catenation (') and Alternation (|).

The example shown in Figure 2 is a "Purchase_items" task. This task is a typical semi-structured clerical task. The IS clause of this task specifies that after a purchase request has been received, either a purchase requisition or a purchase order is processed. The task is completed by the steps involved in the Complete_purchase procedure. To get the details of the steps involved in the Complete_purchase task, we would have to examine the corresponding descriptions. The more detailed

```
PROC  Purchase_items

DESC  Procedure for puchasing items with non-state funds.

IS    Receive_purchase-request
      ' (Process_purchase_order | Process_purchase_requisition)
      ' Complete_purchase

WITH  *Items
      *Purchase_request = Receive_purchase_request.*Purchase-request
      *Purchase_order = Process_purchase_order.*Purchase_order-form
      *Invoice_mail_form = Complete_purchase.*Invoice_mail_form
      *Purchase_requisition =
      Process_purchase_requisition.*Purchase_requisition_form

COND  IF *Purchase_request/total-field <= 250
         THEN Process_purchase_order WILL-EXIST

      IF *Purchase_request/total-field > 250
         THEN Process_purchase_requisition WILL-EXIST

      FOR-PROPERTIES ("total-field" "itemized-order" "vendor-field")
                     (*Purchase_order MATCHES *Purchase_request  AND
                      *Purchase_order MATCHES *Invoice_mail_form) OR
                     (*Purchase_requisition MATCHES *Purchase_request  AND
                      *Purchase_requisition MATCHES *Invoice_mail_form)

PRECONDITIONS --

SATISFACTION  *Items/status = delivered
```

Figure 2: An Example Procedure Specification.

(or less abstract) tasks contain links to the tools available in the system. The lowest level procedures in this hierarchy correspond (approximately) to tool invocations.

The attributes of a task are defined by the WITH clause. Task attributes may be constant values, associated objects, or attributes of constituent tasks. These attributes described by the WITH clause of a task may then be used by higher level subsuming tasks. In the example, the *Purchase_request object attribute is obtained from a parameter of the Receive_purchase_request subtask. Note that a reference to an object is marked by a preceding asterisk.

Constraints may be placed upon the values of task attributes and the relationships between task attributes. The COND clause is used to describe these constraints. In addition to rules specifying restrictions on constant values of attributes, the COND clause may also include rules establishing relationships between associated object and/or task attributes. For example, in Figure 2 we see that among other constraints, the *Purchase_requisition_form object used in the Process_purchase_requisition subtask must match the *Invoice_mail_form object used in the Complete_purchase subtask in terms of their total, itemized-order, and vendor fields. COND clause rules may also be used to distinguish between two tasks where only one of the two can occur (Alternation). In the example, the value of the "total-field" of the *Purchase_request attribute determines which of Process_purchase_order or Process_purchase_requisition can occur as a subtask.

Often a constraint (like the one just mentioned aboved) can be represented within an object description itself (e.g. a "*Purchase_order_form" must have its "total-field" less than or equal to $250.00) instead of in the relevant task's COND clause. However, additional power and clarity can result from the redundant expression of the constraint in the task (as shown in Figure 2). Thus, when object-related constraints may be used to guide task choice, it is appropriate to represent them in both the task and object descriptions.

The COND and WITH clauses, along with the temporal ordering constraints found in the IS clause, describe the flow of objects between the subtasks of the higher-level task being described. The WITH clause depicts the vertical flow of objects through the task abstraction hierarchy.

The POISE formalism also contains a description of the state of the knowledge base that must exist in order for the task to begin. The PRECONDITION clause specifies this set of conditions.

Upon completion of a task, certain conditions must be satisfied. This information serves both as an aid to the planner and as an alternate means of recognizing the completion of a task. The SATISFACTION clause specifies these conditions on database state. The example task specifies that the items of concern in the procedure Purchase_items must have been delivered when the task has finished. Goal specification in terms of database state allows the interface to bypass the usual mode of plan interpretation, which is based on a strict algorithmic ordering of plan substeps. Each substep of the higher level plan (in this case Purchase_items) may also be characterized by its goals (which contribute toward the highest-level goal); together, the goals of the top-level plan and those of its substeps constitute an implicit goal hierarchy.

B. Object Descriptions

A data model for the office has been suggested by Gibbs and Tsichritzis [GIBB83,GIBB84]. It is an example of a semantic data model that was designed specifically for the office domain. We have adopted a subset of the features in this data model for the external representation of the objects in the POISE knowledge base. A summary of this subset follows.

An object type is the structural specification of a class of objects. The general form of an object type definition is:

define *object-type object-name*
  begin
    properties: {property definitions}
    constituents: {constituent definitions}
    mappings : {mapping definitions}
    constraints: {constraint specifications}
  end

The *properties* section defines the attributes possessed by that object. Properties of an object type may be hierarchically decomposed. For example, a vendor object may have an address as one of its properties. The address property may be decomposed into street and city, street may be further decomposed into number and street name, etc. Each successive decomposition would be listed in the properties section of the object. Properties that are multi-valued are also easily specified.

The *constituents* section describes an object in terms of constituent objects rather than properties. The *mappings* section defines relationships among the constituents specified in the *constituents* section. Therefore, a *mappings* section will only be present when a *constituents* section is present.

The *constraints* section is used to specify data type constraints on properties, object type constraints on constituents and uniqueness constraints on properties and/or constituents.

Specialization relationships are defined via a special declaration of the form *Object2 ISA Object1*, where Object1 and Object2 are declared object names. This declaration implies inheritance from Object1 to Object2 of all Object1's properties, constituents, mappings and constraints.

In addition to *object types*, *domains* and *triggers* may be specified. *Domains* are abstract datatypes which are used to define non-primitive data types. In the office automation application, for example, *dates* may be a domain and its associated functions would parse date specifications, modify them, display them, compute time differences, etc. *Triggers* are demon-like entities that carry out specified actions when certain conditions are met upon the invocation of specified database operations.

The use of domains has been presented as a declarative method for representing constraints that are imposed on single properties. Triggers are procedural rather than declarative, and they represent object-related constraints dealing with relationships between different parts of a single object. Triggers are used for constraints that hold between one or more slots of a single object, or for other such constraints which are more complex than simple data type specifications. They can also be used to implement simple tasks, such as sending mail out on a specified schedule. The POISE system treats tasks implemented this way as part of the available tool set.

The object data model provides operations to add and remove object instances in the knowledge base, to modify object instances, to retrieve object instances and to define transactions. The use of transactions in a task-oriented environment such as POISE raises some interesting questions. Tasks such as filling out a form could be specified as transactions but, unlike the execution of an application program, a task can be suspended indefinitely at any point by the user. The locks associated with these suspended transactions could cause significant problems. Another problem is that POISE provides assistance by performing constraint checking as soon as possible whereas the normal definition of transactions can involve more than one user action and allows constraint violation during the transaction. These issues are being explored in the current application.

It may be possible to represent task descriptions as objects using the model just presented. Substeps can be listed in the *constituent* section of the specification, WITH clause attributes can be represented in the *property* section, and the COND constraints can be represented either in the *constraint* section or as *triggers*. The formalism would require extensions to allow for the representation of the PRECONDITION and SATISFACTION clauses of the EDL task descriptions.

C. Tool Descriptions

The monitor acts as the POISE interface to the external world by performing mappings between events in the user and task domains and events in the lowest level of POISE task descriptions. It recognizes user actions on behalf of the planning and interpretation components and performs actions

at the request of the planner. The mappings between tool functionality, user actions and task descriptions constitute the tool descriptions that are defined whenever a new tool is added to the system.

The tool mappings can be quite complex; one can usually access a tool function only through a pre-defined interface that was designed for human users. A major part of the tool mappings are the functions that access tool functionality through the tool's existing interface. The use of access functions implies that changes to the tool set need affect only the tool mappings in the monitor, and do not affect the POISE task descriptions.

A set of mappings is also defined between objects that tools know about and objects in the POISE database. Tools manipulate objects and POISE maintains instantiations of selected objects. If an object changed by a tool is stored in the POISE knowledge base, the monitor informs POISE of the change. Note that these mappings would not be necessary if the tool set and the intelligent interface were designed as an integrated package. For example, the forms tool in the current POISE system is implemented directly using forms and operators defined by the object data model.

The lowest level POISE task descriptions, called primitives, have an important role in the tool descriptions. The primitives define what user actions POISE will be interested in, and what tool functionality will need to be accessed by POISE. Consequently, for each action corresponding to a POISE primitive, the capability must exist for both recognizing and performing that action through the tool interface. The POISE primitives thus effectively describe the granularity of the tool descriptions.

III INSTANTIATION MANAGEMENT

The POISE interface is capable of running in two different modes: interpretation and planning. While in interpretation mode (the start-up and default mode in the current application), POISE maintains consistent interpretations or integrated views of user actions as they are being performed. These interpretations are represented by hierarchical structures of instantiations of static task templates. That is, when a user action is recognized by the interface, a structure is created which is based on the appropriate static task template; this structure is then incorporated into a hierarchical interpretation. The instantiation will contain the dynamically determined parameter values particular to that user action, as well as the static constraints found in the template.

The dynamic portion of the knowledge base contains all current interpretations of user actions; some of which may have been designated as more likely than others by the focuser as the interpretation process proceeds. In addition, as each new step is taken by the user, POISE retains a copy of the previous interpretation in order to facilitate backtracking in the event of an interpretation error.

The second function or mode of the POISE interface is to provide the automation of user tasks through the use of the planning subsystem. The user is able to request the system to carry out a task or a series of tasks by specifying a higher-level POISE task (task-oriented planning) or by indicating conditions which must exist in the dynamic knowledge base that would represent the completion of the targeted task (goal-oriented planning). In the first case, the user may invoke the planning subsystem to complete an existing (partially instantiated) task or to carry out a new task. In the latter case, the intelligent interface is responsible for determining the most appropriate way (e.g. sequence of POISE tasks) of achieving the specified goals. For example, the user may explicitly request the interface to complete the task of purchasing a desk once a purchase request has been received (task-oriented planning). The user could also specify the acquisition of a desk as a goal, and the system could determine how to achieve that goal (e.g. through an instantiation of the Purchase_Items task).

An example snapshot view of the dynamic knowledge base (Figure 3) illustrates some characteristics of procedure and object instantiation management. As procedure instantiations are created, object instantiations are also created to represent the associated database objects. These object instantiations are created by the POISE monitor when a tool creates an object in the application domain. The object instantiations created in this way are known as "base objects", and they uniquely correspond to the real objects being manipulated by the user. "Base-object" instantiations are linked to the task descriptions which refer to them as well as to related object instantiations. For example, a base-object may be connected to another base-object which contains it.

In the example, we can see that when the user invokes the mail tool to read his/her mail, the interface creates an instantiation of the Receive_information procedure, and POISE also creates associated object instantiations to represent the Purchase_request_form and the

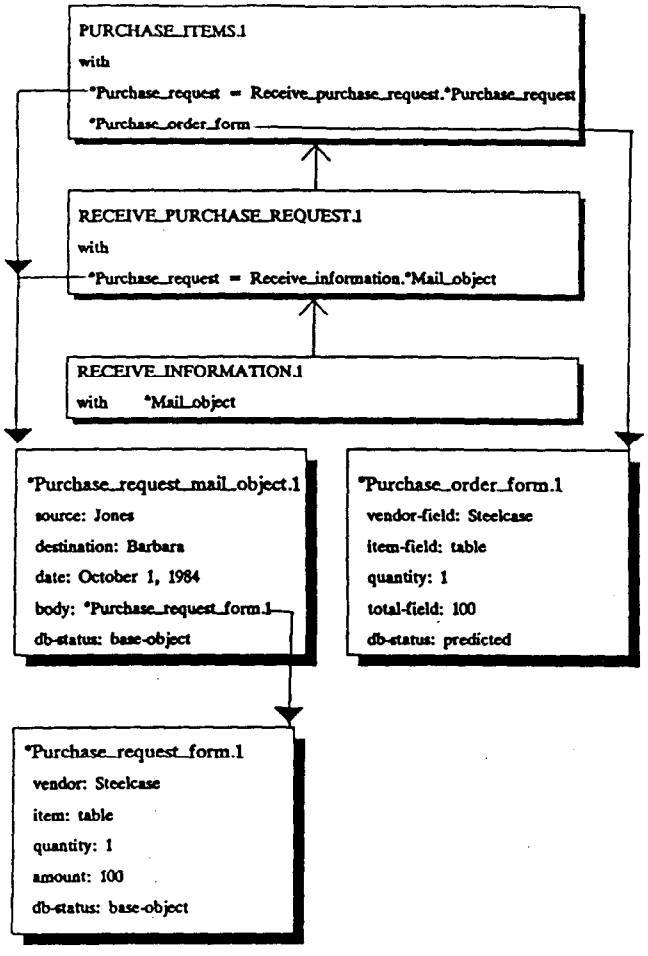

Figure 3: Simplified example of instantiations during task execution.

Purchase_request_mail_object manipulated by the tool. The database status of these objects indicates that they are base objects. These two objects are linked together to show that the Purchase_request_form is part of the Purchase_request_mail_object.

A second type of object instantiation (in addition to "base-objects") are "predicted" object instantiations. Predicted objects are represented in much the same way as base objects, but they are conceptually different. Predicted objects serve as placeholders for constraints associated with an interpretation, and do not correspond to an object that the user (or tool) has actually created or manipulated. The constraints embodied by the predicted objects are derived from the descriptions of tasks making up the interpretation as well as from the object descriptions associated with the interpretation. The use of predicted objects provides additional guidance during the parsing of user actions into consistent and unambiguous interpretations, and facilitates the propagation of constraints among related object instantiations.

Again looking at the example in Figure 3, we see that as interpretation proceeds by abstracting unambiguously up to Receive_purchase_request and Purchase_items, the object associated with the new instantiation of Purchase_items is represented by a predicted instantiation, since this object has not actually been created by a tool. However, the predictions of this object and the constraints it embodies are recorded in the dynamic knowledge base.

In contrast to traditional database management, where the requirement is to maintain object instantiations, the maintenance of a knowledge base for an intelligent user interface involves multi-leveled demands. These demands include the management of the base object instantiations, predicted objects, task instantiations, and the relationships between all of these different types of instantiations. Also, interpretations of instantiations exist as another type of unit to maintain, and the knowledge base manager must handle multiple copies of possibly conflicting interpretations. Many of these management requirements can be handled by representing additional information in the object data model schemas. For example, a base object may have defined relationships to all predicted objects which may constrain it, and triggers may be attached to propagate changes directly affecting a base object to its associated predicted objects for consistency checking.

IV A FRAME-BASED IMPLEMENTATION

Frame-based languages [BARR81] and other artificial intelligence representation tools are appropriate choices for implementing the knowledge base described thus far. These knowledge representation techniques offer a skeletal structure which is much less restrictive than traditional data models, and offer the needed flexibility for representing a broad variety of types of knowledge and their associated constraints. While a frame-based representation language can be used to represent the entire knowledge base, only the data model is currently implemented with such a tool, while the other sections are under development. The data model's view of an object as an aggregate of properties or constituents is easily mapped to a

frame-based view where a frame represents a concept as an aggregate of its slots. The frame-based implementation of the object portion of the knowledge base is described in the remainder of this section.

Object descriptions specified in the external language of the data model are converted by POISE to the chosen internal representation, which is SRL (Schema Representation Language) [WRIG83]. SRL is a frame-based language that has been a testing ground for exploring issues in inheritance. SRL was designed to allow maximum flexibility in definition of the desired representation.

Some of the features that SRL provides are: multiple roles (and thereby multiple inheritance paths), multiple contexts, facilities for the specification of default values, a search path specification language, demons, user-defined relations as well as useful system-defined relations, and various accompanying packages such as a query interface and some data base utility functions. In the development of SRL, special attention has been paid to dealing with the problems of differentiating multiple-path inheritance, allowing users to define their own relations and inheritance semantics, and allowing selective search specifications made by the user.

In our implementation of the knowledge base in SRL, we have used the IS-A link to define simple inheritance paths. Objects are implemented as "schemas" in SRL, with the "slots" of the schemas corresponding to what the data model refers to as object constituents and properties.

Constraints on property values are specified both declaratively and procedurally, following the distinction outlined in the data model. Single-property constraints are specified declaratively using domains. Domain definitions are embodied in a special type of object. More complex constraints relating more than one property of an object are expressed procedurally using a trigger mechanism (called "demons" in SRL).

Simple data-type restrictions are specified by an "range" attachment to the slot of interest, and SRL has a built-in mechanism for executing the procedural check corresponding to the declarative form whenever a value is put into that slot. More complex data-checking, specifically that involving inter-slot dependencies, is performed via a procedural demon mechanism. The triggers in the formal model are implemented using SRL's demon facility. Demons are attached to the slots of concern, and fire when triggered by a specified type of access (determined by the system designer). Demons are used in the current application for both complex value-checking and automatic slot-filling when possible.

SRL offers rather complex facilities for specialized inheritance (other than the traditional IS-A link) and additional type of inheritance links, which for the most part were judged to be too complex for the present application. An additional link which is used is the *instance* link, which is similar to the *is-a* link in terms of inheritance, but allows one to distinguish between the concepts in the hierarchy and the actual world objects corresponding to instantiations of concepts.

V SUMMARY

An intelligent user interface that can both recognize and carry out user tasks requires complex knowledge representation and management techniques. A typical database system can only partially fulfill those requirements. In particular, a database system does not provide facilities for defining semi-structured tasks and managing the instantiations of those tasks that arise from different interpretations and predictions of user actions. The major components of the intelligent user interface knowledge base (task descriptions, object descriptions and tool descriptions) can, however, be regarded as extensions of components of database systems (application programs and schemas).

In the system described in the paper, formalisms for defining tasks and tools are combined with a data model for describing the objects they manipulate. Constraints can be related both to objects and to the way in which objects are used by tasks. Instantiations of the tasks and associated objects are created and used by the focuser, planner and interpreter modules of the POISE system. A more integrated and uniform formalism for the representation of objects, tasks and tools may be appropriate, and is currently under investigation.

Acknowledgements

The POISE system was developed with V. Lesser, N. Carver, A. Hough, and L. Lefkowitz. This research was funded in part by the Digital Equipment Corporation External Research Program and a contract with Rome Air Development Center (RADC).

VI REFERENCES

BARR81 Barr, Avron; Feigenbaum, Edward A.; eds. *The Handbook of Artificial Intelligence*, Vol. 2; 1981.

BARB83 Barber, G. "Supporting organizational problem solving with a work station". *ACM Transactions on Office Information Systems*, 1: 45-67; 1983.

BATE84 Bates, P.C.; Wileden, J.C. "High-level debugging of distributed systems: The behavioral abstraction approach". *Journal of Systems and Software*, 3: 255-264; 1984.

CARV84 Carver, Norman F.; Lesser, Victor R.; McCue, Daniel L. "Focusing in Plan Recognition". *Proceedings of the National Conference on Artificial Intelligence*; Austin, Texas, 1984.

CROF84 Croft, W. Bruce; Lefkowitz, Lawrence S. "Task Support in an Office System". *ACM Transactions of Office Information Systems*, 2: 197-212; 1984.

FIKE80 Fikes, R.E.; Henderson, D.A. "On supporting the use of procedures in office work". *First National Conference on Artificial Intelligence*, Stanford, California, 1980.

GIBB84 Gibbs, S. "An object-oriented office data model." Ph.D. Thesis, University of Toronto, Canada, 1984.

GIBB83 Gibbs, S.; Tsichritzis, D. "A data modeling approach for office information systems", *ACM Transactions on Office Information Systems*, 1: 299-319; 1983.

TSIC82 Tsichritzis, D.; Lochovsky, F. *Data Models*. Prentice-Hall, 1982.

WRIG83 Wright, M.; Fox, M.S. *SRL 1.5 User Manual*, Intelligent Systems Laboratory, Carnegie-Mellon University Robotics Institute, 1983.