SRI International

A Common Knowledge Representation for Plan Generation and Reactive Execution

Technical Note No. 532

June 9, 1993

By: David E. Wilkins, Sr. Computer Scientist
Artificial Intelligence Center
Computing and Engineering Sciences Division

Submitted for publication to the journal *Computational Intelligence*

This work was supported in part by the Defense Advanced Research Projects Agency under the DARPA/Rome Laboratory Planning Initiative under Contracts No. F30602-91-C-0039 and F30602-90-C-0086.

The views, opinions and/or conclusions contained in this note are those of the author and should not be interpreted as representative of the official positions, decisions, or policies, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, or the United States Government.

333 Ravenswood Avenue • Menlo Park, CA 94025-3493 • (415) 326-6200 • FAX: (415) 326-5512 • Telex: 334486

Abstract

This paper describes the ACT formalism, which is designed to encode the knowledge required to support both the generation of complex plans and reactive execution of those plans in dynamic environments. ACT is an heuristically adequate representation that is useful in practical applications, and serves as an interlingua for Artificial Intelligence (AI) technologies in planning and reactive control. The design of the formalism is discussed and example uses from practical applications are presented. These applications show that the ACT representational constructs have reasonable computational properties as well as being adequately expressive.

Contents

1 Introduction **1**

2 Planning and Execution Technologies **4**

2.1 SIPE-2 5

2.2 PRS 10

3 Developing a Common Representation **14**

4 The ACT Formalism **17**

4.1 ACT Slots 18

4.2 Plots 23

4.3 Metapredicates 25

4.4 Using ACTs 30

4.5 ACT-Editor 34

4.6 Translators 37

5 Conclusion **38**

A Syntax for ACT Metapredicates **42**

B Creating ACTs Programmatically **45**

List of Figures

1	Sipe Operator for Deploying an Air Force	7
2	Sipe Deductive Rule for Removing Located Facts	9
3	Structure of the Procedural Reasoning System	10
4	Portion of a KA for Deploying an Air Force	12
5	Portion of a KA for Leak Isolation	13
6	ACT Slots	17
7	ACT Metapredicates	19
8	SOCAP Deploy Airforce Operator as an ACT	22
9	ACT with Graphical Display of Plot	24
10	Plot of Deploy Airforce ACT	25
11	An ACT for Establishing a Lookout	32
12	An ACT for Deducing Locations	34
13	Plot of Deploy Airforce ACT	35

1 Introduction

Our research is developing software to take appropriate actions in a dynamic and uncertain environment. The dynamic environment requires that the execution system be able to respond to stimuli and make decisions in "real time." Furthermore, we are interested in applications complex enough that the ability to synthesize plans and use these plans to guide the execution system is critical. We have developed a representation, called the ACT formalism, for representing the knowledge necessary to both generate plans and execute them while responding to external events. This paper describes the ACT formalism and an implementation of it.

Two features distinguish our approach: (1) the development of an heuristically adequate system that will be useful in practical applications, and (2) the need to generate complex plans with parallel actions of multiple agents. To achieve these requirements, we started with artificial intelligence technologies in planning and reactive control (described in Section 2) that had already been implemented and shown to be useful in practical applications. However, these technologies did different types of reasoning using different knowledge representations — Sections 3 and 4.3 describe the difficulties in using a common representation for these tasks. In this paper, we describe the ACT formalism, which is a common representation that planning and reactive control systems can use for representing and reasoning about plans and the knowledge necessary to generate and execute them. The planning and execution technologies were extended and merged to fully implement the ACT formalism.

Developing an interlingua so that different reasoning systems can cooperate on different aspects of the same problem is a central challenge for the field of artificial intelligence. This paper presents an attempt to do this for a narrowly focused problem, namely getting planning and execution systems to use a common language. We view this narrow focus as critical to achieving success, since a common representation that tries to cover too broad an area runs into serious problems that have been described elsewhere [7].

Our integration of planning and control focuses on domains with the following

properties (among others):

- Reactive response is required.
- Plans must be generated to achieve acceptable performance.
- The system must be able to act without a plan.
- Complex plans (e.g., including parallel actions) must be supported in both planning and execution.

We give two examples of domains that meet these requirements. Our software has been used to do planning and reactive execution in both. The first domain is controlling an indoor mobile robot. A reactive control system is necessary to respond to people and obstacles that may suddenly and unexpectedly appear in the robot's path. Deliberative planning is necessary so that the robot can achieve purposeful behavior, such as retrieving a book from the library and bringing it to someone's office. The second domain is military operations planning. Certainly one would not engage in an operation like Desert Storm without first doing deliberative planning to achieve the requirements of the mission. Reactive response is also necessary because execution of military plans is often interrupted by unexpected equipment failures, weather conditions, and enemy actions. Application of our systems to this domain was done as part of the ARPA-Rome Laboratory Planning Initiative (DRPI) [17].

An important design goal of our research is to develop an heuristically adequate system that will be useful in practical applications, thus the need to generate complex plans with parallel actions that are interrelated. For example, in military operations planning, many actions are being executed simultaneously, and changes in the way one action is accomplished may affect the execution of other parallel actions. Because of the complexity of the plans, a graphical user interface is required to understand the plans and system behavior.

The complexity of the plans is what differentiates our approach from previous approaches to the integration of planning and execution. Lyons and Hendricks [11] describe an approach in which the planner monitors the execution of the reactive system

and specifies *adaptations* to the reactive system that will improve its performance relative to achievement of the given goals. They require these incremental adaptations to *improve* system performance before they are added to the reactor. The RAP system [4] also uses simple plans to modify the reactive control of a robot. Such approaches work in the robot problem, where plans are relatively simple, but in the military planning problem, the planner must exercise more control over the execution system. In our system, the planner produces much more complex plans than in either of the above systems, and does not modify the reactor to improve its performance — rather, the planner generates a plan that will be the principal guidance for the reactor. The reactor will generally have no idea how to accomplish the top-level goals appropriately until the planner has generated a plan. The reactor will implement appropriate lower-level behaviors while it is waiting for a plan, and may pose additional problems to the planner during execution.

The development of the ACT formalism is similar in motivation to the development of the KRSL language [10] that is being done as part of the DRPI. However, the ACT formalism is more focused, trying only to get planning and execution systems to speak a common language, while the KRSL effort is more ambitious, trying to develop a common language for many types of systems, including systems for planning, execution, scheduling, simulation, temporal reasoning, database management, and other tasks. The ACT formalism has been one of the formalisms driving the design of the KRSL specification for plans.

The purpose of this paper is to describe the ACT formalism and the design decisions behind it. There are many different ways in which planning and execution systems can interact, but an analysis of such alternatives is beyond the scope of this paper. Although we mention how this interaction takes place in our implementation (Section 4.4), this is still an area of ongoing research. A complete description of the applications of the ACT formalism to the robot and military problems is also beyond the scope of this paper, although we do give some examples. The application of the planner to these domains has been described elsewhere [13, 17], and the translation from its representation to ACT is described in this paper.

2 Planning and Execution Technologies

To achieve an heuristically adequate system that will be useful in practical applications, we started with AI technologies in planning and reactive control that had already been implemented and shown to have these properties. In particular, the basis for our system is the SIPE–2 (System for Interactive Planning and Execution) planning system and the Procedural Reasoning System (PRS).[1] Before the research reported in this paper, these two systems did different types of reasoning using different knowledge representations. The ACT formalism is a common representation that both systems can now use for representing and reasoning about plans and the knowledge necessary to generate and execute them.

The development of PRS and SIPE–2 has been driven by their applications to numerous problem domains. Similarly, the development of the ACT formalism has been driven by the representations of these two systems and by the military operations planning domain. The complete inputs for both SIPE–2 and PRS can be specified in the ACT formalism. Since we have implemented translators from the ACT formalism to both SIPE–2 and PRS, as described in Section 4.6, it is now the recommended language for encoding knowledge in both systems.

Because both SIPE–2 and PRS have been applied to several different domains of practical interest, the ACT formalism is sufficient for a wide range of interesting problems. However, the ACT formalism should not be considered a fixed representation that has been designed to cover every anticipated need. We expect extensions to be made to it as new domains require new features. This is consistent with our goal of handling practical applications, and has the advantage that our representational constructs have their properties tested in real domains as they are added to the formalism. By so doing, we avoid the problem of developing extremely expressive representations that have poor computational properties.

Before describing the ACT formalism, we briefly describe the two systems for which it provides a common representation. More detailed descriptions can be found else-

[1] SIPE–2 and PRS are trademarks of SRI International.

where [5, 13, 14, 16].

2.1 SIPE-2

SIPE–2 is a partial-order AI planning system that supports planning at multiple levels of abstraction. It provides a formalism for describing actions as *operators* and utilizes knowledge encoded in this formalism, together with heuristics for reducing the computational complexity of the problem, to generate plans for achieving given goals. Given an arbitrary initial situation, the system either automatically or under interactive control combines operators to generate plans, possibly containing conditionals, to achieve the prescribed goals. SIPE–2 is capable of generating a novel sequence of actions that responds precisely to the situation at hand. The generated plans include information so that during plan execution the system can accept descriptions of arbitrary unexpected occurrences and modify its plans to take these into account.

Because this technology is generic, domain-independent, and heuristically adequate, it has the potential to impact a large variety of problems. Example applications include planning the actions of a mobile robot, planning the movement of aircraft on a carrier deck, travel planning, construction tasks, the problem of producing products from raw materials on process lines under production and resource constraints, and joint military operations planning [14, 17]. SIPE–2 provides a powerful graphical user interface to aid in generating plans, viewing complex plans and other information as graphs on the screen, and following and controlling the planning process [17].

SIPE–2's formalism allows reasoning about resources, the posting and use of constraints on planning variables, and the description of a deductive causal theory to represent and reason about the effects of actions in different world states. In contrast to most AI planning research, heuristic adequacy (efficiency) has been one of the primary goals in the design of SIPE–2. Techniques have been developed for efficiently implementing each of the features mentioned above.

Examples in this paper are drawn primarily from the domain of military operations planning which was implemented as part of the DRPI [17]. SIPE–2 is the core reasoning

engine in SRI's SOCAP system (System for Operations Crisis Action Planning) [17], which was used for the second Integrated Feasibility Demonstration of DRPI. This system successfully generated employment plans for dealing with specific enemy courses of action, and expanded deployment plans for getting the relevant combat forces, supporting forces, and their equipment and supplies to their destinations in time for the successful completion of their mission. Input to the system includes threat assessments, terrain analysis, apportioned forces, transport capabilities, planning goals, key assumptions, and operational constraints.

A brief description of how SIPE–2 represents operators will help to explain the central ideas of the ACT formalism and the translation process from SIPE–2 to ACT and back. Operators represent the actions, at different levels of abstraction, that the system may perform in the given domain. The primary representational task of an operator is to describe how the world changes after the action it represents is executed. SIPE–2 makes the assumption that the world stays the same except for the effects explicitly listed with each action in its representation of the plan.

Many of the effects explicitly listed in the plan are not explicitly listed in the operators from which the plan was produced, since they are deduced by the system during generation of the plan from the deductive causal theory of the domain. Operators must explicitly list only effects that are required to trigger all the necessary deduced effects. Since the causal theory will deduce different effects depending on the situaton, the operators can be applied in any situation. Without such an ability, a huge number of operators may be needed since there must be a different operator (with different effects) for every different situation in which an action might be performed.

In addition to effects, operators contain information about the objects that participate in the actions, the constraints that must be placed on them, the goals that the actions are attempting to achieve, the way actions in this operator relate to more or less abstract descriptions of the same action, and the conditions necessary before the actions can be performed (the action's preconditions). These features will be presented by discussing the SIPE–2 operator in Figure 1 for deploying an air force in the military domain. The ACT translation of this operator will be given in Section 4.

Operator: Deploy-airforce
Arguments: airforce1,airfield2,end-time1,
location1,airfield1,cargobyair1,cargobysea1,
seaport1,seaport2,sea-loc1,air-loc1;
Purpose: (deployed airforce1 airfield2 end-time1)
Precondition: (located airforce1 location1)
(near airfield1 location1) (near seaport1 location1)
(partition-force airforce1 cargobyair1 cargobysea1)
(transit-approval airfield2)(transit-approval seaport2)
(near seaport2 airfield2)
(route-aloc airfield1 airfield2 air-loc1)
(route-sloc seaport1 seaport2 sea-loc1)
Plot:
Process
Action: mobilize
Arguments: airforce1,location1;
Effects: (mobilized airforce1 location1)
Parallel
Branch 1:
Goal: (located cargobyair1 airfield1)
Goal: (located cargobyair1 airfield2)
Branch 2:
Goal: (located cargobysea1 seaport1)
Goal: (located cargobysea1 seaport2)
Goal: (located cargobysea1 airfield2)
End Parallel
Process
Action: join-aggregate
Arguments: airforce1,airfield2,cargobyair1,cargobysea1;
Effects: (located airforce1 airfield2)
(not (located cargobyair1 airfield2))
(not (located cargobysea1 airfield2))
End Plot End Operator

Figure 1: Sipe Operator for Deploying an Air Force

The *purpose* of an operator determines which goals the operator can solve — in this case, to deploy an air force to a particular air field by a certain time. The *precondition* must be true in the world state before the operator can be applied. The precondition in Figure 1 requires the initial position of the air force to be known, and finds intermediate seaports and airports that are on known routes to the destination and which have transit approval. The *arguments* of an operator are templates for creating planning variables and adding constraints to them. In Figure 1, airforce1 and airfield1, for example, are variables that are constrained (by virtue of their names) to be in the classes airforce and airfield respectively. The operator's preconditions and purpose are both encoded as first-order predicates on the arguments of the operator, which can be variables or objects in the domain.

Applying an operator involves interpreting its *plot* as a subplan for achieving its purpose. The plot of an operator provides a partially ordered sequence of actions and goals for performing the higher-level action represented by the operator. When expanding a plan to a lower level of detail, the planner uses the plot as a template for generating nodes to insert in the plan. The plot may be at the same level of abstraction as the purpose of the operator (e.g., in the standard blocks world the level of description never changes), or it may use a lower level of abstraction. In Figure 1, the plot consists of mobilizing the air force, then getting the subparts of the air force to travel in parallel by air and by sea via different locations to the destination, and finally aggregating the subparts together when they have all reached the destination.

The plot of an operator can be described in terms of *goal* nodes, which require a certain predicate to be achieved, *choiceprocess* nodes, which require that one of a given set of operators be applied to solve a certain predicate, and *process* nodes, which require a specific operator or primitive action to be applied. In the example operator, the plot uses process nodes for mobilizing and aggregating the air force, and uses goal nodes to achieve changes in location of the subparts.

Many previous domain-independent planners required add and delete lists to be provided in operators. In SIPE-2, this is not necessary because the deductive causal theory deduces most of the effects of the nodes in a plan. In the example operator, all

Causal-Rule: Remove-located
Arguments: unit1, location2, location1 is not location2
Trigger: (located unit1 location2)
Precondition: (located unit1 location1)
Effects: (not (located unit1 location1))

Figure 2: Sipe Deductive Rule for Removing Located Facts

the effects of moving a subpart to a new location are deduced from the *located* goal predicate. In particular, the deduced effects will include the fact that the unit is not at its previous location and its location at other abstraction levels has changed.

The deductive rules for specifying the causal theory also use the above operator syntax. Instead of having a plot to direct plan expansion, they have an *effects* slot that specifies the deduced predicates to be added to the effects of an action. The rule for deducing that a unit is not at its previous location is shown in Figure 2. Deductive operators allow expression of domain constraints within a world state, as well as permit access to the previous world state. Rules that allow the former are called *state rules*, while rules that allow the latter (such as the rule in Figure 2) are called *causal rules.* By accessing both the current and previous world states, the system can react to *changes* between two states, thus permitting effects concerning what happened during an action to be deduced even though these effects might not be implied by the final world state.

To access both the previous and current world states, causal rules have both a *precondition* and a *condition.* When deducing the effects of an action, the condition is matched in the world state after the action, while the precondition is matched in the world state before the action. Thus, state rules may have a condition but not a precondition, while causal rules may have both. Two other slots are needed on a deductive operator, *trigger* and *effects.* A deductive rule is applied whenever an action being inserted in a plan has an effect that matches the predicate given as the rule's trigger. The addition of deduced effects to an action also triggers deductive rules. If the precondition and condition of a triggered rule are both true, then the effects of the rule are added as deduced effects of the action.

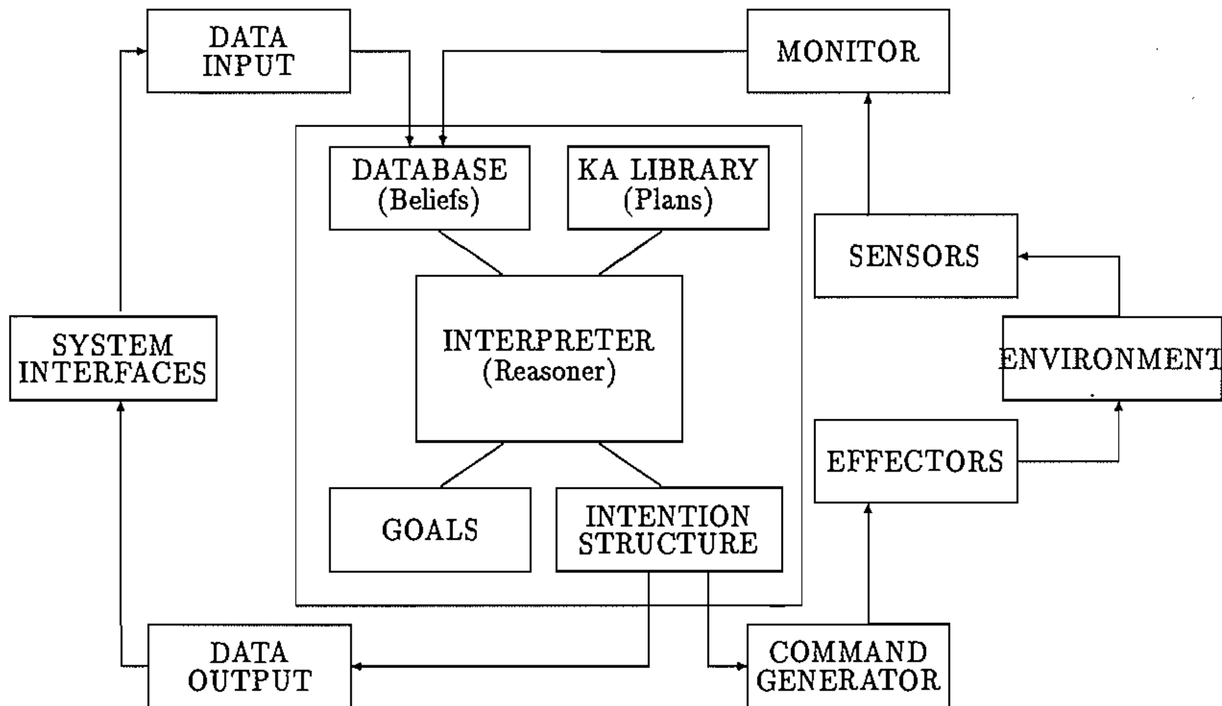

Figure 3: Structure of the Procedural Reasoning System

2.2 PRS

PRS [5] is a reactive system for reasoning about and performing complex tasks in dynamic environments. This powerful and theoretically sound scheme for representing and reasoning about actions and procedures is shown in Figure 3.

PRS consists of (1) a *database* containing current *beliefs* or facts about the world; (2) a set of current *goals* to be realized; (3) a set of *plans* (called Knowledge Areas or KAs) describing how certain sequences of actions and tests may be performed to achieve given goals or to react to particular situations; and (4) an *intention structure* containing all KAs that have been chosen for [eventual] execution. An *interpreter* (or *inference mechanism*) manipulates these components, selecting appropriate plans based on the system's beliefs and goals, placing those selected on the intention structure, and executing them.

The system interacts with its environment, including other systems, through its

database (which acquires new beliefs in response to changes in the environment) and through the actions that it performs as it carries out its intentions. The PRS architecture provides a framework to define reactive systems. While executing plans, PRS constantly monitors incoming database changes. This provides rapid respond to events in the world since the inference mechanism used guarantees that any new fact in the database is noticed in a bounded time.

A brief description of how PRS represents KAs will help explain the central ideas of the ACT formalism and the translation process from ACT to PRS. Knowledge about how to accomplish given goals or react to certain situations is represented in PRS by declarative procedure specifications called *Knowledge Areas* (KAs). Each KA consists of a *body*, which describes the steps of the procedure, and an *invocation condition*, which specifies under what situations the KA is useful. Together, the invocation condition and body of a KA express a declarative fact about the results and utility of performing certain sequences of actions under certain conditions [6]. The body and invocation condition will be presented by discussing the KA in Figure 4 for deploying an air force in the military domain. The ACT translation of this KA will be given in Section 4; the SIPE–2 representation of this knowledge was shown in Figure 1.

The invocation condition has two components: an *invocation* part and a *context* part. Both must be satisfied for the KA to be invoked. The invocation is a logical expression describing the *events* that must occur for the KA to be executed. Usually, these consist of some *change* in system goals (in which case, the KA is invoked in a goal-directed fashion) or system beliefs (resulting in data-directed or reactive invocation), and may involve both. In Figure 4, the invocation specifies that this KA will be considered when PRS acquires the goal to achieve the deployment of an air force to an air field.

The context of a KA is a logical expression specifying those conditions that must be true of the current state for the KA to be executed. The context in Figure 4 requires the initial position of the air force to be known, and finds intermediate seaports and airports that are on known routes to the destination and which have transit approval. Matching the context will bind the variables in it to specific objects.

DEPLOY-AIRFORCE

```
INVOCATION:
(*GOAL (! (DEPLOYED $AIR $AIRFIELD2 $END-TIME)))

CONTEXT:
(AND (*FACT (LOCATED $AIR $LOCATION))
     (*FACT (NEAR $AIRFIELD1 $LOCATION))
     (*FACT (NEAR $SEAPORT1 $LOCATION))
     (*FACT (PARTITION-FORCE $AIR
                             $CARGOBYAIR
                             $CARGOBYSEA))
     (*FACT (TRANSIT-APPROVAL $AIRFIELD2))
     (*FACT (TRANSIT-APPROVAL $SEAPORT2))
     (*FACT (NEAR $SEAPORT2 $AIRFIELD2))
     (*FACT (ROUTE-ALOC $AIRFIELD1
                        $AIRFIELD2
                        $AIR-LOC))
     (*FACT (~ (EQUAL $AIRFIELD1 $AIRFIELD2)))
     (*FACT (ROUTE-SLOC $SEAPORT1 $SEAPORT2 $SEA-LOC))
     (*FACT (~ (EQUAL $SEAPORT1 $SEAPORT2)))
     (*FACT (TYPE AIRFIELD $AIRFIELD1))
     (*FACT (TYPE AIRFIELD $AIRFIELD2))
     (*FACT (TYPE CARGO-BY-AIR $CARGOBYAIR))
     (*FACT (TYPE CARGO-BY-SEA $CARGOBYSEA))
     (*FACT (TYPE SEAPORT $SEAPORT1))
     (*FACT (TYPE SEAPORT $SEAPORT2)))

GOAL ACHIEVER?:
T

EFFECTS:
NIL

PROPERTIES:
((AUTHORING-SYSTEM SIPE-2) (CLASS OPERATOR))

DOCUMENTATION:
""
```

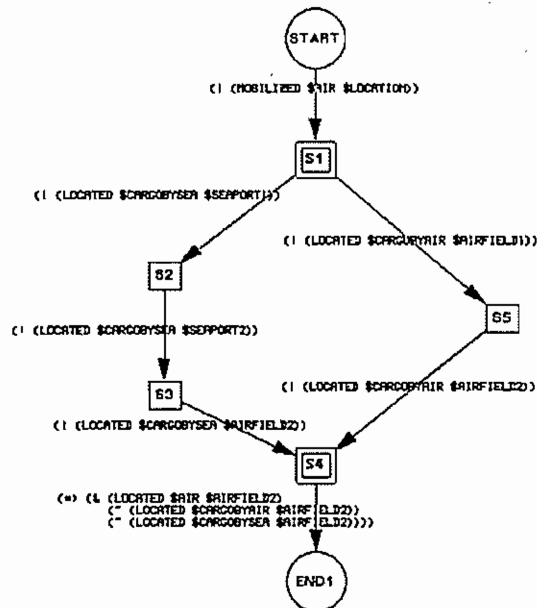

Figure 4: Portion of a KA for Deploying an Air Force

The KA body describes what to do if the KA is chosen for execution. The body is represented as a graphic network in which execution begins at the `start` node in the network, and proceeds by following arcs through the network. Execution completes if an end node (with no outgoing arcs) is reached. Each arc of the network is labeled with a goal to be achieved by the system. To traverse an arc, the system must either (1) determine from the database that the goal has already been achieved or (2) find, and successfully execute, a KA that achieves the goal labeling that arc. If the system fails to achieve the goals on all of the arcs emanating from a double-border node, or at least one of the goals on arcs emanating from a single-border node, then the KA as a whole will fail.

In Figure 4, the first arc of the body is labeled with the goal of achieving the mobilization of the air force. Following arcs are labeled with goals to move the subparts of the air force, and the last arc in the body is labeled with a goal to post certain facts in the database about the air force and its subparts.

There are some properties of KAs that are crucial for the correct functioning of the system. The *goal-achiever* slot, for example, is a Boolean value telling whether or

Leak Isolation

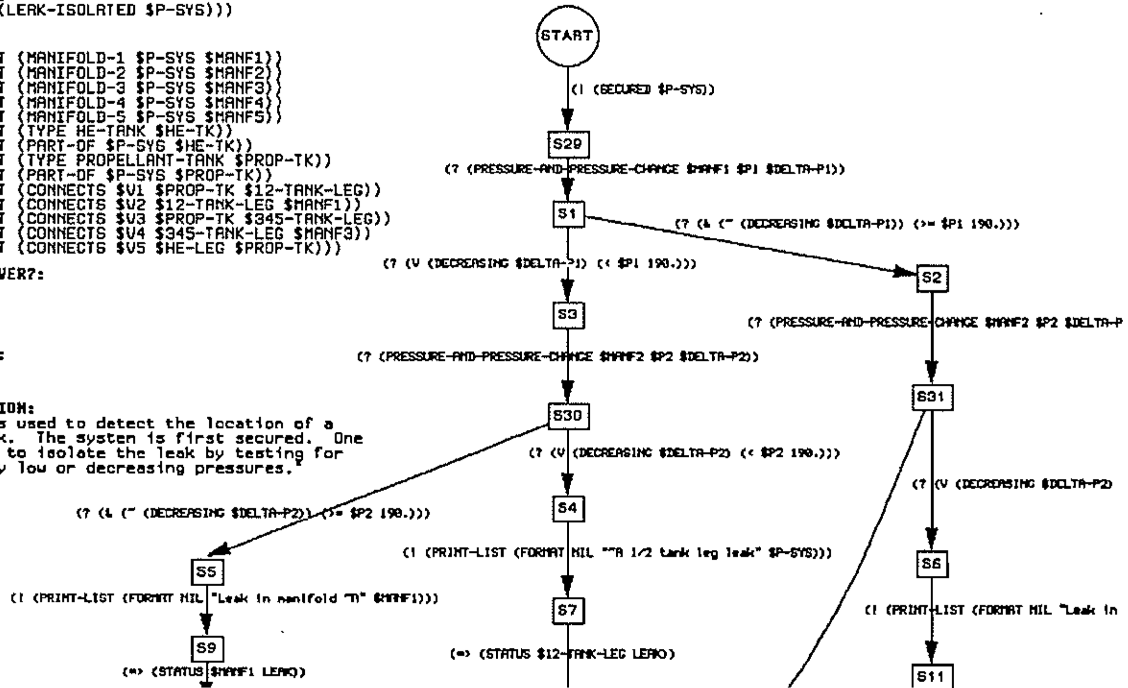

Figure 5: Portion of a KA for Leak Isolation

not successful execution of the KA (i.e., reaching an end-node) achieves the goal that triggered the KA. In Figure 4, this slot is T, since the KA does deploy the air force. Similarly, KA properties that are not essential to the functioning of the interpreter but which may be required by application-specific metalevel KAs can be placed in a special *property* slot of a KA. Such properties, for example, might include information on the likelihood of success of the KA or its average execution time.

While the KA for deploying an air force is useful for comparisons to SIPE-2 and will be shown later as an ACT, it was written as a high-level planning operator and is not the type of knowledge that would typically be encoded for execution by PRS. A more typical PRS KA would be the KA in Figure 5. It describes a procedure to isolate a leak in a reaction control system (RCS) of the space shuttle. The invocation of this KA triggers when the system acquires the goal to isolate a leak in an RCS (`$p-sys`), provided the various type and structural facts given in the context are true.

The KA body in Figure 5 is typical of many PRS KAs; there are several condi-

tional branches, and the arcs are labeled with low-level conditions that are close to the primitive execution level. In this body, whenever more than one arc emanates from a given node, only one of the arcs emanating from that node needs to be traversed (and PRS can choose any one). For example, to traverse the arc emanating from the start node requires either that the system be already secured or that some KA for securing the RCS be found and successfully executed. Similarly, to transit the next arc requires that some KA be found for determining the pressure change `$delta-p1` in the manifold `$manf1`. If the system fails to traverse an arc emanating from some node, other arcs emanating from that node are tried. Since only one arc emanates from the start node in Figure 5, if all attempts to secure the RCS fail, this procedure for isolating a leak in the system will also fail. The full KA for this procedure consists of over 45 nodes.

In addition to KAs that express knowledge directly about the world, the set of KAs also includes *metalevel* KAs. Metalevel KAs encode knowledge about the manipulation of the beliefs, goals, and intentions of PRS itself. For example, typical metalevel KAs encode various methods for choosing among multiple applicable KAs, determining how to achieve a conjunction or disjunction of goals, and computing the amount of additional reasoning that can be undertaken, given the reactive constraints of the problem domain.

3 Developing a Common Representation

In order to combine the capabilities of plan generation and reactive control systems, we have designed a uniform representation for domain knowledge and plans. Our uniform representation refers to knowledge provided by the user, e.g., plan fragments and standard operating procedures (SOPs), as *ACTs*. ACTs are used by both the planning and execution systems. This allows both systems to cooperate on the same problem, and allows sofware tools for graphically editing and creating ACTs to enhance both systems simultaneously and uniformly.

In our implementation, ACTs represent operators in SIPE–2, and KAs in PRS, and are the recommended method for encoding knowledge. We have implemented

translators from the ACT formalism to the internal representations of both SIPE–2 and PRS, as described in Section 4.6. The plans generated by the planner are translated to instantiated ACTs. The execution system uses procedures encoded as ACTs to respond to minor problems during execution, and may also reinvoke the planner when appropriate. We have implemented an ACT-Editor, described in Section 4.5, that supports graphical input and modification of ACTs.

Obviously, the new representation must be capable of supporting the types of reasoning done by both planning systems and reactive control systems. This is no small task, since each system stresses a different type of knowledge and reasoning process. For example, PRS frequently uses conditionals and loops in its procedures, and can monitor the world to determine what is true. Before extensions were made to support ACTs, PRS did not support parallel execution of separate actions. SIPE–2, on the other hand, stresses parallel actions and uses the planner's reasoning to predict what is true (since sensors cannot sense future states). The ACT representation supports all these types of reasoning, and PRS and SIPE–2 have been extended to support certain aspects of the ACT formalism. These extensions are described in detail elsewhere [15].

A central issue in designing the ACT formalism is the difference between planning and execution. While executing a plan and generating a plan require much of the same knowledge, there are still some important differences between the two activities. The planner is attempting to look ahead to the future consequences of particular courses of action, while the execution system is sensing the world and reacting to incoming information. This difference manifests itself in several ways, including system response to violations of requirements, the triggering of ACTs, and the representation of knowledge about the world. These manifestations result in different interpretations of the ACT formalism during planning and execution. How these differing interpretations are implemented in SIPE–2 and PRS is described in Section 4.3.

One important difference is that the planner can permit requirements to be violated, because its plan-critic algorithms can then be used to patch the plan to prevent the violations from occurring. The execution system, however, must fail as soon as a requirement is violated since the world is actually in a state that violates the re-

quirement. It is generally difficult to determine the most appropriate response to an execution failure; thus, the execution system will need knowledge, unnecessary to the planner, about how to respond to failures.

In order to be reactive, the execution system must have ACTs that are both goal-driven and event-driven. In addition to trying to achieve system goals, it must respond appropriately to events that take place in the world. The planner, in contrast, need respond only to its own goals since it is reasoning about hypothetical future states. Ideally, the planner would modify partially generated plans to react to changes in the world during the planning process, although current AI planning systems have only minimal capabilities in this area. The requirement for PRS to be event-driven results in distinguishing between goal predicates and fact predicates in the ACT formalism.

Representation of the world state is generally different in planning and execution systems. For example, to be efficient enough to be reactive, PRS maintains only one world model that reflects the current state of the world. By contrast, SIPE–2 must be able to reason about the world in many different future states, which necessarily adds overhead to the representation and reasoning. Most planning systems, including SIPE–2, encode a new world state by recording changes since a prior world state. Thus, the planner requires knowledge, unnecessary to the execution system, relating to modeling how the world state changes as actions are taken. Another aspect of this difference is that the execution system can use the world as an information source and use sensors to query some property in the world. Because the planner reasons about world states that may exist in the future, it must model every proposition in which it is interested, again requiring knowledge that may be unnecessary to the execution system.

A final issue of practical importance is that the ACT representation must be clear enough to enable users to understand plans and SOPs represented in it, and to allow knowledge engineers to encode domain knowledge as ACTs. Thus, it was necessary to balance the power of the representation with its perspicuity. Section 4 describes the decisions we made in this regard.

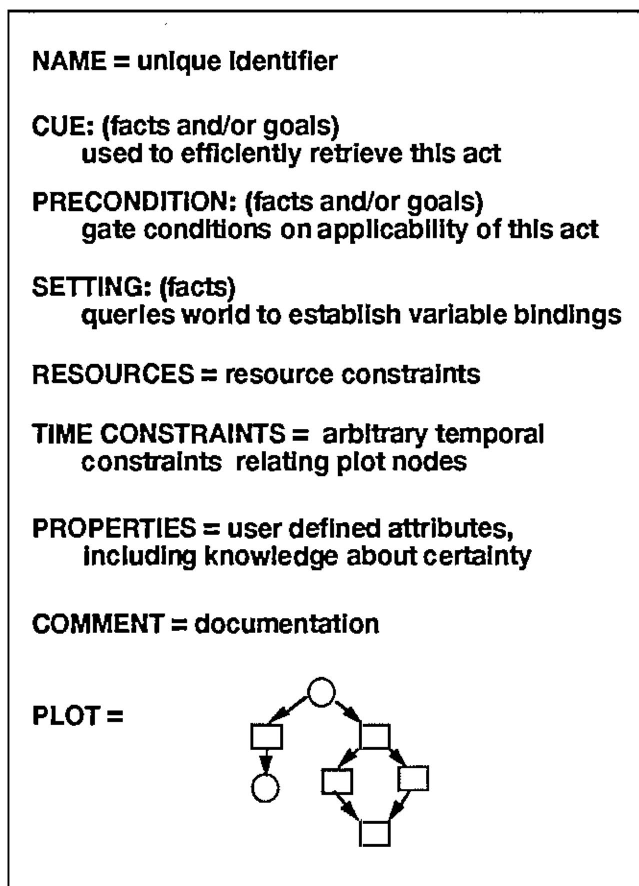

Figure 6: ACT Slots

4 The ACT Formalism

Each ACT represents a plan or a piece of domain knowledge about an action or set of actions that can be taken. Each ACT consists of a number of predefined slots and a set of nodes representing actions. Each slot and node uses a combination of a set of predefined metapredicates and logical formulae to represent its knowledge. The predefined slots occur in every ACT, and are shown in Figure 6, which describes briefly the idea behind each. The slots *Cue*, *Precondition*, and *Setting* have as values logical formulae describing goals and/or facts, while the other slots represent information in other forms.

Each ACT also has a *Plot* which is a directed graph of nodes representing actions

and arcs representing a partial temporal order for execution. The nodes in such a graph are referred to as *plot nodes.* A plot has one and only one start node, but may have multiple terminal nodes. A terminal node is a node with no outgoing arcs. Loops can be specified by connecting the outgoing arc of one node to an ancestor node in the graph. The actions represented by each plot node are specified by metapredicates and logical formulae described below.

While the metapredicates that may appear on the plot nodes and slots are described in detail in Section 4.3, a brief summary of the metapredicates, listed in Figure 7, is necessary before explaining the slots. The *Test* metapredicate is used to determine whether a formula is true in the database or world without taking actions to achieve it. The *Use-Resource* metapredicate makes a declaration of resources that will be used by the ACT. Three metapredicates can be thought of as specifying actions: *Achieve*, *Achieve-By*, and *Wait-Until.* Achieve directs the system to accomplish a goal by any means possible, Achieve-by is similar but directs the system on how to accomplish the goal, and Wait-Until directs the system to wait until some other process, agent, or action has achieved a particular condition. The *Require-Until* metapredicate sets up conditions that must be maintained over a specified interval, and the *Conclude* metapredicate specifies any effects that are accomplished by an action (in addition to the predicates mentioned in the action-metapredicate specifying the action).

4.1 ACT Slots

The Cue, Precondition, and Setting slots, as well as plot nodes, are filled in with logical formulae. Each slot can have an entry for zero or more metapredicates. The formulae used as the values for each metapredicate are built from predicates specified in first-order logic, connectives, and the names of ACTs. The metapredicates recognized in the ACT formalism are shown in Figure 7, which hints at the syntax accepted by each where P stands for a formula composed of first-order predicates and connectives. The syntax for metapredicates is described fully in Appendix A. The predicates will have three truth values: true, false, and unknown. Predicate names can be declared as `open` or `closed`, to instruct the system as to whether the closed world assumption applies.

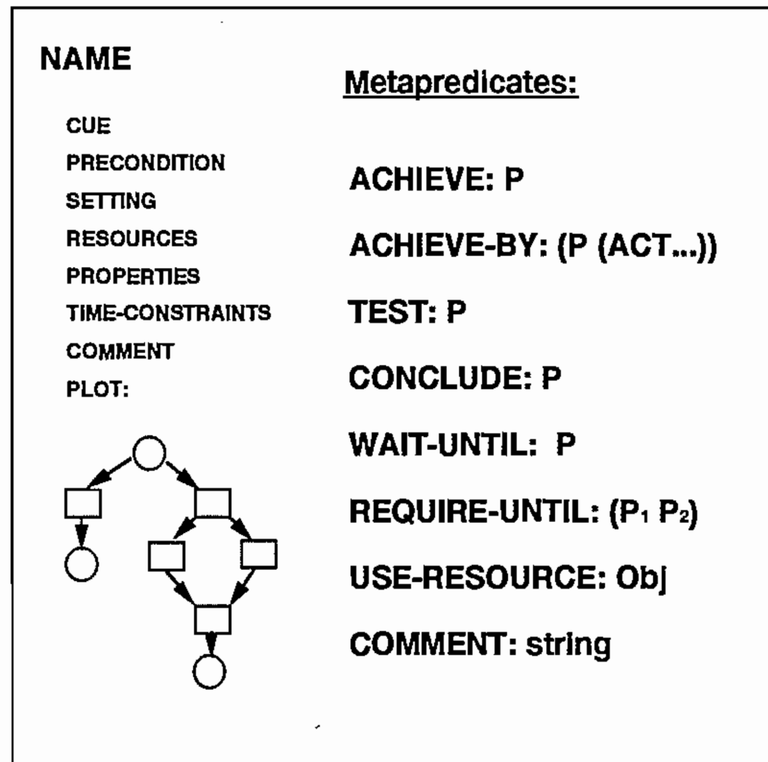

Figure 7: ACT Metapredicates

In our implementation, PRS currently restricts the use of unknown truth values to open predicates, while SIPE–2 implements it for all predicates. If necessary, PRS could be extended to handle unknown truth values in all cases.

Static information represented as binary predicates can be stored in a sort hierarchy, and the variables in our formalism are typed, with the types corresponding to classes in the sort hierarchy.[2] The sort hierarchy should be used whenever possible for efficiency [13].

All plot nodes and all slots except *Name* allow arbitrary user properties (which can be used by users and their programs but are ignored by the system) and the *Comment* metapredicate. Table 1 lists the metapredicates the system recognizes for each slot. The slots with no metapredicates require further explanation. Name is a symbol that

[2]SIPE–2 has such a sort hierarchy implemented and this is currently used in both PRS and SIPE–2. This may someday be replaced by a more expressive frame system, e.g., LOOM [12].

Slot	Metapredicates
Cue	Achieve, Test
Preconditions	Achieve, Test
Setting	Test
Resources	Use-resource
Time-constraints	none
Properties	none
Comment	none
Name	none

Table 1: Metapredicates Allowed on Slots

names the ACT, and Comment provides documentation (generally, just a string). Plot is a set of plot nodes, and all metapredicates are valid for a plot node.

Properties is a user-defined property list (e.g., author, priority). While Properties does not necessarily expect any properties, some are recognized by the system and are recommended. Recommended properties are *Authoring-System* and *Author.*

In our ACT implementation, SIPE–2 recognizes the properties *Class* and *Variables.* Class denotes the type of operator to which this ACT should be converted. Allowable values for Class are: state-rule, causal-rule, init-operator, operator, and both.operator. SIPE–2 will also check the Variables property for declarations about variables. The value of variables should be a list of quantifier-pairs. Each quantifier-pair whose first element is `existential` or `universal` will be processed, and its second element should be a list of variable names.

The Cue is used to index and retrieve an ACT for possible execution. Cue specifications should be short so that the system can rapidly identify a subset of potentially executable ACTs. This is important in PRS to maintain real-time response. Complicated conditions should be put in the Precondition or Setting. Restrictions on the use of metapredicates in the Cue are given in Section 4.3. Generally, a Test metapredicate

in the Cue indicates an event-driven ACT that responds to some new fact becoming true, while an Achieve metapredicate indicates a goal-driven ACT that will be used to solve system goals. The Cue corresponds to the "purpose" in a SIPE–2 operator, the "trigger" in a deductive operator, and the "invocation condition" in a PRS Knowledge Area.

The Precondition specifies the gate conditions (in addition to the Cue) on applicability of the ACT. The ACT is not invoked unless its Cue and Precondition are true. The Setting specifies conditions that are expected to be true in the world, and is used to instantiate variables; O-Plan refers to such conditions as "query conditions" [3]. Although both are gating conditions, the Setting is separated from the Precondition to make the ACT more easily understood. Generally, the Precondition and Setting will consist of a Test metapredicate (see Section 4.3).

The *Time-Constraints* slot is used to specify time constraints between plot nodes that cannot be represented by ordering arcs. Many dependencies between different military actions require such constraints. For instance, cargo offload teams should arrive at the same time as the first air transport arrives at the airport. The ACT syntax for time-constraints allows specification of any of the 13 Allen relations [1].

The syntax does not use the metapredicates; instead, there are two types of constraints that can be represented: time windows on nodes and inter-node constraints. A time window for a node specifies its earliest and latest allowable start times, earliest and latest finish times, and minimum and maximum durations. An inter-node constraint represents a constraint between the endpoints of a pair of events or plan nodes. It is represented as an 8-tuple composed of the minimum and maximum distance between the start of the first event and the start of the second event, and likewise the minimum and maximum start-finish, finish-start, and finish-finish distances. The syntax for specifying time constraints is documented in Appendix A.

In our ACT implementation, the temporal constraints in the Time-Constraints slot are translated for processing by General Electric's Tachyon system. Tachyon [2] was chosen as the external software module for propagating the temporal constraints because it had the required capabilities, was readily available, and was known to be

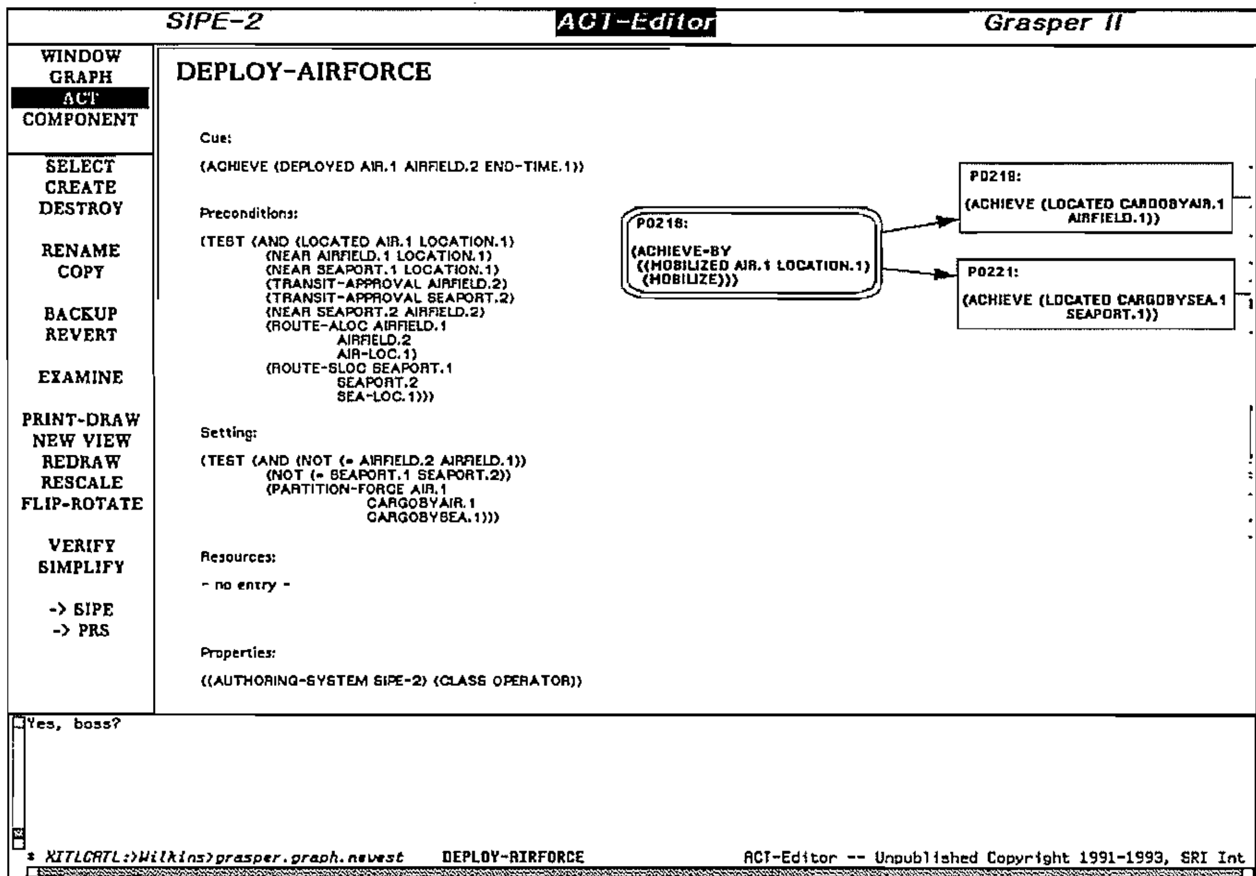

Figure 8: SOCAP Deploy Airforce Operator as an ACT

efficient. Tachyon is an efficient implementation of a constraint-based model for representing and maintaining qualitative and quantitative temporal information. SIPE-2 calls Tachyon, which propagates these constraints and combines them with the "commonsense" constraints that it represents internally, returning an updated set of time windows on the plan nodes which are inserted into the plan. The temporal constraints are currently ignored by PRS, although PRS could be extended to support them.

Figure 8 is an example ACT (not all of it fits on the screen) taken from the screen of the ACT-Editor, a system we implemented for viewing and editing ACTs that is briefly described in Section 4.5. It is an ACT that was originally written as a SIPE-2 operator in the military operations domain and was translated by our ACT-to-Sipe translator. The slots are displayed on the left side of the screen and the first few plot

nodes are on the right. This ACT is a planning action designed to deploy an air force to a particular location. The Cue is used to invoke the ACT when the system has the goal of achieving such a deployment. The Preconditions enforce various constraints on the intermediate locations to be used in the deployment. The Setting essentially looks up the cargo that must go by air and sea for this deployment. The plot is described in Section 4.2.

ACTs slots support the representational requirements for the applicability conditions of both SIPE–2 operators and PRS KAs. For example, when translating an operator to an ACT, the purpose becomes a Cue composed of goal formulas, the trigger of a deductive operator becomes a Cue composed of fact formulas, the precondition becomes fact formulas in both the Precondition and the Setting, and constraints on variables become fact formulas in the Setting. The SIPE–2 "instantiate" slot corresponds to an entry by that name on the properties of an ACT. SIPE–2 could be extended to support goal formulas in the preconditions and settings to provide additional power for interacting with parallel goals. PRS KAs can be translated to ACTs as follows: the "goal achiever" can be ignored, the "invocation condition" becomes both fact and goal formulas in the Cue, the "context" becomes both fact and goal formulas in both the Precondition and the Setting, and the "effects" will be encoded in the plot.

4.2 Plots

The plot of an ACT specifies an action network that is to be executed or used for plan elaboration. Each node in the network has its own set of metapredicates for specifying its action, as described below. The arcs coming into a node are said to be *disjunctive* when the node can be executed if only one of its incoming arcs is activated. Incoming arcs are *conjunctive* when all of them must be active before the node can be executed. Similarly, outgoing arcs are disjunctive if the system needs to execute only one of them, and are conjunctive if all of them must be executed.

We investigated several alternative representations for plots that combine the conditionals and loops typically used in execution systems with the parallel actions typical

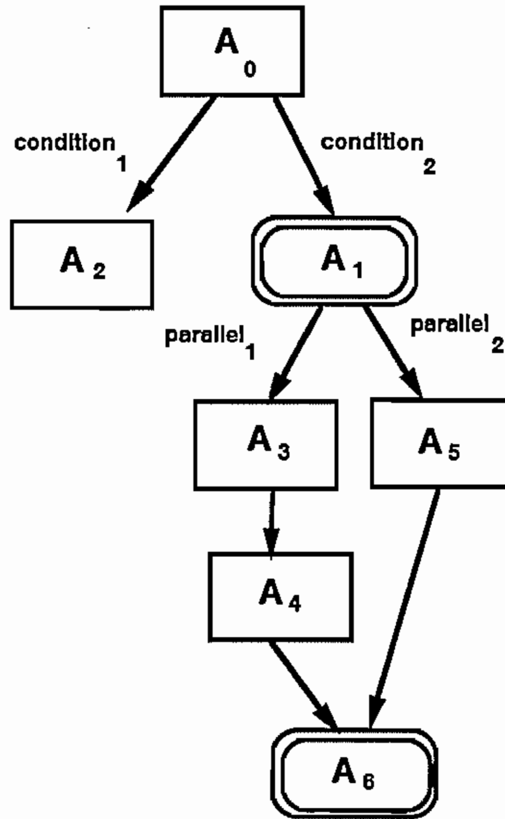

Figure 9: ACT with Graphical Display of Plot

of planning systems. The problem with most alternatives is that the complexity of the representations makes them hard to understand when nodes have various combinations of disjunctive and conjunctive arcs either incoming or outgoing. Our solution, which is reasonably perspicuous and still general, is to define two types of nodes, *conditional* (or disjunctive) and *parallel* (or conjunctive). All arcs coming into and going out of a conditional node (drawn as a single-border rectangle) are disjunctive, while all arcs coming into and going out of a parallel node (drawn as a double-border oval) are conjunctive. This is depicted in Figure 9.

Two consequences of this handling of multiple edges are important: (1) If a node has zero or one incoming edges and zero or one outgoing edges, it is irrelevant whether it is a conditional or a parallel node — they are equivalent. (2) If one action is to be activated by only one of its incoming edges and must activate all of its outgoing

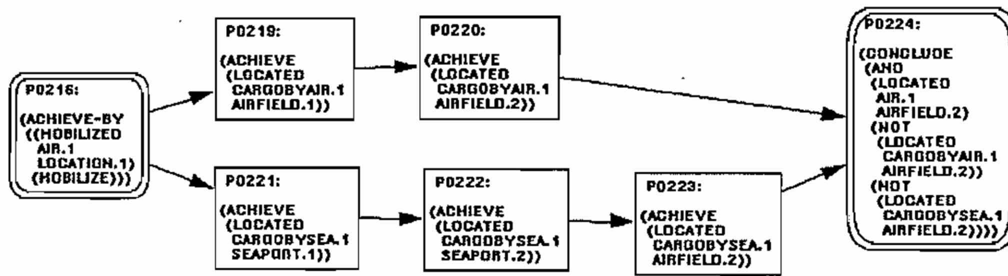

Figure 10: Plot of Deploy Airforce ACT

edges, then it must be represented by a conditional node that collects the incoming edges followed by a parallel node that collects the outgoing edges. The metapredicates may occur on either of these two nodes, while the other node need not have any metapredicates.

Figure 10 shows the plot of the Deploy Airforce ACT previously discussed. The first plot node specifies that the air force is to be mobilized. It is a parallel node, so its successors will be invoked in either order or at the same time. The successors specify that the air cargo will be moved to the final destination in parallel with moving the sea cargo to two intermediate ports and finally to the final destination. The final node joins the parallel actions and posts conclusions about the locations of the air force and its subparts.

4.3 Metapredicates

While executing a plan and generating a plan require much of the same knowledge, there are still some important differences between the two activities, as discussed previously. These differences result in different interpretations of the ACT metapredicates in the planning and execution systems. Our implementation of ACT metapredicates in SIPE-2 and PRS, as described below, specifies how the metapredicates should be interpreted given the constraints imposed by the planning and execution environments. However, it is possible that small advantageous changes in this interpretation could be made based on special properties or features of the particular planning and exe-

Metapredicate group	Metapredicates
Environment	Test, Use-Resource
Action	Achieve, Achieve-By, Wait-Until
Effects	Require-Until, Conclude

Table 2: Metapredicate Grouping for Execution in Plot

cution systems being used. For example, planning systems with powerful temporal reasoning capabilities may handle the Time-constraints slot differently, and execution systems with architectures different from PRS may handle failures of Require-Until metapredicates differently.

Figure 7 lists the metapredicates that may appear on each plot node. For purposes of ordering their execution, these metapredicates are partitioned into three groups, as shown in Table 2. The environment-metapredicates, Test and Use-Resource, are executed first by the execution system. During planning, Use-Resource makes a declaration that will be used by the planner (in SIPE–2, the plan critic algorithms satisfy resource requirements), and a Test is generally ignored while expanding a plan with an ACT except in Cues and Preconditions. The action-metapredicates, Achieve, Achieve-By, and Wait-Until, are executed next, and there should be only one of these on a node. The effects-metapredicates, Require-Until and Conclude, are executed last by the execution system. During planning, Require-Until sets up a protection interval that the planner must maintain, and Conclude specifies any effects that are additional to the predicates mentioned in the action-metapredicate of the node.

The meanings of the ACT metapredicates are described below. As described above, the ACT metapredicates sometimes have different semantics in the planning and execution systems. When the term "the system" is used below, it refers to the behavior of both the planning and execution systems. The syntax of ACT metapredicates is fully described in Appendix A, which also mentions the parts of the syntax not accepted by SIPE–2 and/or PRS.

ACHIEVE: This metapredicate is primarily used on plot nodes, where it specifies a set of goals the ACT would like to achieve at that point in the partial plan. In the Cue, an Achieve metapredicate indicates a goal-driven ACT and tells the system what goals the ACT should be used to achieve. Thus in planning, Achieve in the Cue is used for expanding a plan and is not used for deduction. Similarly, a Test in the Cue is used during deduction and is ignored during plan expansion. (The *class* property on the Properties slot can be used to direct the planner's use of an ACT for deduction or plan expansion.) The execution system allows the Cue to have both a Test and an Achieve. While rarely used, such a construct would trigger an event-driven ACT only when the system had posted the given goals to achieve.

In the Precondition and Setting slots, the planner ignores an Achieve since it achieves goals in the plan, not during matching of these conditions. While not recommended, an Achieve can be used in the Precondition or Setting of an ACT executed by the execution system. However, in PRS, this will match only if PRS has posted a matching goal during the current execution cycle. This is not recommended and should be done only by users who understand PRS's main control loop.

TEST: This metapredicate specifies facts (formulae containing predicates) that can be queried in the database and/or, for the execution system, by sensing the world. When a Test is in the Precondition or Setting slots, it is used to determine if the ACT should be executed after it becomes relevant. A Test in a Cue is used to determine whether the ACT is eligible for execution, and indicates an event-driven ACT that responds to some new fact becoming true. During planning, this is used to trigger deductive operators to deduce context-dependent effects of an action (a Test in the Cue of a nondeductive operator will be ignored). During execution, if one or more of the specified facts in the Cue has just been posted in the database and the remaining facts are true in the database, then the ACT will be considered relevant but not yet ready for execution.

A Test in a plot node is also handled differently by the two systems. During execution, a Test in a plot node is used to determine whether the specified facts already exist in the database, and if not, it is used to determine which ACTs should be executed to determine if the specified facts are true in the world. During planning, a Test in a plot node can occur only after a conditional branching and is interpreted as a run-time test to determine which conditional plan to execute. Any other Test in a plot node is ignored.

CONCLUDE: This metapredicate appears only in plot nodes, and specifies the facts (predicates) to be concluded in addition to the predicates mentioned in the action-metapredicate of the node. Disjunctions are not allowed. During planning, Conclude is used to reason about a different possible future world, and during execution, Conclude formulas are posted to the database, possibly triggering the execution of event-driven ACTs.

WAIT-UNTIL: This metapredicate appears only in plot nodes, and specifies that execution of the ACT is to be suspended until the indicated event occurs. This is identical to PRS's wait-until construct. The planner implements this metapredicate by ordering the Wait-Until node after some action that achieves the required condition. In SIPE–2, this is accomplished by using the `external condition` construct.

ACHIEVE-BY: This metapredicate appears only in plot nodes, and specifies the goals to be achieved as well as a restricted set of ACTs, one of which must be used to achieve the goals. This provides a means for guiding the system on how to achieve a goal. In SIPE–2, this is translated to a `process` node for singleton ACTs and a `choiceprocess` node when more that one ACT is given.

REQUIRE-UNTIL: This metapredicate appears only in plot nodes, and specifies a condition to be maintained until another indicated condition for terminating this requirement occurs (i.e., a protection interval). The accepted syntax for Require-Until has two options. The general one is `(req-wff term-wff)`. Here, `req-wff` is a formula to be maintained until the termination condition `term-wff`

becomes satisfied. The shorter option is simply `term-wff`, where the `req-wff` is assumed to be the goal formula specified for either the Achieve or Achieve-By metapredicate in the same node. An error arises if no such goal can be identified. In SIPE–2, a Require-Until is specified by the `protect-until` construct, and the plan critics will modify any partial plan that violates a Require-Until so that the final plan will satisfy all Require-Untils.

Require-Until is more difficult to implement in execution systems, since it is not clear what to do upon failure. Our implementation in PRS attempts to handle failures as well as the constraints of execution and PRS's architecture allow. In PRS, Require-Untils are directly mapped onto a require-until operator that was recently added. A PRS require-until fails when either `req-wff` is unsatisfied at the point when `term-wff` becomes satisfied, or `req-wff` is unsatisfied and there is no means to repair it (see below). A PRS require-until succeeds when either: `req-wff` is satisfied when `term-wff` becomes satisfied, or the KA containing the require-until terminates without the goal having failed.

While one could imagine having required formulas that are to be protected beyond the scope of the ACT that posted them, this is problematical in PRS for two reasons: (1) the system architecture would have to be changed to allow ACTs to post goals that would remain for processing after the ACT posting the goal had succeeded, and (2) there is no reasonable action to take after a failure when the posting ACT no longer exists. Should the `req-wff` become unsatisfied before `term-wff` occurs, the Require-Until does not necessarily fail; rather, it is said to be violated. PRS will post a goal to repair the violation, and the user can provide domain-dependent ACTs to perform the repairs. If no ACTs respond to the require-until violation, then PRS will fail from the ACT containing the Require-Until.

COMMENT: This metapredicate can appear in any node or slot. It provides a means for associating some documentation with the node or slot and is usually a string.

USE-RESOURCE: This metapredicate in the Resources slot means each of its ar-

guments is a resource throughout the plot. Our interpretaton of Use-Resource is based on the resource reasoning abilities of SIPE–2 and PRS. This interpretation is an obvious place for extending the ACT formalism when more powerful resource reasoning capabilites are available. On a plot node, PRS ignores Use-Resource while SIPE–2 indicates the Use-Resource arguments will be resources only at that node (assuming they are not mentioned in the Resources slot). Currently, these are reusable resources (i.e., they are not consumed), and the system will prevent other simultaneous actions from using the same resource. SIPE–2 translates Use-Resource directly into its resource construct. PRS handles the Resources slot by requiring that the specified resources be available before the ACT can be executed. PRS actually translates the Resources slot into appropriate predicates that are appended to the Test metapredicate of the Setting slot. The resources are allocated when the ACT is intended for execution and are released when the ACT either succeeds or fails.

4.4 Using ACTs

To show how the various constructs in the ACT formalism are used to represent plan generation and execution knowledge, we will look at three example ACTs from the military domain. A description of this domain can be found elsewhere [17], but the ACTs shown are self-explanatory. By using ACTs, we have demonstrated SIPE–2 and PRS cooperating on the same problem for the first time. SIPE–2 generates large military operations plans while PRS responds to events, executes the plans produced, and performs lower-level actions that have not been planned.

First, we briefly describe how the interaction between SIPE–2 and PRS currently takes place in our implementation, although this interaction is still an area of ongoing research. While both the planning and execution systems can use ACTs at all levels of detail, it is necessary in realistic domains to fix a level of detail below which the planner will not plan. Planning to the lowest level of detail is often not desirable, and the combinatorics can be overwhelming. For example, it is not desirable to plan large military operations down to the minutest detail. Similarly, it is often not desirable

for the execution system to respond to certain high-level goals without a plan. For example, a reactive system should not start executing a Desert Storm-sized operation without first having a plan. ACTs at interim levels of detail may be advantageously used by both planning and execution systems.

In the military domain, we fixed a level of detail to which SIPE–2 will plan. The more detailed actions are left to PRS. This is accomplished by using the *class* property to specify which ACTs are to used by SIPE–2 and/or PRS. SIPE–2 produces complex plans at this level that are translated to one large instantiated ACT for execution by PRS. (We are developing techniques for splitting such an ACT into a set of smaller ACTs.) This ACT will be the only one that responds to the top-level goal (in the military domain), so that PRS will simply react to events until the planner generates a plan and the top-level goal is posted in PRS. This plan will be the principal guidance for the reactor. The reactor will implement appropriate lower-level behaviors while it is waiting for a plan, and may pose additional problems to the planner during execution.

The execution system will use ACTs to respond to minor problems during execution, and to initiate detailed actions. We are currently investigating the research issues involved in having PRS invoke SIPE–2 to replan. This replanning can be triggered either by PRS directly when PRS considers it appropriate as a response to failure, or by actions embedded in the original plan. One obvious alternative is to using ACTs for communicating replanning information from the reactor to the planner. For efficiency, we are also considering the solution of the reactor communicating individual goals and changing a common database so that the planner can immediately begin replanning without a translation process.

We will consider two ACTs that operate in this context. Figures 8 and 10 showed the ACT for deploying an air force that encodes the same knowledge as the SIPE–2 operator shown in Figure 1 and the PRS KA shown in Figure 4.

The Cue tells the system to use this ACT to achieve a goal of deploying an air force to a particular air field by a certain time. The Precondition and Setting must be true in the world state before the operator can be applied. The Precondition in Figure 8 requires the initial position of the air force to be known, and finds intermediate

LOOKOUT-RED

Cue:

(ACHIEVE (LOOKOUT UNIT.1 LAND-SECTOR.1))

Preconditions:

(TEST (CODE-RED LAND-SECTOR.1))

Setting:

(TEST (AND (VANTAGE-POINT SITE.1 LAND-SECTOR.1)
(ABOVE AIR-SECTOR.1 LAND-SECTOR.1)))

Resources:

(USE-RESOURCE TFIGHTER.1)

Properties:

((ARGUMENTS (UNIT.1 LAND-SECTOR.1))
(AUTHORING-SYSTEM ACT-EDITOR)
(CLASS NONPRIMITIVE-EXECUTION-ACTION))

Comment:

INSTALL A LOOKOUT USING AIR COVER

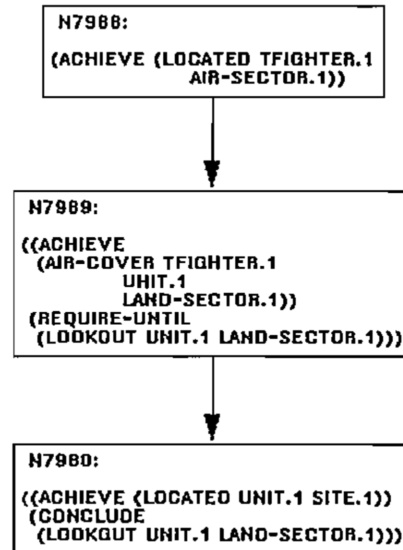

Figure 11: An ACT for Establishing a Lookout

seaports and airports that are on known routes to the destination and which have transit approval. The Setting constrains the two airports and seaports to be different and partitions the air force into two subparts (by matching a predicate in the database). Planning variables are constrained to be in a certain class based on their name; for example, seaport1 and airfield1 are variables that are constrained to be in the classes seaport and airfield respectively.

The Plot, shown in Figure 10, is used either to expand a plan by inserting the plot as a subplan or to begin execution of deployment. The Plot begins with a parallel node that uses an Achieve-By metapredicate to force the mobilization of the air force by using the Mobilize ACT. The nodes following this can then be executed in any order or simultaneously. These nodes begin two sequences of conditional nodes that us Achieve metapredicates to get the subparts of the air force to travel in parallel by air

and by sea via different locations to the destination. Finally, there is a parallel node, which joins these two parallel threads and aggregates the subparts together when they have all reached the destination.

Figure 11 shows the Lookout-Red ACT for establishing lookouts. In our current implementation of the military domain, the planner does not plan down to the level of establishing lookouts, so this ACT is only used by PRS during execution, although it could be used by SIPE-2 if the planning was extended to a lower level of detail. Lookout-Red is one alternative way to achieve a lookout, and the Precondition specifies it should only be used when the terrain is dangerous (represented here by the `code-red` predicate). The Setting finds the correct site for the lookout and the correct sector for the supporting air cover. There is a Use-Resource metapredicate in the Resources slot which requires that a `tfighter` resource be available before this ACT can be executed.

The plot of Lookout-Red first uses an Achieve metapredicate to get the fighters to the correct location. The second node in the plot achieves an air cover of a certain land sector and uses a Require-Until metapredicate to specify that this air cover must be maintained until the lookout has been achieved. Thus, if the air cover is terminated for some reason before the lookout is established, PRS will post a violation and use appropriate ACTs to respond to this situation. SIPE-2 would modify a plan being generated to avoid such a violation. Finally the plot uses an Achieve metapredicate to get the lookout unit to the correct location, and a Conclude metapredicate to note that the lookout has now been established (which terminates the require-until condition).

Another type of knowledge used by some plan generation and execution systems is knowledge for deducing the context-dependent effects of actions. Such a capability greatly enhances the expressive power of a planning system, and is a feature of SIPE-2. The ACT formalism supports this type of knowledge. For example, the Located-sector-up ACT in Figure 12 is used to deduce the new region in which a movable object is located after that object has just moved to a new sector (a region is at a higher level of abstraction and may contain several sectors). This ACT is used by both SIPE-2 and PRS to update the world model whenever an object is moved. The Cue of Located-sector-up specifies an event-driven ACT that responds to new facts about the location

LOCATED-SECTOR-UP

Cue:

(TEST (LOCATED MOVABLE.1 SECTOR.1))

Preconditions:

- no entry -

Setting:

(TEST (LOCATED-WITHIN SECTOR.1 REGION.1))

Resources:

- no entry -

Properties:

((VARIABLES (NOCONSTRAIN (REGION.1)))
(AUTHORING-SYSTEM SIPE-2)
(CLASS STATE-RULE))

N9143:
(CONCLUDE (LOCATED MOVABLE.1 REGION.1))

Figure 12: An ACT for Deducing Locations

of an object in a sector. The Setting instantiates *region*.1 to be the region of the new sector, and the single plot node uses the Conclude metapredicate to specify that the object is now located at *region*.1. Similar rules deduce that the object is no longer located at its previous sector or region.

These three ACTs show how ACT constructs are used to represent knowledge that is typical of plan generation and execution. They show event-driven and goal-driven ACTs, applicability conditions, resources, specification of conditions to be maintained over an interval, parallel plan fragments, and deduction. In particular, the ACT formalism supports the knowledge necessary to implement the deductive causal theories used by SIPE–2 for deducing context-dependent effects of actions. It is easy to create and modify ACTs using the ACT-Editor described in the next section.

4.5 ACT-Editor

In our domains, the complexity of the plans requires a graphical user interface to input ACTs, understand generated plans, and monitor system behavior. We have

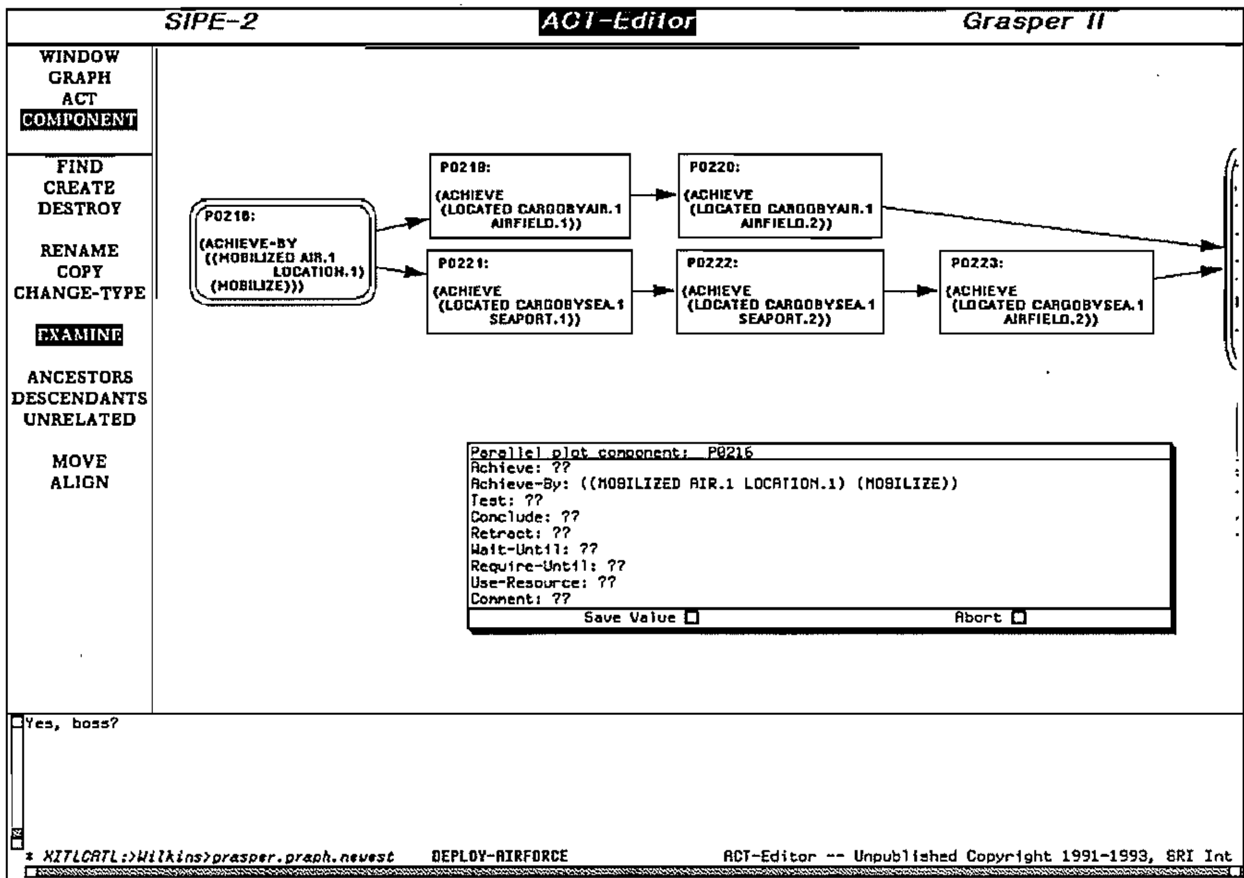

Figure 13: Plot of Deploy Airforce ACT

implemented an ACT-Editor for displaying, editing, and inputing ACTs. It effectively provides a graphical knowledge-editor for both SIPE–2 and PRS.

The user interface for SIPE–2, PRS, and the ACT-Editor is based on another proven and extensively used system, Grasper.[3] Grasper [9, 8] is a programming-language extension to Lisp that introduces graphs — arbitrarily connected networks — as a primitive data type. It includes procedures for graph construction, modification, and queries as well as a menu-driven, interactive display package that allows graphs to be constructed, modified, and viewed through direct pictorial manipulation.

The ACT-Editor displays ACTs graphically as shown in Figures 8 and 13. Grasper allows great flexibility for the user to customize the display of ACTs. As with SIPE–2

[3]Grasper is a trademark of SRI International.

and PRS interfaces, noun/verb menus are used for interaction. Figure 8 showed the commands available for acting on a whole ACT. The Examine command was used to change the print width of the ACT to produce the drawing of the plot in Figure 10. At the bottom of the command menu are two commands to translate this ACT to either SIPE–2 or PRS. Figure 13 shows the commands available for acting on individual slots and plot nodes. In this figure, the user has clicked the Examine command and then clicked the first node of the plot. The resulting pop-up window allows the user to edit any of the metapredicates on that node — currently Achieve-By is the only metapredicate on node P0216.

A user can input ACTs graphically using the menu-based interface. A user can create or modify an ACT by selecting the appropriate actions from the menu; the ACT-Editor will prompt for all necessary information. For example, in Figure 13 the pop-up window allows editing of the metapredicates, e.g., additional effects of the mobilize action could be added under the Conclude metapredicate. A user who has finished creating or modifying an ACT can use the Verify command to invoke the ACT-Verifier to check the syntax and topology.

ACTs produced by the Sipe-to-ACT translator can often be quite complex. This is certainly the case when plans generated by SIPE–2 for nontrivial problems are converted into ACTs; for example, a typical plan in the military domain produces an ACT with over 200 plot nodes in it. User-generated ACTs can also be complicated, especially if the user is inexperienced. Two capabilities were added to the ACT-Editor with the purpose of reducing the complexity of manipulating and viewing ACTs: a *simplifier* and the *New View* command.

The simplifier is used to streamline the logical structure of an ACT. The simplifier will eliminate both unnecessary plot nodes and redundant ordering links. These simplifications produce ACTs that are semantically equivalent but often noticeably more compact. The compactness is of benefit primarily for viewing purposes, making the intent of an ACT more apparent to a user. In addition, simplification can lead to improved plan execution performance when unnecessary components of ACTs are eliminated.

The New View command does not modify the structure of an ACT but rather modifies its presentation on the screen. While the underlying ACT remains unchanged, a user can vary the amount of detail to be presented on each plot node. Having the ability to modify the display characteristics in this manner is of value most notably when examining ACTs generated by SIPE–2 for complex problems, since their size can render them unwieldy.

4.6 Translators

SIPE–2 and PRS run their own internal representations for efficiency reasons. Thus, it is necessary to translate ACTs into these representations and back. Section 4.3 described some of the details of translating metapredicates.

The Sipe-to-ACT translator can translates all SIPE–2 operators to the ACT formalism, as well as translating fully instantiated plans. Thus, all SOCAP operators originally written for SIPE–2 can be viewed and modified with the ACT-Editor (as shown by the examples in Figures 8 and 13), and new operators can be created using the ACT-Editor. The system cannot yet translate a partial plan with uninstantiated variables into ACT, since not all possible constraints on variables generated by SIPE–2 have a corresponding formulation in ACT. This could be handled by using the Properties slot to store an internal SIPE–2 representation, but we have not found it necessary to translate partially developed plans.

The ACT-to-Sipe translator translates all ACTs into either SIPE–2 operators or plans. This enables SIPE–2 to plan with ACTs that have been input by the user in the ACT-Editor. The temporal constraints in the Time-Constraints slot are translated for processing by Tachyon. The ACT-to-PRS translator converts ACTs into a representation that can be interpreted and executed in real time under the control of PRS. This makes all SIPE–2 operators accessible to PRS by running Sipe-to-ACT and ACT-to-PRS. Section 4.3 described this process. The constraints in the Time-Constraints slot are currently ignored when translating to PRS, although it is clear that PRS could be extended to support them.

We have not implemented a PRS-to-ACT translator, as PRS does not have structures that will be passed to SIPE–2. We are currently investigating the research issues involved in having PRS invoke SIPE–2 to replan; one solution is for PRS to communicate individual goals and change a common database rather than communicate complete ACTs.

5 Conclusion

The ACT formalism encodes knowledge necessary for the generation and execution of complex plans, with reactive response to a changing environment. ACT is an heuristically adequate representation that is useful in practical applications and serves as an interlingua for AI technologies in planning and reactive control. The development of the ACT formalism is similar in motivation to the development of the KRSL language [10] that is being done as part of the DRPI. However, the ACT formalism is more focused, trying to get practical planning and execution systems to use a common language.

The use of ACT by SIPE–2 and PRS in a practical application has been demonstrated. The development of PRS and SIPE–2 have been driven by the applications to numerous problem domains. Similarly, the development of the ACT formalism has been driven by the representations of these two systems and by the military operations planning domain. (We view the ACT formalism as an evolving entity that will have extensions made as driving domains require new features.) The ACT formalism has been used in this domain to represent both knowledge about actions and generated plans containing several hundred nodes.

ACTs have enabled SIPE–2 and PRS to cooperate on the same problem for the first time — SIPE–2 generating plans while PRS responds to events, executes the plans produced, and controls the execution of the lower-level actions that have not been planned. The ACT-Editor provides a graphical knowledge-editor for both SIPE–2 and PRS, since ACTs can be translated into either system.

The integrated planning and execution we have demonstrated in this problem shows that ACT representational constructs have reasonable computational properties as well as being expressive enough for this domain. The ability to represent all the constructs in PRS and SIPE–2 implies the ACT formalism is sufficient for a wide range of interesting problems, since both these systems have been applied to several practical domains.

Acknowledgments

The research described in this paper was supported by the DARPA/Rome Laboratory Planning Initiative under Contracts F30602-91-C-0039 and F30602-90-C-0086. The people who designed and implemented the ACT formalism are John Lowrance, Leonard Wesley, Janet Lee, Karen Myers, and Peter Karp. Thanks to Karen Myers for many excellent suggestions on the presentation in this paper.

References

[1] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the Association for Computing Machinery*, 26(11):832–843, 1983.

[2] Richard Arthur and Jonathan Stillman. Tachyon: A model and environment for temporal reasoning. Technical report, GE Corporate Research and Development Center, 1992.

[3] K. Currie and A. Tate. O-plan: The open planning architecture. *Artificial Intelligence*, 52(1):49–86, 1991.

[4] R. J. Firby. An investigation into reactive planning in complex domains. In *Proceedings of the 1987 National Conference on Artificial Intelligence*, pages 202–206, American Association for Artificial Intelligence, Menlo Park, CA, 1987.

[5] Michael P. Georgeff and François Félix Ingrand. Decision-making in an embedded reasoning system. In *Proceedings of the 1989 International Joint Conference on Artificial Intelligence*, American Association for Artificial Intelligence, Menlo Park, CA, August 1989.

[6] Michael P. Georgeff and Amy L. Lansky. A procedural logic. In *Proceedings of the 1985 International Joint Conference on Artificial Intelligence*, American Association for Artificial Intelligence, Menlo Park, CA, August 1985.

[7] M. L. Ginsberg. Knowledge interchange format: The KIF of death. *AI Magazine*, 12(3):57–63, 1991.

[8] P.D. Karp, J.D. Lowrance, T.M. Strat, and D.E. Wilkins. The Grasper-CL graph management system. Technical Report 521, SRI International Artificial Intelligence Center, Menlo Park, CA, 1992.

[9] Peter D. Karp, John D. Lowrance, and Thomas M. Strat. *The Grasper-CL Documentation.* SRI International Artificial Intelligence Center, Menlo Park, CA, 1992.

[10] Nancy Lehrer. KRSL specification language. Technical Report 2.0.2, ISX Corporation, 1993.

[11] D. M. Lyons and A. J. Hendricks. A practical approach to integrating reaction and deliberation. In *First International Conference on Artificial Intelligence Planning Systems*, pages 153–162, College Park, Maryland, 1992.

[12] R. MacGregor and M. H. Burstein. *Using a Description Classifier to Enhance Knowledge Representation*, June 1991.

[13] David E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm.* Morgan Kaufmann Publishers Inc., San Mateo, CA, 1988.

[14] David E. Wilkins. Can AI planners solve practical problems? *Computational Intelligence*, 6(4):232–246, 1990.

[15] David E. Wilkins. Planning in dynamic and uncertain environments. Annual report, SRI International Artificial Intelligence Center, Menlo Park, CA, September 1992.

[16] David E. Wilkins. *Using the SIPE Planning System: A Manual.* SRI International Artificial Intelligence Center, Menlo Park, CA, 1992.

[17] David E. Wilkins and Roberto V. Desimone. Applying an AI planner to military operations planning. In M. Fox and M. Zweben, editors, *Intelligent Scheduling.* Morgan Kaufmann Publishers Inc., San Mateo, CA, 1993.

A Syntax for ACT Metapredicates

The following Backus-Naur Form (BNF) documents the syntax for ACT metapredicates. For those unfamiliar with BNF, the form on the left of the ::= symbol should be written using the form on the right. Items in the typewriter font (e.g., `item`) represent the exact characters to use. The | symbol represents "or", the * represents any number of repetitions including zero, and the ** represents any number of repetitions, but there must be at least one occurrence. The braces {} designate optional elements.

The right-hand side of the first group of rules is an English description of the lowest-level placeholders.

```
pred-name       ::= the name of a predicate
pred-arg        ::= the name of a variable, a function, or an object
act             ::= the name of an ACT
prop            ::= any s-expression
val             ::= any s-expression
plotnode        ::= the name of a plot node (a symbol)
integer         ::= any integer

pv-pair         ::= ( prop val )
pred            ::= (pred-name {pred-arg}*) | (not (pred-name {pred-arg}*)) |
                    (unknown (pred-name {pred-arg}*))

conj-wff        ::= pred | (and {conj-wff}**)
disj-wff        ::= pred | (or {disj-wff}**)
wff             ::= conj-wff | disj-wff
wff-pair        ::= ( wff wff )
wff-list        ::= ( {wff}** )

time-constraint ::= (allen node node {{integer}-{integer}}) |
                    (qual-relation (point node) (point node))
allen           ::= starts | overlaps | before | meets | during |
                    finish | finishes | equals
qual-relation   ::= later | later-eq | earlier | earlier-eq | equals
point           ::= start | end
node            ::= pred-name{.integer} | act{.integer} | plotnode
```

Wait-Until	::= wff | wff-list
Conclude	::= conj-wff
Achieve-By	::= {(wff ({act}**))}*
Require-Until	::= wff | wff-pair
Use-Resource	::= pred-arg | ({pred-arg}**)
Comment	::= val
Property	::= pv-pair | ({pv-pair}**)
Time-constraints	::= {time-constraint}*
Achieve	::= wff | wff-list
Test	::= wff | wff-list

On a Cue slot:

Achieve	::= wff | disj-wff
Test	::= wff | disj-wff

While the above syntax is accepted by the ACT-Editor and ACT-Verifier, PRS and SIPE–2 restrict the forms they can handle. These restrictions are mentioned here. Currently, PRS ignores the temporal constraints in the Time-Constraints slot. In both systems, an `or` is equivalent to the first disjunct that succeeds (as in Lisp). Thus, it is not a logical disjunction that will consider instantiations from all disjuncts. PRS currently restricts use of unknown truth values to open predicates. The definition of *wff* allows arbitrarily deep nesting of `and` and `or`. This is not in general supported by SIPE–2.

The following are additional properties recognized by SIPE–2:

- The system will use only ACTs whose Class property in Properties is one of: `state-rule`, `causal-rule`, `init-operator`, `init.operator`, `operator`, `both.operator`.
- The system will use a Variables property in the Properties slot whose value is a list of quantifier-pairs. Each quantifier-pair whose first element is `existential` or `universal` will be processed.

The following are restrictions placed on the ACT formalism by SIPE–2 (a warning message is printed whenever something is ignored):

- In the Precondition slot, Achieve is ignored.
- In the Cue slot, Achieve is ignored in a deductive ACT, and Test is ignored in a nondeductive ACT.
- An `and` should not be nested inside an `or`.
- Predicates that will be translated to SIPE–2 constraints should not be inside an `or`. These predicates are $= \quad class \quad in - range \quad > \quad < \quad <= \quad >=$.
- An `or` is allowed only in a Test metapredicate.
- For deductive operators, there must be a Test in the Cue slot, all plot nodes except the start node are ignored, and all metapredicates except Conclude on that plot node are ignored.

B Creating ACTs Programmatically

Normally, ACTs are created interactively with the ACT-Editor. Sometimes it is desirable to create ACTs programmatically, e.g., when writing a translator from some langauge to ACT. This appendix documents the functions of the ACT-Editor that can be used for that purpose

The ACT metapredicates can be thought of as properties of a plot node or slot. For each slot that can have a metapredicate, there is a function named CREATE-ACT-slot that creates the corresponding slot. The metapredicates are implemented as properties in the Grasper representation.

`create-act-cue` *act &key achieve test comment tuples append?* [*Function*]

`create-act-precondition` *act &key achieve test comment tuples append?* [*Function*]

`create-act-setting` *act &key test comment tuples append?* [*Function*]

`create-act-resources` *act &key use-resource comment tuples append?* [*Function*]

`create-act-property` *act &key comment tuples append?* [*Function*]

`create-act-properties` *act &key comment tuples append?* [*Function*]

`create-act-comment` *act &key comment tuples append?* [*Function*]

The plural names are synonyms for the singular names. The first argument of each function is the name of the ACT, and the allowed metapredicates are given as keywords. In addition, the keywords `:tuples` and `:append?` are accepted. `:Tuples` accepts the syntax for arbitrary properties, and can be used as an alternative means to specify metapredicates by using the metapredicate name as the property name. `:Append?` is a Boolean value that when true specifies that values for a given property are to be appended to any values already present. The default is that the most recent property-value pair would override any existing value for that property. Below are some example calls to these functions.

```
(create-act-cue 'ACT1 :achieve '(P1 A1 B1))
(create-act-property 'act1 :tuples '((act::authoring-system act::sipe-2)
                                     (act::class operator)))
(create-act-cue 'ACT1 :tuples '((ACHIEVE (P1 A1)) (ACHIEVE (Pn An)))
                      :append? t)
```

Plots are networks of plot nodes that are ordered by edges. Plot nodes are either parallel or conditional nodes. A parallel node cannot be activated unless all incoming edges are active, and it activates all outgoing edges. A conditional node may be activated when only one of its incoming edges is active, and it in turn can succeed if only one of its outgoing edges is successfully activated.

`create-parallel-node` *name act &key location achieve achieve-by test conclude retract wait-until require-until use-resource comment tuples append?* [*Function*]

This creates a parallel node with the given name in the given act. Metapredicates are specified by keywords, with `:tuples` and `:append?` also accepted. The keyword `:location` is also accepted. Its value should be a list of numbers of length two that specify the X and Y coordinates of the node.

`create-conditional-node` *name act &key location achieve achieve-by test conclude retract wait-until require-until use-resource comment tuples append?* [*Function*]

This creates a conditional node with the given name in the given act. The keywords are the same as for `create-parallel-node`.

`create-ordering-edge` *source-node destination-node act &key location* [*Function*]

This creates an ordering edge from source-node to destination-node in the given act. `:Location` is a list of (x y) locations that specify knots along the edge.

Below are some example calls to these functions.

```
(create-parallel-node 'foo 'act1 :conclude (predlist->act (effects opr))
                                 :location (list start-x start-y))
(create-parallel-node 'baz 'act1 :location (list (+ 200 start-x) start-y))
(create-ordering-edge 'foo 'baz 'act1)
```