Details of the CFOR Planner

Jonathan Gratch
Institute for Creative Technologies
University of Southern California

ICT Technical Report
No: ICT TR 01.2002

Details of the CFOR Planner

Jonathan Gratch
University of Southern California Information Sciences Institute
4676 Admiralty Way, Marina del Rey, CA 90405
gratch@isi.edu

1 Overview

The CFOR planning system combines aspects of two separate families of AI planning systems. On the one hand, it builds on ideas for integrating planning and execution, exemplified by Ambros-Ingerson and Steel's IPEM system (1988) and Golden et al.'s X11 planner. On the other hand, it builds on plan adaptation strategies initially proposed by Hayes (1975) and significantly elaborated by Kambhampati and Hendler (1992). The resulting hybrid supports planning, execution and replanning for environments where actions have duration and the world may change in surprising ways. Additionally, CFOR is designed to support planning in multi-agent environments, and its characteristics are influenced by this requirement.

CFOR plans via constraint posting as in other classical planners such as SNLP (McAllester and Rosenblitt, 1991). Constraints are added in response to perceived problems or ambiguities in the current plan. For example, if an action in the plan has an unestablished precondition, this threat may be resolved by identifying an existing action that establishes the effect (*simple-establishment*) or introducing a new action (*step-addition*). Either "fix" introduces new constraints to the current plan network. Simple-establishment asserts a *protection constraint* that protect the effect from the moment it is created until it is used by the precondition, and binding constraints that ensure the effect unifies with the open-precondition. Step-addition posts a constraint to include the new action in addition to the constraints posted by simple-establishment. Plan generation can be viewed as a sequential decision process where the planner repeated analyzes the current plan network and chooses on of a set of possible fixes.

CFOR adopts IPEM's basic approach to integrating execution with this basic plan generation scheme. Besides maintaining a plan network, the planner maintains a declarative representation of the perceived state of the world or *current world description* (CWD). The CWD allows the planner to monitor the execution of tasks and detect any surprising changes in the environment. The planner also incorporates a set of execution "fixes" which may be interleaved with plan generation fixes. The planner may initiate tasks whose preconditions unify with the CWD (and are not preceded by any uninitiated tasks), terminate tasks who's effects appear in the CWD, and fail tasks if some pre-specified criterion is satisfied. As the CWD reflects the perceived state of the world, it may change in ways not predicted by the current plan network. For example, some external process modifying the environment is detected by changes to the CWD not predicted by the current set of executing tasks. These changes may provide opportunities (as when an unsatisfied precondition is unexpectedly observed in the world). They may also threaten constraints in the plan network, forcing the planner to modify the task network to resolve them.

As mentioned, planning proceeds through a sequence on planning decision. Sometimes these decisions are inconsistent, or inappropriate given a change in the environment. To support plan repair, CFOR records the interdependencies between planning decisions in a data structure, and then uses this "dependency graph" to identify the decisions that contribute problems in the current plans. This dependency graph has been variously called a decision graph (Hayes, 1975) and a validation structure (Kambhampati and Heldler, 1992). CFOR uses this structure to propose planning decisions that might be retracted. CFOR can interleave decision retraction with its plan generation and plan execution decisions.

2 Representations

The planner must be able to represent its beliefs about the current state of the world, its goals, its plans, and possibly its beliefs about the plans of other agents (as discussed later). These representations are currently built on top of a general reasoning system called Soar (Newell, 1990) that uses a simple attribute value knowledge representation and rule-based reasoning (although the planning algorithm is not Soar-specific).

2.1 Representation of Plans

Plans are maintained in a data structure called a *plan network*. The plan network consists of a set of tasks and a set of constraints over tasks or their sub-components.

2.1.1 ***Tasks***

Tasks (sometimes referred to as operators, actions, or steps) represent the basic activities that an agent can perform in the world. Tasks are represented using the STRIPS formalism (with slight modification). Tasks have preconditions and effects, an execution state, a set of binding constraints, and some other more specialized fields that will be described below. An example of a task definition is:

```
defTask Move {?group ?start ?end ?route} {
  :actual-name PERFORM_TACTICAL_MOVEMENT
  :reference {{if type{?end} == BATTLE_POSITION}
                  then PS:Move_to_BP}
                {if type{?end} != BATTLE_POSITION }
                  then PS:Move}}
  :pre {{pre-at:    at{?group ?start}}}
  :add {{add-at:    at{?group ?end}}}
  :del {{del-at:    at{?group ?start}}}
  :bindings {{?start != ?end}}
  :commands
        {{:at-start resume{?group ?start}}
         {:when-added ?route = route{?start ?end}}
}
```

Parameters: A task has a set of parameters (variables) that define the task's execution behavior. Parameters are denoted by a text string preceded by a "?" symbol. The parameters are a superset of the variables mentioned in the preconditions and effects of the task. Traditionally, parameters only take on constants as their value. However, I additionally allow parameters to refer to tree-structured objects defined using attribute/value syntax. Some task-definition fields may refer to these sub-components via accessor functions. In the example above, the reference field tests the "type" attribute of the "?end" parameter. These accessor functions cannot be used in precondition and effect definitions.

Preconditions: Preconditions describe what must (necessarily) hold to successfully executed the tasks. The preconditions of a task correspond to a logical conjunction of predicates. When defining preconditions, each predicate is preceded by a unique identifier (unique within the task definition). This identifier is used when defining task decomposition schema (as described below).

Effects: The consequences of executing a task are described by an "add" list and "delete" list of predicates. Informally, these lists describe facts that are made true or false as a consequence of executing the tasks. As tasks have duration, we must reconsider the standard STRIPS semantics that assumes that effects occur instantaneously (i.e., tasks define a discrete transition two quiescent states). Under the STRIPS assumption, add and delete lists correspond to logical conjunctive expression describing the difference between these two states. In the real world, effects do not happen instantaneously, nor do they occur simultaneously. Another complication is that, though real-world actions sometimes fail, we don't want to model this explicitly (using something like a probabilistic representation which adds complexity). This precludes a strictly logical semantics of action effects.

As a consequence of these factors, I provide an alternative (informal) semantics for effects. The add list is a set of predicates that (individually) will (1) be satisfied by the CWD at some point during the execution of the task and (2) persist until some other activity in the world negates this fact. The delete list has the same semantics except that the predicates are implicitly negated. There is no closed world assumption.

A number of factors may prevent all of the effects of a task from being simultaneously realized. Tasks may fail causing only a subset of their effects to occur. Other simultaneously executing tasks may undo an effect during the task's execution. We allow multiple tasks to execute simultaneously but the representation language cannot generally express interactions between tasks (though Pednault illustrates how to represent some interactions in Pednault, 1986). If a task has different effects when other tasks are executing simultaneously, the planner won't properly predict these conditional outcomes (but it can react to them after the fact when the CWD changes in ways not predicted by the planner).

Binding constraints: Tasks may contain a set of codesignation or non-codesignation constraints. If two parameters codesignate, then they must take on the same value. If they non-codesignate, they cannot take on the same value.

Execution State: Each task instantiation in the plan network has an execution state attribute that describes its current execution status. Before a task executes it is in a *pending* state. After it has been initiated, its state transitions to *executing*, and after termination it becomes *executed*.

When tasks fail it is often convenient to support some form of handshaking between the planner and the executor. For example, when the executor signals a failure conditions, one may wish to explicitly send a command to abort the task and place the executor in a coherent state. I support this type of reasoning via a two-step failure protocol. A task failure is initiated via a failure signal, arising from some domain-specific failure detector associated with the task. At this point a task transitions into a *failing* state. At this point, one can send failure-handling commands to the execution system (see below). Subsequently, and again initiated via a domain specific signal) tasks transition into a *failed* state.

When a task transitions into a failed state, the planner assumes that any effects that have not as yet been observed in the CWD will never occur.

Primitive vs. Abstract: CFOR is a hierarchical planner. *Abstract* tasks may be decomposed into partial sequences of more primitive tasks. *Primitive* tasks are tasks that cannot be further decomposed.

Commands: Commands are my own convention and are not used in other planners. They support a variety of sometimes used capabilities including procedural attachment and run-time variables (Ambrose-Ingerson and Steel, 1988), and provide a more modular way to specify the effectors associated with task execution. Each command specification includes (1) a time tag that specifies when the command is to be executed, (2) an optional codesignation specification that codesignates any result returned by the command with some task parameter, and (3) the name of the procedure to be called and the parameters it is to be called with. Several time tags are currently implemented:

When-added: Indicates that the procedure should be called whenever an instantiation of the task is inserted into the plan network

At-start: Indicates that the procedure should be called whenever a task is initiated

At-end: Indicates that the procedure should be called whenever a task is terminated.

At-failure: Indicates that the procedure should be called if the task transitions to a failing state.

After-failure: Indicates that the procedure should be called if the task transitions to a failed state.

If multiple commands have the same time tag, their order of execution is determined randomly, with the following caveat: a command's execution is deferred if some of its parameters are unbound. This allows one to define a cascade of commands where some commands generate bindings for other commands and the planner automatically determines a valid execution ordering. If a command has unbound parameters that no other command generates bindings for, an error results.

The following two fields do not effect the planner's behavior and only serve as input to a plan visualization tool associated with the CFOR Planner:

Actual-name: Some of the military domains we have implemented provide quite long designators to refer to certain tasks. To display tasks more succinctly, one can use a short name when defining the task and specify another "actual-name" that is to be used when communicating with the execution system. If the actual-name field is omitted, the actual-name defaults to the defined name.

Reference: The reference field is used to associate documentation information with tasks. Each reference refers to a documentation filename and is preceded by the file type. When prompted, this information is displayed by the plan visualization tool. One can indicate that different information should be displayed conditional on the value of task parameters. In the above example, the reference field indicates two possible postscript files to display depending on the value of the "type" attribute of the ?end parameter.

2.1.2 Constraints

In addition to tasks, plans contain a number of constraints of various types. As mentioned above, CFOR is a constraint-posting planner. Planning is seen as a process of looking for possible violations of existing constraints (threat detection) and asserting new constraints to resolve possible violations (threat resolution). Constraint posting planners also perform some limited inference on constraints (or constraint propagation) and consistency checking (to look for possible constraint violations).

Ordering constraints: The planner can represent a partial ordering relationship between tasks. Following standard planning convention, an ordering constraint is a binary relation between tasks. Asserting *before(T1,T2)* means that task T1 occurs before task T2. Actually, things are a little more complicated because tasks have duration and are explicitly initiated and terminated. The constraint *before(T1,T2)* is interpreted to mean that task T1 will be initiated before task T2 is initiated (i.e., the start of T1 occurs before the start of T2).

Note that this interpretation doesn't prevent T1 and T2 from executing simultaneously. In other words, *before(T1,T2)* does *not* mean that task T1 ends before task T2 begins. I use a representational "trick" to express the later statement. One can associate a dummy "end-of-task" task with any given task in the plan network. Such dummy tasks have no preconditions or effects and transition directly to an executed state whenever its associated task terminates. Using this representation, the statement "T1 ends before T2 begins" is expressed as *before(end-of(T1),T2)*.

Ordering constraints are transitive: *before(T1,T2)* and *before(T2,T3)* implies *before(T1,T3)*. It is also assumed that plans are acyclic. Therefore, a set of ordering constraints containing a cycle is considered inconsistent.

Supporting hierarchical planning further complicates the picture. Abstract tasks may be decomposed into a partially ordered sequence of more primitive tasks. This raises the following question. Say that T1 and T2 are tasks, T1 is before T2, and we decompose T1 into T1a and T1b where T1a is before T1b. What, if any, ordering relations exist between T1a and T2? What about T1b and T2? In the current implementation I adopt the (overly) restricted assumption used by IPEM. If T1 is before T2, then T2 and any of its children must be initiated before T2 and any of its children.

Binding Constraints: The plan network contains two classes of binding constraints over variables in the plan network. Codesignation constraint asserts an equality relationship between a pair of variables. Non-codesignation constraints assert an inequality relationship between a pair of variables.

Protection Constraints and Causal Links: Interval protection constraints (IPCs) assert that the truth value of some predicate must hold during the interval that occurs between two tasks. For example *IPC(T1, P(x,y), T2)* corresponds to the constraint that the predicate *P(x,y)* must be true in any state occurring between the initiation of task T1 and the initiation of task T2. As in ordering constraints, a predicate may be protected from (to) the end of a task by using dummy "end-of" tasks. So, *IPC(T1, P(x,y), end-*

of(T2)) asserts that *P(x,y)* is to be protected to the end of task T2.

Most classical planning systems use the term *causal link* to refer to protection constraints. However, Kambhampati has noted that causal links are really a special case of IPCs and that planners have need for using IPCs for reasons other than protecting causal links. Following Kambhampati's terminology, a causal link is a commitment to use a particular effect of some task to satisfy a precondition of a subsequent task. This commitment is subsequently protected with an interval protection constraint). In addition to protecting causal links, the CFOR planner uses IPCs to represent failure conditions for task (e.g., task T fail if P(x) becomes false during its execution), and to represent conditions on decomposition schema, as discussed below.

Hierarchy Relations: CFOR is a hierarchical planner. Abstract tasks may be decomposed into a partial sequence of more primitive tasks. Some hierarchical planners, such as IPEM, remove abstracts tasks from the plan network as they are decomposed. CFOR, however, retains abstract tasks in the plan network. The *subtask* relation captures the relationship between an abstract task and its descendents.

2.2 Representation of Goals

Plans are constructed with the aim of achieving a set of goals. The CFOR planner handles two basic classes of goals: goals of achievement and maintenance goals.

Goals of Achievement: Most planning algorithms focus on goals of achievement. The idea here is that a plan should be a sequence of steps that transform the world into one that satisfies some goal expression. In the CFOR planner, as with most planning algorithms, the goal expression is a conjunction of predicates (e.g. on(Block-A, Block-B) and clear (Block-A)). Following standard convention, CFOR represents goals as a *goal* task within the plan network. Ordering constraints are used to ensure that this task is the last task in the network. The *goal* task is a task with no effects and who's preconditions correspond to the desired goal state.

CFOR provides a mechanism for dynamically adding and retracting goals in the plan network. The defGoal construct allows one to specify specific conjuncts to add to the goal expression under specific circumstances. It is our intention to expand the syntax of the defGoal construct to allow the creation of new *goal* tasks that can be ordered at intermediate points in the plan. This generalization would allow different goals to be achieved at qualitatively different time in the plan. An example of the current goal definition syntax is as follows:

```
defGoal handle-new-order {?me ?friend} {
:when {
  (<state> ^my-name <?me>
           ^mental-state BORED
            ^friend <?friend>)}
  :goal {{ at{?me MOVIES}
           at{?friend MOVIES}}}
```

The defGoal expression has a name (handle-new-order), a set of parameters (that correspond to the parameters listed in the goal expression), a "when" expression, and a goal expression. The "when" expression specifies a conjunction of conditions which, when satisfied cause the goal expression to be added to the *goal* task, and when unsatisfied cause the goal expression to be retracted. This expression is formulated using the syntax of the Soar agent architecture, but we will not go into the specific syntax of the when expression at this point. The goal expression is a conjunction of predicates.

2.3 Representation of Current State

The plan network contains a representation of the current (sensed) state of the world called the current world description (CWD). The CWD is a conjunction of predicates that are assumed to currently hold in the environment. The planner provides a construct to specify the meaning of these predicates. The defGoal construct specifies how to compute the truth value of predicates in terms of lower-level symbols that are maintained by the interface between the planning algorithm and the environment. An example is:

```
defIO at {?group ?loc} {
  :vars {
     (<s> ^current-loc <cloc>)
     (<cloc> ^group <?group> ^loc <?loc>)}
  :test {{<?group> ^active-unit *yes*)}
  :when-true {(write (crlf) <?group> | is at | <?loc>)}
  :when-false {(write (crlf) <?group> | is not an active unit|)}
}
```

The example defines a predicate "at" with two variables, group and location. The :vars are :test fields together describe a conjunction of attribute value tests that, when satisfied, denote when the predicate "at(?group, ?loc)" holds. If just the :vars field holds, the predicate is as asserted to be false. The :when-true and :when-false allow certain facts to be asserted in the interface or procedures to be called (such as display text).

3 Modifying the Plan Network

The planner can modify the current plan network by adding or retracting constraints via a set of "planning decisions." These decisions, for the most part, corresponding to the standard operations of a partial-order planner (e.g., step-addition, conflict-resolution, etc.).

3.1 Task Decomposition

A plan is considered incomplete (threatened) if it contains any abstract tasks that have not been specialized into a set of primitive actions. The user may define a set of task-decomposition schema that specify alternative ways of specializing abstract tasks. A task may be specialized in one of two ways, either by providing values for unbound task parameters, or by breaking an abstract task into a partially-ordered sequence of (more) primitive tasks. Either of these possibilities are represented using the defRefinement syntactical construct illustrated in the following two examples:

```
defRefinementl CrossCarefully {
  :task {Move{?group ?start ?end ?route}}
     :conditions {
        {:filter busy-road{?cross-street} :at-start step2}
        {:test  cross-streets-of{?route} == ?cross-street}}
     :expansion {
        {step1: Move{?group ?start ?intersection ?head}}
        {step2:Look-both-ways{?group ?intersection}}
        {step3: Move{?group ?intersection ?end ?tail}}}
     :orderings {{step1 < step2} {step2 < step3}}
     :links {
        {step1:add-at == step3:pre-at}}
     :commands {
        {when-added
           ?intersection = intersect{?route ?cross-street}}
        {:when-added
           ?head =  route-head{?route ?intersection}}
        {:when-added
           ?tail = route-tail{?route ?intersection}}}}


defRefinement FindRoute
  :task {Move{?group ?start ?end ?route}}
  :conditions {
     {:unbound ?route}}
  :bindings {{?route == OFF_ROAD}}
```

The first refinement breaks a move tasks down into a sequence of three subtasks. The second refinement specializes the Move tasks by assigning a value to one of its parameters. The defRefinement construct has the folowing fields:

:task – This indicates the abstract tasks to which the refinement applies. It is a task name and a list of the task parameters (which may be constants or variables). Only abstract tasks that unify with the task field will be considered by this refinement

:conditions – Conditions further limit the applicability of the refinement. The refinement is only considered if all of its conditions are satisfied. Conditions come in several types.

:filter conditions specify predicates that must be true prior to the execution of the abstract task. For example, the CrossCarefully refinement is only considered if the current plan network indicates that the cross-street will be busy when we get to the move task (In this sense, filter conditions are like the :use-when conditions of NONLIN.) When the refinement is executed, these filter conditions are incorporated into the plan network as interval protection constraints. One can also modulate the duration of this protection constraint by using "at-start" and "at-end" modifiers after the filter specification. In the CrossCarefully example, the street must persist in being busy at least until step 2 in the decomposition (Look-both-ways) is initiated.

:test conditions test the value of certain variables or variable sub-components. One can test if the value of variable equals (or does not equal) a constant or some other variable. The refinement is only allowed if the variable is bound and is consistent with the test.

:bound/:unbound conditions test if variables are bound or unbound

:expansion – This field is where one specifies any subtasks that the abstract is broken down into. Each task is preceded with a unique identifier.

:orderings – This field specifies any ordering constraints between subtasks in the expansion. The orderings are expressed using the unique identifiers mentioned in the expansion section

:links – This field is used to specify causal links between subtasks in the expansion field. Recall that when defining tasks, one must assign unique identifiers to precondition and effects. The links field makes used of these identifiers. For example, in the CrossCarefully refinement, the refinement specifies that moving to the intersection (step 1) establishes the precondition of moving away from the intersection (step 3). More specifically, the effect named "add-at" of step 1 should be used to establish the precondition name "pre-at" of step 3.

:bindings – The bindings field allows one to imposing binding constraints on variables mentioned in the refinement. In the FindRoute refinement, a move task is specialized by asserting a binding constraint

:commands – Commands may be associated with refinements in the same way they are associated with tasks.

3.2 Simple Establishment

For a plan to be complete, all preconditions must be established by some effect of some other task in the plan. Simple establishment takes an effect that is already in the plan network and uses it to establish some open precondition. The effect must unify with the precondition. Performing simple establishment introduces several constraints to the plan network. It introduces codesignation constraints between the corresponding parameters in the effect and precondition, and it creates an IPC that protects the truth-value of the effect until the start of the precondition's task.

3.3 Step Addition

Step addition is like simple-establishment, except that instead of using an existing effect, it adds a new task to the plan network to achieve the desired effect. Thus, in addition to the constraints added by simple establishment, it also adds a new task instantiation to the plan network

3.4 Conflict Resolution

Three planning decisions, promotion, demotion, and separation, are used to specialize away possible constraint violations in the plan network. Since the plan network is partial (there are partial orderings between tasks and partial codesignations between variables), there are cases where constraints in the plan network are possibly, but not necessarily violated. Conflict resolution decisions specialize the partial plan to try and insure that these possible violations cannot happen. Specifically, conflict resolution deals with possible violations of IPC constraints. For example, say the plan contains a IPC that protects predicate P(x) between step 1 and step 3. Say further that step 2 asserts -P(y). Then depending on the ordering constraint between the steps and the codesignation constraints between x and y, there may or may not be a violation of the protection constraint.

The promotion and demotion planning decisions try to resolve the conflict by posting ordering constraints to move step 2 before or after the protection interval, respectively. Separation tries to resolve the potential conflict by adding a non-codesignation constraint between x and y. Currently, separation remains unimplemented.

3.5 Initiate Task

The interleaving of planning and execution is supported through a set of execution related plan decisions that may be interleaved with other planning decisions. Tasks have duration and are explicitly initiated and terminated.

Only primitive tasks are explicitly initiated. Whenever a primitive task is selected for initiation, all of its pending ancestors are first initiated (and any "at-start" commands associated with those ancestors are executed). Next, the primitive task's "at-start" commands are executed (if any), its execution state is change to executing, and ordering constraints are added to order the task before any pending ones. Finally, the task is "locked in" to the plan network, meaning that if the planning decision that introduced the task is subsequently retracted (see below), the task remains in the network as one can not undo the past.

There are several constraints on when a task can be initiated. A primitive task is a candidate for initiation if the following conditions hold:

(1) The task is pending.
(2) No pending task precedes it.
(3) All the task's preconditions are established
(4) The task's preconditions unify with the CWD.
(5) There is no pending effect of another task that could clobber a desired effect of this task. An effect is "pending" if it is an effect of an executing task that does not yet unify with the CWD. An effect is "desired" if it serves as an establisher of some precondition. (This prevents unintended negative interactions between concurrently executing tasks.)
(6) There is no uninitiated ancestor of the task who's preconditions are unestablished or unsatisfied by the CWD. Note that technically we should verify conditions 1-5 for all of the ancestors but for efficiency I ignore condition 5.

3.6 Terminate Task

When a task is initiated, its effects are individually unified against the CWD. If an effect successfully unifies, it is noted as having occurred and is tagged with the current time. If all effects have been observed, the task is a candidate for termination. A task cannot be terminated unless all of its subtasks have terminated.

When selected for termination, all of a task's "at-end" commands are executed and its execution state is changed to executed. If the task has an "end-of-task" task associated with it (see the section on ordering constraints), the end-of-task task's execution state is also changed to executed.

3.7 Initiate Fail Task

In the real-world, task execution occasionally fails. The CFOR planners supports two failure modes for tasks; a hand-shaking mode and a non-handshaking mode. In the non-handshaking mode, a task is simply marked as failed, as discussed in the next subsection. In the handshaking mode, the task first transitions to a "failing" state and then transitions to the failed state as described in the next subsection. The advantage of the handshaking approach is that one can model the following behavior: (1) the planner gets some feedback that indicates that the task is failing, (2) the planner decides to fail the tasks and sends a message to the controller to terminate the task and cleanup any intermediate subprocesses, and (3) the planner receives an acknowledgement that it has successfully terminated the task execution.

Currently, task failure is initiated based on a signal coming up from the executor that states if the task is failing or has failed. It is a candidate to initiate failure on if the signal states the task is failing. It is a candidate to fail the task if the signal states the task has failed. Alternatively, one can associate maintenance conditions with tasks (IPCs that span the duration of the task execution) and signal a failure mode if the maintenance condition is violated. Currently the representation syntax doesn't support this alternative.

When this decision is executed, first the planner executes any "at-failure" commands and changes the state of the task to failing.

3.8 Fail Task

A task is a candidate for failure if the execution system generates a "task failed" signal. Upon executing this planning decision, any "after-failure" commands are executed and the task execution state is changed to "failed." When a task fails, some of its effects may not have occurred (they do not unify with the CWD). Any effects that have not occurred by the time the task has failed are assumed to never occur. They are marked as "failed" and any causal links they participate in are automatically retracted.

If an abstract task fails, any pending subtasks (and dependent structure) are automatically retracted (see the section on retracting refinements).

3.9 Handling Unexpected Events

The plan network can be seen as forming predictions about the future state of the world. The CWD captures the current perceived state of the world. The effects of currently executing tasks form predictions about expected changes to the CWD. Occasionally, the world will change in ways that violated these expectations. For example, if tasks fail or another agent manipulates the world without our knowledge, certain facts may be added to the CWD that are not added by currently executing tasks. Similarly, certain facts may become false even though they are not deleted by currently executing tasks.

The planner has mechanisms that automatically detect these deviations from expectation. When this occurs, the planner offers a "handle-unexpected-event" decision for consideration. If this decision is adopted, the planner creates a "unexpected-event" task who's effects capture the discrepancy. This task's add effects correspond to any unpredicted additions to the CWD and it's delete effects correspond to any unpredicted deletions. This task is given an "executed" execution state and is ordered before any pending tasks.

3.10 Retract Refinement

During the curse of planning, the planner may detect inconsistencies between constraint in the plan network that cannot be resolved by conflict resolution, or it may discover that there is no way to make further progress on the plan. In this circumstance, the planner must retract one of its planning decisions. CFOR differs from most planners in that it can retract planning decisions not chronologically. For example, it might add decision 1 then decision 2 then retract decision 1 while leaving decision 2. CFOR uses a dependency structure to understand which decisions lead to which constraints and to understand the dependencies between decisions. This works somewhat like an assumption-based truth maintenance system (ATMS). Decisions can be thought of as assumptions. Whenever there is an inconsistency, the planner identifies the set of decisions contributing to the inconsistency. Each of these decisions is a candidate for retraction.

If a decision is retracted, all of the constraints introduced by the decision are removed from the plan network (except those "locked-in" as discussed above). Additionally, all decisions that depend on the retracted decision are additionally retracted, as well as their respective constraints. For example, if an abstract task is retracted, all of the decisions that specialized that task will also be retracted from the plan network. The maintenance of this dependency structure is described in the next section.

4 Dependency Graph

The dependency graph is used when retracting planning decisions and serves two roles. First, it is used to identify the dependencies between decisions, so when a decision is retracted, the decisions that depend on that decision are retracted as well. Second, it is used to track the dependencies between constraints and decisions, so when a constraint violation is detected, the set of contributing decisions may be identified.

4.1 Decision Dependencies

Dependencies are maintained between the planning decisions that change the basic structure of the graph. Establishment decisions (simple or step-addition) depend on whatever decision lead to the creation of the precondition that they are establishing. Simple-establishment, furthermore, depends on the decision that led to the creation of the establisher. Decomposition decisions depend on the decision that led to the creation of the abstract task they are decomposing. They also depend on the decisions that led to the creation of whatever effects, or bound values they test in their condition field. (I currently do not track this later dependency.) Ordering decisions (promotion, demotion) depend on whatever decision introduced the task that threatened an IPC (though I also do not track this dependency).

Decomposition decisions can create a lot of structure; creating a set of subtasks and a set of establishment relations. To allow greater flexibility in plan repair, we actually model decomposition as a nested decision. Each causal link that is created by the decomposition decision is treated as a separate sub-decision that is dependent on the main decision. This allows threatened causal links to be retracted without retracting the whole decomposition.

4.2 Constraint Dependencies

Each time the planner must retract decisions (because of a constraint violation or an inability to make further progress), the dependency graph is queried to identify the set of decisions contributing to the problem. This involves reasoning about the various constraints introduced by decisions and how they interact. We discuss how this works in the context of each possible event that could lead to a decision retraction. Note that it sufficient (for completeness) to identify a superset of the decisions involved in the problem, but for search efficiency, this set should be minimal as possible.

Open-precondition flaw: The planner will have to retract some decision if there is an open precondition that cannot be established. The decisions implicated in this flaw are the decision that added the task associated with the precondition, and any decisions that introduced binding constraints on any of the variables mentioned in the precondition. Note that, since codesignation constraints are transitive, the fact that ?A equals ?B may involve a chain of codesignation constraints, each introduced by some decision. All of these decisions are implicated.

Abstract-task flaw: If an abstract cannot be decomposed, the decision that added the abstract task is implicated in the failure.

IPC threat: An IPC threat arises when the effect of some tasks violates an IPC. In this case, the decision that introduced the IPC and the decision that introduced the threatening task are both implicated.

Codesignation violation: A codesignation violation occurs when two variables are constrained to both condesignate and noncodesignate with each other. In this case, all of the decisions that are involved in the codesignation and noncodesignation of that variable are implicated.

Order-cycle: An order cycle occurs whenever there is a loop in the ordering relation over tasks: e.g. before(A,B) and before(B,A). As in codesignation constraints, we have to consider the decisions involved in the transitive computation of this cycle.

References

Ambros-Ingerson, J. A. and Steel, S. 1988. "Integrating Planning, Execution and Monitoring," in AAAI-88.

Bratman, M., Israel, D. and Pollack, M. 1988. "Plans and resource-bounded practical reasoning," *Computational Intelligence, 4,* pp 349-355.

Cohen, P. and Levesque, H., 1990. "Intention is choice with commitment," *Artificial Intelligence, 42(3).*

Durfee, E. H., Kenny, P. G., and Kluge, K. C. 1997. Integrated Permission Planning and Execution for Unmanned Ground Vehicles. *Proceedings of the First International Conference on Autonomous Agents,* pp. 348-354.

Firby, J. 1987. "An investigation into reactive planning in complex domains," In AAAI-87.

Georgeff, M. P., and Lansky, A. L. 1987. "Reactive reasoning and planning," in AAAI-87, pp. 677-682.

Golden, K., Etzioni, O., and Weld, D. 1994. "Omnipotence without Omniscience: Efficient Sensor Management for Planning." in AAAI-94, Seattle, WA.

Gratch, J. 1997. Task-decomposition planning for command decision-making, *6th Conf on Computer Generated Forces and Behavioral Representation.*

Grosz, B., and Kraus, S. 1996. "Collaborative Plans for Complex Group Action," Artificial Intelligence, 86(2).

Hill, R., Chen, J., Gratch, G., Rosenbloom, P., and Tambe, M. 1997. "Intelligent Agents for the Synthetic Battlefield," in AAAI-97/IAAI-97, pp. 1006-1012.

Kambhampati, S. and Hendler, J. 1992. A validation-structure-based theory of plan modification and reuse. *Artificial Intelligence 55, pp 193-258.*

Kautz, H. and Selman, B. 1996. "Pushing the Envelope: Planning, Propositional Logic, and Stochastic Search," in AAAI-96, Portland, OR, pp. 1194-1201.

Knoblock, C. 1995. "Planning, executing, sensing, and replanning for information gathering," *IJCAI*-95

McAllester, D. and Rosenblitt, D. 1991. "Systematic Nonlinear Planning," in AAAI-91.

Minsky, M. 1985. *The Society of Mind.* Simon and Schuster

Newell, A. 1990. *Unified Theories of Cognition.* Harvard Press.

Pednault, E. P. D, 1986. Formulating Multiagent, Dynamic-world Problems in the Classical Planning Framework. Reasoning about Actions and Plans, Georgeff, M. and Lansky, L. (eds.). Morgan Kaufmann Publishers, Inc.

Pollack, M.E., 1992. "The uses of plans," *Artificial Intelligence, 57(1)*, pp 43-68.

Rickel, J. and Johnson, L. 1997. Intelligent tutoring in virtual reality," in *Proc. of World Conf on AI in Education.*

Traum, D. and Allen, J., 1994. Towards a formal theory of repair in plan execution and plan recognition. *Procedings of UK planning and scheduling special interest group.*

Tambe, M. 1997. "Agent Architectures for Flexible, Practical Teamwork," in AAAI-97, pp. 22-28.

Wilensky, R. 1980. "Meta-Planning," Proceedings AAAI-80, Stanford, CA, pp. 334-336.

Wilkins D. E. and Myers, K. 1996. Asynchronous dynamic replanning in a multiagent planning architecture. *Advanced Planning Technology,* Austin Tate, AAAI Press.

Wolverton, M. J. and desJardins, M. 1998. Controlling Communication in Distributed Planning Using Irrelevance Reasoning. *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI98).* Madison, WI.