A Common Knowledge Representation for Plan Generation and Reactive Execution

David E. Wilkins Karen L. Myers
SRI International
Artificial Intelligence Center

8 July 1994

This paper has been accepted to the *Journal of Logic and Computation*, and should appear in 1995. Meanwhile, it is also available in postcript form on the World Wide Web in http://www.ai.sri.com/people/ wilkins, and as SRI AI Center Technical Note 532R.

SRI International, 333 Ravenswood Ave., Menlo Park, Ca. 94025

Abstract

The ability to integrate sophisticated planning techniques with reactive execution systems is critical for nontrivial applications. Merging these two technologies is difficult because the forms of knowledge and reasoning that they employ differ substantially. The ACT formalism is a language for representing the knowledge required to support both the generation of complex plans and reactive execution of those plans in dynamic environments. A design goal of ACT was its adequacy for practical applications. ACT has been used as the interlingua in an implemented system that links a previously implemented planner with a previously implemented executor. This system has been used in several applications, including robot control and military operations, thus attesting to its expressive and computational adequacy.

Contents

List of Figures

Abstract

The ability to integrate sophisticated planning techniques with reactive execution systems is critical for nontrivial applications. Merging these two technologies is difficult because the forms of knowledge and reasoning that they employ differ substantially. The ACT formalism is a language for representing the knowledge required to support both the generation of complex plans and reactive execution of those plans in dynamic environments. A design goal of ACT was its adequacy for practical applications. ACT has been used as the interlingua in an implemented system that links a previously implemented planner with a previously implemented executor. This system has been used in several applications, including robot control and military operations, thus attesting to its expressive and computational adequacy.

1 Introduction

Our research involves developing systems to select and execute appropriate actions for achieving goals in dynamic and uncertain environments. Dynamic environments require the ability to react to unexpected events and to make appropriate decisions quickly. These systems should be capable of acting in a reasonable manner without a plan; however, the ability to synthesize plans and use these plans to guide the execution of the systems is critical for nontrivial tasks.

Traditionally, plan generation and reactive execution have been considered as separate activities, with few attempts to integrate them within a single system. Such integration is difficult, given that generation and execution involve different kinds of knowledge and reasoning capabilities. We have developed the ACT formalism for representing the knowledge necessary to support both plan generation and reactive execution. This paper describes ACT, the design decisions behind it, and its use in sharing knowledge between an implemented planner and an implemented executor.

Development of an interlingua that enables multiple reasoning systems to cooperate on different aspects of the same problem is a central challenge for the field of artificial intelligence (AI). ACT addresses the restricted case of getting planning and execution systems to cooperate, thus allowing heterogeneous combinations of different planning and execution systems. Such focus is critical to achieving success in terms of near-term use in software tools, since a common representation that tries to cover too broad an area runs into problems that have been described elsewhere [10].

Two features distinguish our approach: (1) the development of a heuristically adequate system that will be useful in practical applications, and (2) the need to generate and execute complex plans with parallel actions of multiple agents. To aid in achieving these requirements, we began with AI technologies in planning and reactive control that had been developed previously and shown to be useful in practical applications. The representational capabilities of these systems provides a minimal level of functionality to be incorporated into ACT. However, ACT is intended to serve as a general-purpose representation language that could be used to share knowledge between many different execution and planning systems. In fact, several extensions were made to the existing planning and execution technologies to fully implement ACT. We anticipate that ACT will evolve over time to incorporate new concepts that prove to be useful.

1.1 Validation

The representational and computational adequacy of ACT has been validated by implementing the Cypress system, which uses ACT as an interlingua to enable run-time interactions between planning and execution subsystems. Cypress has been applied in two domains that are complex, uncertain and dynamic. The first domain is controlling an indoor mobile robot. A reactive controller 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 necessary because execution of military plans is often interrupted by unexpected equipment failures, weather conditions, enemy actions, and other events that may require changes to the overall strategic plan.[1]

A complete description of the applications of ACT to the robot and military problems is beyond the scope of this paper, although self-explanatory examples are given. More detailed descriptions can be found elsewhere [17, 19, 4]. Examples in this paper are drawn primarily from the domain of planning military operations.

1.2 Overview of the Paper

Section 2 begins with brief descriptions of the initial AI technologies. Several difficulties in defining a common representation for planning and execution are described and addressed in Section 3. Section 4 presents the definition of ACT itself. Section 5 describes the use of ACT in Cypress and the extensions made to the existing planning and execution technologies to fully implement ACT. This is followed by a comparison to related work.

2 Planning and Execution Technologies

To achieve a 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). These two systems originally employed vastly different formalisms for encoding knowledge about actions. The ACT formalism is a common representation that both systems can use for representing and reasoning about plans and the knowledge necessary to generate and execute them. The complete inputs for both systems can be specified in ACT. Translators have been implemented from the ACT formalism to both SIPE–2 and PRS (see Section 5.1), so ACT is now the preferred language for encoding knowledge in both systems.

The development of PRS and SIPE–2 was driven by their applications to numerous problem domains. Similarly, the design of the ACT formalism has been driven by the

[1] Application of our systems to this domain, as well as the work described here, was done as part of the ARPA-Rome Laboratory Planning Initiative (ARPI) [19].

functionalities provided in these two systems. As a result, the ACT formalism is sufficient for a wide range of interesting problems. However, ACT should not be considered a fixed representation that has been designed to cover every anticipated need. Extensions will no doubt be made to it as new domains require new features. This philosophy 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 doing so, the problem of developing extremely expressive representations with poor computational properties and little practical merit is avoided.

Before describing the ACT formalism, the two systems for which it provides a common representation are briefly described. More detailed descriptions can be found elsewhere [9, 17, 18].

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 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.

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.

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 [19]. This technology is generic and domain-independent, and has proven useful on a large variety of problems. Example applications include planning the actions of a mobile robot, managing aircraft on a carrier deck, containing oil spills, travel planning, construction tasks, producing products from raw materials under production and resource constraints, and joint military operations planning [18, 19].[2] In the military domain used for most examples in this paper, SIPE–2 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.

[2] SIPE–2 is the core reasoning engine in SRI's SOCAP system (System for Operations Crisis Action Planning) [19], which was used for the second Integrated Feasibility Demonstration of ARPI.

2.1.1 Operators

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 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 list only effects that are required to trigger all the necessary deduced effects. Since the causal theory will deduce different effects depending on the situation, 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.

The *purpose* of an operator determines which goals the operator can solve — in this case, to deploy an air force to a particular airfield 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 determines intermediate seaports and airports that are on routes to the destination given in the database and 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 precondition 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 actions 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 abstraction never changes), or a lower level. 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

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

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.

2.1.2 Deductive Rules

In SIPE–2, the deductive rules typically deduce most of the effects of the actions in a plan. In the example operator, all 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

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

abstraction levels has changed.

The deductive rules that specify 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 permitting 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.

2.2 PRS

The Procedural Reasoning System (PRS) [9] is a framework for constructing reactive control systems that can perform complex tasks in dynamic environments. PRS attempts to achieve any goals it might have, given its current beliefs about the world, while simultaneously reacting to new events that occur, thus providing a framework in which goal-directed and event-driven behaviors can be integrated smoothly. Here, we describe PRS as it was defined when work on ACT first began. As discussed in Section 5.2, many capabilities in ACT were added to to this version of PRS, resulting in a new system called PRS-CL.

The architecture of PRS is depicted 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 operators, called *Knowledge Areas (KAs)*, describing how sequences of actions and tests may be performed to achieve certain goals or to react to particular situations; and (4) *intentions* containing those KAs that have been chosen

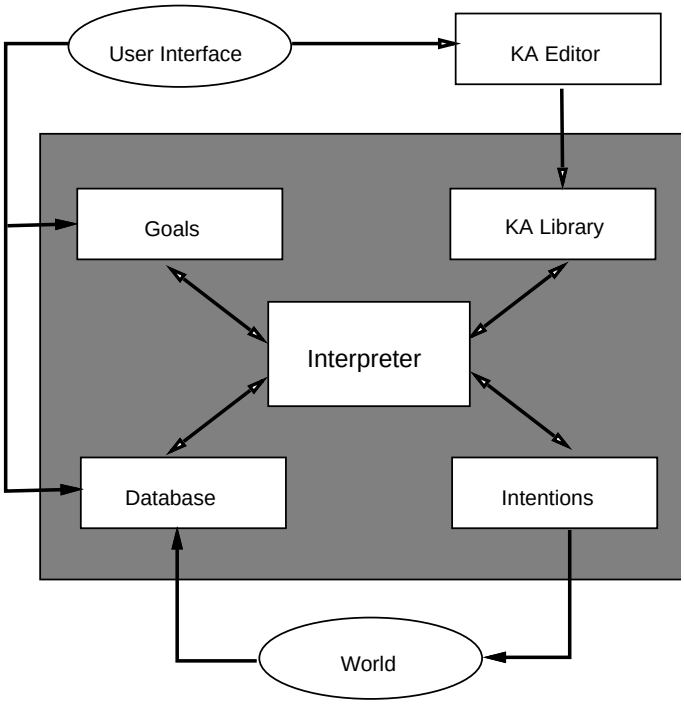

Figure 3: Architecture of the Procedural Reasoning System

for (eventual) execution. An *interpreter* manipulates these components, by selecting appropriate KAs for execution based on the system's beliefs and goals, then creating the corresponding intentions, and finally executing them.

The system interacts with its environment 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. While executing plans, PRS constantly monitors incoming database changes. The inference mechanism used guarantees that any new fact in the database is noticed in a bounded time, thus providing rapid response to new events.

PRS has several capabilities that make it a powerful system for developing real-time applications. Multiple copies of PRS objects, each referred to as an *agent*, can be run in parallel. Agents operate asynchronously but can communicate through message passing to solve problems in a distributed, cooperative manner. In addition, *meta-KAs* can be used to implement complex control and scheduling behaviors, as required for individual applications. KAs can manipulate the internal beliefs, goals, and intentions of PRS, and KAs that do so are referred to as *meta-KAs*. For example, a meta-KA might modify the intentions of the system after computing the amount of reasoning that can be undertaken given the real-time constraints of the problem domain. Further details are given elsewhere [8].

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:
""
```

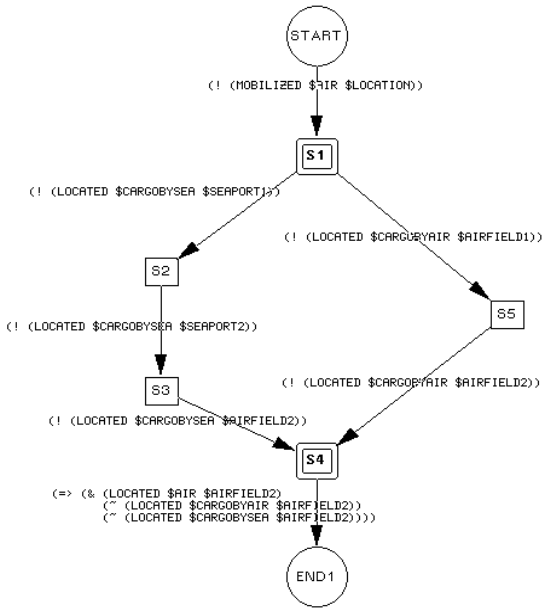

Figure 4: KA for Deploying an Air Force

2.2.1 Knowledge Areas

Describing the representation of KAs in PRS will help explain the central ideas of the ACT formalism and the translation process from ACT to PRS. The KAs encode knowledge of how to accomplish goals or react to certain events. Each KA consists of an *applicability condition*, which specifies under what situations the KA is useful, and a *body*, which describes the steps of the procedure. Together, the applicability condition and body of a KA express a declarative fact about the results and utility of performing certain sequences of actions under certain conditions.

The body and invocation condition will be presented by discussing the KAs in Figures 4 and 5. The former is a KA for deploying an air force that is useful for comparisons with SIPE–2 operators and the ACT formalism. (The ACT version of this KA will be given in Section 4; the SIPE–2 representation of this knowledge was shown in Figure 1.) However, this KA encodes the kind of knowledge generally used by a high-level planning operator and is not representative of the knowledge that is typically used by PRS. Therefore, the KA in Figure 5 for isolating a leak in a complex physical system is also described. In fact, the `DEPLOY-AIRFORCE` KA can be executed only in PRS-CL since the version of PRS described in the literature [9] does not support parallel execution of actions within a KA.

The applicability condition of a KA has two components: an *invocation* part and a *context* part. Both must be satisfied for the KA to be applied. The invocation part of a KA 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. The invocation part of the `DEPLOY-AIRFORCE` KA specifies that the KA will be considered for application when PRS is given the goal to achieve the deployment of an air force to an airfield.

The context of a KA is a logical expression that specifies conditions that must be true in the current state in order for the KA to be executed. The context for the `DEPLOY-AIRFORCE` KA requires (among other things) that the initial position of the air force be known, and that intermediate seaports and airports on known routes to the destination have been granted transit approval. Satisfaction of the context may involve binding variables within its logical expression to specific domain objects.

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 traversing arcs through the network. 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. When more than one arc emanates from a given node, only one of the arcs needs to be traversed. PRS will test the arcs emanating from a node in an arbitrary order until one is found that can be successfully traversed. PRS then "commits" to this choice, ignoring all other arcs from the node. Thus, multiple outgoing arcs from a node are used to implement conditional branching, as shown in Figure 5. Execution completes successfully when an end node (i.e., a node with no outgoing arcs) is reached. If the system cannot successfully traverse any of the arcs emanating from a node, then execution fails for the KA.

In Figure 4, the first arc of the body is labeled with the goal of achieving the mobilization of the air force. As described in Section 5.2, PRS-CL supports parallel execution of nodes — to successfully execute the nodes with double borders in Figure 4 (`S1` and `S4`), PRS-CL must successfully traverse all their incoming and outgoing arcs.[3] Subsequent 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.

The KA in Figure 5 is more typical of the procedural knowledge used by PRS. Arcs are labeled with low-level goals that are close to the primitive execution level, and numerous branches are used to select courses of actions conditionally at run time. This KA describes a procedure for isolating a leak in a reaction control system (RCS) of the space shuttle. It is applicable when the system acquires the goal to isolate a leak in an RCS (`$p-sys`), provided the various conditions in the context are true. The full KA for this procedure consists of more than 45 nodes.

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`. 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.

[3] The original PRS did not support parallel execution within a KA.

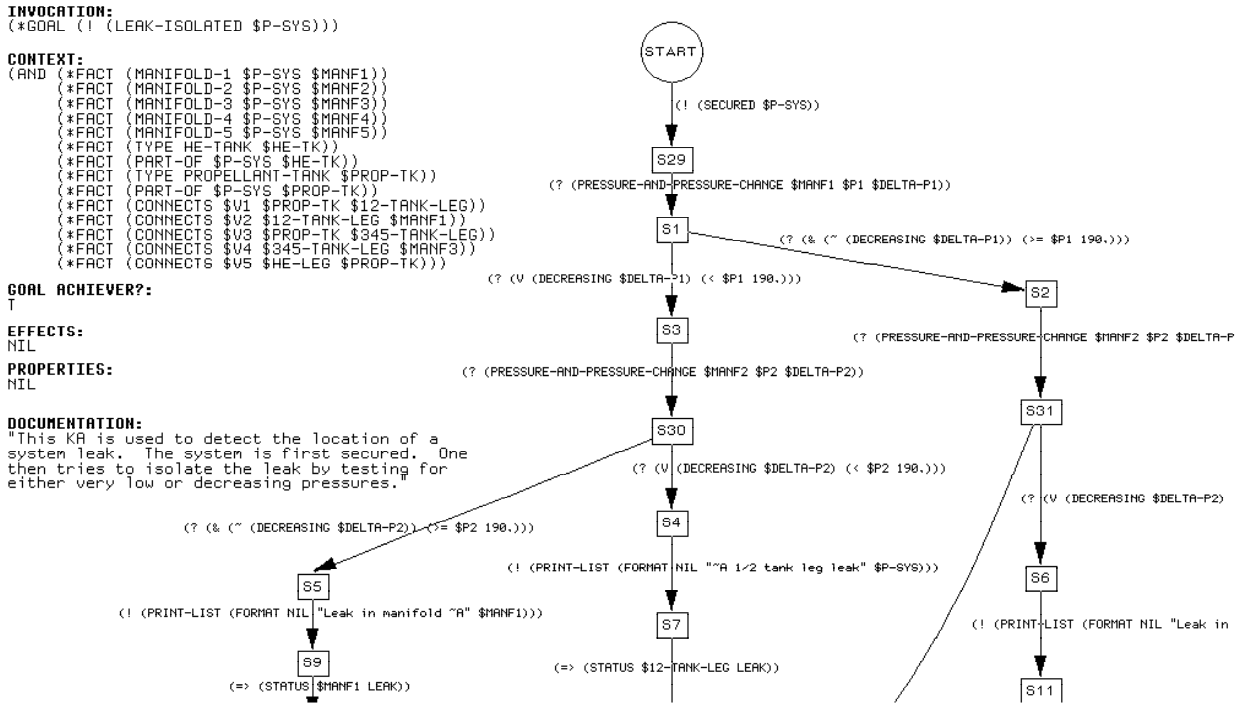

Figure 5: Portion of a KA for Leak Isolation

3 Developing a Common Representation

Consolidating the representational needs of both plan generation and reactive execution systems is no small task, given the widely different functionalities of the two classes of systems. For example, execution systems frequently use conditionals and loops, and monitor the world to determine what is true. Generative planners stress parallel actions and predicting the results of taking certain actions. There is, however, much overlap in the representational requirements for the two classes of systems. We discuss both the commonalities and differences between their respective representational needs.

3.1 Commonalities

Plan generation and reactive execution systems share many representational requirements. At the heart of both plan is the need for a language in which to express beliefs and goals. Such a language provides the basis upon which to build representations of actions, plans, and operators. Goals having different modalities are important to both classes of systems, including goals of achievement, testing, and maintenance (among others). The ability to express protection intervals for conditions to be maintained is a common necessity.

The notion of *applicability* of a procedure or operator to accomplish a goal is shared by both classes of systems. Certain conditions must be met before a procedure or operator can be used. Both classes of systems require the means to order goals and actions. Flexibility in this regard is important: it should be possible to represent both sequential and parallel activities, as well as cycles and conditional branching.

Another shared requirement is the ability to represent information required for deductive reasoning. Plan generation systems require deductive reasoning to keep track of the context-dependent effects of actions. Such a capability greatly enhances the usefulness of a planning system, and is an important part of most applications of SIPE–2. Plan execution systems make use of deductive reasoning for keeping track of consequences of changes in the world.

While the above shared representational needs are critical to both classes of systems, additional capabilities may be needed for certain applications. Examples include abilities to specify and allocate resources, and abilities to specify and reason about temporal constraints.

3.2 Differences

Generative planners and reactive executors (referred to hereafter as *planners* and *executors*) have widely differing operational characteristics. Those that lead to representational differences between the two classes of systems are described here, and those that lead to differing interpretations or implementations of a common representation are described in Sections 4 and 5.

Executors are designed to respond to new goals and information and take appropriate steps as a consequence. They are expected to be *embedded* in the real world, and typically employ a single world model corresponding to the actual perceived state. In contrast, planners explore a search space to generate a partially ordered set of activities designed to satisfy a particular goal. Planners must reason about hypothetical future states that would result from taking certain actions. This distinction between a single world state versus numerous hypothetical states impacts how these systems respond to failures and model the world.

Since planners reason about possible future world states, they must model every proposition of interest internally. In contrast, executors are embedded systems that can gather information from the world. Thus, executors treat the world as an information source and apply sensors to collect additional information about the world. Representations for actions must support the specification of procedures for *actively testing* properties of the world.

Failure has different meanings for the two classes of systems. Failure for planners means that either a plan (or partial plan) that has been constructed is not suitable. Unsuitability can arise because a complete plan does not satisfy the original goal, a constraint (e.g., a goal of maintenance or a resource requirement) is violated, or there is no completion of a partial plan that could satisfy the goal. Upon failure, planners have several options. They can backtrack to consider alternative plan choices, or employ various algorithms to modify the plan so that it will satisfy the goal or not violate the constraints (e.g., there are algorithms to eliminate resource conflicts). Failure has different consequences for executors since they are taking actions in the world. Thus, failure generally indicates that the current set of activities should be aborted, but

it is difficult to determine the most appropriate response. Thus, executors will need to encode procedures, unnecessary to planners, that will allow recovery from failure states.

Other differing operational characteristics also impact any common representation. Executors do not reason ahead about the consequences of their actions. As a result, many executors contain some form of *reflective reasoning* that allows them to reason about current activities in order to decide on future actions. From a representational perspective, such reflective reasoning requires the means to represent notions of system goals and activity within the representation language. Planners don't require such facilities, as they have no corresponding need for reflective reasoning. While a planner could control its search by reflective reasoning, most AI planners instead have specialized search engines to quickly change focus in the search space.

Planners proceed from a fixed initial state, while executors operate in environments that are highly dynamic and may change in unpredictable ways during execution. Executors must be able to respond to such changes in a timely fashion, and therefore require the ability to represent *event-driven* behavior. Executors will need knowledge, unnecessary to planners, about how to respond to events as they occur.

Planners and executors rely on different kinds of action sequences. Executors rely on constructs such as conditional actions and iteration, while planners emphasize parallel execution. Parallel execution is often ignored in executors; for example, before extensions were made to support ACT, PRS did not support parallel execution of actions within a KA. An adequate shared representation of actions for planners and executors must support all of these sequencing operations.

4 The ACT Formalism

The ACT formalism is a domain-independent language for representing the kinds of knowledge about activity used by both plan generation and reactive execution systems. The basic unit of representations is an *Act*, which can be used to encode both plan fragments and standard operating procedures (SOPs).

An Act describes a set of actions that can be taken to fulfill some designated purpose under certain conditions. The purpose could be either to satisfy a goal or to respond to some event in the world. The purpose and applicability criteria for an Act are formulated using a fixed set of *environment conditions*. Action specifications are called the *plot*, and consist of a partially ordered set of actions and subgoals.

The environment conditions and plots are specified using *goal expressions*, each of which consists of one of a predefined set of *metapredicates* applied to a logical formula. The metapredicates permit the specification of many different modes of activity, including goals of achievement, maintenance, and testing.

Our discussion begins with an overview of the goal expressions supported in ACT, next describes the environment conditions and plots, then explains variable usage in Acts, and finally describes how the metapredicates are interpreted by both the planner and the executor. An additional issue in the design of ACT was to make it clear enough to enable users to understand Acts 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 (e.g., see Section 4.3). The complete grammar

Environment Slot	Role
Name	identifier
Cue	used to efficiently retrieve this Act
Precondition	gating conditions on applicability of this Act
Setting	queries world to bind local variables
Resources	resource constraints
Properties	user-defined attributes, temporal constraints
Comment	documentation

Table 1: Environment Conditions and Their Roles

for ACT is given in the Appendix, which also documents restrictions on this syntax imposed by our implementation (see Section 5).

4.1 Goal Expressions

Goal expressions describe requirements on the planning/execution process and desired states to be reached. They consist of an ACT metapredicate applied to a logical formula built from predicates specified in first-order logic, connectives, and the names of Acts. The predicates describe possible goals and beliefs of the system. Goal expressions are used to specify both applicability conditions for environment conditions and subgoals for plot nodes. The interpretation of the goal expression can vary slightly, depending on whether it is in an environment condition or plot node. The following summary introduces the metapredicates; more precise meanings are given in Section 4.5.

The *Test* metapredicate specifies a formula whose truth value must be ascertained. The *Use-Resource* metapredicate makes a declaration of resources that will be used by the Act, and hence that must be available for an Act to be applied. Three metapredicates can be thought of as specifying actions: *Achieve*, *Achieve-By*, and *Wait-Until*. Achieve metapredicates direct the system to accomplish a goal by any means possible; the Achieve-By metapredicate is similar but specifies a restricted set of Acts that can be used to accomplish the task. Wait-Until directs the system to wait until some specified condition holds. The *Require-Until* metapredicate designates conditions that must be maintained over a specified interval. The *Conclude* metapredicate designates information about changes in the world caused by an action.

4.2 ACT Environment Conditions

The ACT environment conditions are defined as a series of fixed *slots*, shown in Table 1. Name and Comment are straightforward; the former is a unique identifier for the Act, and the latter is a string that provides documentation. The slots Cue, Precondition, Setting, and Resources are referred to as the *gating slots* for an Act because they specify conditions that must be satisfied in order for the Act to be applicable in a given situation. The gating slots are filled with one or more goal expressions. The environment conditions are discussed in detail below, including an explanation of what it means for a condition to be satisfied. Table 2 displays the metapredicates allowed in each of the gating slots.

Gating Slot	Metapredicates
Cue	Achieve, Test, Conclude
Preconditions	Achieve, Test
Setting	Test
Resources	Use-Resource

Table 2: Metapredicates Allowed in Gating Slots

Cue

The Cue indicates the purpose for which the Act can be used. The Cue can contain either an Achieve, Test, or Conclude metapredicate. An Achieve metapredicate in the Cue indicates that the Act can achieve some condition – that is, it can be used for subgoaling. A Test metapredicate indicates that the Act can *actively* test some condition. Active testing is important for situations where the truth-value of a formula needs to be attained, but the information is not contained in the system database. For example, one might create an active-test Act for a procedure to obtain a sensor reading.

The use of the goal expression `(CONCLUDE (P))` in a Cue indicates that the Act should be invoked when `(P)` is added to the database. A Cue containing a Conclude metapredicate can react to events that arise during the course of execution. An Act whose Cue contains an Achieve or Test is said to be *goal-invoked*, while an Act whose Cue contains a Conclude is *fact-invoked*. Execution systems may require short cue specifications so that potentially applicable Acts can be rapidly identified.

Precondition

The Precondition slot specifies situational constraints that must be satisfied for the Act to be applicable. It can contain both Achieve and Test metapredicates. The meaning of `(TEST P)` in the Precondition is that `P` must be true in order for the Act to be applied. The meaning of `(ACHIEVE G)` is that the system must currently have `G` as a goal in order for the Act to be applied.

Setting

The Setting specifies additional Test metapredicates for the applicability of an Act. This slot is equivalent functionally to the Precondition slot but typically is used to separate out those conditions whose purpose is to instantiate variables.[4] The Setting is separated from the Precondition to make the Act more easily understood.

Resources

The Resources slot indicates resources that are to be allocated for the duration of the Act. This slot can be filled only with the Use-Resource metapredicate. For an Act containing an expression of the form `(USE-RESOURCE (A B C))` in its Resources

[4] O-Plan refers to such conditions as "query conditions" [5].

slot to be applied, it is necessary that the resources `(A B C)` be free. These resources would then be unavailable for use by other processes until execution of the Act finishes.

Properties

The properties slot is a list of property/value pairs for the Act. Properties are used for several purposes: to provide documentation, to represent information specific to a particular application or planning/execution system, and to represent knowledge that is not directly supported in ACT, generally because it is needed by either the planner or the executor, but not by both. For example, SIPE–2 recognizes the property `Variables` for quantifying variables as existential or universal, and PRS-CL uses the property `Decision-Procedure` to designate an Act that is used for metalevel reasoning. The most interesting property from a representational standpoint is `Time-Constraints`, the syntax for which is given in the Appendix and allows specification of any of the 13 Allen relations [1]. This property is used to specify time constraints between plot nodes that cannot be represented by ordering arcs, e.g., two actions must end at the same time. These constraints could alternatively be represented as arcs of different color between plot nodes. There are no required properties, although some are recommended for documentation purposes (such as `Author`). The user is free to supply additional properties, as desired.

Figure 6 is an example act taken from the screen of the ACT-Editor, a system for viewing and editing Acts (see Section 5). This act was originally written as a SIPE–2 operator in the military domain and was translated by our SIPE-to-ACT translator. The environment conditions are displayed on the left side of the screen and the plot nodes on the right side. This Act describes an operator for deploying 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 Precondition enforces 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 4.3.

4.3 Plots

The plot specifies the activities for accomplishing the purpose of an Act. The plot consists of a directed graph, whose nodes represent actions and whose arcs impose a partial order for execution. (Any temporal relationship between two nodes can be represented using the Time-Constraint property.) Associated with each plot node is a list of goal expressions for the node.

4.3.1 Metapredicates in Plot Nodes

All ACT metapredicates are allowed on plot nodes. However, at most one goal expression on each node can be built using the same metapredicate. Furthermore, the metapredicates Achieve, Achieve-By, and Wait-Until are mutually exclusive: if one is used on a node, the other is prohibited. Empty plot nodes are allowed.

For purposes of ordering their execution, the metapredicates are partitioned into three groups, as shown in Table 3. The Context metapredicates, Test and Use-

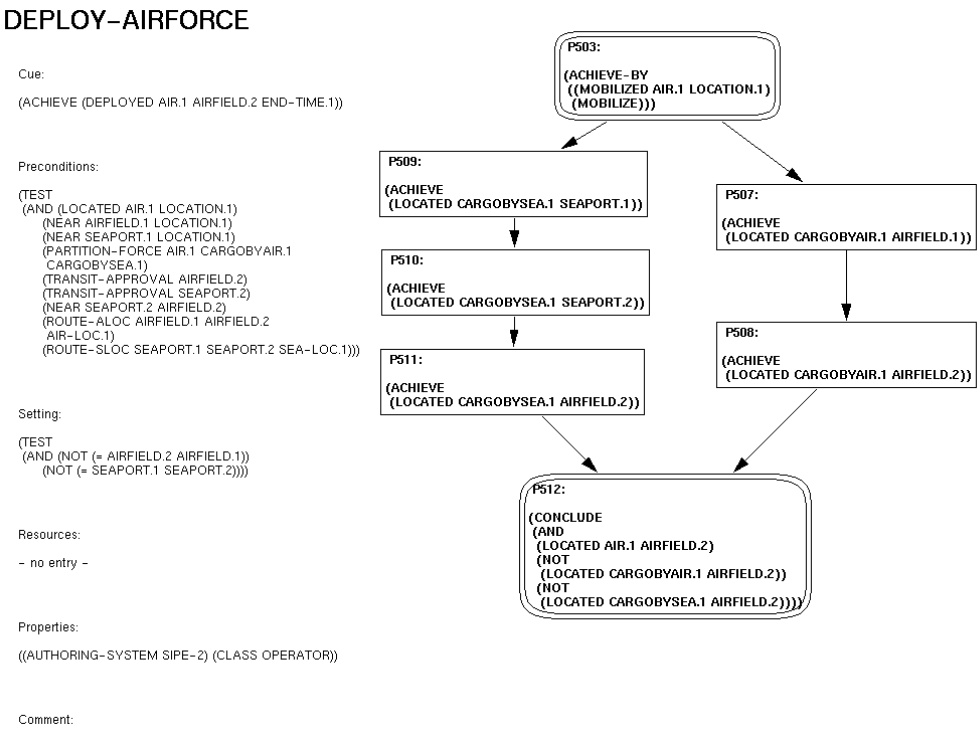

Figure 6: Deploy Airforce Act

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

Table 3: Metapredicate Grouping for Execution in Plot

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). The Action metapredicate for the node, either Achieve, Achieve-By, or Wait-Until (if there is one), is executed next. The Effects metapredicates, Require-Until and Conclude, are executed last by the execution system. Require-Until sets up a protection interval that must be maintained, and Conclude specifies any effects to be added to the system database.

4.3.2 Plot Topologies

A plot has a single *start node* (a node with no incoming arcs) but may have multiple *terminal nodes* (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, as in the `Iterative Factorial` Act in Figure 7. Execution of a plot requires successful execution of all nodes along some path from the start node to some terminal node. Successful execution of a node requires satisfaction of all of the node's goal expressions.

Plot nodes come in two types, *conditional* and *parallel*. Conditional nodes are drawn as single-border rectangles, and parallel nodes are drawn as double-border ovals. In Figure 6, nodes `P503` and `P512` are parallel, while all other nodes are conditional. Arcs coming into and going out of a parallel node are *conjunctive*, meaning that all of the arcs need to be executed. For the plot in Figure 6, `P503` specifies that the air force is to be mobilized. Since it is a parallel node, its successors can be invoked in either order or at the same time. `P512` joins the parallel actions and cannot begin execution until both of its incoming branches have completed. During planning, both branches are inserted into the plan as unordered subplans.

Arcs coming into and going out of a conditional node are interpreted as *disjunctive*, meaning that only one of the arcs need be executed. Consider first a disjunctive node with multiple successor nodes. A planner produces a conditional plan following this node. An executor executes the successor nodes until one is found whose goals are satisfied. At that point, execution 'commits' to the branch headed by that successor node and ignores all other branches. A disjunctive node with multiple incoming arcs can be executed as soon as one of its ancestor nodes has been successfully executed. As an example, consider the Act in Figure 7 for computing the factorial of a number in an execution system (this Act is not intended for use by a planner). After execution of `N11817`, the executor will nondeterministically choose one of its successor nodes for execution. If the goal expression on this choice is satisfiable, then the executor continues executing that branch. If not, it will try to satisfy the other successor. In this Act, one of the two successors will always succeed. In general, they might both fail and then `N11817` is said to fail.

Two consequences of the typing conventions should be noted: (1) if a node has zero or one incoming edge and zero or one outgoing edge, it is irrelevant whether it is a conditional or a parallel node, and (2) if one action is to be activated by only one of its incoming edges and must activate all of its outgoing 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 appear on either one of the nodes, while the other node would be empty. We considered alternative representations for plots that combine disjunctive incoming edges and conjunctive outgoing edges, but the complexity of such representations makes them hard to understand.

4.3.3 Temporal Reasoning

Any temporal relationship between two plot nodes can be represented in ACT. Many applications require only strict orderings between nodes (represented as arcs in the plot graph), but some require other temporal relationships (represented as `Time-Constraints`). For example, in the military domain, cargo offload teams should arrive at the destination either before or simultaneously with the first air transport. Moving the offload teams cannot be ordered before moving the first transport; they must have the option of traveling at the same time. While such constraints could

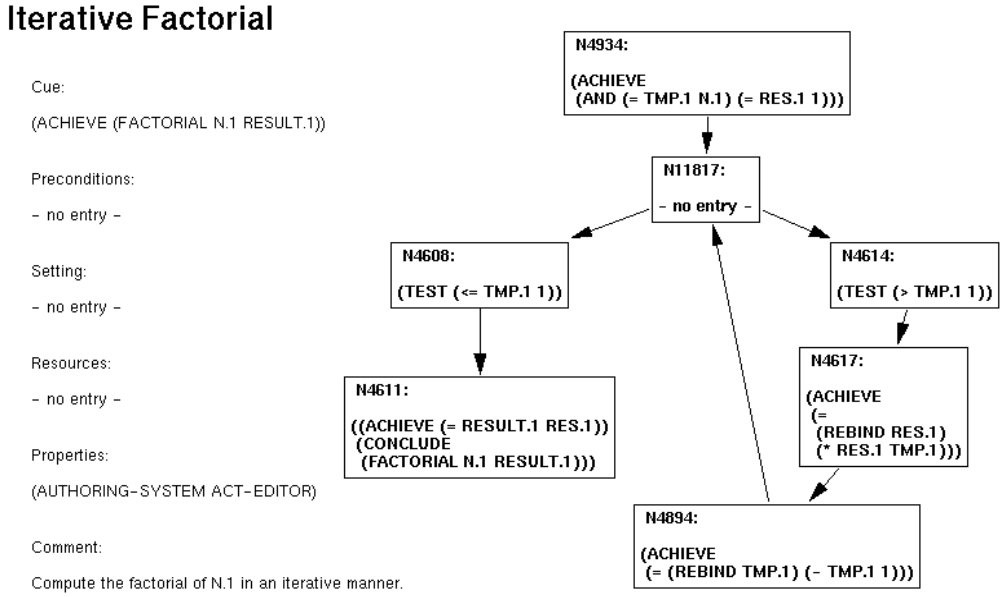

Figure 7: Iterative Factorial Act

be represented graphically, they are currently represented on the Time-Constraints property as described in Section 4.2.

Arbitrary temporal constraints are currently used by planners but not by executors. Satisfying many temporal constraints involves planning/scheduling the execution times of actions, e.g., computing when to start two actions so that they will end at the same time or be guaranteed to overlap. Execution systems could be extended to handle temporal constraints, but this brings up interesting issues of how much planning and scheduling the executor should do to satisfy such constraints, and whether it is still an execution system if it does significant planning and scheduling.

In addition to constraints between nodes, ACT allows the representation of time windows on individual plot nodes. 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 – between the endpoints of a pair of plan nodes – is represented as an 8-tuple composed of the minimum and maximum distance between the start of the first node and the start of the second node, and likewise the minimum and maximum start-finish, finish-start, and finish-finish distances. Section 5 describes the processing of temporal constraints in SIPE–2.

4.4 Variables

ACT uses *typed variables*. In particular, the name of the variable indicates a class to which any instantiation of the variable must belong. e.g., `airplane.1` is a variable for objects in the class `airplane`. By default, variables are treated as *logical*, meaning that they denote a single, fixed object. As such, they can be bound to at most one value during execution of an individual Act. (Each application of an Act is associated with its own local variables, so the variables in different applications of the same Act are distinct.) Logical variables can be contrasted with the *dynamic* variables employed in standard programming languages; a dynamic variable is rebound to different values throughout its lifetime.

It is sometimes necessary to use dynamic variables within Act plots. One such situation arises during the writing of an Act that must execute a loop a certain number of times. Another situation arises when an Act is to be used to monitor the value of some changeable feature of the environment (such as the pressure measured by a gauge) and to take certain steps when the value crosses some threshold. Such Acts are difficult to write using only logical variables.

For this reason, the ACT language supports rebinding of variables during execution of an Act, provided the user explicitly specifies where the rebindings are allowed. These specifications are made by applying the function `REBIND` to variables for which rebinding is to be allowed. For example, the expression

```
(ACHIEVE (= (REBIND X.1) (+ 1 X.1)))
```

expresses the goal of rebinding the variable `X.1` to the value that is 1 greater than the current value of `X.1`. This expression is different from

```
(ACHIEVE (= X.1 (+ 1 X.1)))
```

which seeks to equate a value with a value that is 1 larger, and hence will always fail. The Act `Iterative Factorial` in Figure 7 illustrates how the `REBIND` function can be used to specify plots with loops.

4.5 ACT Metapredicates

In our description of the ACT metapredicates, the term "the system" is used generically to refer to any particular implementation (whether a planner or executor) that makes use of the ACT language. The distinctions between executing and generating a plan previously discussed result in slightly different intended interpretations of some of the ACT metapredicates by planning and execution systems. Here, we describe these different interpretations. It is possible that small advantageous changes in these intended interpretations could be made based on special properties of the particular planning and execution systems being used.

ACHIEVE The use of Achieve on a plot node 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 the goals for which the Act can be used. Thus, Achieve in the Cue is used for subgoaling.

In the Precondition and Setting slots, the planner ignores an Achieve since it achieves goals in the plan, not during matching of these conditions. An Achieve can be used in the Precondition or Setting by the execution system to check for

the existence of other system goals. This capability can be useful when writing metalevel procedures that reason about the current goals of the system.

TEST This metapredicate specifies formulae whose truth-value is to be ascertained. While a planner must query its internal world model, an executor can query its database and/or sense the world. When a Test is in the Precondition or Setting slots, it determines if the Act should be executed after its Cue indicates its relevancy.

A Test in the Cue is used only by the executor and indicates that the Act can be used to actively test some condition by sensing the world. 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 formulae are true, and if not, it determines 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 can be used on plot nodes to specify formulae to be added to the system database. A Conclude in a Cue indicates an event-driven Act that responds to some new fact becoming true. Such an Act may effectively deduce consequences of an action by concluding further formulae.

WAIT-UNTIL This metapredicate appears only in plot nodes. In the executor, it specifies that execution of the Act is to be suspended until the indicated event occurs. The planner implements Wait-Until by ordering the node containing the Wait-Until after some other action that achieves the required condition.

ACHIEVE-BY This metapredicate appears only in plot nodes, and specifies the goals to be achieved as well as a set of Acts, one of which must be used to achieve the goals. This focuses the system on a limited number of means for achieving a goal.

REQUIRE-UNTIL This metapredicate appears only in plot nodes, and specifies a *protection interval*, namely a condition to be maintained until another indicated condition for terminating this requirement occurs. The 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 formula can be identified.

Planners generally have a mechanism for maintaining protection intervals. Planning algorithms can modify partial plans that violate 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. Each executor may have its own approach to handling failures; the implementation in PRS-CL is described in Section 5.2.

USE-RESOURCE This metapredicate in the Resources slot means each of its arguments is a resource throughout the plot. On a plot node, Use-Resource indicates resources required only at that node. Currently, these are reusable resources (i.e., they are not consumed), and the system will prevent other simultaneous actions from using the same resource. Different types of resources are an obvious place for extending the ACT formalism. In the executor, the resources are allocated before the plot begins execution and are released when the Act either succeeds or fails. The planner constructs a plan without resource conflicts.

COMMENT This metapredicate can be used in any node or slot to provide documentation. It accepts a string as argument.

4.6 Examples

Three example Acts from the military domain illustrate how the ACT language can be used to represent the kinds of knowledge about actions that are typical of plan generation and execution systems. A full description of this domain can be found elsewhere [19, 4], although sufficient details are provided here for a full understanding of the examples. The examples show event-driven and goal-driven activities, applicability conditions, resources, specification of conditions to be maintained over an interval, parallel actions, and deduction.

Figure 6 shows the Act for deploying an air force that encodes the same knowledge as the SIPE–2 operator shown in Figure 1 and the PRS-CL KA shown in Figure 4. The Cue indicates that the Act can be employed for the goal of deploying an air force to a particular airfield by a certain time. The Precondition and Setting must be true in the world state before the operator can be applied. The Precondition requires the initial position of the air force to be known, and determines intermediate seaports and airports that are on known routes to the destination and that have transit approval. The Setting constrains the two airports and seaports to be different and partitions the air force into two subparts.

The Plot 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 an Act named Mobilize. The successor nodes can then be executed in any order or simultaneously. These nodes begin two sequences of conditional nodes that use Achieve metapredicates to cause 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 8 shows the Act Lookout-Red for establishing lookouts. Lookout-Red is one means to achieve a lookout, and its Precondition specifies it should be used only when the area is under code-red conditions. The Setting finds a suitable site for the lookout and the correct sector for the supporting air cover. There is a Use-Resource metapredicate in the Resources slot that requires the availability of a `tfighter` resource for the Act to be applicable.

The plot of Lookout-Red first uses an Achieve metapredicate to move 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 be

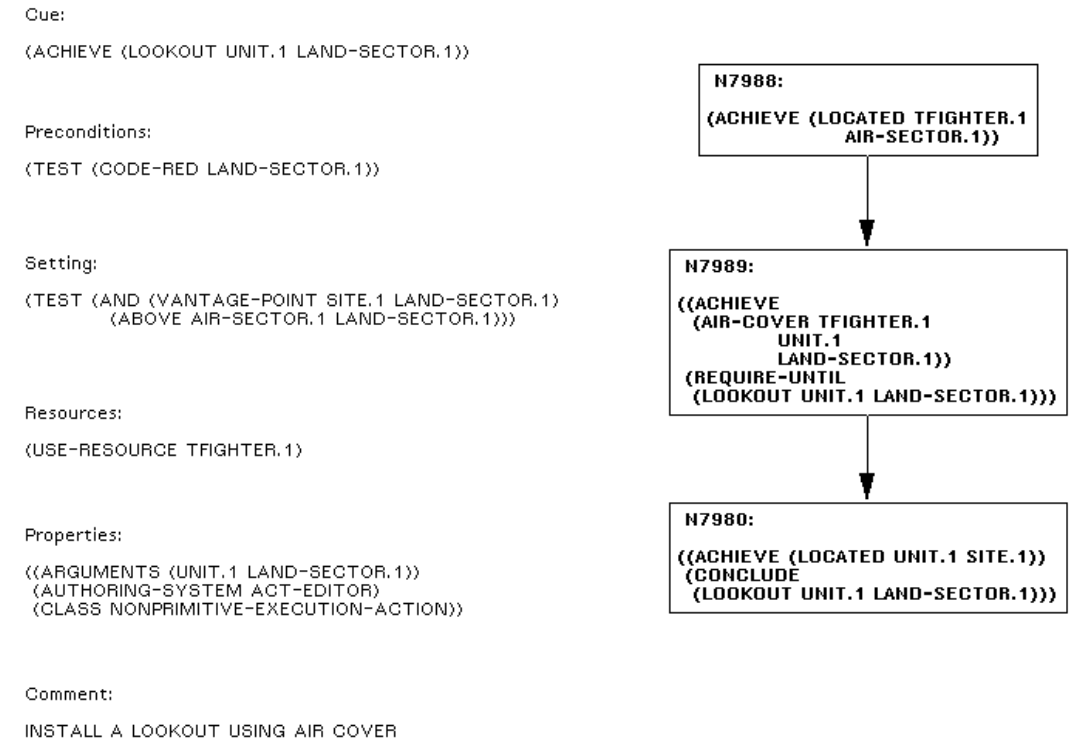

Figure 8: An Act for Establishing a Lookout

maintained until the lookout has been achieved. If the air cover is terminated for some reason before the lookout is established, the executor will use appropriate Acts to respond to the violation, while the planner will modify the 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 record that the lookout has now been established (which terminates the Require-Until condition).

Knowledge for deducing the context-dependent effects of actions is also commonly used by planning and execution systems. Such a capability greatly enhances the expressive power of a planning system, and is a feature of SIPE–2. ACT supports this type of knowledge. For example, the Located-sector-up Act in Figure 9 deduces 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 could be used by both the planner and executor to update their world models whenever an object is moved. The Cue of Located-sector-up specifies an event-driven Act that responds to new facts about the location 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

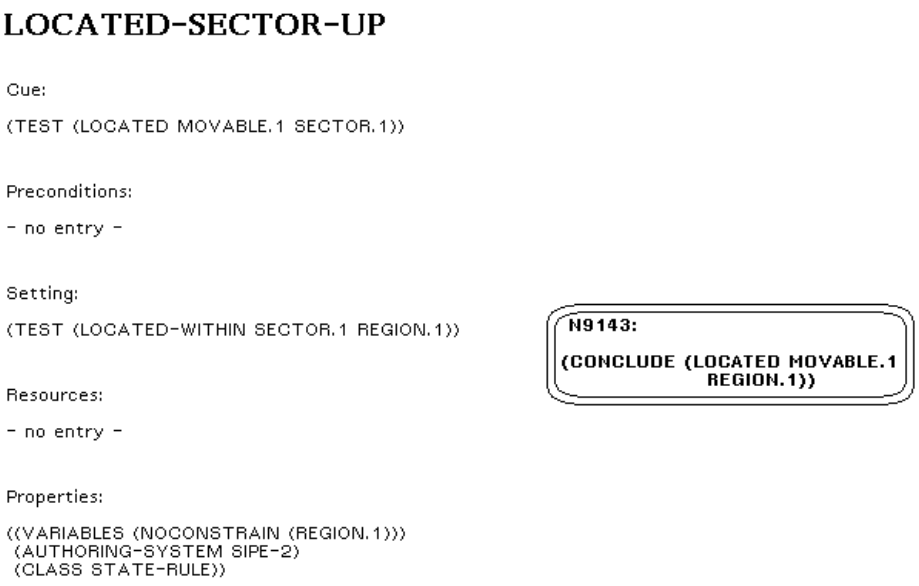

Figure 9: An Act for Deducing Locations

located at `region.1`. Similar rules deduce that the object is no longer located at its previous sector or region.

5 Implementing ACT

The ACT formalism has been implemented in the Cypress system. Cypress is based on versions of SIPE–2 and PRS-CL extended to support ACT; its architecture is depicted in Figure 10.[5] Cypress provides a framework in which to create taskable, reactive agents and supports the generation and execution of complex plans with parallel actions, the integration of goal-driven and event-driven activities during execution, and the use of replanning to handle run-time execution problems. Details on Cypress can be found elsewhere [20]; here we describe the implementation and use of ACT within the system.

For efficiency, PRS-CL and SIPE–2 employ their own internal representations for plans and actions. Cypress therefore uses ACT as an *interlingua* that enables these two systems to share information. Cypress includes translators that can automatically map Acts onto SIPE–2 and PRS-CL structures, along with a translator that can map SIPE–2 operators and plans into Acts. Using ACT, PRS-CL can execute plans produced by SIPE–2 and can invoke SIPE–2 in situations where run-time replanning is required. The ACT-Editor subsystem supports the graphical creation and display of Acts. The Gister-CL subsystem implements a suite of evidential reasoning tech-

[5] In particular, Cypress = SIPE + PRS. SIPE–2, PRS-CL, ACT-Editor, Gister-CL and Cypress are trademarks of SRI International.

Figure 10: The Architecture of Cypress

niques that can be used during both planning and execution to analyze uncertain information about the world and possible actions [16]. For example, Gister-CL could be used to reason about uncertain information in order to choose among candidate Acts in either the planner or executor.

In contrast to many other agent architectures, planning and execution operate asynchronously in Cypress, in loosely coupled fashion. This approach makes it possible for the two systems to run in parallel, even on different machines, without interfering with the actions of each other. In particular, PRS-CL remains responsive to its environment during plan synthesis.

5.1 Adequacy of ACT for Cypress

The ACT formalism supports the representational requirements for PRS-CL KAs and SIPE–2 operators, deductive rules, and plans. When translating a SIPE–2 operator to an Act, the purpose becomes a Cue with an Achieve metapredicate, the trigger (of a deductive rule) becomes a Cue with a Conclude metapredicate, the precondition becomes a Test metapredicate in both the Precondition and the Setting, and constraints on variables become a Test metapredicate in the Setting. PRS-CL KAs can be translated to ACT as follows: the "invocation condition" becomes both Test and Achieve metapredicates in the Cue, the "context" becomes both Test and Achieve metapredicates in both the Precondition and the Setting, and the "effects" will be encoded by Conclude metapredicates in the plot. The examples in Section 4.6 show event-driven and goal-driven Acts, applicability conditions, resources, specification of conditions to be maintained over an interval, and deduction.

The SIPE-to-ACT translator can translate all SIPE–2 operators to ACT, as well as translating fully instantiated plans. 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 ACT-to-PRS translator converts Acts into a representation that can be interpreted and executed in bounded time under the control of PRS-CL, thus making all SIPE–2 operators accessible to PRS-CL. We have not implemented a translator from KAs to ACT, as PRS-CL does not create Acts for use by other systems.

5.2 Extending SIPE-2 and PRS-CL for ACT

Since ACT generalizes the internal representations of SIPE–2 and PRS-CL, we extended these systems to support it. Fewer extensions were made to SIPE–2, since it does not use Acts provided specifically for the executor. However, the executor must execute plans generated by the planner, and this required several extensions to the version of PRS described in the literature [9], resulting in PRS-CL.

Acts permit the representation of a number of important constructs not supported by the original PRS. For instance, the metapredicates Use-Resource, Require-Until, and Achieve-By have no counterparts in KAs. Furthermore, ACT plots provide a more general representation of actions than is allowed by KAs, which do not support parallel execution of actions within individual KAs. PRS-CL executes directed, possibly cyclic, graphs of actions that support parallel topologies, including all SIPE–2 plans and operators. This was a complex extension to PRS, as it involved a major overhaul of the run-time representation structures and the introduction of control mechanisms for activating/deactivating parallel execution threads. PRS-CL includes a limited resource-handling capability to support the Use-Resource metapredicate. Resources can be allocated and released, but only for the entire scope of an Act. This supports the use of Use-Resource in the Resource environment condition of an Act; Use-Resource in plot nodes is not currently supported by PRS-CL.

PRS-CL supports the Require-Until metapredicate, which can be used to specify that a required condition (`req-wff`) remain true until some termination condition (`term-wff`) is satisfied. PRS-CL provides the Require-Until metapredicate with an *active maintenance* semantics, which allows the required condition to be temporarily violated without leading to a complete failure of the goal. Doing so enables the use of *repair procedures* that can be applied in order to actively re-establish the required condition. Repair procedures are generally domain-specific Acts designed for fixing individual conditions. When the `req-wff` of a Require-Until is initially detected as unsatisfied, PRS-CL will post a repair goal of the form `(ACHIEVE (REPAIR req-wff))`. If no repair Acts have been specified or none succeed in re-establishing `req-wff`, the Require-Until is then considered to have failed.

A Require-Until succeeds in PRS-CL when either `req-wff` is satisfied when `term-wff` becomes satisfied, or the Act containing the Require-Until terminates with `req-wff` satisfied. 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-CL 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.

The original PRS allows the specification of goals but not any information indicating which KAs should be used to accomplish those goals. To support the Achieve-By metapredicate, the PRS-CL interpreter loop was modified to filter candidate Acts to select only from among those specified by the Achieve-By. Both SIPE–2 and PRS-CL were modified to use the typed variable syntax of ACT.

SIPE–2 already supported the ACT metapredicates: it translates Use-Resource directly to its resource construct, translates Wait-Until to its `external condition` construct, translates a Require-Until using its `protect-until` construct combined with using its plan-critic algorithms to modify any partial plan that violates a Require-Until so that the final plan will satisfy all Require-Untils, and translates Achieve-By to a `process` node for singleton Acts and a `choiceprocess` node when more than one Act is given in the Achieve-By.

The only major extension to SIPE–2 was to support the Time-Constraints property. At certain intervals in the plan generation process (the frequency can be optionally changed), SIPE–2 will translate its current actions, ordering links, and temporal constraints for processing by General Electric's Tachyon system. Tachyon [2] is an efficient implementation of a constraint-based model for representing and maintaining qualitative and quantitative temporal information. Tachyon propagates the constraints from SIPE–2 and combines them with the "commonsense" constraints that it represents internally, returning an updated set of time windows on the actions that are inserted into the plan [3].

The Appendix details portions of the ACT syntax not supported by SIPE–2 and PRS-CL. Three limitations are important enough to mention here. First, the Time-Constraints property is ignored by PRS-CL, although PRS-CL could be extended to support most temporal constraints (see the discussion in Section 4.3.3). Second, using an Achieve metapredicate in the Precondition or Setting slots will match only if PRS-CL has posted a matching goal during the current execution cycle, making this capability complicated to use properly. Third, the `REBIND` function is not supported by SIPE–2, can be used only in plot nodes, can apply only to variables that do not appear in gating slots, and can be used only for goals of the form

`(ACHIEVE (= (REBIND X.1) <expression>))`.

5.3 Using Acts

ACT is now the preferred representational medium for encoding knowledge about actions for use by either SIPE–2 or PRS-CL. One consequence of this shared representation is that the two systems can now be used cooperatively to complete the planning/plan-execution cycle. The military operations domain demonstrated SIPE–2 and PRS-CL cooperating on the same problem for the first time.

In nontrivial domains, the complexity of plans and knowledge about actions requires a graphical interface to input Acts, understand generated plans, and monitor system behavior. We have implemented the ACT-Editor for displaying, editing, and inputing Acts. Acts generated by SIPE–2 can be quite complex; for example, a typical plan in the military domain produces an Act with over 200 plot nodes in it. The ACT-Editor includes a simplifier that streamlines the logical structure of an Act,

eliminating both unnecessary plot nodes and redundant ordering links. In addition, a user can vary the amount of detail to be presented on each plot node — an essential feature since plot nodes generated by SIPE–2 often have several metapredicates with large goal expressions. The ACT-Editor effectively provides a graphical knowledge editor for both SIPE–2 and PRS-CL, and is described in detail elsewhere [20].

A typical use of Acts in Cypress works as follows. The plans generated by the planner are translated to Acts for execution. The executor executes the plans produced and uses Acts to respond to events, perform lower-level actions that have not been planned, and invoke the planner when replanning is required.

While both the planner and executor 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 most minute detail. Similarly, it is often not desirable for the executor to respond to certain high-level goals without a plan. For example, a reactive system should not attempt to implement a Desert Storm-sized operation without first having a strategic 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 abstraction to serve as an interface level for the planner and executor. The planner plans down only to this level to produce a plan, while the executor will accept plans at that level and attempt to execute them using actions from that level of abstraction or below. The level of interface is currently encoded into the Acts themselves using the `class` property to specify which Acts are to be used by the planner and/or executor. For example, the Deploy Airforce Act from Figure 6 is used only by the planner, the Lookout Act from Figure 8 is used only by the executor, and the Located-sector-up Act from Figure 9 is used by both.

Once the planner has produced an Act, the execution is initiated by posting a goal that matches the Cue of the plan Act. The executor will then recognize that the newly loaded Act can be used to solve the goal and will begin executing it. This Act serves as a kind of outline for establishing the top-level goal, with the executor choosing appropriate lower-level Acts to satisfy subgoals in the plan. During execution of the plan, the executor monitors the world for any events that might trigger additional activity, such as new goals being posted or changes in the world that require immediate attention. Provided that Acts have been written to address such events, the executor will respond by simultaneously executing those Acts in addition to the original plan. Certain failures will cause the executor to invoke the planner to do run-time replanning. The planner attempts to modify the failed plan and ensures that the future consequences of the new plan do not interfere with the still-active execution threads of the original plan [20].

6 Comparison to Other Work

The complexity of the plans and the necessity of using them is what differentiates our work from previous approaches to the integration of planning and execution. Most of the previous work on combining planning and reactive execution has been driven by robotics applications. Systems for mobile robots generally emphasize execution, using only precomputed plans or simple plans generated at run-time. Such systems often

operate without a plan and use a plan only when the planner happens to produce a plan that will work in the current situation. In contrast, domains such as military operations planning require that the planner exercise greater influence over the execution system. The planner should generate a plan that will serve as the principal guidance for the executor, which will generally have no idea how to accomplish the top-level goals appropriately until the planner has generated a plan. There may be serious consequences if the plan is ignored.

Lyons and Hendricks [14] 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 that these incremental adaptations *improve* system performance before they are added to the reactor. The RAP system [6] also uses simple plans to modify the reactive control of a robot. These and similar approaches use adaptive modifications to rule sets, rather than employing the full look-ahead reasoning of a generative planner. The ATLANTIS system [7] is a heterogeneous, asynchronous architecture for controlling mobile robots. Its deliberator employs a "simple linear planner" and its controller does not understand the plan representation (thus allowing different planners to be used). Rather, the planner provides some previously designated inputs that the sequencer can use.

These approaches suffice for robot applications, where complex plans are not necessary. In more complex domains, high-level plans are essential. Approaches that adapt the executor to improve its performance may not be feasible because of the complexity of having many parallel execution threads. In our implementation, the executor will suspend problematic execution threads, request the planner to modify the plan, continue execution of nonproblematic actions while waiting for the planner, implement lower-level behaviors while waiting, and restart execution when a modified plan is received. Hanks and Firby [11] discuss the issues involved in integrating planning and execution, including issues not addressed in this paper, such as reasoning about the utility of deliberation. They do not present an implemented formalism beyond RAP.

McDermott [15] describes RPL, a Lisp-like programming language for writing programs that will produce goal-directed behavior and reactive response in a robot executing the program. This work is tailored to the robot domain, and RPL is obscure for use as a plan representation with which humans will interact. RPL is suited for producing specialized modules that implement a particular behavior, while ACT is designed as an interlingua for general-purpose functional modules. Rex [12] is another system designed for programming behaviors for intelligent reactive systems.

The motivations for ACT are similar to those of the KRSL language [13]. In fact, ACT has been one of the formalisms driving the design of the KRSL specification for plans. There are a number of differences, however. ACT 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. Furthermore, KRSL does not yet provide the same degree of support for execution activities as does ACT.

7 Conclusion

The ACT formalism supports the representation of knowledge necessary for the generation and execution of plans suitable for use in complex and dynamic environments. ACT serves as an interlingua that supports heterogeneous combinations of AI technologies in planning and reactive control.

The use of ACT as an interlingua to represent knowledge about actions for both SIPE–2 and PRS-CL in practical applications attests to the practicality of the language. Acts have enabled SIPE–2 and PRS-CL to cooperate on the same problem for the first time, using a common library of action representations. The planner generates and modifies plans while the executor executes the plans produced and uses Acts to respond to events, performs lower-level actions that have not been planned, and invokes the planner when replanning is required.

The development of SIPE–2 and PRS-CL has been driven by their applications to numerous problem domains; similarly, the development of ACT has been driven by the representations of these two systems. The ability to represent all the constructs in PRS-CL and SIPE–2 implies that the ACT formalism is sufficient for a wide range of interesting problems, since both these systems have been applied to several practical problems. The integrated planning and execution demonstrated in the military problem shows that ACT has reasonable computational properties, as well as being reasonably expressive. For this application, ACT was used to represent knowledge about actions and plans containing several hundred plot nodes.

ACT is not specific to our implementation and is a general-purpose representation language that could be used to share knowledge between many different execution and planning systems. However, there are many types of knowledge that could be of use to an integrated planning-execution system that are not yet included in ACT. ACT is an evolving entity that will be extended as additional features are required. Possible future extensions include goal expressions that are matched in the presence of uncertainty, actions with uncertain effects, knowledge about the utility of actions or Acts, reasoning about the beliefs of other agents, partial achievement of goals, and extended resource reasoning and scheduling capabilities.

Acknowledgments

The research described in this paper was supported by ARPA and Rome Laboratory as part of their Planning Initiative under Contracts F30602-91-C-0039 and F30602-90-C-0086. The people who designed and implemented the ACT formalism in Cypress are David Wilkins, Karen Myers, John Lowrance, Leonard Wesley, Janet Lee, and Peter Karp. Marie Bienkowski, Marie desJardins, and Roberto Desimone designed and implemented the processing of temporal constraints and the military operations planning operators.

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] Marie Bienkowski and Marie desJardins. Planning-based integrated decision support systems. In *Proceedings of the 1994 Conference on AI Planning Systems*, Chicago, IL, 1994.

[4] Marie Bienkowski, Marie desJardins, and Roberto Desimone. Generative planning to support military operations planning. In *1994 Symposium on Command and Control Research and Decision Aids*, Monterey, CA, 1994.

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

[6] 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.

[7] Erann Gat. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proceedings of the 1992 National Conference on Artificial Intelligence*, pages 809–815, American Association for Artificial Intelligence, Menlo Park, CA, 1992.

[8] Michael P. Georgeff, F. Félix Ingrand, and Amy L. Lansky. *Procedural Reasoning System User Guide*. SRI International Artificial Intelligence Center, Menlo Park, CA, 1989.

[9] 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, 1989.

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

[11] Steve Hanks and R. James Firby. Issues and architectures for planning and execution. In Katia P. Sycara, editor, *Proceedings of the Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 59–70. Morgan Kaufmann Publishers Inc., San Mateo, CA, November 1990.

[12] L. P. Kaelbling. An architecture for intelligent reactive systems. In *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*, pages 395–410. Morgan Kaufmann Publishers Inc., San Mateo, CA, 1987.

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

[14] 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.

[15] D. McDermott. Transformational planning of reactive behavior. Technical Report CSD-RR-941, Yale University, Department of Computer Science, 1992.

[16] Thomas M. Strat and John D. Lowrance. Explaining evidential analyses. *International Journal of Approximate Reasoning*, 3(4):299–353, July 1989.

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

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

[19] David E. Wilkins and Roberto V. Desimone. Applying an AI planner to military operations planning. In M. Fox and M. Zweben, editors, *Intelligent Scheduling*, pages 685–709. Morgan Kaufmann Publishers Inc., San Mateo, CA, 1994.

[20] David E. Wilkins, Karen L. Myers, John D. Lowrance, and Leonard P. Wesley. Planning and reacting in uncertain and dynamic environments. *Journal of Experimental and Theoretical AI*, 7(1):197–227, 1995.

Appendix: ACT Syntax

This appendix presents the syntax for defining Acts and documents the restrictions on Acts that are to be translated to SIPE–2 operators and PRS KAs.

ACT Specification

The following Backus-Naur Form (BNF) documents the syntax for ACT metapredicates accepted by the ACT-Editor and ACT-Verifier. For those unfamiliar with BNF, the form on the left of the symbol `::=` can be replaced by the form on the right. Items in the typewriter font (e.g., `item`) represent actual primitives to be used while italicized text (e.g., *sexprression*) defines primitives descriptively. Square brackets [ ] are placed around optional objects. The symbol | represents "or", the * represents any number of repetitions including zero, and the $^+$ represents any number of repetitions greater than one. Braces {} without * or $^+$ appended simply indicate grouping.

Logical Formulas

wff ::= (pred-name {term}*) | **(UNKNOWN** (pred-name {term}*)) |
(NOT wff) | **(AND** {wff}$^+$) | **(OR** {wff}$^+$)
term ::= simple-term | function | **(REBIND** variable)
simple-term ::= individual | variable
variable ::= {class} . {integer}
function ::= (fn-name {term}*)

pred-name ::= *the name of a predicate*
fn-name ::= *the name of a function*
individual ::= *a domain object*
class ::= *the name of a class*
integer ::= *a positive integer*

Metapredicates

meta-pred ::= test | conclude | achieve | achieve-by
| use-resource | wait-until | require-until

test ::= **(TEST** {wff | wff-list})
achieve ::= **(ACHIEVE** {wff | wff-list })
achieve-by ::= **(ACHIEVE-BY** {wff+acts | ({wff+acts}$^+$))
conclude ::= **(CONCLUDE** {wff | wff-list})
use-resource ::= **(USE-RESOURCE** {simple-term | ({simple-term}$^+$) })
wait-until ::= **(WAIT-UNTIL** {wff | wff-list })
require-until ::= **(REQUIRE-UNTIL** { wff | wff-pair })
wff-pair ::= (wff wff)
wff-list ::= ({wff}$^+$)
wff+acts ::= (wff ({act}$^+$))
act ::= *the name of an Act*

Plot Nodes The actions associated with plot nodes are specified using metapredicates. All metapredicates can be used on plot nodes. However, at most one instance of each metapredicate is allowed per node. Furthermore, the metapredicates `ACHIEVE`, `ACHIEVE-BY` and `WAIT-UNITL` are mutually exclusive: if one is used on a node, the other is prohibited. Plot nodes can contain a comment string. Finally, no variable mentioned in the environment nodes can appear within the scope of a `REBIND` operator on a plot node.

Environment Conditions (Slots) The *gating* slots, namely Cue, Setting, Test and Resources, are filled with metapredicates. Each gating slot has its own list of accepted metapredicates. As with plot nodes, at most one instance of each metapredicate is allowed per slot. Unlike plot nodes, certain environment nodes *require* the presence of some metapredicates. The constraints on the use of metapredicates for gating environment nodes is summarized here, building on the BNF notation defined above. All gating slots also support a Comment entry, specified as a string. Metapredicates in the environment slots cannot use the `REBIND` operator.

Cue	::= (test) | (achieve) | (conclude)
Preconditions	::= (test) | (achieve) | (test achieve)
Setting	::= (test)
Resources	::= (use-resource)

The Comment and Properties slots are *non-gating*. The Comment slot can be filled with a string that documents the Act.

The Properties slot is filled with a property list. The SIPE–2 recognizes two special properties: Variables for declaring variables, and Time-Constraints for specifying temporal constraints. The Variables property should have as its value a list of quantifier-pairs, each of whose first element is `existential` or `universal`. The Time-Constraints property specifies ordering constraints between plot nodes that cannot be represented by the precedence arcs of the plots. Two types of constraints are used: time windows on nodes and inter-node constraints. The syntax for these properties is presented below.

Values for Time-Constraints Property

TC-value	::= ({time-constraint}$^+$)
time-constraint	::= (allen-reln node node [{integer}-{integer}]) | (qual-relation (point node) (point node))
allen-reln	::= `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
plotnode	::= *a symbol*

Values for Variables Property

Var-value	::= ({var-decl}$^+$)
var-decl	::= (decl variable)
decl	::= `univeral` | `existential`

PRS Restrictions

The following restrictions are placed on Acts to be translated to PRS *knowledge areas (KAs)*.

- The `UNKNOWN` operator for construction of formule is not supported.
- `USE-RESOURCE` metapredicates on plot nodes are ignored.
- The `Time-Constraints` property is ignored.
- The `REBIND` function can be used only in plot nodes, can only apply to variables that do not appear in gating slots, and can only be used for goals of the form `(ACHIEVE (= (REBIND X.1) <expression>))`.
- The `CONCLUDE` metapredicate cannot be applied to either an explicit disjunction such as `(OR` P_1 $\ldots P_k$`)`, or an explicit disjunction embedded within a conjunction such as `(AND (Q) (OR` P_1 $\ldots P_k$`) )`. This restriction is necessary because (a) PRS does not support the insertion of disjunctive facts into its database, and (b) when given a conjunctive fact to be concluded, PRS adds the individual conjuncts into the database.

  (NOTE: PRS will only flag the use of explicit disjunctions written using the operator `OR`. Implicit disjunctions such as `(NOT (AND (P) (Q)))` are not recognized.)

In addition, the following conventions should be noted:

- An `OR` is equivalent to the first disjunct that succeeds (as in Lisp), rather than a logical disjunction that considers instantiations from all disjuncts.
- Metapredicates of the form

  `(ACHIEVE (`$\texttt{wff}_1$ $\ldots\texttt{wff}_k$`))`
  `(ACHIEVE (AND` $\texttt{wff}_1$ $\ldots$ $\texttt{wff}_k$`))`

  are equivalent for plot nodes. For environment nodes, the latter is translated to `(GOAL* (ACHIEVE (AND` $\texttt{wff}_1$ $\ldots$ $\texttt{wff}_k$`)))` while the former is translated to `(& (GOAL* (ACHIEVE` $\texttt{wff}_1$`)) ...(GOAL* (ACHIEVE` $\texttt{wff}_k$`)) )`. The analogous convention holds for the translation of metapredicates of the form `(TEST (`$\texttt{wff}_1$ $\ldots\texttt{wff}_k$`))`.

SIPE-2 Restrictions

The following are restrictions placed on Acts that are to be translated to SIPE–2 operators. The ACT->SIPE translator issues a warning message whenever a restriction is violated. A plot must have only one terminal node, and not contain loops (although loops in SIPE–2 can be represented by a special *parallel-loop* action in an Achieve-By metapredicate).

The system translates only Acts whose *class* property (in the Properties slot) is one of: `state-rule, causal-rule, init-operator, init.operator, operator, both.operator`. Acts with class `operator` correspond to action operators in SIPE–2 while Acts having any other of the above class properties correspond to SIPE–2 deductive rules. We use the term *nondeductive* to refer to the former class of operators and *deductive operator* to refer to the latter class.

- For logical formulae:
  - `AND` should not be nested inside `NOT`, `OR`, or `AND`.
  - `OR` should not be nested inside `NOT` or `OR`.
- Formulae containing `OR` are allowed in Test metapredicates only.
- The `REBIND` construct is not supported.
- The predicates $=$ $class$ $in-range$ $>$ $<$ $<=$ $>=$ are translated to SIPE–2 constraints. They should not be used inside an `OR`.
- An `OR` is equivalent to the first disjunct that succeeds (as in Lisp), rather than a logical disjunction that will consider instantiations from all disjuncts.
- For the Cue slot:
  - `TEST` is not supported.
  - `ACHIEVE` and `CONCLUDE` can be applied only to atomic formulas, negated atomic formulas, and conjunctive formulas.
  - Deductive operators must have a `CONCLUDE` metapredicate in the Cue, and all plot nodes except the start node are ignored. Nondeductive operators must have an `ACHIEVE` metapredicate in the Cue.
- `ACHIEVE` is ignored in the Precondition slot.
- The ACT->SIPE translator does handle the option of using the names of plotnodes in the Time-Constraints property.