Action-Based Planning

Amy L. Lansky

Recom/NASA Ames Research Center
MS 269-2, Moffett Field, CA 94035-1000
(415) 604-4431 LANSKY@PTOLEMY.ARC.NASA.GOV

Abstract

This paper presents an approach to domain representation and planning that is fundamentally different from traditional methods – an approach based strictly on actions and their interrelationships, rather than on state-based goals and preconditions. In particular, we focus on the action-based planner COLLAGE, describe its methods for plan-construction, and contrast them with more traditional planning techniques.

1 Introduction

Over the years, the term "planning" has become synonymous with state-based reasoning – especially the kind of reasoning associated with STRIPS-based domain descriptions. In such a framework, the world is viewed as a sequence of states and is modeled in terms of state predicates. Planning algorithms add actions and relations into a plan in order to ensure that state-based propositions (goals, preconditions) hold at particular points in the execution of a plan. As a result, actions are treated as "second-class citizens," serving as means toward ultimately state-based ends.

However, in the real world, when we look beyond puzzles and blocks stacked on tables, we find that people usually think directly in terms of the actions they perform.[1] A plumber thinks: "I'm going to install this faucet, but first I have to prep the wall." He or she rarely thinks: "I want to attain the state *installed-faucet*, which I can achieve by installing the faucet. Before I can do the installation, I have to attain the state *prepped-wall*, which I can achieve by prepping the wall." Such reasoning would simply be an inefficient way of thinking about the world. The central point of this paper is that planners can do what people do – simplify domain characterization and planning by reasoning directly in terms of actions rather than state.

Efforts at applying planning technology to realistic domains have constantly been pulled towards some form of action-based reasoning. For example, SIPE, OPLAN and other hierarchical task network (HTN) planners [3, 13] use task decomposition; it is inevitably the most commonly used mechanism in the synthesis of real-world plans. Although task networks are usually viewed as a means of "goal" decomposition (i.e. in terms of state), practically speaking, they merely specify how to decompose some high level action into a network of subactions that perform the requisite task. Similarly, much of the work in reactive and case-based planning has focused on the use of procedures to capture real-world planning knowledge [4, 5, 6]. Although a procedure can be represented in terms of the state-based goals it achieves, for all practical purposes it is simply a compact structure for representing a sequence of actions to be performed. People inevitably utilize procedures in this way to represent knowledge about "how to do things" – i.e. their plans.

Of course, viewing the world primarily in terms of actions instead of state is essentially a "dual" approach; one can argue that the two are equivalent. Yet, how often a change in representation can enable more facile domain encoding and reasoning! This is particularly true for coordination-intensive or logistical domains, the target domain class for our work. When confronted with the task of coordinating multiple streams of activities, we have found that people tend to consciously manipulate action-ordering requirements; the state-based rationale for action orderings are left implicit. One reason for this may be the limited nature of human perceptive capabilities. If many activities are going on in parallel, it is easier for us to focus on actions and their sequencing than it is to sense the unpredictable and, in some cases, unobservable sequence of global domain states.

Consider the encoding of resource usage policies, which are quite common to coordination-intensive domains. A plan may include "request" actions that register a request to use a resource and "serve" actions that utilize the resource. In order to enforce a "first-come-first-serve" policy, a planner must ensure that "serve" actions are ordered in the same way as their corresponding "request" actions. In a state-based encoding, a request-queue state object must be used to represent the ordering of request actions. Request and serve operators manipulate this queue (in terms of preconditions and postconditions) to attain the correct behavior – a formulation that can be diffi-

[1] We use the term *action* to refer to both high-level "abstract" actions as well as low-level executable actions.

cult to construct correctly [8]. This representation is also distributed among the various operator descriptors, rather than stated in a unitary compact form. In contrast, it is easy to represent this policy in terms of a single and compact action-based requirement:[2]

`(req ?id1)=>(req ?id2)` $\supset$ `(serve ?id1)=>(serve ?id2).`

Over the past few years, we have developed representation and planning techniques that focus on the action-based approach. Our first planner, GEMPLAN [8, 9], also incorporated state-based reasoning. Our new system, COLLAGE [10], is strictly action-based. The primary purpose of this paper is to describe this unique approach to planning. Section 2 begins by describing COLLAGE's constraint-satisfaction architecture. Section 3 describes the semantics of the COLLAGE constraint-forms and contrasts their use with more traditional planning methods. Finally, Section 4 provides further discussion about the relationship between action-based and state-based planning.

We hope that readers leave this paper with one key idea: *extremely useful and powerful planning can be done without reasoning about state conditions.* Two realistic domains we have tackled – building-construction planning [10] and data analysis planning [11] – have not required any traditional state-based reasoning. Indeed, COLLAGE's action-based approach was so natural to these domains that we feel a state-based encoding would seem awkward in comparison. The initial office building domain we worked on was also the focus of a study using SIPE [7]. SIPE's difficulties in handling this domain ultimately led to the revisions in SIPE-2. In contrast, not only was COLLAGE's implementation straightforward, but empirical tests have shown significant speedups over the run-times achieved with SIPE. Although the most significant speedups were attained using COLLAGE's localized (partitioned) search mechanism [9, 10], even a non-localized version of this domain did much better than SIPE. This result is the best evidence we have to date of the relative efficiency of action-based planning over traditional, state-based planning. Please see [10] for full details about these experiments as well as a more thorough treatment of COLLAGE's plan construction methods and localized search.

2 Planning With Constraints

Intrinsic to COLLAGE is the notion of planning as "constraint satisfaction." Each COLLAGE domain/problem description includes: a set of action types (*ActionTypes*), instances of which may appear in a plan; a set of constraints (*Constraints*) that must be obeyed by the final plan; and a static set of domain- and problem-specific facts and functions (*KnowledgeBase*) that may be used during the planning process. COLLAGE's task is to create a plan consisting of actions of the types in *ActionTypes* that satisfies all constraints in *Constraints*. As we will show, one of the primary purposes of *KnowledgeBase* is to provide information that conditionalizes the application of these constraints.

In contrast to a STRIPS operator, a COLLAGE action type description simply provides an action name and the types of its parameters. For instance,

`:action-type (pick ?b_block)`

defines a `pick` action type, an instance of which might be `(pick A)`.

Each COLLAGE plan consists of: *actions*, *relations between actions*, and *unary and binary binding requirements on action parameters*. There are several kinds of COLLAGE relations, including the temporal relation (=>), a causal relation ($\rightsquigarrow$) and a subaction relation. Since we are focused on coordinative (multiagent) domains, the temporal relations in a plan form a partial ordering. COLLAGE's planning algorithms account for all plan executions consonant with a partial order, including those with action simultaneity. The binding requirements embedded within a plan form a network (much like a CSP-network [12]) and are propagated as part of the planning process.

Each constraint in *Constraints* must be an instance of a constraint form in the COLLAGE constraint library. Note that we use the term "constraint" here very loosely: it is any kind of property that the planner knows how to test and enforce. For example, a *decompose* constraint requires that each instance of a high-level action type be decomposed into subactions in one of a set of prescribed ways. When enforcing a constraint of this form, the planner will find instances of the high-level action type, test to see if they are decomposed, and will do so if they are not.

Each domain constraint C is associated with three mechanisms that it inherits from its constraint form and instantiates for itself: a *check*, a set of *fixes*, and an *activator set* (accompanied by an *initial activation setting*). The role of C's check is to test whether or not C is satisfied by a plan. Each check method returns a list of *bugs* – violation instances within the plan. For example, the check for a decompose constraint for an action type A would return all A actions that are not yet decomposed. The fix methods for C implement the various possible plan "repairs" that will satisfy C – they are COLLAGE's plan-construction methods. For example, if an A action can be decomposed in n possible ways, there will be n possible fixes for its decompose constraint. All COLLAGE fixes are additive – i.e. they add new actions, relations, and variable binding requirements into the plan.

The purpose of C's activator set is to indicate when C may be violated; the initial activation setting states whether C is considered to be active when planning begins. C's activator set is composed of a set of action

[2] => denotes the temporal relation; action parameters ?id1 and ?id2 are used to capture the correspondence between request-serve pairs.

types that could trigger a violation of C. If any instance of one of these types is added into the plan or if one of its parameter bindings is further constrained, C will be activated for consideration by the planner. For instance, the decompose constraint for A has activator set $\{A\}$. Activators are designed to be conservative; i.e. a constraint must be activated when it is violated, but may sometimes be activated when it is not.

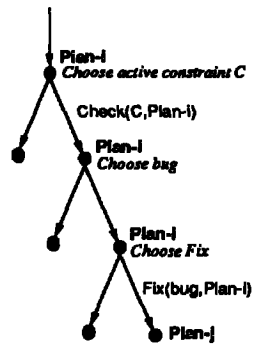

Figure 1: Constraint Satisfaction Search

When a constraint is activated by the addition of new plan information, it is posted to a constraint agenda that is used to drive planning search (see Figure 1). Each node in the search space is associated with a plan constructed up to that point in the reasoning process. COLLAGE's search cycles through the following steps: choose an activated constraint and apply the constraint check; if the constraint is not satisfied, choose a bug from the set of constraint bugs; choose and apply a fix, yielding a new plan and, possibly, newly activated constraints. At this point, remaining bugs might be tackled or the set of active constraints might be reconsidered – COLLAGE currently does the former by default. The branching factors in this space include the set of activated constraints, the set of bugs, and the set of possible fixes to satisfy a constraint. All choice points within this search space are backtrackable, including choice points *within* a fix (i.e. there may be several possible ways to repair a plan within the context of a specific fix). However, to attain practical performance, COLLAGE does limit the completeness of this space in practice. Note that constraint activations are stored in a plan-relative fashion, enabling old activations to be reconsidered upon backtracking.

Traditional planning may be viewed as a specialization of this constraint-satisfaction approach. In traditional frameworks, the only kind of "constraint" is the attainment of a state condition at a specific point in the plan (a goal or precondition). The only "fixes" employed by traditional planners are those based on the modal truth criterion [2]. Like COLLAGE's activators, the justification structures employed by many traditional planners provide a way of monitoring potential constraint violations.

However, in contrast to many classical planners, COLLAGE constraint fixes may add new actions, relations, and bindings *anywhere* within a plan – where they are added depends on the nature of the constraint and the current plan structure. As a result, the plan "grows" in a way determined by constraint application ordering and the nature of the constraints themselves – not in forward (or backward) execution order, as is common in some state-based frameworks.

3 Constraint Forms

Each COLLAGE constraint form is associated with a name and a set of "slots" that parameterize its use. A particular constraint is specified by supplying the name of its form and a set of slot values. Each COLLAGE constraint may also be associated with two additional slots, `condition` (used for constraint conditionalization) and `binding-req`. We begin this section with a discussion of the use of this information. We then describe the basic constraint forms.

Once an activated constraint has been chosen for application, its associated condition (if it has one) will be tested. A constraint condition is a list (conjunction) of boolean tests of the following types: queries on plan content, queries on facts in *KnowledgeBase*, and calls to boolean functions. Wild card formal variables may be used in these tests. When a condition is tested, all possible combinations of values for these formals will be found. COLLAGE will then apply the constraint in each of these individual binding contexts that satisfies the condition. For example, suppose that *KnowledgeBase* includes information about the floors in an office building. Given[3]

```
:condition ((fact (floor ?f))
            (test (> ?f 5))
            (action (build-column ?f ?c)))
```

COLLAGE would return all possible binding pairs for the variables `?f` (of type `floor`) and `?c` (of type `coordinate`) that satisfy the condition. These would correspond to all floors (greater than `5`) and column coordinates in the office building for which a `build-column` action has been added into the plan.

We have found that the use of constraint conditionalization yields great expressive power. A constraint will be applied exactly and only in those situations in in which its condition is true – often causing a single constraint application to branch into multiple application instances (potentially in reaction to the external "state" of *KnowledgeBase* or the current form of the plan). This use of conditionalization often enables the same constraints to be utilized in numerous contexts.

After a particular bug is handled by a fix algorithm, COLLAGE applies the binding requirements supplied in `binding-req`. Examples of such binding requirements

[3]Each formal must be associated with a binding before it is passed into a boolean function. The action types queried in a constraint condition are, by default, added to the set of action types in the constraint's activator set. This enables COLLAGE to activate constraints when the truth of its condition is nonmonotonic. For further discussion of monotonicity, see [10].

are provided in the next section. COLLAGE's variable binding propagation facility is extensive, allowing for variables from both enumerable and nonenumerable domains. Binding requirements can be defined by functions or via unification with facts provided in *KnowledgeBase*. Numeric variables can be constrained by linear functional requirements, which enables metric-time reasoning. The facility also allows for variables with internal structure (i.e. variables that are composed from other variables, each of which may be individually constrained).

3.1 Action Constraint

The simplest constraint form in COLLAGE is the *action* constraint. It is used to ensure that particular actions – usually "high-level" actions – are present in the plan. As such, it is typically used to express problem-specific goals. Each action constraint provides a set of action descriptors $A1...An$.[4] The constraint is satisfied if, for each Ai, there exists some action in the plan that matches Ai. Its semantic truth criterion is thus:

$$(\forall\ Ai \in \{A1...An\})\ (\exists\ ai{:}Ai\ ).$$

The check algorithm for an action constraint tests to see if matching actions exist. If not, the fixes either add new matching actions or specialize existing actions (by imposing binding requirements) so that they now match their corresponding action descriptor. All action constraints are active when planning begins (since they represent "goals"). Once satisfied, action constraints cannot be violated; their activator set is empty.

As an example, consider the following constraint, used to add high-level `build-beam` actions into an office-building construction plan. Given a description of the building in terms of a set of "pod" facts in *KnowledgeBase*, a beam is added for each side of a pod.[5] Notice how the formulation below allows a single constraint to be used in numerous contexts, as defined by the contents of *KnowledgeBase* (e.g. every building must have beams – but each specific building has different beam requirements).

```
(action
  :condition ((fact (pod ?f ?c1 ?c2 ?c3 ?c4)))
  :actions ((build-beam ?f ?c1 ?c2)
            (build-beam ?f ?c3 ?c4)
            (build-beam ?f ?c1 ?c3)
            (build-beam ?f ?c2 ?c4)))
```

3.2 Temporal/Causal Constraints

This class of constraints consists of four constraint forms: *tempbefore, tempafter, enable,* and *cause*. Each temporal/causal constraint provides two action descriptors, A and B. The constraint is satisfied if matching action instances of type A and B exist in the plan which are temporally or causally related in the specified way. Below are the semantic truth criteria for all four forms:

`tempbefore(A B)`: $(\forall\ b{:}B)\ (\exists\ a{:}A)\ (a \Rightarrow b)$

`tempafter(A B)`: $(\forall\ a{:}A)\ (\exists\ b{:}B)\ (a \Rightarrow b)$

`enable(A B)`: $(\forall\ b{:}B)\ (\exists\ a{:}A)\ (a \rightsquigarrow b)$

`cause(A B)`: $(\forall\ a{:}A)\ (\exists\ b{:}B)\ (a \rightsquigarrow b)$

The checks, fixes, and activators for all four constraint forms are quite similar. For example, `tempbefore(A B)` has activator {`B`} – it will be activated any time an action of type `B` is added into the plan. The check makes sure that an `A` action exists in the plan that is temporally before the `B` action. If there is not, the fixes will either add a new `A` action into the plan or reuse an existing `A` action, and will also make sure that it precedes the `B` action.

`Tempbefore` and `enable` constraints are often used to express domain requirements traditionally expressed as state preconditions. For example,

```
(tempbefore
  :actions ((paint ?floor ?color)
            (carpet ?floor ?carpet))
  :binding-req ((color-carpet-match ?color ?carpet)))
```

requires that painting on a particular floor precede carpeting. The attached binding requirement forces paint color to match the carpeting. The predicate `color-carpet-match` would be defined by a function or via unification with facts in *KnowledgeBase*.

In contrast to `tempbefore` and `enable`, the `tempafter` and `cause` constraint forms have less obvious analogues in traditional backchaining planners. This is because such planners satisfy preconditions in order to assure the "safety" of operator use, but do not typically assure eventualities (other than goals) that depend on the emerging form of the plan. For example,

```
(tempafter
  :condition ((fact (floor ?f))
              (test (> ?f 3)))
  :actions ((build-balcony ?f ?c1 ?c2)
            (install-escape-ladder ?f ?c1 ?c2)))
```

will add `install-escape-ladder` actions only when a balcony is being built above the third floor.

3.3 All-Match Constraints

This class of constraints is also composed of four constraint forms: *all-match-before, all-match-after, all-match-enable,* and *all-match-cause*. These constraints ensure that matching actions in the plan are related in a particular way. Unlike temporal/causal constraints, the fixes for these constraints do not add new actions into a plan – they only create new relations between existing actions. The truth criteria are:

`all-match-before(A B)`: $(\forall\ b{:}B)\ (\forall\ a{:}A)\ (a \Rightarrow b)$

`all-match-after(A B)`: $(\forall\ a{:}A)\ (\forall\ b{:}B)\ (a \Rightarrow b)$

[4] An action descriptor is similar to an action type description but can have instantiated parameters. For example, `(put A ?s)` matches any `put` action that places block `A` onto some surface `?s`. We use $a{:}A$ to denote that action a matches descriptor A.

[5] A pod is a cube-like building-block described by a floor and four corner coordinates.

all-match-enable(A B): $(\forall\ b{:}B)\ (\forall\ a{:}A)\ (a \rightsquigarrow b)$

all-match-cause(A B): $(\forall\ a{:}A)\ (\forall\ b{:}B)\ (a \rightsquigarrow b)$

All four constraint forms have the same activator set, {A,B}. The check and fix algorithms are also similar. For example, given **all-match-before(A B)**, the check will find all B actions, and for each, will find all matching A actions. If an A action does not precede its matching B action, the pair will be returned as a bug, which the fix will repair by adding the necessary temporal relation.

All-match constraints are useful when a particular action B does not *require* a corresponding activity A in relation to it, but if corresponding activities do exist, they must have some particular temporal or causal relationship. For example,

```
(all-match-before
  :actions ((install-embedded-slab-utilities ?slab)
            (pour-slab ?slab)))
```

requires any needed utilities to be embedded in a slab before the slab is poured. However, if no such utilities are needed, the constraint would have no effect. This kind of requirement is somewhat awkward to describe in a STRIPS-based framework – in particular, *negative preconditions* must be used. In this example, **(pour-slab ?slab)** would have effect *(slab ?slab)* and **(install-embedded-slab-utilities ?slab)** would have precondition *(not (slab ?slab))*.

3.4 Decompose Constraint

The COLLAGE *decompose* constraint is similar to task reduction in a traditional planner or problem decomposition as used by systems like REAPPR [1]. Each decompose constraint provides an action descriptor, A, and a set of decomposition descriptors, *Decomps*. The constraint is satisfied if each action matching A is decomposed in exactly one of the possible ways provided by the decomposition descriptors.

Each decomposition descriptor includes the following slots: (1) **SubActions**: action descriptors for the subactions in the decomposition; (2) **FirstSubActions**: the (temporally) first subactions of the decomposition; (3) **LastSubActions**: the (temporally) last subactions of the decomposition; (4) **Relations**: required relations between subactions; (5) **Condition**: a condition that constrains the situations in which the decomposition can be used. The form of the condition is the same as a general constraint condition. However, rather than applying a decomposition for all possible binding contexts in which the condition is true, a single context is chosen; and (6) **Binding-Req**: additional binding requirements on subaction parameters.

The truth criterion for this constraint form requires that each A action be decomposed in one of the ways described in *Decomps*. The check finds all A actions that are not decomposed (note: an action can only be decomposed once). The fixes decompose these actions in one of the prescribed ways. COLLAGE also assures that the temporal relationships between a high-level action, its subactions, and other actions in the plan are coherent. A slight variant of the decompose constraint form, *decompose-reuse*, reuses existing actions as part of a decomposition instead of creating new subactions.

COLLAGE decompose constraints have advantages over traditional HTN approaches. First, the use of decomposition-specific conditions and binding requirements yields great expressive power. Second, the relations between subactions can form any arbitrary partial order. Finally, COLLAGE retains high-level actions in a plan after they are decomposed rather than replacing them. As a result, all constraints can be applied to actions at mixed levels of detail.

3.5 Pattern Constraint

The pattern constraint form has no real analogue in traditional planners. Each pattern constraint provides a set of action descriptors, *PatternActions*, and a regular expression, *Pattern*, in a language that admits repetition and disjunction. A pattern constraint is satisfied if all action instances that match the action descriptors in *PatternActions* are totally ordered and this total order satisfies *Pattern*.

Consider the following example from a Blocks World domain, where **(pick ?x)** represents the act of picking up block **?x** and **(put ?y ?z)** represents the act of putting block **?y** onto a surface **?z**.

```
(pattern
  :actions ((pick ?a) (put ?b ?c))
  :regexp  (((pick ?x) => (put ?x ?y))*=>
            [rebind ?x ?y]))
```

This constraint applies to all **pick** and **put** actions in a plan. It requires that these actions alternate between **pick** and **put**, beginning with a **pick** action and ending with a **put** action.[6] Moreover, each **pick** action for some block must be immediately followed in the sequence by a **put** that places that same block onto some surface.

The activator for a pattern constraint is *PatternActions*. The check algorithm gathers all matching actions and tests to see if they satisfy *Pattern*. The fix algorithm assures this by adding appropriate relations and binding requirements. Thus, the fix does not *create* actions in order to satisfy a pattern – it only restricts their ordering and parameter bindings. Alternative solutions will be generated if backtracking occurs within the fix.

3.6 Binding-Requirement Constraint

A binding-requirement constraint simply conditionalizes the application of binding requirements. For example, the following would assure that carpeting on the penthouse floor is luxurious:

[6] An expression of form **(Pattern)*=>** denotes zero or more repetitions of **P**. The **rebind** expression indicates which parameter descriptors can be rebound after each iteration.

```
(expl-binding-req
  :condition ((action (carpet ?floor ?carpet))
              (test (top-floor ?floor)))
  :binding-req ((luxurious ?carpet)))
```

4 Discussion

In our experiences with COLLAGE, it has been straightforward to describe and plan for several realistic domains without resorting to any traditional form of state-based reasoning. Indeed, traditional formulations often strain to achieve the same effect as a simple action-based constraint. Consider a construction requirement that tile be laid before faucet installation. This requirement would be encoded by requiring that a state condition *laid-tile* hold before each faucet-installation action. But if there is only one way to achieve *laid-tile* and no way of undoing it, it is easier to simply require a temporal relationship between tile-laying and faucet-installation actions:

```
(tempbefore
  :actions ((lay-tile) (install-faucet)))
```

We might also decompose `lay-tile` in various ways, thereby attaining the same effect as allowing for many ways of "achieving" *laid-tile*. When "clobberers" do exist (e.g. `strip-tile`), we can often use an a pattern to represent desired forms of behavior. For instance,

```
((lay-tile=>strip-tile)*=> => lay-tile=>install-faucet)
```

would allow tile to be repeatedly laid and stripped, but ultimately, a `lay-tile` action must be the last tile-related action before `install-faucet`.

Of course, there are ways in which state manifests itself in an action-based framework. For example, a constraint condition may be viewed as describing the "states" in which the constraint should be applied. Indeed, such conditions often serve a function similar to that played by filter conditions. For example, the conditionalized `tempafter` constraint in Section 3.2 filters the situations in which `install-escape-ladder` actions are added. *KnowledgeBase* also contains "state" information. For instance, the `pod` facts defining a building may be viewed as goals. The initial state of a construction problem (e.g., a definition of available resources) would also be supplied in *KnowledgeBase*. And if we extended *KnowledgeBase* to allow for dynamic addition of facts, it could be used to track the state of the world during plan execution. Indeed, we plan to eventually extend COLLAGE to allow for more dynamic forms of constraint application, especially in reaction to run-time information.

5 Conclusions

This paper presented an alternative approach to planning based on the satisfaction of action-based constraints and contrasted it with traditional state-based methods. The constraint forms of the COLLAGE planner were described, including temporal/causal constraints forms, action decomposition, and a unique pattern constraint form. COLLAGE also includes powerful mechanisms for constraint conditionalization and variable-binding requirements. Action-based planning possesses at least three features that underly its utility: (1) *expressiveness*: action-based constraints are powerful, natural to use, and compact; (2) *efficiency*: the approach is tuned to coordination-intensive reasoning; (3) *extendibility:* rather than constructing a plan using one method (based on the modal truth criterion), COLLAGE uses an extendible suite of constraint forms, each associated with its own plan-construction method. This approach allows use of the most natural constraint form for each domain requirement.

Acknowledgments

I would like to acknowledge the contribution of past and present members of the COLLAGE project: Lise Getoor, Andrew Philpot, Phil Chu, and Scott Schmidler. John Bresina, Nicola Muscettola, and John Allen also provided useful comments about this paper.

References

[1] Bresina, J. "REAPPR – An Expert System Shell for Planning," LCSR-TR-119, LCSR, Rutgers (1988).

[2] Chapman, D. "Planning for Conjunctive Goals," *Artificial Intelligence*, Volume 32, pp. 333-377 (1987).

[3] Currie, K. and A. Tate, "O-Plan: The Open Planning Architecture," *Artificial Intelligence*, Volume 52, pp. 49-86 (1991).

[4] Firby, R.J. *Adaptive Execution in Complex Dynamic Worlds*, Yale University Computer Science Department, TR 672 (1989).

[5] Georgeff, M.P. and A.L. Lansky, "Reactive Reasoning and Planning," *AAAI-87*, pp. 677-682 (1987).

[6] Hammond, K.J. "CHEF: A Model of Case-Based Planning," *AAAI-86*, pp. 261–271 (1986).

[7] Khartam, N.A. *Investigating the Utility of Artificial Intelligence Techniques for Automatic Generation of Construction Project Plans*, PhD Dissertation, Stanford University Civil Engineering Dept. (1989).

[8] Lansky, A.L. "Localized Event-Based Reasoning for Multiagent Domains," *Computational Intelligence Journal (4,4)* pp. 319-340 (1988).

[9] Lansky, A.L. "Localized Search for Multiagent Domains," *IJCAI-91*, pp. 252–258 (1991).

[10] Lansky, A. L. "Localized Planning with Diverse Plan Construction Methods," NASA TN FIA-TR-93-17 (1993). Submitted to *Artificial Intelligence*.

[11] Lansky, A.L. and A.G. Philpot, "AI-Based Planning for Data Analysis Tasks," *Ninth IEEE Conference on Artificial Intelligence for Applications (CAIA-93)*, Orlando, Florida (1993).

[12] Mackworth, A. K. "Consistency in Networks of Relations," *Artificial Intelligence*, Volume 8, pp. 99-118 (1977).

[13] Wilkins, D. *Practical Planning*, Morgan Kaufmann Publishers, San Mateo, California (1989).