The Role of Plan Recognition in Design of an Intelligent User Interface

Carol A. Broverman *Karen E. Huff* *Victor R. Lesser*

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

Abstract

We report on the design and implementation of intelligent interface that assists users performing computer-based professional work by recognizing sequences of actions that are globally consistent and meet desired goals. Our approach is based on hierarchical plans that represent user tasks. Recognition of instantiations of these plans occurs by predicting future actions from past events and then matching new actions to these predictions. The intelligent interface GRAPPLE (Goal Recognition and Planning Environment) extends a previous system primarily through a reformulation of the plans, incorporating more knowledge about the plans and the domain. We present this new formalism, which lays the groundwork for the development of meta-plans and reasoning from first-principles.

1 Introduction

In complex domains of computer-based professional work, there is a need for intelligent interfaces which assist practitioners (as opposed to novices) with sequences of actions which meet desired goals and are consistent in their global context. It is not a question of replacing the practitioner with an expert system, but rather of cooperatively supporting the work of the practitioner with an intelligent assistant. This assistant would bridge the gap between the practitioner's perspective on problem-solving activities and the computer system's perspective on tool invocations and resource usage.

Using predefined, hierarchical activity definitions (plans), the intelligent interface can monitor the conversation between user and computer system, recognizing the commands issued as instantiations of primitive plan definitions. Predictions of expected user commands can be made as a result of the successful integration of a primitive action into a more abstract plan representation. The interface can also operate in an alternate mode and automatically generate sequences of primitive commands in response to a user request for a high-level plan. Such an intelligent interface would thus be a mixed-initiative system, combining facilities for plan recognition and plan automation, with an embedded planner used to extend the predefined plans as needed. The assistance provided to users would include:

- Detecting actual and potential errors, including errors of global strategy;
- Recovering from and correcting errors using context and goal information;
- Creating and managing agendas of work to be performed;
- Summarizing accomplishments of terminal sessions;
- Automatically performing steps in a plan or completing a plan.

Realizing this type of intelligent interface will require AI techniques for several reasons. During recognition, the information needed for a definitive interpretation of a plan may not be complete. Further, the plan definition may be approximate and based, in part, on heuristic knowledge. The interface will have to generate selected alternatives, make choices, and be prepared to retract interpretations at a later time. It will usually be too burdensome to write predefined plans to cover all possible situations, so there will be a need to generate new plans dynamically (either to interpret user actions or to carry out some high-level plan). An embedded planner will ensure robustness of the interface over a wider range of activities. Finally, a deep knowledge of the domain, together with appropriate automated reasoning techniques, will add to the power of all aspects of the interface.

In this paper, we discuss a research effort to investigate the role of plan recognition in the design of intelligent interfaces. A first-generation system, called POISE (Procedure Oriented Interface for Supportive Environments) is briefly presented as an intelligent interface which incorporates plan recognition viewed from an event-based perspective. The limitations of this approach are cited, and a second-generation system is being designed and imple-

CH2364-8/86/0000-0863$1.00 © 1986 IEEE

mented to extend the earlier effort. This new goal-based system, called GRAPPLE (Goal Recognition and Planning Environment), is presented in detail. Fundamental changes and the resulting capabilities are highlighted. The last section of this paper is devoted to a discussion of current research directions, such as the incorporation of meta-plans, and the representation and use of first principles knowledge about the domain.

2 An Event-Based Intelligent Interface

We have implemented an intelligent interface called POISE [1,2,5] which performs simple plan recognition and plan automation in the domain of office automation. POISE is written in Common Lisp and is integrated with DEC FMS office automation tools and VMS mail facilities.

The original POISE system successfully addresses the goal of implementing a mixed-initiative intelligent interface which recognizes and automates plans which are represented in a hierarchical plan library. The plans POISE uses are behavioral (event-based) in nature; an abstract plan is decomposed by specifying a combination of lower-level plans, using grammar rules with temporal operators. These temporal operators include the regular operators of concatenation, alternation, and repetition, supplemented with a few concurrency operators[4]. Thus, an ordering of subplans is specified explicitly within the more complex plan, and this ordering constitutes the skeletal definition of the abstract plan.

A semantic database is accessed by the POISE interface to model all domain objects being manipulated by the plans. Objects are represented using a frame-based language[1], exploiting the inheritance feature for ease of specification, and including facilities to represent semantic constraints within and between objects.

The POISE system provides solutions to the problems of incomplete information and search complexity through its focusing mechanisms. A set of heuristics is used to limit the generation of alternatives when ambiguity arises, and truth-maintenance techniques are applied to retract incorrect interpretations [2]. Constraints from both the semantic database and within the plans are propagated statically and dynamically throughout the active instantiation network. Constraint propagation results in further pruning of predicted user actions, a limited capacity for error detection, and an ability to automate the completion of a partially instantiated plan.

2.1 Limitations in the First Generation System

POISE is limited in the types of reasoning which can be performed about relationships among plans. The behavioral plan definitions specify decomposition solely in terms of temporally ordered subplans and do not represent plan preconditions or goals. This characterization of a plan is insufficient for a planner whose role is to synthesize new plans, since there is no representation of the reasons behind the substeps, and little knowledge is available about the relationships between different plans. For example, a POISE plan may require that subplan B follow subplan A, but it does not describe the semantic database state established by subplan A and required by subplan B. Also, the temporally-ordered substep form of plan decomposition is rigid and lacks the modularity needed to easily integrate new plans into the plan library. Since environments can be arbitrarily complex and dynamic, limitations imposed by a purely event-based representation are significant.

Also, without explicit goals, there is no way to note that an action may be omitted because its goal has already been met. Nor is there enough knowledge to support a robust approach to recognition and recovery from plan failure. With an event-based approach, recovery must be built into the plan rules, making the rules unwieldy and complex and discouraging deeper reasoning about failure and recovery. Since in most semi-structured environments, the "main-line" or standard definition of a plan is actually less common than the variations and exceptions which occur [7], a serious attempt to overcome these limitations imposed by the event-based approach should be made.

Another limitation of POISE is inherent in the way the various types of knowledge are represented. In POISE, the domain knowledge is expressed using a frame-like model of domain objects, while the plan knowledge is represented using a completely different formalism and underlying representation tool. The use of a uniform representation for both domain plans and domain objects would allow the system designer to tie together constraints related to both plans and objects, and provide greater coherency[1]. In addition, general domain knowledge that is not directly related to either a plan or object definition is not represented in POISE. Thus the knowledge needed for a deeper model of the domain is lacking, seriously limiting reasoning that can be done about plan failure or while handling exceptions that arise during plan recognition.

3 A Second Generation System

We are currently designing and implementing a successor to POISE called GRAPPLE. This system is being developed in order to address shortcomings inherent in the ear-

inal POISE system, and to pursue further problems which arise when performing plan recognition in a largely unstructured domain. We are particularly interested in exploring potential sources of deeper domain knowledge than those exploited by POISE, thus motivating a reevaluation of the characterization and interpretation of plans. As a testbed for GRAPPLE, we are using the domain of software development, which is a complex domain and offers rich sources of knowledge, yet is relatively self-contained.

3.1 Fundamental Changes

An overriding theme of the GRAPPLE plan formalism is an expanded representation of knowledge about plans and their interrelationships. The *goal* of a plan and the *effects* of a plan on the domain model are explicitly represented, as are *preconditions*, which must be satisfied before the plan can be executed. A deeper representation of the plan increases the capacity for reasoning during plan recognition and automated planning and affords the system a much richer knowledge base from which to reason about plan failure. The system also has a larger store of semantic knowledge which it can use to "understand" and accomodate exceptional scenarios during plan recognition.

The use of a state-based, goal-oriented perspective in GRAPPLE is in contrast to the POISE event-based substep plan characterization and follows the classical planning formalism. A *goal* is specified as a partial state of the semantic database. A goal can be decomposed into *subgoals*, each of which also is expressed as a semantic database state specification. Achievement of all the subgoals, along with the posting of the *effects* of the plan, should lead to satisfaction of the goal of the plan. *Effects* can be expressed in high-level as well as primitive plans, allowing for the expression of complex semantic changes to the semantic database.

A state-based approach to plan representation provides the system more modularity. For example, if one of the subgoals for a plan is to *have-more-disk-space*, a number of plans may be retrieved that achieve this subgoal; for instance: *delete-a-file*, *purge-directory*, and *increase-quota*. The multiple possible plans need not be specified statically; they can be determined dynamically in order to exploit the rich sources of contextual knowledge at runtime. Representing goals as states in GRAPPLE also allows the interface to avoid a potentially redundant execution of a plan. If a plan has a subgoal which is already satisfied, then no plan need be executed to achieve the subgoal. The overall ordering of the plans that can achieve subgoals of a complex plan is determined dynamically by monitoring the satisfaction of *preconditions*. The state-based approach thus allows for the easy addition and removal of plan definitions from the plan library, without necessitating a recompiling of all the plans and their subgoals. In POISE, the event-based plan specification is "hard-coded," thus rendering the plan library inflexible to dynamic modifications.

GRAPPLE also attempts to overcome limitations imposed by POISE's nonuniform representations. In GRAPPLE, plans are represented with the same knowledge representation tool/language as domain objects. Therefore relationships between certain plans and objects can be easily recorded and constraints relating to both plans and associated domain objects are uniformly specified. The groundwork is thus laid for a more powerful object representation language and more powerful reasoning capabilities.

In order to provide complete coverage of relevant activities, GRAPPLE also models objects and processes which are not directly monitorable. Certain actions, such as decision making by the user, always occur "offline." Other actions, such as communication, may occur "online" through mail facilities *or* "offline" via phone or in person. Even when "offline," these actions cannot be ignored in constructing a total picture of the user's activities; for example, such actions may satisfy the preconditions of later actions. To handle this, any plan can be denoted "offline", in which case GRAPPLE must deduce when its execution has occurred and when its effects should be posted to the semantic database.

3.2 GRAPPLE Plan Formalism

A plan definition in GRAPPLE consists of a set of clauses: *Plan-name, Goal, Plan-Vars, Builtin-Vars, Precondition, Subgoals, Constraints*, and *Effects*. Plans may be either "builtin" or "complex". "Builtin" plans are those plans which map directly to primitive commands that may be issued by the user in the programming environment and will have a *Builtin-vars* clause but will never have a *Subgoals* clause. "Complex" plans, by definition, require multiple subplans to achieve their goals. Each intermediate step corresponds to a subgoal in the *Subgoals* clause. Plan may also be "offline" if they model user decision-making or some other non-monitorable activity. All types of plans have a *Goal* clause, but the other clauses do not have to be present in the plan definition, except where just mentioned.

The *Goal* clause identifies a state[1] of the semantic database that is achieved by the successful completion of the associated plan. The goal is expressed as a predicate calculus proposition whose truth can be determined by querying the semantic database. The *Goal* of a plan is distinct from its more abstract *purpose*, which is determined dynamically by the integration of an instantiation of this plan into a

[1] Obviously, this as well as other semantic database state specifications are *partial* states of the semantic database, since it deals with only a few aspects of the entire state, which is the conjunction of every fact in the semantic database.

hierachical interpretation at runtime.

The *Builtin-vars* clause is present only for *"builtin"* plans, and defines the primitive values that are determined by the filter program[2]. The *Plan-vars* clause defines the names and types of the input and output parameters of the plan. The directional flow of the parameters is determined by the goal statement; those parameters which are bound in the *Goal* are the output parameters.

The *Precondition* clause defines the initial state of the semantic database that must hold in order for the plan to be allowed to begin. It is expressed as a proposition in predicate calculus. The *precondition* may be *"locked,"* in which case, once it is achieved, it cannot be negated until the plan actually begins[3].

The *Subgoal* clause is present for *"complex"* plans, and consists of a set of semantic database states, again expressed as predicate calculus propositions. The *Subgoals* represent the decomposition of the plan *Goal*. Thus, complex plans are not defined in terms of other plans, but indirectly through states of the semantic database which may be achieved by other plans. Individual subgoals in the *Subgoals* clause may also be *"locked,"* which indicates that once achieved, a subgoal must persist until the completion of the entire plan. In general, though, it is not required that all subgoals be true at the completion of a plan. The order in which subgoals are to be achieved is determined dynamically, dictated by the *preconditions* of the plans chosen to accomplish them. A notation is provided for denoting "iterated subgoals", indicating that a subgoal may be achieved repeatedly with different variable bindings while completing the plan.

The *Constraints* clause specifies constraints that must hold within and between variables used by any of the clauses of the plan. They are expressed as predicates on the semantic database.

The *Effects* clause specifies modifications that are to be made to the semantic database upon completion of the plan. New objects can be created, and attributes and entities can be added or deleted. Additions and deletions from the semantic database are specified as predicates and qualified by the type of modification (ADD or DELETE). New entities are specified with the NEW qualifier. All types of plans may have an *Effects* clause, allowing the expression of a complex, high-level change to the semantic database.

The semantic database, used in POISE to model the world of the user, serves the additional role in GRAPPLE as the state description of a classical planning system. It is consulted to determine if a goal, precondition, subgoal, or constraint is true. The effects clause serves as a non-procedural description of a state transition. The semantic database may be thought of as an entity-relationship model, with entities, attributes of entities, and relationships between entities. The usual translation to predicate calculus notation may be made, whereby the entities become constants, the attributes map into functions, and the relationships map into predicates and (optionally) additional functions[3]. These constructs are then used in the sentences of which the various plan clauses are composed. An example GRAPPLE plan definition is given in Figure 1, and a portion of the semantic database, with links and attributes corresponding to available predicates and functions, is shown in Figure 2.

(PLAN-NAME *edit* IS-COMPLEX

PLAN-VARS:	(t : text; ds : abstract-spec)
GOAL	EXISTS (t) | latest-realization(t, ds) AND consistent-with(t,ds)
PRECOND	EXISTS (b: text) | baseline(b,ds)
SUBGOALS	(NAME Accessible) (VARS (f : file) (t : text)) (STMT (EXISTS (f) | stored-in(t,f)))
	(NAME Created) (VARS (nt, ot : text) (ds : abstract-spec)) (ITERATED) (STMT (EXISTS (nt) | latest-realization(nt,ds) AND successor(nt,ot)))
	(NAME Accepted) (VARS (t:text) (ds : abstract-spec)) (COMPLETES Created) (STMT (believed-to-be(t,ds))
CONSTRAINTS	(Accessible.t = Created[first].ot) (Created[this].nt = Created[next].ot) (Created[any].ds = ds) (Accessible.t = b) (ds = Accepted.ds) (Created[final].nt = Accepted.t)
EFFECTS	*None* -(accomplished by subgoals)

Figure 1: A GRAPPLE Plan Definition

3.3 Plan Recognition in GRAPPLE

The plan recognition component of GRAPPLE is currently being designed and implemented. Basic mechanisms have been established for predicting expected actions based on occurrences already seen and for incorporating an occurrence of an action into an interpretation structure. An *expected-actions* list is maintained for each top-level plan to record the monitorable user actions predicted by the interface. A *pending-conditions* list is also associated with each top-level plan to record those *goals, subgoals, and pre-*

[2]The filter program monitors user actions, and traps all command issued by the user. It is responsible for determining the type of *"builtin"* plan that corresponds to the command, and presenting the primitive parameters of that invocation to the intelligent interface in a standardized form.

[3]The start of a plan is defined by either the occurrence of a primitive action which is integrated as a subpart of that plan or the occurrence of the plan itself, if "builtin".

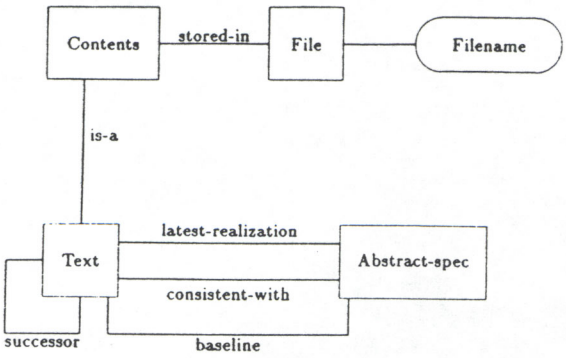

Figure 2: Subset of Semantic Database in GRAPPLE

conditions that are awaiting satisfaction.

At any point in time during the running of the intelligent interface, there are one or more top-level plans which are in progress. They are represented by instantiations of those plans on the *active plan blackboard*. When a plan is instantiated, each of its goals and subgoals is instantiated as well and maintained as *pending conditions* for that plan. A backward-chaining approach is then taken to predict which plans could achieve these *pending conditions*.

Predictions are currently[4] made by matching the subgoal/goal conditions with the goals of other plans in the plan library. Once a prediction is made, an instantiation structure is created for the predicted plan and its precondition is posted to await satisfaction. If the plan is a primitive one, it is posted to the list of *expected-actions* for the top-level plan that subsumes it.

When a user-action occurs, a *matcher* is invoked to determine which of the *expected-actions* is being performed. Values determined by the filter program, which directly monitors user actions, are passed up to the designated expected action structure, and bindings of variables are propagated. *Pending-conditions* are reevaluated and the plan recognizer generates new expectations after integrating the action occurrence.

Choice points have been identified at various stages during plan recognition. For example, one choice point occurs when many plans qualify as achievers of an outstanding goal or subgoal. Another arises when an incoming action matches several predicted actions on the *expected-actions* list and there is not enough information to disambiguate. Heuristics to guide the focus of control and to limit search are currently under investigation.

[4]A more complete and sophisticated prediction mechanism will be incorporated upon the addition of a more sophisticated *planner* module, which will analyze the interactions between *effects* of plans and pending *goal* conditions.

3.4 Current Research Directions

Work is in progress to develop a model for describing and using meta-plans, which are special plans describing the use of the domain plans, and to explore the potential for reasoning using first-principles knowledge about the domain.

3.4.1 Meta-Plans

We are currently working on a meta-plan approach to provide more types of relationships between plans, in additional to subgoal decomposition. Recognizing plan failure and integrating the resulting recovery actions are particularly important in domains like software development, where the basic paradigm of work is "trial and error." Also, during informal analysis of programmer terminal sessions, we have noticed other plan interrelationships. Obtaining on-line help provides the user with specific information to be able to formulate and issue some other command. Gathering information via tools to analyze, reorganize, condense, and present data supports the user in making key decisions about how to carry out some plan. At times, programmers will model a plan with dummy input in order to see if it will work as they predict. Occasionally, work is undertaken in experimental mode, where the initial state is explicitly saved in advance, the work then performed, and a decision made as to whether to accept the results or back-up to the initial state and try again.

Meta-plans allow us to capture these general patterns as a context for executing any plan, without having to write out all the details in every plan. In our work with meta-plans to date, we have found that the same basic plan formalism with goal, precondition, subgoal, constraints, and effects clauses can be used. The meta-plan variables are not domain objects, rather, they are domain plans, their goals, effects, etc. While it was not one of our original goals, we found that meta-plans can be written so that the effects manipulate the actual recognition data structures described in section 3.3. Thus, we can implement the intelligent interface at the topmost level as a simple plan execution system, where execution of the meta-plans causes recognition of the domain plans.

3.4.2 Incorporating First Principles Knowledge

As we have worked with plan definitions of either the state-driven or event-driven type, we have recognized that there is additional knowledge about the domain which is not appropriately expressed in the plans themselves. This is particularly true in specialized domains such as software development, where there is a rich set of technical concepts (such as versions, history, configurations, properties and bugs of modules) and a broad range of *first principles* knowledge about programming. This knowledge forms a

self-contained world for reasoning about actions, and will, we believe, be an important addition to the intelligent interface.

This first principles knowledge can be used in the intelligent interface in several different ways, to improve interface performance and extend more assistance to the user. Using the first principles knowledge to generate tentative bindings of plan parameters will result in earlier, more detailed prediction, and will also limit the number of alternatives to consider during recognition or execution of plans. It provides an alternative to simple heuristics such as "prefer the continuation of a plan already in progress to the start of a new plan" for choosing among alternatives, which may be increasingly important as the number of alternatives grows or when plans are inherently underspecified. It can be used to double-check decisions made by the programmer (modeled in the offline plans). Finally, first principles knowledge can provide additional semantic distinctions between apparently equivalent actions (fixing a bug versus adding a new feature) so that future programmer decisions (such as what tests to run) can be anticipated and double-checked.

4 Status

We have defined the GRAPPLE plan and semantic database formalism and are currently completing the plan recognition algorithms, including constraint handling and focusing. Knowledge Craft [6] , a knowledge representation tool package that offers a logic programming environment built on top of a frame-based knowledge representation, is being used to implement the system. A large set of plans for a Unix[5]/C software development environment has been written in the GRAPPLE formalism, and we are starting to formalize the first principles knowledge for this domain. We have also started work on appropriate meta-plans in order to provide integrated interpretations for the entire spectrum of user actions.

References

[1] Broverman, C.A.; Croft, W.B. "A knowledge-based approach to data management for intelligent user interfaces," Proceedings of Conference for Very Large Data Bases 11, Stockholm Sweden, 1985, pp.96-104.

[2] Carver, N.; Lesser, V., McCue, D. "Focusing in Plan Recognition," Proceedings of AAAI, Austin, Texas, pp.42-48, 1984.

[3] Chen, P.P. "The entity-relationship model: toward a unified view of data," *ACM Transactions on Database Systems*, 1:1, pp.9-36, March 1976.

[4] Croft, W.B.; Lefkowitz, L.S. "An office procedure formalism used for an intelligent interface." COINS Technical report 82-4, University of Massachusetts, 1982.

[5] Croft, W.B.; Lefkowitz, L.S., "Task Support in an Office System," *ACM Transactions on Office Information Systems*, vol. 2, pp.197-212, 1984.

[6] Knowledge Craft Manual Guide, Vax/VMS Version 3.0, Carnegie Group Inc., March 1986.

[7] Kunin, J.S. Analysis and Specification of Office Procedures. Ph.D. thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Ma., February 1982.

[5]Unix is a trademark of AT & T Bell Laboratories.