Reasoning About Exceptions During Plan Execution Monitoring[1]

Carol A. Broverman and W. Bruce Croft
Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

Abstract

In a cooperative problem-solving environment, such as an office, a hierarchical planner can be incorporated into an intelligent interface to accomplish tasks. During plan execution monitoring, user actions may be inconsistent with system expectations. In this paper, we present an approach towards reasoning about these *exceptions* in an attempt to accommodate them into an evolving plan. We propose a representation for plans and domain objects that facilitates reasoning about exceptions.

I. Interactive planning and exceptional occurrences

Hierarchical planners incrementally develop a plan at different levels of abstraction, imposing linear orderings at each stage of the expansion to eliminate subgoal interactions [Sacerdoti(1977), Tate(1977), Wilkins(1984)]. The execution of the plan's primitive actions must be monitored to ensure success. Exceptions and interruptions are common occurrences, and the planner must react to new information made available during the various stages of plan construction and execution. Existing plans may require modification or new plans may have to be generated.

We are concerned with using a planner as a support tool in a cooperative problem-solving environment such as an office [Broverman and Croft(1985), Croft and Lefkowitz(1984)]. In such an environment, the planner is not viewed as an omnipotent agent with complete knowledge of the domain and procedures for accomplishing all plan steps. Rather, it aids the user in performing correct and consistent tasks. The operation of the planner depends heavily on interaction with the user in order to allow user control and to draw on the users' domain knowledge. Interactive planners necessarily interleave plan generation and execution since user actions determine the course of future events.

Previous planners have provided general replanning actions which are invoked in response to problems in the plan resulting from the introduction of an arbitrary state predicate or "fact" [Hayes(1975), Sacerdoti(1977), Wilkins(1985)]. In these systems, the replanning techniques provided do not attempt to reason about failing conditions or possible serendipitous effects of the exception. These methods simply make use of the explicitly linked plan rationale to detect problems and determine what violated goals need to be reachieved. We view this type of replanning as a "reactionary" tactic involving little intelligence, and reserve its use for exceptions generated by *external agents*[2].

To address the problems associated with interactive planning, we propose extending the traditional replanning approach. When a user action deviates from the planner's predictions, the system should exploit available knowledge in an attempt to explain the exceptional behavior. Such a constructive approach is preferred to replanning, since replanning, in this case, would attempt to achieve goals that the user deliberately chose not to pursue. This paper discusses *reasoning about exceptional occurrences* as an approach towards incorporating exceptions into a consistent plan. In the next two sections, we describe an interactive planner and the elements of our representation which are used to support the reasoning process. We then outline the types of exceptions that can occur and algorithms for handling them, within the context of an example taken from the domain of real estate.

II. An interactive planner

Input to our interactive planner is provided as an abstract goal specification, and the output is a partially or fully expanded procedural net, with partial temporal ordering (similar to other hierarchical planners [Sacerdoti(1977), Tate(1977), Wilkins(1984)]). A procedural net contains goal nodes, action nodes, and phantom nodes (goal nodes which are trivially true), along with links representing the causal structure of the plan. Since complete expansion of the initial goal may require additional information from the user, only action nodes are considered primitive, and thus executable.

[1]This work is supported by the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, New York 13441-5700, the Air Force Office of Scientific Research, Bolling Air Force Base, District of Columbia 20332, under contract F30602-85-C-0008, and by a contract with Ing. C. Olivetti & C.

[2]The planner attempts to satisfy a number of *agents*. The user(s) are regarded as *internal* agents, while agents are considered to be *external* if the system lacks a model for their behavior (e.g., the *real world*).

We distinguish between those primitive action nodes which the system is able to carry out using available tools (*system-executable*) and those which must be executed by the user (*user-executable*). An action node may be both system-executable and user-executable, in which case automation is preferred. An example of an action which may be solely user-executable could be the cancellation of an order; the decision to cancel must be initiated by the user and thus can be modeled as a decision action occurring "offline" [Broverman *et al*(1986)]. Transferring information from a purchase request to an order form, however, is a primitive action which may either be performed by the user or automated.

At any point during the planning and execution of a task, an *expected-action list* contains the set of user-executable primitive actions which are not preceded by unexpanded goal nodes. This is the set of actions which are predicted by the system to occur next. As each system-executable or user-executable action is performed, the procedural net is expanded further, producing an updated expected-actions list. A user action may be inconsistent with system expectations, in which case it is flagged as an *exceptional occurrence*.

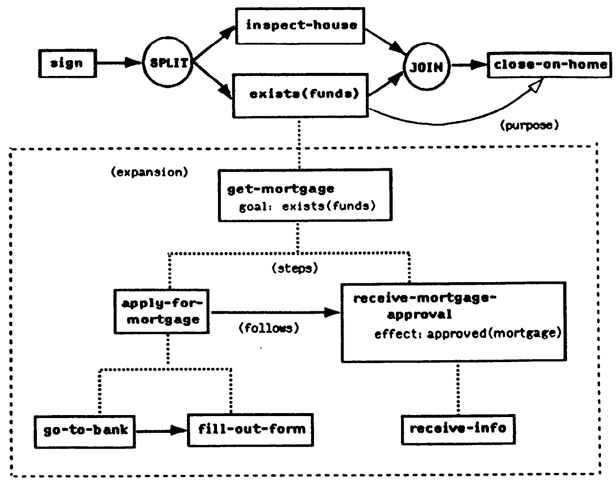

Figure 1: Example procedural net fragment

III. A representation for plans and domain objects

An important part of our approach is a uniform object-based representation of *activities, objects, agents and relationships*[3] [Broverman and Croft(1985)]. An integrated abstraction hierarchy (see Figure 2) combined with a powerful constraint language facilitates the representation and use of more sophisticated knowledge about plans, such as the *policies* of McDermott [McDermott(1978)]. The reasoning process described in the next section exploits this object-based representation. A similar approach has been used by Alterman [Alterman(1986)] and Tenenberg [Tenenberg(1986)] to represent old plans that are adapted to new situations.

The major features of our representation are a *taxonomic knowledge, aggregation, decomposition, resources, plan rationale* and *relationships*. Each of these is defined and illustrated using an example from the domain of house-purchasing, shown in Figures 1 and 2. Figure 1 depicts a partially expanded procedural net fragment which represents the portion of a house-buying task which remains after a house has been selected for purchase. Figure 2 shows a portion of the domain knowledge relevant to this task.

Any complex entity can be viewed as a composition of several other objects as well as an aggregation of properties. An abstract activity object which can be decomposed into more detailed substeps has a *steps* property containing a partial ordering of more detailed activity steps. Decom-

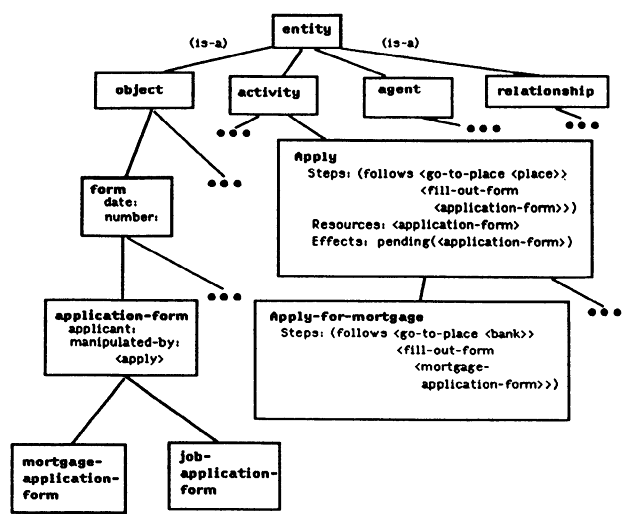

Figure 2: Fragment of knowledge base

[3]In the remainder of the paper, we refer to plan descriptions as *activities* and objects of the domain simply as *objects*.

position of a domain object into other objects is expressed as a set of object types named in a *parts* property. The aggregation of all properties of either an activity or domain object, including decomposition information, constitutes the object definition.

All entities are represented in a type hierarchy, with inheritance along *is-a* links between types and their subtypes. Entities inherit the properties and constraints of their supertypes. For example, a *mortgage-application-form* has various fields which are inherited from the more general *form* object, and obeys the constraint stating that it can be manipulated by an *apply* type of activity (inherited from *application-form*). Activities inherit the preconditions and effects of their supertypes, as well as decomposition information. For example, any apply activity may be decomposed into an activity of type *go-to-place* followed by *fill-out-form*. *Apply-for-mortgage* is a subtype of *apply* and thus inherits and specializes this decomposition. *Apply-for-mortgage* also inherits the effect of *pending(application-form)*.

An activity has an associated set of *effects* which are asserted upon its completion. Effects are represented as predicates on domain objects. The *goal* of the activity is a distinguished main effect and is used for matching during plan expansion. An activity schema also includes a declaration of the types of domain objects it may manipulate. The inverse of this *resources* property is the *manipulated-by* property expressed in domain objects to indicate which types of activities may affect them. The union of an activity schema with the descriptions of associated object types provides a rich semantic representation of the domain, incorporating objects and operators.

Causal knowledge is represented by *goal* properties and *purpose* links. *Goals* are of a global nature, in that they relate an activity to a representation of its intent; that is, they state what this activity accomplishes regardless of the context of the current procedural net. *Purpose* links may be placed between two plan substep nodes in both static and dynamic plan representations, to indicate that a substep of a plan produces a state required for the proper execution of a later substep, much like NONLIN's goal structure [Tate(1977)]. The *purpose* links prove to be particularly important in determining whether or not an exception can easily be incorporated into an existing plan.

Arbitrary *relationships* may also exist between domain objects. For example, a *seller* relation may be depicted between an individual and a certain *house*, expressing the fact that someone is selling a particular house. A special type of relationship which may exist between two objects is a *transformation* relation, which contains a procedural attachment for producing the correct instance of one type of object associated with the instance of the second object type. For example, the abstract class object *address* may be related to *telephone-number* through a special transformation specification which indicates that a phone call using a *phone-number* may produce the corresponding *address*.

IV. Unexpected occurrences

A user action occurs within the context of predictions made by the system. Exceptions can be generated by unanticipated user actions. Because of the inherent open-endedness of the domain, an unexpected occurrence may in fact be a valid semantic action, not recognized as such because of an inaccurate or incomplete activity description.

Referring back to our example depicted in Figures 1-2, we can imagine the following possible scenarios:

(a) Suppose *receive-mortgage-approval* has occurred. We are expecting an *inspect-house* action by the user. Instead, the user executes the first step of the *close-on-home* procedure, *go-to-closing-location*. This is an instance of a *step-out-of-order* exception, since this step is expected, but not until later in the plan.

(b) Suppose the *purchase-and-sale-agreement* has been signed, and the system next expects the user to start carrying out the steps to obtain a mortgage (*go-to-bank*). Instead, a *sell-stock* action is taken by the user, generating an *unexpected-action* exception.

(c) Suppose that while the user is waiting for his mortgage to be approved, his friend from the bank stops in the office and hands him a hard-copy of the approval. Since the normal way of receiving approval is in the form of an electronic message, the user simply offers a *user-assertion* by introducing the predicate *approved(mortgage)*.

(d) Suppose, that while executing the *fill-out-form* substep of the *apply-for-mortgage* step, the user fills in the address field with a *phone-number* instead of an *address*, triggering a constraint violation. This is a case of an *expected action, unexpected parameter* type of exception, where a static object constraint violation has occurred. Unexpected parameters can result in violations of other types of constraints, such as a static constraint in the activity schema, or a constraint dynamically posted on a domain-object by an activity instance.

The above scenarios illustrate the classes of unexpected occurrences which can arise. Actions can be *out-of-order* or completely *unexpected*. A *user-assertion* arbitrarily introduced to the system may have implications for the current plan. A user assertion is modeled as an unexpected action with the assertion as its main effect, and is treated as an unexpected action. An expected action may occur with an *unexpected parameter*, resulting in the violation of a static or dynamically posted object constraint, or the violation of a constraint within the plan itself. In the following sections, we develop algorithms for reasoning about the various types of exceptions, and show how each of the above scenarios can be resolved, resulting in a consistent plan.

V. A general architecture for exception handling

While this paper focuses primarily on the reasoning process used to handle exceptions, a general architecture designed to accommodate exceptional occurrences is shown in Figure 3. Several of the modules are similar to those described in other hierarchical planners, specifically [Wilkins(1985)]. We have extended the basic replanning model to include additional modules (highlighted in Figure 3) to address exception handling. Exceptions are detected by the *execution monitor* and classified by the *exception classifier*. Violations in the plan caused by the introduction of an exception are computed by the *plan critic*. Real-world (not user-generated) exceptions are handled by the *replanner*. The replanning approach we have adopted is similar to that of [Wilkins(1985)], where one or more of a set of general replanning actions is invoked in response to a particular type of problem introduced into a plan by an exceptional occurrence. For interactive planning, we extend the set of general replanning actions to include the insertion of a new goal into the plan.

The *exception analyst* applies available domain knowledge in an attempt to construct an explanation of an exception. Its primary function is to determine the relationships and compatibility of the actual events to the expected actions, goals and parameters. The particular entity relationships investigated by the exception analyst are determined by the type of internal exception. The exception analyst may be triggered by both external and internal exceptions, although it is primarily used for internal exceptions.

The paradigm of *negotiation* [Fikes(1982)] has been used as a model for reaching an agreement among agents on a method for accomplishing a task. We propose to use negotiation for establishing a consensus among agents who are affected by an exception. The *negotiator* determines the set of affected agents and uses the information provided by the exception analyst to present suggested changes to the original plan.

We distinguish between *effecting* and *affected* agents with regard to the occurrence of an exception. The *effecting* agent is that agent who has caused the exception. An *affected* agent is one whose interests are influenced (either positively or negatively) by the exception. Affected agents are those who are "responsible" for the parts of the plan where problems are detected by the plan critic. An external agent can never be an *affected* agent, since the system has no model of an external agent's interests or behavior.

Using information provided by the exception analyst about relationships between actual and expected values, the negotiator initiates an exchange between the effecting agent and the affected agents. The negotiator and plan critic execute in a loop in which the plan critic analyzes changes suggested by the negotiator to detect any problems introduced. This loop is exited when no further problems are detected by the plan critic and all affected agents are satisfied.

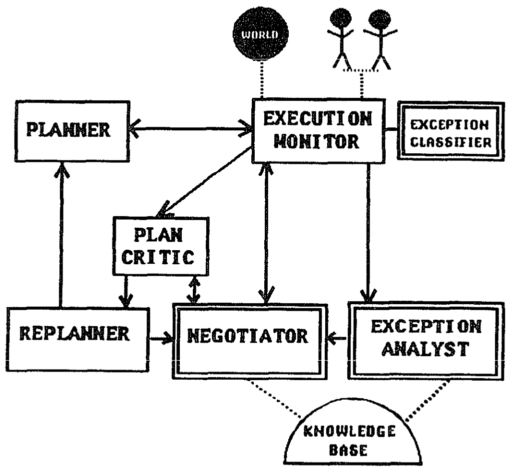

Figure 3: An architecture for a cooperative planner

The negotiator also directs the acquisition of information from the user, if required, again using a trace of the exception analyst's search to guide the questioning. Negotiation may also be invoked upon the failure of replanning. If the negotiator or replanner produces a consistent explanation of the exception, control is returned to the planner to continue plan execution and generation. A successful negotiation can result in a system which has "learned," that is, the static domain plans may be augmented with knowledge about the exception and thus enhances the system's capability to handle future similar exceptions.

VI. Reasoning about exceptions

The behavior of the exception analyst is guided by some general principles derived from the type of the exceptional occurrence. A *step-out-of-order* exception, for example, may imply that the user may be attempting a short-cut, while an *unexpected action* exception may be eventually recognized as an intentional substitution of the unanticipated action for the expected action. The exception analyst performs a controlled exploration throughout the knowledge base which is guided by the current state of the procedural network as well as the type of exception which has occurred. If a number of strategies are possible, the least costly is attempted first. In the following sections, we present algorithms for handling the various types of exceptions, illustrating (where relevant) with the example scenarios developed in section 3.

A. When the action taken doesn't match an expected one

If a user performs an action which doesn't have a match on the *expected-actions list*, the exception classifier is invoked to determine whether this action is entirely *unexpected* or between the object provided as the *actual* parameter value and the object which was *expected* as the parameter value. The exception analyst attempts to establish the following:

1. The two objects may have a *common ancestor* in the object hierarchy. If so, the exception analyst constructs the set of features *unique* to the expected object, since the lack of these features in the object actually provided as the parameter value may be problematic.

2. The two objects may both be *manipulated-by* activities which belong to a common activity superclass. If so, they probably are utilized in similar fashions.

3. There may be any number of other *relationships* between the two objects. Specifically, a *transformation* relationship may link the object provided with the expected object, describing a method to the obtain the expected parameter value.

   To handle scenario (d), the exception analyst notes that the *phone-number* object and *address* objects are linked through a *transformation* relationship, specifying that a procedure *call* may be used on the phone number to produce the corresponding address.

4. The discrepancy between the two parameters may result from *differing quantities* of the object type. If so, an excess may or may not be allowable. The semantics associated with the underlying data type are particularly important when handling quantity discrepancies, since commonsense reasoning may be required. For example, if the *go-to-bank* step was supposed to result in withdrawing 50 *dollars*, emerging with 100 may not be problematic, but baking a cake in a 450 *degree* oven when the recipe calls for 350 degrees may have unsatisfactory results.

This information collected by the exception analyst is used during *negotiation* to establish whether the exceptional parameter should be allowed. The scope of the knowledge base which may be affected by the exception is dependent on the type of constraint violation which has occurred. Modifications and consequences which may result from a static object constraint violation, for example, are localized to the static knowledge base, while plan constraint violations and dynamic object constraint violations may have more far-reaching consequences for the remainder of the plan.

VII. Status

Implementation of a prototype which incorporates the ideas presented in this paper is currently underway. One of the major aims of this work is to augment domain plans with knowledge acquired during exception handling. We are currently looking at the issue of propagating change in an object-based representation.

References

[Alterman(1986)] Alterman, R. "An adaptive planner", *Proceedings of AAAI-86*, 65-69, 1986.

[Broverman and Croft(1985)] Broverman, C,; Croft, W.B. "A knowledge-based approach to data management for intelligent user interfaces", *Proceedings of VLDB 11*, Stockholm, 96-104, 1985.

[Broverman *et al*(1986)] Broverman, C.A., Huff, K.E., Lesser, V.R. "The role of plan recognition in design of an intelligent user interface", *Proceedings of IEEE Conference on Man, Machine, and Cybernetics*, 863-868, 1986.

[Croft and Lefkowitz(1984)] Croft, W.B.; Lefkowitz, L.S. "Task support in an office system", *ACM Transactions on Office Information Systems*, 2: 197-212; 1984.

[Fikes(1982)] Fikes, R.E. "A commitment-based framework for describing informal cooperative work", *Cognitive Science*, 6: 331-347; 1982.

[Hayes(1975)] Hayes, P.J. "A representation for robot plans", *Proceedings IJCAI-75*, 181-188, 1975.

[McDermott(1978)] McDermott, D.V. "Planning and Acting", *Cognitive Science*, 2, 1978.

[Sacerdoti(1977)] Sacerdoti, E.D. *A Structure for Plans and Behavior*, Elsevier North-Holland, Inc., New York, NY, 1977.

[Tate(1977)] Tate, A. "Generating project networks", *Proceedings IJCAI-77*, Boston, 888-893, 1977.

[Tenenberg(1986)] Tenenberg, J. "Planning with Abstraction", *Proceedings of AAAI-86*, 76-80, 1986.

[Wilkins(1984)] Wilkins, D.E. "Domain-independent planning: Representation and plan generation", *Artificial Intelligence*, 22: 269-301; 1984.

[Wilkins(1985)] Wilkins, D.E. "Recovering from execution errors in SIPE", SRI International Technical Report 346, 1985.