# The Act Formalism

## Version 2.2

Karen L. Myers  David E. Wilkins
SRI International
Artificial Intelligence Center
myers@ai.sri.com  wilkins@ai.sri.com

**Working Document:** Revisions as of September 25, 1997

**Abstract**

This document describes Version 2.2 of SRI's Act formalism. The Act formalism is a domain-independent language for representing task networks whose actions manipulate both an external world and an internal database. It is intended to serve as an interchange language that will enable a broad range of action-related technologies to share information. The most recent version of the Act specification can be found at the URL:

`http://www.ai.sri.com/~act/act-spec.ps`

# 1 Overview

This document describes Version 2.2 of SRI International's *Act formalism*. The Act formalism is a domain-independent language for representing plans and action networks whose actions manipulate both an external world and an internal database. Act is designed to be an interchange language that will enable a broad range of action-related technologies to share information. In contrast to a number of recent efforts to define *ontologies* for action representations, Act grew out of an effort to enable mature planning-related technologies to interoperate [8] (namely a generative planner [6] and a reactive execution system [5]). Since that time, Act has been extended to support plans, interactions with a sophisticated temporal reasoning system [2], and use of Acts by another reactive execution system [3]. In the future, it is expected that Act will evolve to support the representational requirements of additional action-related technologies.

This document describes the Act formalism in brief; additional information on the background and semantics of Act can be found in [7]. Section 2 provides a high-level overview of Act representational constructs. Section 3 describes how plans at multiple levels of abstraction are represented, while Section 4 defines a BNF grammar for Act. Finally, Section 5 summarizes a set of functions for manipulating ASCII representations of Acts.

The Act syntax presented in this document supersedes specifications in any earlier documents (including [7], which corresponds to Version 1.0 of Act). Suggested extensions to the formalism would be welcomed. The most recent version of the Act is published at the URL:

`http://www.ai.sri.com/~act/act-spec.ps`

To support the use of Act, SRI has built the Act-Editor[4], a graphical tool for interactively viewing, creating, modifying, and verifying Acts. The Act-Editor is freely available; instructions for obtaining a copy of the system can be found at the URL: `http://www.ai.sri.com/~act`

# 2 The Act Formalism

The basic unit of organization in the Act formalism is an *Act*. Each 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. An Act can represent, among other things, a procedure, a planning "operator" [6], or a plan at one particular level of detail. Section 3 describes how multiple Acts are used to represent a single plan at multiple levels of abstraction.

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.

## 2.1 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 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 [7].

## 2.2 Act Metapredicates

The term *action metapredicates* refers to the metapredicates Achieve, Achieve-By, Achieve-All, and Wait-Until. Achieve directs the system to accomplish a goal by any means possible; Achieve-By is similar but specifies a restricted set of Acts that can be used to accomplish the task. Achieve-All directs the system to accomplish a set of Achieve metapredicates in parallel for all objects that satisfy a given pattern predicate. Wait-Until directs the system to wait until some specified condition holds.

The *Test* metapredicate specifies a formula whose truth value is to be ascertained. The *Use-Resource* metapredicate makes a declaration of resources that will be used by an Act, and hence that must be available for the Act to be applied. The *Require-Until* metapredicate designates conditions that must be maintained over a specified interval. The *Conclude* and *Retract* metapredicates designate changes to the internal database.

The distinctions between executing and generating a plan [7] result in slightly different intended interpretations of some of the metapredicates by planning and execution systems. Here, we describe these different interpretations and describe each metapredicate in more detail. 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.

**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.

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.

**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.

**ACHIEVE-ALL** This metapredicate appears only in plot nodes, and directs the system to accomplish a set of Achieve metapredicates in parallel. The first formula gives a goal template for the Achieve metapredicates to be generated, and the second formula is a pattern. One Achieve metapredicate is generated for each possible match of the pattern formula. It is assumed that at least one variable in the pattern formula occurs in the goal template formula.

**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.

**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.

**RETRACT** This metapredicate can be used on plot nodes to specify formulae to be deleted from the system database. For closed world predicates, `(retract P)` is equivalent to `(conclude (not P))`, where P is a single predicate instance. A Retract in a Cue indicates an event-driven Act that responds to some new fact becoming true.

**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 elsewhere [7].

**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 in the Resources slot 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.

| Environment Slot | Role |
|---|---|
| Name | identifier |
| Cue | the purpose of the Act |
| Precondition | gating conditions on the applicability of the Act |
| Setting | conditions for binding local variables |
| Resources | resource constraints |
| Properties | user-defined attributes, temporal constraints |
| Comment | documentation |

Table 1: Environment Conditions and Their Roles

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

Table 2: Metapredicates Allowed in Gating Slots

## 2.3 Environment Conditions

The Act environment conditions are defined as a set 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. Table 2 displays the metapredicates allowed in each of the 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 to describe the actions required for a robot to obtain an image of a object from a certain perspective (e.g., move to a given position, rotate the camera, tilt the camera, etc.).

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*.

**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 express conditions that are expected to be satisfied for some set of bindings. The Setting conditions are included expressly for the purpose of extracting a satisfying set of bindings for such variables. For example, in a domain where there is an active 'clock' (either real or simulated) that keeps track of the current time, the Setting may include a metapredicate `(TEST (TIME time.1))` to bind `time.1` the current time at.

**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. To apply an Act containing an expression of the form `(USE-RESOURCE (A B C))` in its Resources slot, it is necessary that the resources `(A B C)` be available. These resources would then be unavailable for use by other processes until execution of the Act completes.

**Properties**

The properties slot is a list of property/value pairs. 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 declaring variables as either existentially or universally quantified, and PRS-CL uses the property `Decision-Procedure` to designate an Act that is used for metalevel reasoning. The `Time-Constraints` property is used to encode ordering constraints on nodes, in terms of the Allen relations [1] (as discussed further in Section 2.5). 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 1 presents a sample Act as displayed by the Act-Editor. 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 2.4.

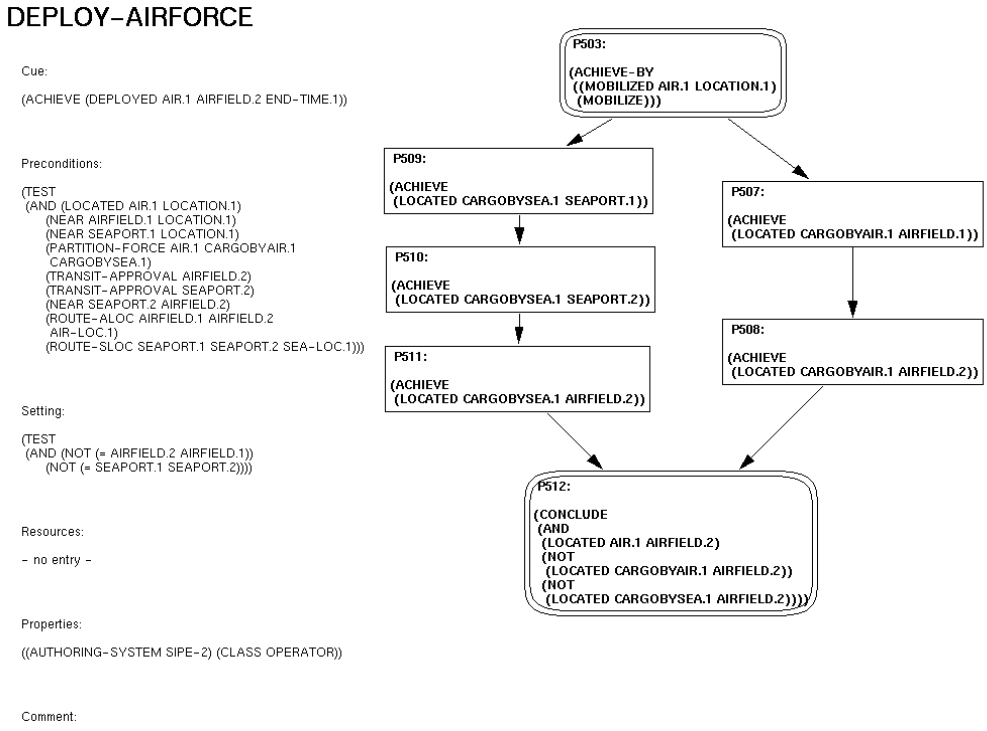


Figure 1: Deploy Airforce Act

## 2.4 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. Associated with each plot node is a list of goal expressions for the node, along with a predefined set of *attributes*.

### 2.4.1 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. The action metapredicates are mutually exclusive: if one is used on a plot node, the others are prohibited. Empty plot nodes are useful for beginning and/or ending plots that expand into parallel or conditional goals.

For purposes of ordering their execution, the metapredicates are partitioned into three groups, as shown in Table 3. The Context metapredicates, Test and Use-Resource, are executed first by the execution system. The action metapredicate for the node (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.

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

Table 3: Metapredicate Grouping for Execution in Plot

### 2.4.2 Attributes for Plot Nodes

Individual plot nodes contain a number of predefined attributes that complement the action-specifications provided by the metapredicates. The `Identifier` is a required attribute that provides an identifier for the node that is unique within the Act. The optional `Comment` attribute can be used to document the node. The `Time-Window` slot provides absolute temporal boundaries on the execution time for that node, in terms of the earliest and latest allowable start times, earliest and latest allowable end times, and minimum and maximum durations.

### 2.4.3 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 Act `Iterative Factorial` in Figure 2. 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 1, 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 1, `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 2 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.

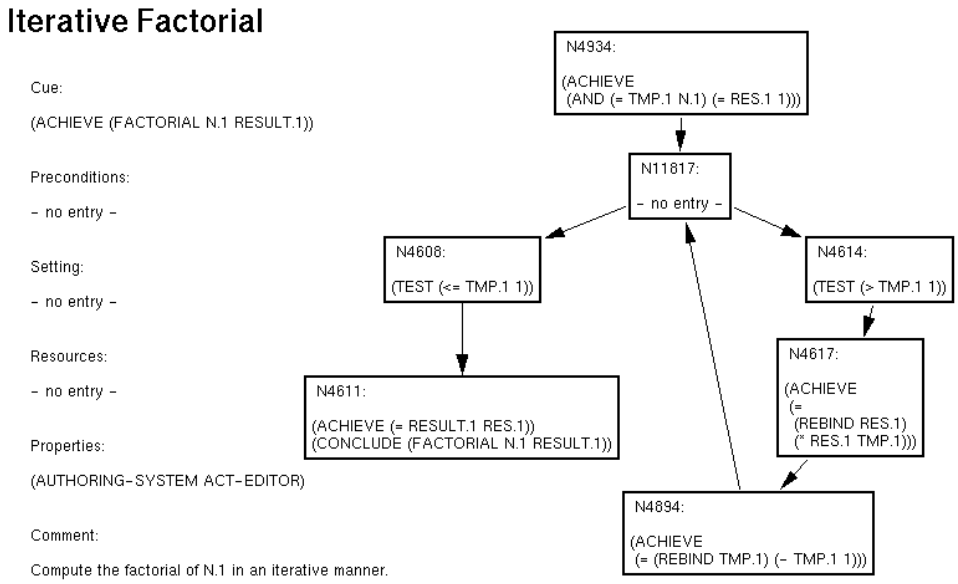


Figure 2: An Act for Computing Iterative Factorial

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.[1]

## 2.5 Temporal Reasoning

A wide range of temporal relationship between two plot nodes can be represented in Act. In terms of Allen relations, a plot node has the relationship `before or meets` with its successor plot node. Other temporal constraints can be represented using either the Time-Constraints property in the environment Property slot or the `Orderings` attribute of plot nodes. In addition, Act allows the representation of *time windows* on individual plot nodes (as noted above). A time window for a node specifies its earliest and latest allowable start times, earliest and latest finish times, and minimum and maximum durations. Section 4 provides the full syntax for these constraints.

## 2.6 Variables

Act uses *typed variables*. In particular, the name of the variable indicates a class to which any instantiation of the variable must belong. For example, `airplane.1` is a variable for objects in the

[1]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.

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 2 illustrates how the `REBIND` function can be used to specify plots with loops.

`REBIND` is not allowed in gating environment slots.

# 3 Representing Plans

An Act can represent a plan at one particular level of detail, or a "slice" of a plan at one point during its development. However, an integrated planning and execution system needs a richer representation of a plan than just the final action network. Generally, plans are constructed at multiple levels of abstraction and replanning during execution requires knowledge of all levels of a plan and the relationship between nodes at different levels.

## 3.1 Tasks, Plans, and Action Networks

To represent such plans, Act distinguishes three concepts depicted in figure 3. A *task* is a set of objectives, the current situation, and other information, assumptions, requirements and constraints affecting the problem to be solved. For example, the requirement that a certain action must be in any acceptable plan would be a constraint on the task.

A number of alternative *plans* can be produced for a given task. Each plan can be at multiple levels of abstraction and consists of the whole set of information about the plan. In particular, a plan is composed of a number of increasingly detailed *action networks*, and a set of *assumptions*. An action

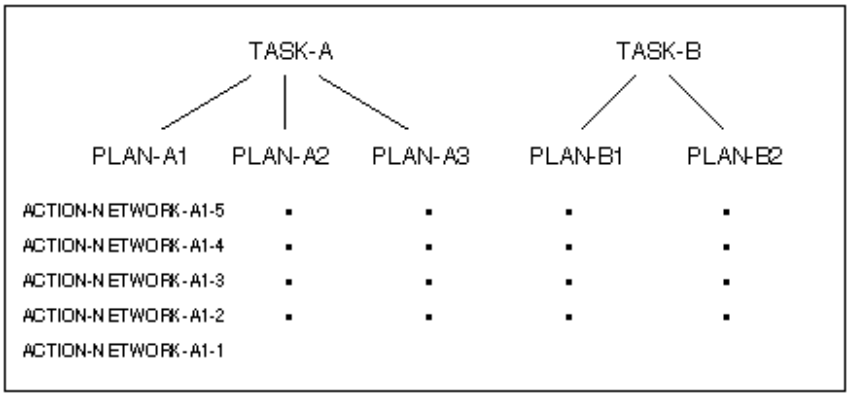


Figure 3: The structure of plan information: Each planning *task* can have several alternative *plans*, which in turn are composed of increasingly detailed expansions of the plan called *action-networks*.

network can be represented as an Act, and is a description of the plan at some point it its development. A plan will contain a set of Acts as well as relationships among the nodes in different Acts.

Except for the first action network in a plan, each action network will specify a *parent* action network. An action network expands upon or refines its parent action network. Figure 3 simplifies plans in that it implies a linear list of action networks — in fact, the action networks are structured as a tree by means of the parent relationship.

## 3.2 Open Issues

The Act formalism, by design, takes no position on many issues that are likely to be planner or domain specific. In particular, the decision about when a developing plan has changed enough to require definition of a new action network is left to the user. In order to coordinate their activities, all participants in a planning process will often need to understand what properties of an action network may change before a new action network is defined. Another decision left to the user is when a subtree of action networks will be named as a separate plan. For example, one could name every path through the tree of action networks as a separate plan, effectively reducing each plan to an ordered list of action networks.

**Plan Information** There is an open issue about what types of information should be represented in an Act action network or plan. Currently, Act does not attempt to represent the state of an entire planning process, but merely the results of the plan generation. For example, to describe a planning process, one would describe the search space, denoting parts of the space that have already been searched and parts that remain to be searched. Similarly, one could keep a list of flaws in the plan that must be corrected. Such information is not currently part of Act.

The information an action network will contain beyond that specified in an Act is not completely determined, and will be extended as Act plans are used more extensively. Certain relationships will be designated as part of Act, and may be stored on the properties and setting slots of the Act. Examples of properties currently defined for action networks include `PLAN`, `TASK`, `PARENT`, `PARENT-GRAPH`, and `CHILDREN` (described in Section 4).

One area in which Act is likely to be extended is to represent more information about constraints on variables in an action network. Some constraints are already represented in the setting, but more constraints generated during plan generation could also be represented. There are also open issues here; for example, if these constraints can be deduced by analysis of the action network, then it may not be necessary to explicitly represent them.

**Assumptions** Currently, no syntax has been defined for specifying assumptions. The syntax will develop over time as this capability is used; for now, users can use any lisp s-expression for their own purposes.

There is some freedom in choosing whether an assumption (which currently includes constraints and other information) is attached to a task or a plan. When doing a series of "what-ifs", for example, it may be more intuitive to attach each different assumption to a plan and have one task, rather than have a set of different tasks. The Act formalism remains flexible in this regard — allowing properties to be attached to plans or tasks as is natural for the domain. However, all users of a particular plan must have a common understanding of the properties used.

### 3.3 Specifying components of plans and tasks

When tasks specify their component plans, or plans specify their component action networks, the component can be denoted either by the Act formalism expression that defines it, or by its name. When a name is used, it is assumed the parser of the expression already has a representation for that object, or knows how to obtain one. This allows an expression to specify or update one component without needing to specify the entire plan structure.

An expression for an action network (Act) can be used without being embedded in a task and plan expression. In this case, the properties slot of the Act should specify the properties `PLAN` and `TASK` whose values are a plan name and a task name, respectively. The parser of the expression can then associate the Act with the correct plan and task. An expression for a plan that is not embedded in a task can also specify a task name, as specified in the syntax.

## 4 Act Syntax

The following Backus-Naur Form grammar (BNF) documents the syntax for the Act formalism. Items in the typewriter font (e.g., `item`) represent actual primitives to be used while italicized text (e.g., *s-expression*) defines primitives descriptively. Square brackets [ ] are placed around optional objects. The symbol | represents alternatives, $^*$ represents any number of repetitions including zero, and $^+$ represents any non-zero number of repetitions. Braces $\{\}$ without $^*$ or $^+$ appended are used to indicate grouping. The notation $<\ >$ represents an arbitrary ordering of the embedded elements.

The given grammar defines the full syntax for Act. It is not expected that all implementations of Act will necessarily support the full syntax. For example, limitations on the implementation of Act for PRS-CL and SIPE–2 are described in [7].

The Act-Editor supports the entire syntax for Acts. Although it does not currently provide graphical support for tasks and plans, it will read task and plan expressions and extract the Acts. Someone

using the Act-Editor need not know the Act syntax above the metapredicate level, because the system builds Acts from metapredicates and provides a graphical display where only metapredicates are displayed in Act syntax. If the structure-based editing option is used, even the syntax of metapredicates is constructed by the system. For this reason, the Act-Editor is the preferred mechanism for constructing Acts.

Acts must still be generated by translators and communicated between agents, and the ASCII syntax presented here supports such capabilities. The Act-Editor can produce expressions in this syntax for its graphically displayed Acts.

### 4.1 Tasks

An example expression for a task might look like:

```
(TASK task24 (PLANS plan32 plan44) (ASSUMPTIONS tbd))
```

task ::= (TASK task-id tasksslots)
tasksslots ::= < [(PLANS plan-spec* )]
[(OBJECTIVES obj-spec* )]
[(ASSUMPTIONS assum-spec)] >
plan-spec ::= plan-id | plan
obj-spec ::= action-mp *(for now, richer language to be determined)*
assum-spec ::= wffs *(for now, richer language to be determined)*

### 4.2 Plans

The action networks in a plan are structured as a tree by means of the parent relationship. Except for the first action network in a plan, each action network will specify a *parent* action network in its properties slot. Similarly, an action network may specify a a *children* property that points to the action networks derived from it. Each action network adds more detail to its parent. An example expression for a plan might look like:

```
(PLAN plan32 (ACTION-NETWORKS act1 act2 act3) (ASSUMPTIONS tbd))
```

Parent/child relationships between nodes of action networks will be specified within the scope of a plot node of an Act (action network) containing the child nodes.

Subplans are used when more than one plan must be generated to solve a task and the plans are not merged, but passed on as a multipart solution. It remains to be specified how nodes in a subplans will link to information in another subplan, although this will be accomplished through properties of Acts and their plot nodes.

plan ::= (PLAN plan-id planslots)
planslots ::= < [(ACTION-NETWORKS act-spec* )]
[(SUBPLANS plan-spec* )]
[(TASK task-id)]
[(ASSUMPTIONS assum-spec* )] >

plan-spec ::= plan-id | plan
act-spec ::= act-id | act

## 4.3 Acts

Figure 4 presents an example of the ASCII representation for Acts. This Act includes examples of all components, such as environment slots and plot nodes, as well as example metapredicates.

If an Act is not embedded in a task and plan expression, the properties slot of the Act should specify the properties `PLAN` and `TASK` whose values are a plan name and a task name, respectively. The properties slot will also specify a `PARENT` and possibly `CHILDREN`, as described earlier, will record any assumptions in the properties `PLAN-ASSUMPTIONS` and `TASK-ASSUMPTIONS`, and will record task objectives in `TASK-OBJECTIVES`.

act ::= (act-id (`ENVIRONMENT` envslots) (`PLOT` plotnode$^{+}$ ))

## 4.4 Environment Slots

The *gating* slots, namely Cue, Setting, Precondition, and Resources, are filled with slot-specific metapredicates. At most one instance of each metapredicate is allowed per slot. 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. SIPE–2 recognizes two special properties: Variables for declaring variables, and Time-Constraints for specifying temporal constraints. The syntax for these properties is presented below. Note that the Cue is the only required environment slot.

envslots ::= < (`CUE` cue-entry [comment])
[(`PRECONDITIONS` [precond-entry] [comment])]
[(`SETTING` [setting-entry] [comment])]
[(`RESOURCES` resource-entry$^{*}$ [comment])]
[(`PROPERTIES` property$^{*}$ )]
[(`COMMENT` [string])] >
cue-entry ::= test | achieve | conclude
precond-entry ::= test | achieve | (<test achieve>)
setting-entry ::= test
resource-entry ::= use-resource
property ::= (symbol val) | temporal-prop | var-prop

## 4.5 Plot Nodes

The actions associated with plot nodes are specified using the full set of Act metapredicates. However, at most one instance of each metapredicate is allowed per node. In addition, the action metapredicates

```
(DEPLOY-AIRFORCE
 (ENVIRONMENT
  (CUE (ACHIEVE (DEPLOYED AIR.1 AIRFIELD.2 END-TIME.1)))
  (PRECONDITIONS
    (TEST (AND (LOCATED AIR.1 LOCATION.1)
               (NEAR AIRFIELD.1 LOCATION.1)
               (NEAR SEAPORT.1 LOCATION.1)
               (PARTITION-FORCE AIR.1 CARGOBYAIR.1 CARGOBYSEA.1)
               (TRANSIT-APPROVAL AIRFIELD.2)
               (TRANSIT-APPROVAL SEAPORT.2)
               (NEAR SEAPORT.2 AIRFIELD.2)
               (ROUTE-ALOC AIRFIELD.1 AIRFIELD.2 AIR-LOC.1)
               (ROUTE-SLOC SEAPORT.1 SEAPORT.2 SEA-LOC.1))))
  (SETTING (TEST (AND (NOT (= AIRFIELD.2 AIRFIELD.1))
                      (NOT (= SEAPORT.1 SEAPORT.2)))))
  (PROPERTIES
    (AUTHORING-SYSTEM SIPE-2)
    (CLASS OPERATOR)))
 (PLOT
  (C19  (TYPE PARALLEL)
    (ORDERINGS (NEXT P87)))
  (P87  (TYPE CONDITIONAL)
    (CONCLUDE (AND (LOCATED AIR.1 AIRFIELD.2)
                   (NOT (LOCATED CARGOBYAIR.1 AIRFIELD.2))
                   (NOT (LOCATED CARGOBYSEA.1 AIRFIELD.2)))))
  (P81  (TYPE CONDITIONAL)
    (CONCLUDE (AND (NOT (LOCATED AIR.1 LOCATION.1))
                   (LOCATED CARGOBYAIR.1 LOCATION.1)
                   (LOCATED CARGOBYSEA.1 LOCATION.1)))
    (ORDERINGS (NEXT C18)))
  (C18  (TYPE PARALLEL)
    (ORDERINGS (NEXT P82) (NEXT P84)))
  (P86  (TYPE CONDITIONAL)
    (ACHIEVE (LOCATED CARGOBYSEA.1 AIRFIELD.2))
    (ORDERINGS (NEXT C19)))
  (P85  (TYPE CONDITIONAL)
    (ACHIEVE (LOCATED CARGOBYSEA.1 SEAPORT.2))
    (ORDERINGS (NEXT P86)))
  (P80  (TYPE CONDITIONAL)
    (ACHIEVE-BY ((MOBILIZED AIR.1 LOCATION.1) (MOBILIZE)))
    (ORDERINGS (NEXT P81)))
  (P84  (TYPE CONDITIONAL)
    (ACHIEVE (LOCATED CARGOBYSEA.1 SEAPORT.1))
    (ORDERINGS (NEXT P85)))
  (P82  (TYPE CONDITIONAL)
    (ACHIEVE (LOCATED CARGOBYAIR.1 AIRFIELD.1))
    (ORDERINGS (NEXT P83)))
  (P83  (TYPE CONDITIONAL)
    (ACHIEVE (LOCATED CARGOBYAIR.1 AIRFIELD.2))
    (ORDERINGS (NEXT C19)))))
```

Figure 4: ASCII representation of the Act from Figure 1

are mutually exclusive for a given plot node: if one is used, the others are prohibited. The specification of the type of a plotnode is optional, with the default value being `conditional`.

The `Parent` attribute specifies parent/child relationships between nodes of different Acts that are part of the same plan. In general, every plot node in an Act has such an attribute specified for it (unless the Act has no parent Act in its plan).

The `Time-Window` attribute provides absolute temporal boundaries on the execution time for that node, in terms of the earliest and latest allowable start times (`start0` and `start1`, respectively), earliest and latest allowable end times (`end0` and `end1`, respectively), and minimum and maximum durations (`min` and `max`). This attribute is expressed in the form

`(Time-Window start0 start1 end0 end1 min max)`

where the start and end times are either explicit numbers, or one of the values `inf` (infinity), `neginf` (negative infinity), `eps` (epsilon), or `negeps` (negative epsilon). The maximum duration must be greater than zero, and the minimum duration must not be negative.

Relative temporal constraints can be expressed on plot nodes, in addition to the properties slot for the Act (as noted above). On plot nodes, the syntax consists of a list of *orderings* of the form *(order-relation plotnode-id)*, where *order-relation* is either `next` or an Allen relation. Here, `next` refers to the successor plot node in the Act specification, which, in terms of Allen relations, is equivalent to `before or meets`. This construct indicates that the specified ordering relation holds between the current node and the node with label *plotnode-id*. The Act-Editor only writes `next` relationships in the plot nodes, other temporal constraints are written in the Time-Constraints property of the Act.

| | | |
|---|---|---|
| plotnode | ::= | (plotnode-id [(TYPE plotnode-type)] plot-metapreds plotnode-attribs) |
| plotnode-type | ::= | conditional \| parallel |
| plot-metapreds | ::= | < [action-mp][test] [conclude] [retract] [use-resource] [require] > |
| action-mp | ::= | achieve \| achieve-by \| achieve-all \| wait |
| plotnode-attribs | ::= | < [parent] [time-window] [orderings] [comment] > |
| parent | ::= | (PARENT plotnode-id) |
| orderings | ::= | (ORDERINGS {(order-reln plotnode-id)}*) |
| order-reln | ::= | next \| allen-reln |
| time-window | ::= | (TIME-WINDOW time time time time duration duration) |
| time | ::= | integer \| eps \| negeps \| inf \| neginf |
| duration | ::= | integer \| inf \| eps |
| comment | ::= | (COMMENT string) |

**Metapredicates**

| | | |
|---|---|---|
| meta-pred | ::= | test \| action-mp \| conclude \| retract \| use-resource \| require |
| test | ::= | (TEST wffs) |
| achieve | ::= | (ACHIEVE wffs) |
| achieve-by | ::= | (ACHIEVE-BY {wff+acts \| {wff+Acts}+}) |
| achieve-all | ::= | (ACHIEVE-ALL wff-pair) |
| conclude | ::= | (CONCLUDE wffs) |
| retract | ::= | (RETRACT wffs) |
| use-resource | ::= | (USE-RESOURCE {simple-term \| (simple-term+) }) |

wait ::= (WAIT-UNTIL wffs)
require ::= (REQUIRE-UNTIL {wff | wff-pair} )

**Logical Formulas**

wffs ::= {wff | wff-list}
wff-list ::= (wff$^+$)
wff-pair ::= (wff wff)
wff+acts ::= (wff ({act-id}$^+$))
wff ::= (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$^*$)

## 4.6 Time-Constraints Property

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.

temporal-prop ::= (time-constraints (time-constraint$^*$))
time-constraint ::= (allen-reln plotnode-id plotnode-id)
allen-reln ::= starts | overlaps | before | meets | during | finishes | equals

**Variables Property**

var-prop ::= (variables ((var-type variable )$^*$))
var-type ::= universal | existential

**Primitives**

task-id ::= *the name of a task*
plan-id ::= *the name of a plan*
act-id ::= *the name of an Act*
plotnode-id ::= *the name of a plot node*
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 ::= *any positive integer*
string ::= *any string*

symbol ::= *any symbol*
val ::= *any s-expression*

**Restrictions**

Additional restrictions are imposed on the Act syntax. These restrictions have been separated from the BNF specification presented above in order to keep the grammar simple.

- There must be exactly one plot node in a given act with no incoming arcs.
- `REBIND` terms cannot be used in gating environment slots.
- Unique identifiers must be used for plot nodes within a given Act.

# 5 ASCII Act Support Functions

The following LISP functions can be used to manipulate ASCII versions of Acts. They are defined as part of the Act-Editor[4].

`print-acts-to-file` *&optional (file *grasper-file*)* [*Function*]
This function takes a Grasper graph (file) and writes the ASCII version of all Acts in the graph to a file named *file*.text.

`print-act-to-file` *&optional (Act (space))* [*Function*]
This function prints an Act in the current graph in ASCII form to the file *actname*.text. The default is to print the Act currently drawn on the GUI.

`print-acts` *&optional (stream *standard-output*)* [*Function*]
This function writes the ASCII version of all Acts in the current Grasper graph to the given stream.

`print-act` *&optional (Act (space)) (stream *standard-output*)* [*Function*]
This function prints an Act in ASCII form to the given stream. The default is to print the Act currently drawn on the GUI.

`print-act-node` *node &optional plot? (Act (space)) (stream *standard-output*)* [*Function*]
This function prints a node in the given Act in ASCII form to the given stream. If *PLOT?* is non-nil, then the type of the node is printed.

`input-acts-from-file` *file* [*Function*]
This function reads the ASCII specifications of Acts from the given file into the current Grasper graph.

`input-act` *form* [*Function*]
This function creates an Act in the current graph from the ASCII specification *form*.

## References


[1] James F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23, 1984.

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

[3] E. H. Durfee, M. J. Huber, M. Kurnow, and J. Lee. Taipe: Tactical assistants for interaction planning and execution. In *Proceedings of Autonomous Agents '97*, 1997.

[4] Karen L. Myers. *The ACT Editor User's Guide*. Artificial Intelligence Center, SRI International, Menlo Park, CA, 1993.

[5] Karen L. Myers. *User's Guide for the Procedural Reasoning System*. Artificial Intelligence Center, SRI International, Menlo Park, CA, 1993.

[6] D. E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm*. Morgan Kaufmann, 1988.

[7] David E. Wilkins and Karen L. Myers. A common knowledge representation for plan generation and reactive execution. *Journal of Logic and Computation*, 5(6):731–761, December 1995.

[8] 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.