Planning and Reacting in Uncertain and Dynamic Environments

By: **David E. Wilkins, Ph.D., Senior Computer Scientist**
Karen L. Myers, Ph.D., Computer Scientist
John D. Lowrance, Ph.D., Program Director
Leonard P. Wesley, Ph.D., Senior Computer Scientist

Artificial Intelligence Center
SRI International
email: {wilkins, myers, lowrance, wesley}@ai.sri.com

December 8, 1994

This paper has been accepted to the *Journal of Experimental and Theoretical AI*, and should appear in volume 6, 1994, pp. 197-227. Meanwhile, it is also available in postcript form on the World Wide Web in http://www.ai.sri.com/people/wilkins/papers.html.

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

Abstract

Agents situated in dynamic and uncertain environments require several capabilities for successful operation. Such agents must monitor the world and respond appropriately to important events. The agents should be able to accept goals, synthesize complex plans for achieving those goals, and execute the plans while continuing to be responsive to changes in the world. As events render some current activities obsolete, the agents should be able to modify their plans while continuing activities unaffected by those events. The Cypress system is a domain-independent framework for defining persistent agents with this full range of behavior. Cypress has been used for several demanding applications, including military operations, real-time tracking, and fault diagnosis.

Contents

1 Introduction

We are interested in developing persistent agents that can achieve complex tasks in dynamic and uncertain environments. An agent of this type requires a number of capabilities. First of all, the agent must be *taskable* in that it can take appropriate responses to orders and goals assigned to it at run time. The complexity of tasks in practical applications generally precludes precompiling plans for all goals in all situations (as proposed by others [31]); hence, the agent must be able to synthesize new plans at run time in order to achieve its goals.

The dynamic nature of the environment necessitates that the agent be able to deal with unexpected changes in its world. Agents must be able to react to unanticipated events by taking appropriate actions in a timely manner, while continuing activities that support current goals. The unpredictability of the world could lead to failure of plans generated for individual tasks. Thus, agents must have the ability to recover from failures by adapting their activities to the new situation. In particular, the ability to modify the plan while continuing its execution is critical in domains where it is infeasible to halt all execution activities while replanning.

Finally, the agent should be able to perform all of the above operations even in the face of uncertainty about the world state. Traditional planning systems typically rely on perfect domain knowledge throughout plan development and execution [30, 33, 34, 37]. Execution systems [10, 26, 7] may sense an uncertain world, but rarely model and reason about the uncertainty. It is necessary to have techniques for representing and reasoning from uncertain information when developing and executing plans. For example, choosing a subplan for inclusion into the plan or for execution must often be weighed within the context of uncertain and partial state information, goals, expected outcomes, and costs of actions.

Many domains of interest require problem-solving agents with the above capabilities. One domain is controlling a mobile robot. Reactivity is necessary for responding to people and obstacles that may appear unexpectedly in the robot's path. Deliberative planning is necessary for purposeful behavior in response to run-time goals. The ability to reason about uncertainty is necessary for tasks such as self-localization and sensor interpretation. Military operations provides a second domain. Certainly one would not engage in an undertaking such as Desert Storm without first formulating a strategic mission plan. Reactive response and failure recovery are necessary for military operations because unexpected equipment failures, weather conditions, and enemy actions (among others) may require changes to the overall strategic plan. Reasoning in the face of uncertainty is critical, since complete knowledge for a given scenario is unlikely.

This paper describes the Cypress system, which provides a framework in which to create taskable, reactive agents that operate in dynamic and uncertain environments. Cypress provides a persistent system that can monitor and react to a dynamic environment by invoking standard operating procedures, by generating and executing plans to meet specific goals, and by applying decision procedures for assessing the situation and selecting among alternative plans of action. One of the major concerns in the development of the system was to build an

heuristically adequate tool that would be useful in practical applications.

1.1 Research Strategy

We began with a collection of mature, powerful AI tools that had been tested in numerous demanding applications: the SIPE–2 generative planner, the PRS-CL reactive execution system, and the Gister-CL system for reasoning about uncertainty. We needed to choose a research approach bounded by two extremes. One extreme was to develop a completely new technology that drew upon our experiences in developing the individual technologies. New data structures would need to be developed to encompass all the information and knowledge required by the different technologies, without sacrificing efficiency. A major drawback to this approach is that interactions among the different technologies would likely lead to unforeseen problems. The other extreme was to retain the independent systems that implement these technologies and coordinate their interactions. This approach has the advantage of building on well-established implementations that have proven useful in practical applications. Its disadvantage is that individual systems may redundantly, even incompatibly, implement common reasoning capabilities, information, and knowledge. In addition, if the individual systems are not modified, then some forms of tight interaction are eliminated from consideration.

We chose to adopt an intermediate path by which we integrated the component technologies but also defined a common representation language that enables the various systems to share knowledge. This approach has the advantages of allowing rapid experimentation and of using established systems with proven practical applicability.

Because of this research strategy, Cypress is a heterogeneous, loosely coupled integration of SIPE–2, PRS-CL, and Gister-CL.[1] Cypress includes a new common representation, ACT, for encoding plans and plan fragments [41]. The basic unit of representations is an *Act*, which can be used to encode both both the planning operators and plans of SIPE–2, and the goal- and fact-invoked reactive procedures of PRS-CL. There is a graphical knowledge editor for ACT, translators for converting from Acts to the internal representations of the component systems, and, in the case of SIPE–2, a translator from internal representations into Acts. The component systems were extended to cover the more diverse forms of knowledge in Acts that were motivated by the other systems, leading to the incremental development of a new composite technology.

1.2 A New Technology

Several features distinguish our approach: (1) the generation and execution of complex plans with parallel actions, (2) the integration of goal-driven and event-driven activities during

[1] In particular, CYPRESS = SIPE + PRS. SIPE–2, PRS-CL, ACT-Editor, Gister-CL and Cypress are trademarks of SRI International. (All other products mentioned are the trademarks of their respective holders.) Cypress is currently available on Unix workstations running Lucid Common Lisp and CLIM, and all machines supporting Symbolics Genera software.

execution, (3) the use of evidential reasoning for dealing with uncertainty, and (4) the use of replanning to handle run-time execution problems. The replanning capabilities are of particular note, as Cypress is the first system of which we are aware that supports *asynchronous* run-time replanning with a general-purpose generative planner. When problems arise during execution of strategic plans, Cypress can invoke a planning module to produce a new plan while continuing to execute portions of the plan that are unaffected by the problems. This mode of operation contrasts with *synchronous replanning*, in which plan execution is halted while an alternative plan is generated. Asynchronous replanning is critical in domains such as military operations or robot control, where it is infeasible to halt all execution activities while replanning some portion of the overall plan.

While there have been numerous efforts to develop the component technologies (planners, replanners, reactive controllers, uncertain reasoners) required for the kind of agent described here, and even to integrate certain combinations of these technologies, we are unaware of any current systems that provide the full functionality of Cypress. Several characteristics distinguish Cypress from other systems that provide both planning and reactive execution. Many systems do not use general-purpose planning and so cannot generate plans of sufficient complexity for many application domains. Previous work in run-time replanning has either been limited to synchronous approaches [19, 36] or focuses on local, adaptive modifications to rule sets, rather than employing the full look-ahead reasoning of a generative planner [26, 7]. The ability to modify a complex, parallel plan at run time and adapt execution activity to the new plan is, to our knowledge, a new accomplishment. There have been many initial investigations of planning under uncertainty, but a large technological gap remains between current planning capabilities and robust planning in uncertain environments. Cypress does not completely bridge this gap, but it does provide a means for using uncertain information when making critical planning decisions (e.g., choosing an action or the objects to use in an action), as well as a framework for evaluating the likely outcomes of a plan that will execute in an uncertain environment with uncertain effects of its actions.

While Cypress does not address all aspects of activity in dynamic and uncertain environments, it does provide a powerful system for defining taskable, reactive agents that can operate successfully in challenging domains. We have applied Cypress to a number of demanding problems, including real-time tracking, fault diagnosis, and military operations [40].

2 Overview of Cypress

This chapter presents an abstract model of taskable, reactive agents and describes Cypress, a particular implemented instantiation of this model. After describing the Cypress architecture, we briefly summarize each of the major subsystems, describe how a problem should be approached, and outline an example application.

Figure 1: Agent Model

2.1 Agent Model

Our agent model has two main reasoning components, an *executor* and a *planner*, as shown in Figure 1. The two components share a library of possible actions that the system can take. The library encompasses a full range of action representations, including *plans*, *planning operators*, and *executable procedures*. The executable procedures correspond to predefined standard operating procedures for satisfying individual goals. Each of these three classes of actions spans multiple levels of abstraction.

The executor is always active, constantly monitoring the world for goals to be achieved or events that require immediate action. In accord with its current beliefs and goals, the executor takes actions in response to these goals and events. Appropriate responses include applying executable procedures stored in the action library, invoking the planner to produce a novel sequence of actions for achieving a goal, or requesting that the planner modify a previous plan for which problems have developed during execution. The planner should be capable of synthesizing sophisticated action sequences that include parallel actions, conditional actions, and resource assignments.

The planner plans only to a certain level of detail, with the executor taking that plan and expanding it at run time by applying appropriate library actions at lower levels of abstraction. Planning to the lowest level of detail is often undesirable because of the resultant combinatorics of deep searches. Furthermore, it makes sense only to plan down to abstraction

Figure 2: The Architecture of CYPRESS

levels at which actions can be reasoned about ahead of time. For example, it is undesirable to plan large military operations down to the most minute detail since many decisions are conditioned on information that is not available until run time. Rather, it is the responsibility of the executor to further adapt the plan to the actual state of the world during execution. Similarly, it is often undesirable for the execution system to respond to high-level goals without a plan; for instance, a reactive system should not attempt to implement a Desert Storm-sized operation by applying procedures blindly.

An additional benefit to having an executor that can take plans at varying levels of abstraction and expand them at run time, is that the executor can begin taking actions towards meeting a goal without having to wait for a completely finished plan. This ability is critical for applications in which the amount of time that can be allotted to generating plans is limited.

2.2 Cypress Architecture

Cypress constitutes a particular framework in which to define taskable, reactive agents based on the model presented in the previous section. The architecture of Cypress is depicted in Figure 2.

Cypress is built on top of several mature AI systems that have been tested in a number of practical applications. The planner is SIPE–2 (System for Interactive Planning and

Execution), a classical planning system capable of generating plans in a hierarchical fashion [37, 39]. The executor is PRS-CL (the Procedural Reasoning System), a reactive execution system that integrates goal-oriented and event-driven activity in a flexible, uniform framework [11, 28]. The adequacy of PRS-CL and SIPE–2 for our model of intelligent reactive agents is discussed in Sections 2.3.1 and 2.3.2. The Gister-CL system implements a suite of evidential reasoning techniques that can be used during both planning and execution to analyze uncertain information about the world and possible actions [23, 32]. For example, it could use uncertain information to choose among multiple planning operators for a goal by the planner, and to choose among suitable procedures by the executor at run time.

For efficiency, PRS-CL and SIPE–2 employ their own internal representations for plans and actions. An *interlingua*, the ACT representation [41], enables these two systems to share information. ACT provides a language for specifying actions and plans that is suitable for both planners and executors. 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 the ACT interlingua, PRS-CL can execute plans produced by SIPE–2 and can invoke the planner in situations where run-time replanning is required.

The ability to define and manipulate plans and operators graphically greatly improves man-machine interactions. For this reason, the Cypress subsystems share a uniform graphical interface built on top of the Grasper-CL system [18]. Grasper-CL is a programming-language extension to Lisp that introduces graphs — arbitrarily connected networks — as a primitive data type. It includes procedures for graph construction, modification, and queries as well as a menu-driven, interactive display package that allows graphs to be constructed, modified, and viewed through direct pictorial manipulation. Grasper-CL is used both as a uniform basis for the man-machine interface of each subsystem and as a separate subsystem that provides supplementary graphical editing capabilities. In addition, the ACT-Editor subsystem supports the graphical creation, manipulation and display of Acts, thus serving as a graphical knowledge editor for other subsystems.

In contrast to many other agent architectures, planning and execution operate asynchronously within Cypress, in a loosely coupled fashion. The two systems communicate domain knowledge, plans, or planning requests by exchanging messages. This approach makes it possible for the two systems to run in parallel, without interfering with each other. In particular, the executor remains responsive to its environment during plan synthesis.

2.3 Component Systems

We briefly describe the three main component systems of Cypress, namely PRS-CL, SIPE–2, and Gister-CL.

2.3.1 PRS-CL

PRS-CL is a framework for constructing persistent, reactive controllers that can perform complex tasks in dynamic environments. It is a successor of PRS [11] that includes many

capabilities added to support the ACT formalism and the replanning in Cypress [41]. PRS-CL and its predecessor have proven useful in developing several demanding applications that required integration of reactive and goal-oriented behavior, including real-time tracking [8], a monitoring and control system for the Reaction Control System of the NASA Space Shuttle [10], and a control system for naval battle management aboard a Grumman E-2C [15].

Individual instantiations of a PRS-CL system are referred to as PRS *application agents*. A PRS application agent consists of a database containing current beliefs or facts about the world, a set of current goals, a set of predefined *procedures* (referred to as Knowledge Areas in PRS) describing how sequences of actions and tests may be performed to achieve certain goals or to react to particular situations, and *intentions* that keep track of the current procedures being executed by the agent. An interpreter manipulates these components, by selecting appropriate procedures for execution based on the system's beliefs and goals, then creating the corresponding intentions, and finally executing them.

A PRS agent 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 the system is running, it constantly monitors incoming information and goals. The predefined procedures are activated in response to the adoption of a new goal or to some change in the world. This combination of goal- and data-driven activity yields a flexible, adaptive execution framework. In particular, any intention can be interrupted and reconsidered in the light of new information about the world. The monitoring method used guarantees that any new fact or goal is noticed in a bounded time, thus providing rapid response to new events.[2]

Multiple PRS agents can be active simultaneously. Each agent has its own local goals, intentions and database, and runs asynchronously in the overall framework. A message-passing facility enables communication among agents in order to support parallel, distributed problem-solving.

PRS-CL procedures support a broad range of action primitives, ranging from testing conditions to achieving subgoals and waiting for events. In terms of control, the procedures provide conditional branching, iteration, recursion, and parallel action sequences. Procedures are not limited to describing activities in the external world, but can also manipulate the internal beliefs, goals, and intentions of PRS. Such *Metalevel procedures* can encode actions that influence the operation of the system itself, such as methods for choosing among multiple applicable procedures, modifying intentions, or computing the amount of reasoning that can be undertaken given the real-time constraints of the problem domain.

PRS-CL has the properties necessary for the executor component of taskable reactive agents: it is reactive, integrates goal-driven and event-driven activities uniformly, and has proven effective in numerous applications. The ability to define multiple PRS agents supports the simultaneous use of multiple instantiations of our abstract agent model, thus enabling cooperative problem solving among distributed agents.

[2]The bound on the cycle time is not absolute, but rather depends on the time required to execute the primitive actions in a given domain.

2.3.2 SIPE-2

SIPE–2 is an AI planning system that has the properties required by our agent model, including the ability to generate action sequences that include parallel actions, conditional actions, and resource assignments, the ability to modify its plans during execution, and computational efficiency that allows practical application. Here we briefly describe the system, stressing the features relevant to Cypress.

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

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 (from a planner's perspective) is to describe how the world changes after the action it represents is executed. Many of the effects of actions are not explicitly listed in the operators, since they are deduced by the system during plan generation from the deductive causal theory provided for the domain. 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.

SIPE–2 provides access to powerful temporal reasoning capabilities. The operator syntax allows specification of any of the 13 Allen relations [1] or qualitative constraints between the endpoints of any pair of actions in an operator. A temporal reasoner (TR) is called as a plan critic by SIPE–2, and the TR propagates these constraints and combines them with "commonsense" constraints that it represents internally (e.g., that durations cannot be negative), returning an updated set of time windows that are inserted into the plan. In the military operations domain, General Electric's (GE) Tachyon system [2] was used as the temporal reasoner. Tachyon is an efficient implementation of a constraint-based model for representing and maintaining qualitative and quantitative temporal information.

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 [38, 40]. 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 [40].

2.3.3 Gister-CL

Gister-CL implements a suite of *evidential reasoning* techniques that can be used during plan generation and plan execution to analyze uncertain information about the world and possible actions [22, 24, 21]. Evidential reasoning includes formal methods for reasoning under uncertainty, encompassing both probabilistic (i.e., Bayesian and Dempster-Shafer) and possibilistic models (i.e., fuzzy logic). When complete statistical data is available, Bayesian conditioning is performed; when incomplete or no statistical data is available, evidential reasoning techniques relax into Dempster-Shafer reasoning. Evidential reasoning methods require only a range of values within which the required probabilistic values must lie.

We have developed both a formal basis and a framework for implementing automated reasoning systems based upon evidential reasoning techniques. Both the formal and practical approach can be divided into four parts: (1) specifying a set of distinct propositional spaces (i.e., *frames of discernment*), each of which delimits a set of possible world situations; (2) specifying the interrelationships among these propositional spaces (i.e., *compatibility relations* between frames); (3) representing bodies of evidence as belief distributions over these propositional spaces (i.e., *mass distributions*); and (4) establishing paths (i.e., *analyses*) for evidence to flow through these propositional spaces by means of evidential operations (e.g., translation, projection, discounting, fusion), eventually converging on spaces where the target questions can be answered. These steps specify a means for arguing from multiple bodies of evidence toward a particular (probabilistic) conclusion.

Gister-CL is designed as a tool for the domain expert. With this tool, an expert can quickly and flexibly develop an argument (i.e., a line of reasoning) specific to a given domain situation. Gister-CL helps the expert keep track of the complex interrelationships among the components of his arguments, ensure that the relevant information has been properly incorporated, and reveal the more tentative aspects of the arguments. Once an analysis has been established by an expert, it can be instantiated over different situations by nonexperts. To improve run-time efficiency, Gister-CL supports the compilation of such analyses into functions. Gister-CL has been applied to a wide range of problems, including multisensor interpretation, helicopter mission planning, naval intelligence analysis, underwater vehicle tracking, and antiair threat identification.

2.4 Anatomy of a Cypress Application

The subsystems of Cypress can be used independently of each other. Thus, users can choose to run certain of the subsystems but not others. The real advantage of Cypress, however,

is that it can be used as an integrated planning framework that supports a wide range of planning and execution activities, including activities that require interactions among the subsystems. Here, we describe how a user might exercise the full extent of Cypress for a given application.

The first step for any application is to represent the possible actions for the domain. These actions are defined (using the ACT-Editor) as a collection of Acts which can be translated into the internal operator/procedure representations of both the planner and executor.

A typical use of Cypress to implement a taskable, reactive agent has the executor continually monitoring the world for additional goals and events that might require action in response. When a plan is required or requested (perhaps by an event in the world), the executor makes the appropriate request of the planner. The executor continues to monitor and respond to the world while the planner plans. The plans generated by the planner are translated to Acts for execution. The executor executes the plans produced by using Acts at lower levels of abstraction to perform actions that have not been planned, and invokes the planner when replanning is required.

Using the operators translated from the ACT representation, the planner generates plans in response to user-specified goals. The user can optionally instruct the planner to automatically employ the uncertain reasoner to reason about uncertain information while making decisions in the planning process. In particular, the uncertain reasoner can be invoked to choose among alternative objects and resources for use in the plan and to choose among alternative operators that can be applied to achieve a goal, based on user-supplied domain information. The uncertain reasoner can be called by the executor during plan execution when so directed by an Act. Thus, the domain programmer indicates those situations in which the uncertain reasoner should be employed.

Cypress supports run-time replanning. Should an unexpected problem arise during execution of the plan, the executor recognizes the problem, calls the planner to produce a new plan, and continues executing those portions of the plan that were not affected by the problem. Upon receiving a new plan, the executor would reconcile it with its current state and continue execution. Section 4 provides a more complete description of the replanning process.

Cypress has been applied to joint military operations planning, in a manner that parallels the above description (see Section 7). For this application, the planner uses domain knowledge in the form of Acts to generate employment and deployment plans for dealing with specific enemy threats, using the uncertain reasoner to choose military units based on uncertain information about both friendly and enemy units (see Section 5). The plan is translated to an Act for execution, and the executor employs a suite of lower-level Acts during execution that define the execution steps for the actions that the planner assumes as primitive. The executor responds to unexpected events with Acts that encode standard operating procedures. When execution of the plan reaches an impasse because of some unexpected changes in the world, it invokes the planner to provide an alternative plan, while continuing execution of activities not affected by the failure.

A critical issue in any application is dividing responsibility between the planner and

executor. In principle, each can execute Acts at any level of abstraction. In practice (as described in Section 2.1), the planner should not plan to the lowest level of detail and the execution system should not execute very abstract actions, unless forced to do so by time limitations. For these reasons, a fixed level of abstraction is specified for each application that serves as an *interface level* between planning and execution. The planner plans down only to the interface level, while the executor will accept plans at that level and attempt to execute them using actions from that level of abstraction or below. Information is encoded in the Acts themselves to indicate whether they should be used by the planner and/or the executor. It would certainly be preferable to provide a flexible level of interaction between the two subsystems that changes depending on the circumstances. For example, the planner would determine the number of levels deep for which planning is profitable, while the executor would decide when it is appropriate to simply execute Acts rather than invoke a planner. We hope to add such flexibility to the system in the near future.

For the military operations domain, nine levels of abstraction are employed in representing domain actions. The planner plans through five levels of abstraction, at which point actions are at the level of deploying forces, mobilizing units, and establishing patrols.[3] The executor employs Acts from the remaining four levels of abstraction during execution. There are no recursive operators in these levels; thus, the executor performs at most four levels of subgoaling for this domain.

3 A Common Representation for Planning and Execution

In order to integrate the various subsystems within Cypress, a common representation was required for describing actions and plans. 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. Since a narrow focus is critical to achieving success in terms of near-term use in software tools, we address the restricted case of a common language for planning and execution systems. Section 3.1 focuses on the issues involved in developing a common representation, while Section 3.2 summarizes our common representation.

3.1 Planning vs. Execution

Planners and executors share many representational requirements. At the heart of both plan generation and execution is the need for a language in which to express beliefs and goals. Such a language provides the basis on 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. Other common necessities are protection intervals for goals of maintenance, resources, the notion of *applicability* of a procedure or operator, and information required for deductive reasoning.

[3]Note that the planner may generate many more levels of plan refinement, but the level of description changes only five times.

Nevertheless, there are important representational differences for the two classes of systems because of their different functionalities. Planners explore a search space to generate a partially ordered set of activities designed to satisfy a particular goal. 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 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.

Failure has different meanings for the two classes of systems. Failure for planners means that a plan (or partial plan) that has been constructed is not suitable. Planners have several options; they can backtrack to consider alternative plan choices, or employ various algorithms to modify the plan and eliminate the failure (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 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, PRS-CL supports parallel execution of actions within a procedure, but PRS does not. An adequate shared representation of actions for planners and executors must support all of these sequencing operations.

3.2 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. A full

presentation of the ACT language can be found elsewhere [41]. Here, we summarize the main features of the language.

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* of an Act, and consist of a partially ordered set of actions and subgoals. The environment conditions and plot subgoals 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.

3.2.1 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 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 (see Section 3.2.2). Here, we provide an approximate reading for the meanings of the metapredicates.

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 changes in the world caused by an action.

3.2.2 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.

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

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. A Conclude metapredicate indicates that the Act should be invoked when a matching formula is added to the database, thus reacting to events that arise during the course of execution. 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 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. The most interesting property from a representational standpoint is `Time-Constraints`, which 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. The user is free to supply additional properties, as desired.

Figure 3 presents an example Act. 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 3.2.3.

3.2.3 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. All ACT metapredicates can be used in plot nodes.

[4] O-Plan refers to such conditions as "query conditions" [6].

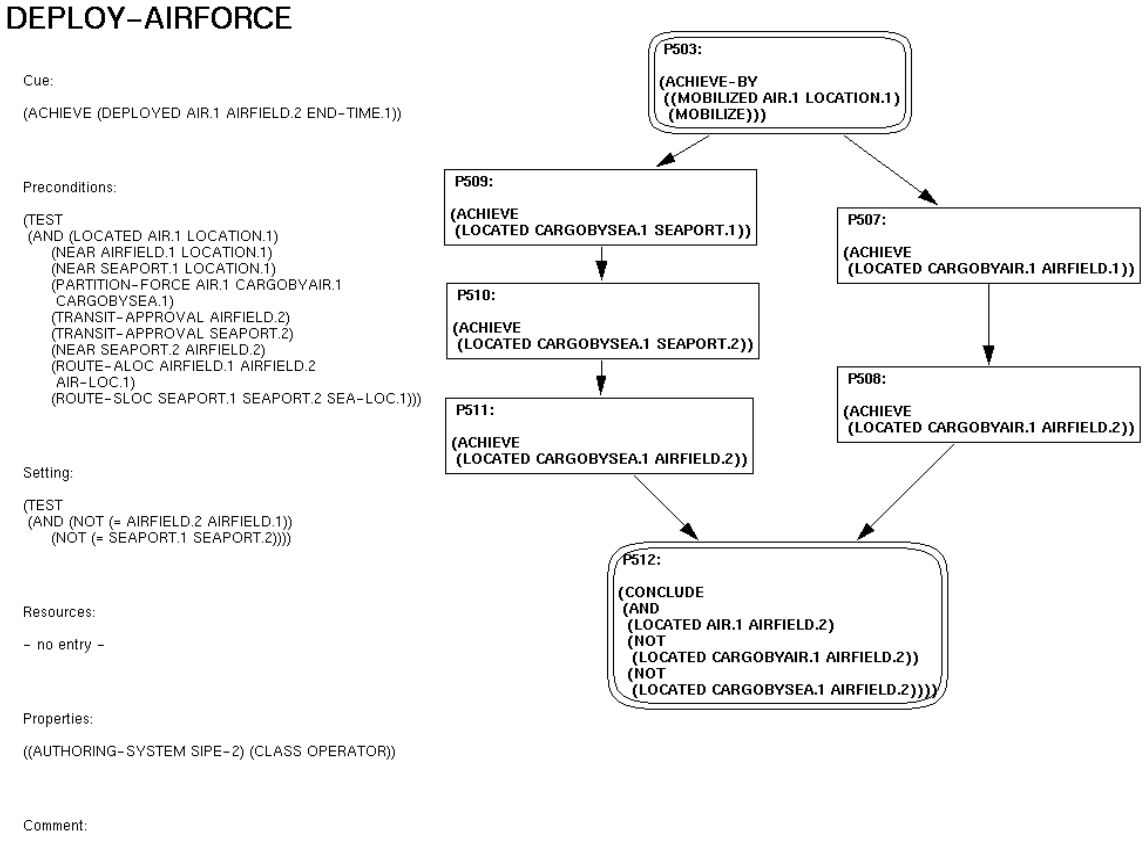

Figure 3: Deploy Airforce Act

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. 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 3, 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. During planning, all 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.

Consider the plot of the Deploy Airforce Act from Figure 3. The start node, `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. The successors specify that the air cargo will be moved to the final destination in parallel with moving the sea cargo to two intermediate ports and finally to the final destination. The final node joins the parallel actions, posts conclusions about the locations of the air force and its subparts, and cannot begin execution until both of its incoming branches have completed.

4 Asynchronous Run-time Replanning

By itself, PRS-CL provides a certain amount of flexibility at run time through its ability to adapt plan execution to the current state of the world. As an example, it can choose among multiple Acts to satisfy a given goal, depending on run-time conditions. PRS-CL can also be made to recover from certain kinds of failures. For instance, it includes a repair facility for recovering from protection failures (i.e., failures to maintain some specified condition over a designated time interval) through the use of domain-specific repair routines [28]. These adaptive mechanisms make PRS-CL a flexible execution system. However, the mechanisms are local in nature in that they can only adapt the plan execution in response to the current situation. They do not suffice for handling failures that require more strategic changes in planned activity. Modifications of a more global nature need the look-ahead reasoning capabilities of a generative planning system.

To this end, we have added a domain-independent *run-time replanning* capability to Cypress, which allows PRS-CL to invoke SIPE–2 to perform replanning for failures that cannot be corrected locally by application of predefined Acts. Replanning proceeds in the following

manner:

1. PRS-CL detects an irrecoverable failure during plan execution.
2. PRS-CL communicates the current state of execution to SIPE–2, then *continues executing* those parts of the plan that are unaffected by the failure.
3. SIPE–2 invokes its replanner to produce an alternative plan.
4. The new plan is translated to an Act and forwarded to PRS-CL.
5. PRS-CL merges the new plan with its current activities and continues execution.

The second step above highlights an important characteristic of our replanning framework, namely its *asynchronous* mode of operation. For asynchronous replanning, plan execution continues on those branches of the plan that are not affected by the failure. This mode of operation contrasts with *synchronous replanning*, in which plan execution is halted while an alternative plan is generated. Asynchronous replanning presents greater technical challenges, the most critical of which is to reconcile the state to which plan execution has progressed during generation of a new plan with the new plan itself. Asynchronous replanning is critical in many domains, since it is infeasible to halt execution while replanning occurs for some parts of the plan.

4.1 Architecture

Our basic model for replanning is bottom-up in nature, being driven by the activities of the executor. The executor is responsible for recognizing failure situations for which replanning is appropriate, requesting new plans from the planner, and finally implementing the new plan.

We developed a *transformational approach* to replanning whereby the activities in the original plan are left unmodified when possible. In particular, the planner modifies the failed plan during replanning rather than generating a completely new plan, and the executor continues execution of threads in the original plan that are unaffected by the failure while replanning takes place. This approach contrasts with many existing methods that simply abort execution of the old plan upon failure, then begin execution of a new plan. The transformational approach has the advantage of preserving undisturbed those activities in progress that remain part of the new plan. This property is essential in domains where it is infeasible to halt all execution activities while replanning some portion of the overall plan.

An application PRS agent initiates the replanning process upon detection of a failure that it recognizes as replannable. (The criteria for deciding when to replan are discussed in Section 4.2.) All requests for replanning are forwarded to a special PRS agent named `Replanner` that performs the necessary communication with SIPE–2, thus enabling the agent that requested replanning to continue execution of those parts of the plan that are unaffected by this failure. The `Replanner` agent is identical to other PRS agents; its specialized behavior

Figure 4: Replanning Architecture

results from the specific set of Acts that it executes. The architecture for the replanning framework is depicted in Figure 4.

The message sent to the `Replanner` agent indicates the name of the application agent requesting the replanning, a unique identifier for this particular failure episode, the plan being executed, the failed goal, and the *execution front*. The execution front is a list of the nodes last successfully executed on each parallel thread of the plan. The `Replanner` agent also sends SIPE–2 relevant information from its database that characterizes the current world state. Without such updates, SIPE–2 could generate plans only for the original state rather than the state in which the failure occurred, since it does not monitor the world during execution. The `Replanner` agent then awaits a response. In general, SIPE–2 will reply with a new plan that addresses the failure. The new plan is forwarded to the PRS agent that requested the replanning, which then integrates it with its current activities. If no new plan can be generated, the `Replanner` notifies the requesting PRS agent, which in turn terminates its execution of the original plan (that is, it gives up trying to accomplish the original goal).

When issued a replanning request, SIPE–2 responds by trying to modify the failed plan rather than producing a new plan from scratch. Doing so is often more efficient in practice (despite the computational complexity of the problem [29, 17]), since only limited changes are generally needed to fix the original plan. SIPE–2 begins by simulating the original plan through the execution front, then comparing its expected world state with the actual state provided by PRS-CL. SIPE–2 collects those formulae not expected to be true and determines how they affect the failed plan. The plan is modified to eliminate any problems by deleting unnecessary subplans and inserting new goals [37]. The planner ensures that the future consequences of the new plan do not interfere with the still-active execution threads.

One of the key technical difficulties in developing the replanning capability was integrating the revised plan with the current activities of the executor. This problem is further complicated for asynchronous planning, since execution of the original plan continues while the new plan is generated. To this end, SIPE–2 provides PRS-CL with both a new plan and a *node map*. The node map is a mapping from nodes in the original plan that have been changed or removed to *continuation nodes* in the new plan. The mapping encodes sufficient information for PRS-CL to transform its current activities during transfer of control to the new plan. Actions/subgoals being executed from nodes that are not in the node map are

not affected and continue execution. For a node mapped by the node map, PRS-CL aborts the associated activities, and begins execution at the corresponding continuation node in the new plan.

4.2 Failure Recognition

Not all execution failures warrant replanning. Some failures are a normal part of plan execution, or can be repaired by various local recovery techniques. For this reason, only failures at the highest level of the executor (that is, in the plans produced by the planner) that cannot be repaired locally are considered for triggering replanning.

Execution failures come in different types, depending on the nature of the goal(s) that has failed (here, we use the term *goal* in a generic sense to denote an objective of the execution system). Our replanning focuses on failures for the most common types of goals used in SIPE–2 plans: goals of testing that some expected condition in the world holds (represented by Test metapredicates in ACT) and goals of achievement (represented by the Achieve and Achieve-By metapredicates in ACT).

A test goal is considered to have failed by the executor when the condition being tested does not hold and no ACTs can be executed that will make the condition true. The planner inserts a test goal into the plan for the precondition of each ACT that is used during plan generation to ensure that it is appropriate to execute the subplan generated from this ACT. Failures of these goals arise because the world has changed in some way that was not anticipated by the planner. For example, when the Deploy Airforce Act from Figure 3 is used to generate a plan, a plot node with a Test metapredicate for the precondition of that Act will precede the plot nodes generated from this Act. In particular, this Test would include predicates requiring transit approval for the destination seaport and airfield. These transit approvals held at planning time or this Act would not have been applied. If, during execution, the transit approvals are rescinded, PRS-CL would detect an unrepairable failure before beginning execution of the part of the plan generated from the Deploy Airforce Act. In general, replanning could either plan some action to re-establish the transit approvals, or replace the deployment by using a different set of actions that did not require these transit approvals. SIPE–2 does the latter, in accordance with its philosophy that conditions for subgoaling should be encoded as goals in the plot and not preconditions.

An achievement goal fails when the condition to be achieved does not hold and there are no Acts that can be executed to make the condition true. In the Deploy Airforce example, a failed achievement goal would occur if PRS-CL could not find lower-level ACTs to execute to achieve the goal of moving the cargo to the destination seaport. Replanning could generate a new plan for achieving the goal (using Acts unavailable to the executor), plan to get a different but sufficient cargo to the seaport, or replace a larger part of the plan that included the failed goal. SIPE–2 may do any of these.

5 Planning under Uncertainty

Agents that operate in dynamic and unpredictable environments must be able to cope with uncertain information. There are several different sources of uncertainty. First of all, it is often impossible to characterize the current state of the world with complete accuracy. Imperfections in the domain knowledge can compound this situation, by adding inaccuracies to the expected outcome of actions. As well, external events may alter the world state in unanticipated ways.

There are several ways in which explicit reasoning about uncertainty can be valuable for taskable, reactive agents, including initial situation assessment, Act selection, Act parameterization, and plan evaluation.

Situation Assessment. Both planners and executors perform better when the current world state is understood. It is rare in practical applications that the state can be characterized with complete certainty. Typically, the initial assessment must be based upon interpreted sensory information and best estimates that pertain to different aspects of the overall situation. The sources are generally fallible and not necessarily current. Therefore, it is important that the available evidence be pooled to lead to a consensus. Even then, the consensus is typically incomplete and uncertain.

Act Selection. Uncertain reasoning can be used by a planner in selecting the operators to use during plan synthesis and by an executor in choosing the procedures to execute. Information that indicates whether the preconditions of competing Acts have been satisfied may be partial and uncertain to varying degrees. For example, in generating a plan to deter a military threat, it may be necessary to decide among different types of deterence operators. The choice may depend on several uncertain factors, such as the readiness, availability, location, and expected costs and benefits of using a particular force. The operator with the highest likelihood of success, given the available information, would be selected for application.

Act Parameterization. Once an Act has been chosen for use, there may be a number of possible objects to use in instantiating it. Uncertain reasoning can assist in this process.

Plan Evaluation. Another application of uncertain reasoning is to evaluate plans in the context of an uncertain environment. Uncertain reasoning can be used to predict the probabilistic results of executing a plan given that neither the initial state of the world nor the effects of applying operators (in known states) are known with certainty. Plan evaluation is useful during planning as a means of comparing alternative plans and during execution in selecting the plan most likely to succeed.

The above examples illustrate why it is important to have techniques for representing and reasoning from uncertain information when developing and executing plans. Gister-CL is particularly useful for performing these uncertain reasoning tasks because of its ability

to function in situations where limited probabilistic information is available. In an analysis developed to estimate the relative strength of military units (described below), a number of key indicators are combined to arrive at an overall assessment. Since these indicators include the morale of the troops and the readiness of their equipment, it is obvious that only limited information is available, particularly when evaluating a relatively unknown opposing force. For some indicators, no relevant information will be available, for others, limited information will justify only rough probabilistic bounds, while for others, adequate experience may justify precise probabilistic estimates. Whatever the available information, Gister-CL will calculate results whose precision is commensurate with the available information; that precision will be captured in the resulting probabilistic bounds which make clear to the decision maker what is objectively known.

We have used Gister-CL within Cypress to aid in Act parameterization and plan evaluation, as described in the next two sections respectively. Gister-CL could also be applied to reasoning about the likelihood of success in applying Acts, although we have not yet made use of that ability.

5.1 Operator Parameterization in Cypress

In our military application, Cypress uses its evidential reasoning capabilities to choose parameters for Acts being used by the planner. Constructing plans that use the most appropriate military units to carry out prescribed missions is important to the overall success of the mission. The choice of the most appropriate unit will typically depend on several factors that range from the availability, readiness, firepower, and so forth to the type of enemy forces expected to be encountered. The choice is difficult because there is generally uncertain information about the relevant factors. We describe how selecting an appropriate Army unit to carry out a specified mission is currently accomplished in the Army, and then describe the use of Gister-CL for automating this process.

Selecting a unit to carry out a particular mission is currently a highly manual process that involves assessing and then comparing the perceived strength of friendly and enemy forces. A unit is chosen based on whether it meets or exceeds specified force-strength requirements. For example, to carry out a deterent mission, sound military practice suggests that the ratio of friendly to enemy force strength be greater than or equal to two. For attack missions, the ratio should be greater than or equal to three. A unit's strength is based on the combined assessments of different characteristics that include

- **Troops:** The degree to which the unit has its full, ready complement of personnel
- **Armament:** The type and extent to which the troops are armed
- **Tradition:** The degree to which the current mission is the type of mission the unit has traditionally fought
- **Equipment:** The type and operational status of the unit's equipment

- **Transport:** The type and readiness of transportation required
- **Posture:** The proximity of the unit to where it needs to be deployed
- **Morale:** The morale of the troops assigned to the unit
- **Training & Experience:** The amount and type of training and experience the unit has for this type of mission
- **Mobility:** The degree and speed with which the unit can be relocated

Friendly as well as enemy unit forces have an associated "base-strength" value, which, in some sense, represents the power of a unit given no deficiencies with respect to its characteristics. If a unit has its full complement of troops, armament, equipment, and so forth, then the unit can be said to be at its designed level of strength. Intuitively, deficiencies in one or more characteristics should be reflected in terms of a reduction in a unit's overall strength. Expressing deficiencies is traditionally done in terms of a "force multiplier" (FM) that is associated with each characteristic. One commonly used method for computing overall strength involves multiplying the FMs and the unit's base-strength value. In practice, it is impossible to precisely assess characteristics of even friendly, much less enemy, forces. At times a range of FM values may be more representative of what is known and at other times, a probability distribution may be the best representation. This full range of expression is available in Cypress.

Gister-CL can be used to straightforwardly automate the selection process. Unit selection involves first comparing the perceived strength of friendly and potential enemy forces, and then selecting a unit based on the degree to which a unit meets or exceeds prespecified strength-ratio requirements. Doing so within an evidential reasoning framework requires constructing frames of discernment and relationships between them for both friendly and enemy unit forces. The frames and compatibility relations are represented as a graph, called a *gallery*. A frame is constructed that represents the nominal "base" strength for each unit, and other constructed frames represent the FM assessments of other force characteristics. We constructed these frames in a gallery as shown in Figure 5. Frames corresponding to a friendly force begin with "F", and frames corresponding to an enemy force begin with "E".

Relationships between the frames are represented by connecting arcs called *compatibility relations*, which compute the result of combining frames. In our example, compatibility relations are often multiplicative. However, some are not. For example, the relative firepower of a unit depends on the relative values of its constituent Troops and Armament FM values, and is, in our implementation, the maximum of the FM values in the Troops and Armament frames. The reason for this dependence is that even if a unit does not have its full complement of troops, the remaining troops may have sufficient armament to balance deficiencies in troops. Conversely, a unit may have sufficient troops to balance deficiencies in armament.

Given a set of FM estimates for a subset of friendly and enemy characteristics, a Gister-CL analysis pools these independent estimates, according to the compatibility relations in

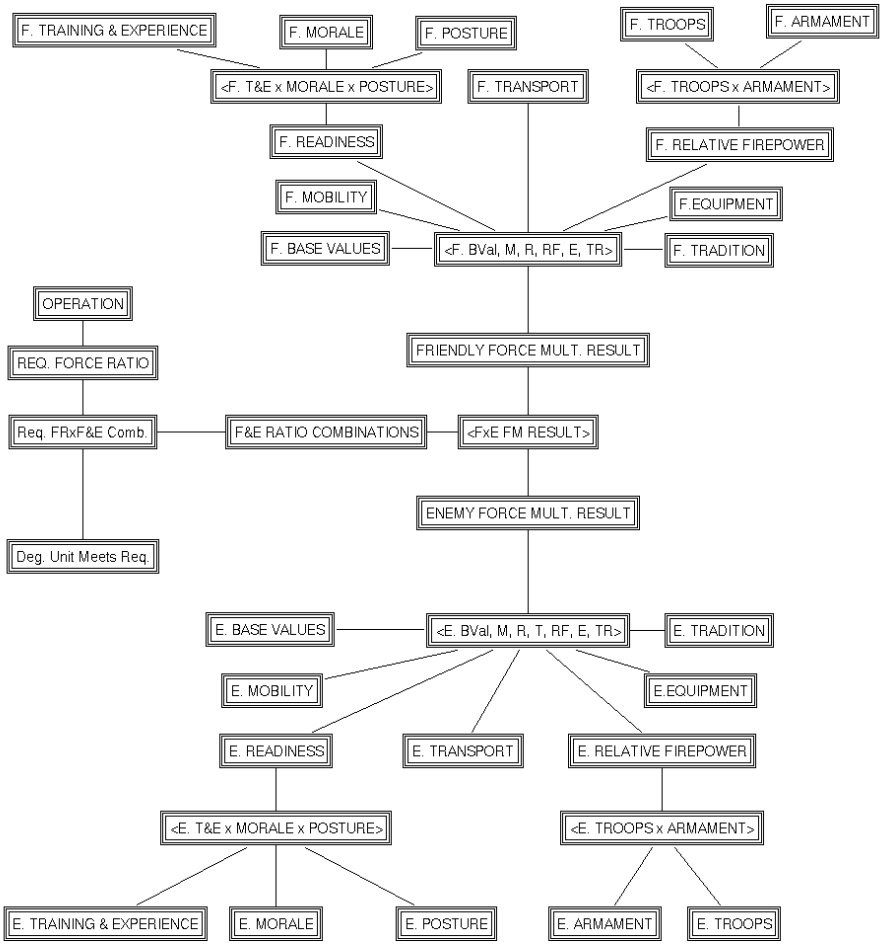

Figure 5: Gister Gallery for Army Unit Selection

```
INTERP. OF DEGREE TO WHICH UNIT MEETS REQUIREMENTS:
  100.                                [100. 100.]   (OR UNKNOWN BELOW REQ. FORCE RATIO EXCEEDS REQ. FORCE RATIO FAR BELOW
REQ. FORCE RATIO FAR EXCEEDS REQ. FORCE RATIO MEETS REQ. FORCE RATIO)
   79.                                [77. 81.]     FAR EXCEEDS REQ. FORCE RATIO
   21.                                [19. 23.]     MEETS REQ. FORCE RATIO
   2.                                 [0. 4.]       BELOW REQ. FORCE RATIO
   2.                                 [0. 4.]       EXCEEDS REQ. FORCE RATIO
   2.                                 [0. 4.]       FAR BELOW REQ. FORCE RATIO
```

Figure 6: Result of Degree to Which Unit Meets Force Ratio Requirements

the gallery, to arrive at a consensus for each unit's strength. Generally speaking, each estimate may consist of a probability distribution over FM interval values; when no estimate is available, the full FM interval is attributed unit belief; when complete knowledge is available, unit belief is attributed to a precise FM value. Depending on the certainty and precision of the inputs, the certainty and precision of the consensus will vary. For example, an analysis of the characteristics of a particular unit for a deterent mission against a particular enemy force results in the distribution shown in Figure 6.

5.2 Plan Evaluation in Cypress

As previously discussed, plan evaluation involves predicting the probabilistic results of executing a plan given that neither the initial state of the world nor the effects of applying operators (in known states) are known with certainty. Such evaluations could be utilized by the execution system to select those procedures for execution that are most likely to achieve their intended goals or by the planner to compare alternative plans.

Cypress employs the planner's logical reasoning capabilities in support of probabilistic reasoning. The planner provides the logic used by the probabilistic reasoner during plan evaluation, with several advantages. Briefly, these advantages are compactness of representation, the planner's efficiency when determining the truth of a proposition in a world state obtained by executing a parallel plan, and the ability of the planner to generate plans automatically when the probabilistic reasoner eventually asks that goals be satisfied rather than that operators be applied.

Gister-CL has two distinct subsystems that perform different types of reasoning. One subsystem focuses on numeric calculations relating to probabilities; the other subsystem performs logical calculations relating to possibilities. As probability distributions are manipulated by the first component, logical questions are posed to the second component. The implementation of each of these components is independent of the other, so long as the logical questions posed by the numeric component are answered by the logical component. Gister-CL includes several different implementations for the logical component (e.g., implementations based on sets, bit vectors, and the real numbers). The most appropriate one to use depends upon the characteristics of the domain of application. Other logical implementations are easily added by defining methods (invoked by calls to generic functions) for the logical connectives and other constructs comprising the logical questions.

We have implemented a SIPE–2 logic for the logical component and have tested it by evaluating plans in the context of uncertain initial states and probabilistic operators. When Gister-CL evaluates plans, this logic provides the algorithms and representations for determining whether a proposition is true in a world state, for determining whether two world states are equivalent, and for deriving new states using compact planning operators. World states are represented as nodes in plans that are created by the planner in response to requests from Gister-CL to perform actions. Gister-CL represents an incompletely specified world state as a set of plan nodes and an uncertain world state as a probability distribution

over (possibly incomplete) world states.

Gister-CL needs to query the truth of propositions in specific world states. In our implementation, propositions and world states are given to the planner, which simply applies its truth criterion to the proposition at the plan node. (The truth criterion uses the whole plan in which the node is embedded to compute its result.) In an incompletely specified state, checking the truth of a proposition or determining its probability will result in several such calls to the planner, one for each node in the set of plan nodes representing the incomplete state information.

The planner derives one world state from another by using its operators, as requested by Gister-CL, to elaborate plans. The plan node representing the requested state is returned. Since the same request may be made several times, SIPE–2 keeps track of all expansions and returns an already constructed plan node whenever appropriate. To apply an action to an incompletely specified world state, the planner is called to apply that action to each of the states that make up the incomplete specification.

The discussion to this point has focused on plan evaluation when the initial (and therefore subsequent) state of the world is uncertain. Another source of uncertainty that needs to be taken into account is the nondeterministic nature of many real-world operators. We represent nondeterministic operators using distributions over deterministic operators. The distribution indicates the frequency by which the nondeterministic operator functions as if it were various deterministic operators. This representation has the advantage of being grounded in deterministic operators, the type of operators that SIPE–2 was designed to manipulate. To predict the effect of a nondeterministic operator when applied to a known state, Gister-CL simply substitutes the corresponding distribution of deterministic operators. If the initial state is itself uncertain, then the operator distribution is applied to all of the possible initial states and the probabilities of the states are multiplied by the probabilities of the operators. Assuming that the nondeterministic nature of the operators does not vary depending on what operator applications have preceded them, then a sequence of nondeterministic actions can be modeled through sequential application of these nondeterministic operators.

In summary, Cypress can evaluate plans in the presence of nondeterministic operators and uncertain initial states. More details are provided elsewhere [25]. This capability was produced by implementing a SIPE–2 logic within Gister-CL; it did not require substantial changes to either system. Plan evaluation in Cypress demonstrates an effective combination of planning and uncertain reasoning technology, exploiting the strengths of each.

6 Human Computer Interaction

In practical applications, plans and domain knowledge are often highly complex. A graphical interface is often essential to inputing operators, understanding generated plans and domain knowledge, and monitoring system behavior. Without natural pictorial representations of the knowledge and the resulting plans, it would be nearly impossible for a human user to understand the domain knowledge, the plan, the planning process, or the execution of the

plan. For persistent agents to be practically useful, significant and important research issues must be addressed in this area. Since this is not the major focus of this paper, we briefly describe the approach taken in Cypress, without analyzing in depth the strengths, weaknesses, or alternatives.

As described in Section 2.2, we have implemented a powerful graphical user interface (GUI) for interacting with Cypress based on the Grasper-CL system.[5] In the GUI, plans are represented as PERT-chart style graphs, and domain knowledge is presented in many ways, from pop-up windows with formatted tables, to graphs representing trees. Many options are provided for labeling and sizing the nodes in any particular graph.

The Cypress GUI is a significant improvement on the GUI of any previous domain-independent planning/execution systems of which we are aware. Because of the domain independence of our system, we chose a graph-based representation. We feel that significant improvement on a graph-based GUI generally requires use of domain-dependent features in the interface, since the plans produced and the important features in them vary widely from one application to another. For example, the preferred user interface for viewing plans in military operations planning is a map with icons moving on it, while Gantt charts are a popular interface for scheduling production lines.

New GUIs were developed for all Cypress subsystems in order to provide a uniform interface for Cypress. The subsystems are controlled through noun menus and command menus. Selecting a noun brings up a menu of commands appropriate to that noun. For example, SIPE–2 has five nouns that let the user choose the level at which the commands will operate, as described in Section 6.2. The other subsystems have their own nouns and command menus in the same style.

Plan drawings often do not fit on the screen in their entirety, and are difficult to visualize. The GUI provides several techniques for viewing them. The graph window is scrollable, so the user can pan over the whole plan. In addition, there are numerous tools for navigating through large graphs. A *birds-eye-view* window provides a low-resolution view of the entire graph. This window can control the scrolling of the full-resolution window, and the user can query graph relationships in the birds-eye window. The GUI also provides the ability to display a truncated subtree of a large graph, and to incrementally expand or collapse nodes within the subtree to walk through regions of interest in the graph. Thus, one can use the low-resolution view of a graph to select the region for high-resolution browsing. Important GUI features of the individual subsystems are described below.

6.1 ACT-Editor

The ACT-Editor is a graphical knowledge-editor for creating, displaying, and manipulating Acts. It provides knowledge-editing capabilities for both SIPE–2 and PRS-CL. A user can create or modify an Act by selecting the appropriate actions from the menu; the ACT-Editor will prompt for all necessary information. Figure 7 shows the graphical display of an Act and

[5] Grasper is a trademark of SRI International.

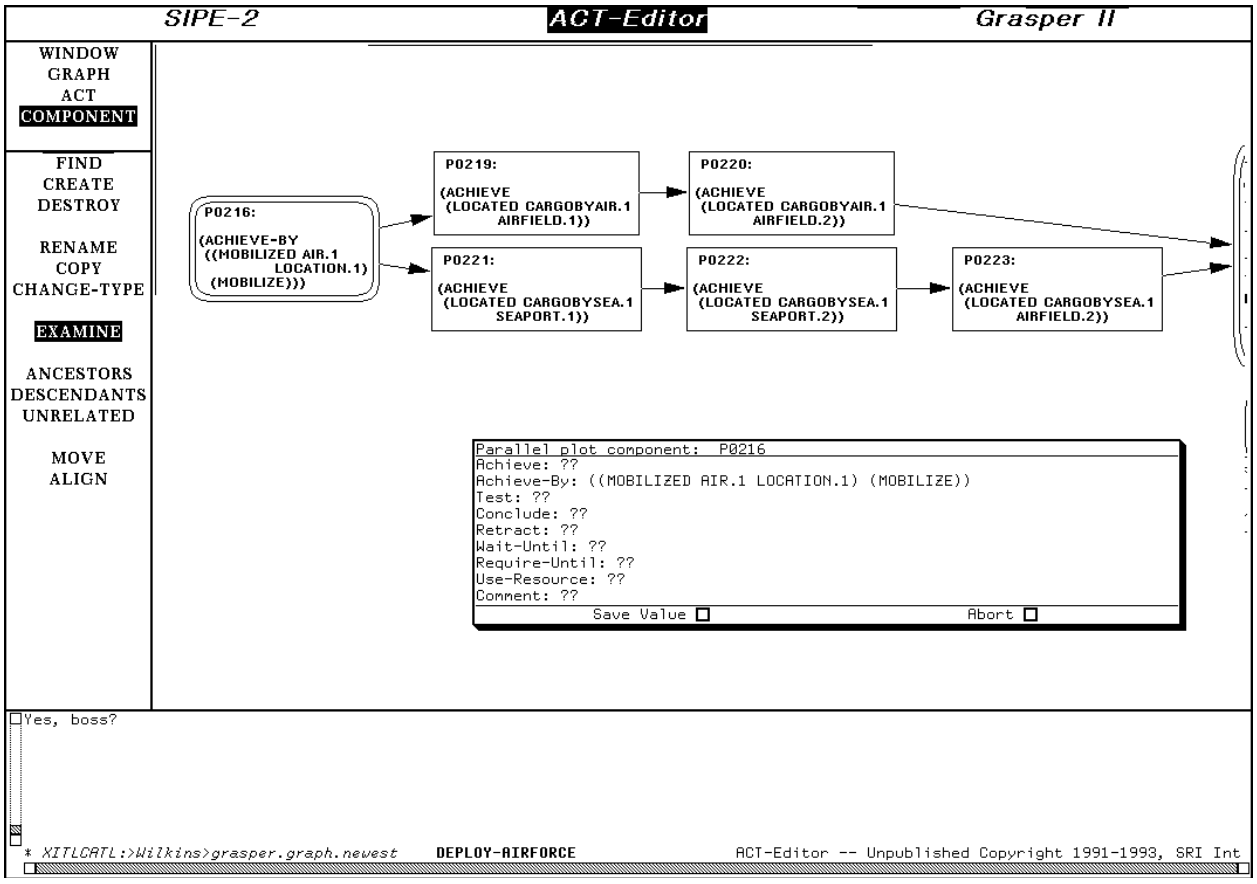

Figure 7: Plot of Deploy Airforce Act

the commands available for acting on individual slots and plot nodes. In this figure, the user has clicked the Examine command and then clicked the first node of the plot. The resulting pop-up window allows the user to edit any of the metapredicates on that node — currently Achieve-By is the only metapredicate on node P0216. For example, additional effects of the mobilize action could be added under the Conclude metapredicate.

In practical applications, Acts can be complex. In particular, ACT representations of SIPE–2-generated plans are often quite large: for example, a typical plan in the military domain produces an Act with over 200 plot nodes. Two capabilities in the ACT-Editor reduce the complexity of manipulating and viewing Acts: a *simplifier* and the *New View* command. The simplifier streamlines the logical structure of an Act, eliminating both unnecessary plot nodes and redundant ordering links. These simplifications produce Acts that are semantically equivalent, are often more compact, often make their intent more apparent, and can lead to improved plan execution performance. The New View command does not modify the structure of an Act but rather modifies its presentation on the screen. 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.

6.2 SIPE-2

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 [40]. Here we briefly mention some of the capabilities that were useful in military operations planning.

SIPE–2 has five nouns that let the user choose the level at which the commands in the current command menu will operate. The PROFILE noun activates commands for setting defaults that allow the user to customize system behavior. The DOMAIN noun activates commands that apply to the problem domain as a whole, e.g., inputing and inspecting the domain knowledge. The PLAN noun activates commands that apply to a specific plan, including executing a plan and generating a plan either interactively or automatically. The DRAWINGS noun activates commands that draw various SIPE–2 structures as graphs. Objects that can be drawn include plans, operators, the sort hierarchy, the initial world model, and problems. Finally, the NODE noun activates commands that apply to specific nodes in the currently drawn plan, e.g., highlighting all actions in parallel with a specific action.

SIPE–2 automatically lays out the nodes of a new plan and gives the user several options concerning which nodes to display and how to label the nodes. One can highlight all the successors of a node, or all the predecessors of a node, or all the nodes unordered with respect to a node, or all nodes that mention a certain object or have certain predicates in their effects. This capability is very useful in plans with many ordering links, such as military operations plans. For example, highlighting all the nodes that mention Fennario Port shows the schedule for that port, and highlighting all the nodes unordered with respect to a certain node could show all actions in the current phase of the operation. All the actions that

accomplish a certain condition can also be highlighted. For example, one can highlight all the actions that move troops in a plan. The user can move nodes around with the mouse to make the graph look exactly as desired.

During interactive planning, the system will highlight actions and goals for which choices are being made by the user. The user may select the next goal to solve after all goals are highlighted. The possible choices for a particular goal (e.g., operators or resources) can then be drawn on request, and the plan drawing will be restored when the user has made a choice. Similarly, nodes involved in resource conflicts are highlighted when the user requests to interactively solve resource conflicts. If an ordering of two nodes is chosen, the GUI will incrementally highlight and draw ordering links as they are added. In particular, it flashes the first node, then draws the ordering link, and then flashes the second node. This gives an excellent visual depiction of how the plan is flowing.

6.3 PRS-CL

Users can interact with PRS-CL agents at run time by modifying their database, stopping/restarting their execution, sending messages, and posting goals. PRS-CL also provides an extensive tracing facility for monitoring run-time behavior. For each agent, a user can activate textual tracing of various aspects of the agent's behavior; including database changes, message passing, resource allocation, and procedure execution. Users can also display the intentions for an individual agent. As described above, the intentions track the current activities and goals of an agent. The intention display makes explicit the hierarchical relationships between goals and subgoals, as well as any prioritization amongst such goal trees.

Procedure execution can also be traced graphically, although for only a single agent at a time. The graphical tracing facility allows the user to specify a set of Acts that are to be displayed when executed. Nodes in the Act are highlighted as their associated goals are undertaken. In this way, the user can visually monitor the activities of the system.

7 An Application of Cypress

An implemented example illustrates the use of Cypress within a military operations domain. This domain is an extended version of the one used for the second Integrated Feasibility Demonstration in the ARPA-Rome Laboratory Planning Initiative [40, 3]. The domain knowledge includes approximately 100 plan operators, 500 objects with 15 to 20 properties per object, and 2200 initial predicate instances. Plans range in size from several dozen to 200 actions, usually spread among numerous parallel threads of activity.

The scenario begins with PRS-CL continually monitoring the world and receiving a request to deter all known military threats in a certain region. The executor does not have an Act to accomplish this, so requests the planner to generate a plan for achieving this goal. SIPE–2 is invoked using a set of Acts previously input to the system. During the planning process, Gister-CL assists the planner in choosing appropriate military forces for particular missions.

The system generates 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.

The plan produced by SIPE–2 contains four main threads of parallel activities: two threads for deterence using ground forces, one for deterence using naval forces, and the fourth using air forces. Throughout the planning process, PRS-CL monitors the world for additional goals and events that might require immediate action. PRS-CL executes the plan by applying appropriate Acts to refine the plan to lower levels of abstractions, eventually bottoming out in actions that are executable in the world. PRS-CL remains responsive to new goals and events throughout.

As part of the air deterence operations, aircraft are moved among various air bases. The use of an air base requires explicit transit approval, which is granted initially for all bases in the domain. An execution failure can be triggered by rescinding transit approval for any of the bases used in the plan. PRS-CL records any such changes, and detects a failure when execution reaches the stage where the rescinded approval is required. No Acts are defined for repairing such a failure locally, thus execution would completely fail at this point without replanning. In Cypress, a replanning request is sent to SIPE–2. Meanwhile, execution of the remaining branches of the original plan continues without disruption.

Replanning for this situation produces a modified plan in which an alternative mobilization strategy is employed. One of our test cases results in the removal of a dozen actions from the plan, replacing them with a new subplan of similar length. The operations in the new plan are selected so as not to interfere with the continuing execution of actions on other parallel threads in the original plan. The new plan is sent to PRS-CL, which integrates the new plan with its current activities and continues its attempts to achieve the original goal.

The ability to modify a complex, parallel plan at run time and adapt continuing execution activity to the new plan is, to our knowledge, a new accomplishment. The computational properties of this application are acceptable. Plan generation time for a small plan of 47 actions, 95 total plot nodes, and 15 parallel branchings is 13 seconds.[6] Plan generation time for a larger plan of 188 actions, 381 total plot nodes, and 72 parallel branchings is 96 seconds. (Non-action plot nodes include nodes that encode plan assumptions and rationale as needed for plan modification.) Modifying the plan as described requires only a few seconds, depending on the modification required. The executor has a cycle time of a small fraction of a second.

[6] All times are user run time in Lucid Common Lisp on a Sun Sparcstation 10.

8 Comparison to Other Work

Evaluations of the component technologies in Cypress (PRS-CL, SIPE–2, and Gister-CL) can be found in the referenced literature for each subsystem (see Section 2). Here, we compare the integrated system as a whole to previous systems that combine planning and execution. Since there has been a marked increase in work on this topic in recent years, we do not attempt to provide a comprehensive comparison; instead, we focus on a few representative approaches that share our objective of producing taskable, reactive agents. Hanks and Firby [12] provide a thorough discussion of the issues in integrating planning and execution, including issues not addressed in this paper, such as reasoning about the utility of deliberation.

The complexity of the plans and the necessity of using them is what differentiates our work from most other efforts. Most of the previous work on combining planning and reactive execution has been driven by the requirements of the execution system, using planning in only very limited ways. The work on planning and execution systems for robotics applications typifies this bias. Current systems for mobile robots emphasize execution and control, using only precomputed plans or simple plans generated at run-time. In contrast, domains such as military operations 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 [26] 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 [7] also uses simple plans to modify the reactive control of a robot. These and other similar approaches use adaptive modifications to rule sets, rather than employing the look-ahead reasoning of a generative planner. 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

The PHOENIX system [14] shares many objectives with the work reported here. The system is reactive, provides certain kinds of failure recovery, and has been applied to the practical problem of firefighting. However, it selects plans from a plan library rather than performing generative planning to produce the high-level plans that guide the system. The ATLANTIS system [9] for controlling mobile robots has an architecture that is similar to that of Cypress. ATLANTIS provides asynchronous communication between a controller and a deliberator that enables simultaneous planning and execution, asynchronous replanning,

and look-ahead detection of possible failures. However, the deliberator employs a "simple linear planner" that interacts with the controller only through previously designated inputs; this contrasts with the general plan interface between planning and execution that Cypress provides. Other systems provide replanning during execution but are limited to synchronous approaches and do not support continued execution during replanning [19]. In contrast to the heterogeneous design of Cypress, Washington's system [35] employs a homogeneous framework in which planning and execution coexist, accessing a single plan structure that is modified over time in response to changes in the world. The system employs forward-chaining expansion of plans and can begin execution without having a complete plan defined at any level of abstraction. This approach enables the system to begin actions towards a goal when time is critical, even when deliberation has not yet finalized a plan for achieving that goal.

The motivations for the ACT representation are similar to those of the KRSL language [20]. 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.

McDermott [27] 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 [16] is another system designed for programming behaviors for intelligent reactive systems.

Finally, we note that there has been previous work on the *static* portion of the replanning task, namely modifying a failed plan from a description of the current world state (without consideration of interactions with a persistent execution system). The most advanced efforts in this regard include SIPE (the predecessor of SIPE–2) [37], and PRIAR [17]. The static replanning techniques in Cypress include all those in SIPE, as well as some minor extensions.

9 Conclusion

Cypress is a powerful framework in which to define taskable, reactive agents that can operate in dynamic and unpredictable environments. While it does not yet address all aspects of activity for such environments, it does provide a strong basis for building intelligent control systems that operate in challenging domains. The integrated planning, execution, and reasoning about uncertainty demonstrated in the military problem attests to this fact, showing both that Cypress is sufficiently powerful and has acceptable computational properties. In this application, the planner generates and modifies plans containing hundreds of plot nodes

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 (while continuing execution). To our knowledge, the ability to do so with sophisticated plans is a new accomplishment.

The development of Cypress involved more than simply engineering the integration of existing systems. The asynchronous replanning facility constitutes one important technological advancement, providing flexible plan execution that can adapt to significant unexpected changes in the world. In addition, interesting technical problems related to the integration of planners and executors had to be addressed, leading to the development of the ACT formalism. ACT supports the representation of knowledge necessary for the generation and execution of plans suitable for use in complex and dynamic environments. As such, it can serve as an interlingua that supports heterogeneous combinations of AI technologies in planning and reactive control. Subtle differences between these two classes of systems were identified and several extensions were made to the existing planning and execution technologies to implement ACT. ACT, which was influenced by the design of PRS-CL and SIPE–2, is intended to be a general-purpose representation language that can be used to share knowledge between many different execution and planning systems.

9.1 Future Work

While SIPE–2, PRS-CL, and Gister-CL represent advanced technologies in their respective areas, each could be strengthened in a number of ways. SIPE–2 would be more useful if it could relax constraints, reason about partial achievement of goals, and weigh the utility of plans with partially achieved goals. Gister-CL could be usefully integrated into this process of predicting the partial achievement of goals and the utility of plans in uncertain environments. The major technological hurdle is the reduction of combinatorics. PRS-CL is a powerful tool for reactive control but is not 'real-time' in that it cannot guarantee response to critical events in small, absolute time intervals. With respect to replanning, the failure recognition component of PRS-CL could be extended to handle additional types of execution failure. Adding these capabilities to the various component systems would greatly increase the scope of applications for which Cypress could be employed. We note that for the most part, the above enhancements reflect open research problems that have yet to be solved adequately in the AI community.

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. 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. We view 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.

For reasons described in Section 1, Cypress was built on top of several predefined technologies. The result is a heterogeneous, loosely coupled system with a very limited set of interactions amongst its components. The heterogeneous model has numerous advantages, including the ability to run subsystems on different machines, and the potential for swapping in different component technologies in a modular fashion. However, strengthening the interactions among the subsystems, particularly between planning and execution, is essential to providing more intelligent activity.

The fixed level of interaction between planning and execution in Cypress is one problem that derives from the loose coupling of the component systems. As described in Section 2.4, we currently fix a level of abstraction to serve as an interface point between the planner and executor. Fixed interaction levels are acceptable in domains like military operations because high-level operations often should not be initiated without a detailed plan, and execution times of hours or even days provide ample time for the planner to respond. In more time-constrained domains where it may be necessary to respond quickly to high-level goals, this approach detracts from the flexibility of the system in two ways.

First, the responsiveness of the executor to newly posted high-level goals that require planning is reduced since the executor must wait for the planner to finish planning to the interaction level before it can initiate actions designed to achieve the goal. Second, it would be preferable to allow the agent to decide in a situation-dependent manner what is an appropriate level of abstraction at which to halt planning by weighing the utility of planning versus acting. The use of ACT as a common representation language for both planning and execution provides the basis for more flexible interactions. In particular, PRS-CL can execute a plan at any level of abstraction by applying the same Acts used for plan expansion (although without the benefit of SIPE–2's look-ahead reasoning). At this point in time, however, we have not yet provided the higher-level capabilities for overseeing such a flexible transfer of control between SIPE–2 and PRS-CL. Architectures and methods that address this problem have been proposed [4, 5, 13]. We hope to develop similar techniques for Cypress, possibly using the evidential reasoning capabilities within the system to weigh the utility of planning versus acting under uncertainty.

Additional communication between the two systems during both execution and replanning could also lead to more intelligent activity. For example, SIPE–2 could receive periodic updates from PRS-CL regarding changes in the world. This information would enable SIPE–2 to reason forward in time to anticipate future plan failures (*envelopes*, in the terminology of [14]), rather than simply responding to failures once they arise. As a second example, it would be useful to have SIPE–2 communicate with PRS-CL during the replanning process to provide information about changes being made to the plan. PRS-CL could then modify its execution of the original plan to avoid undertaking steps that will be eliminated or modified in the new plan being generated.

Acknowledgments

The research described in this paper was supported by the ARPA/Rome Laboratory Planning Initiative under Contract F30602-90-C-0086. The implementation of the military operations domain was done under ARPA/Rome Laboratory Contract F30602-91-C-0039 by Marie Bienkowski, Marie desJardins, and Roberto Desimone. Our thanks to the people who helped implement the GUI: Peter Karp, Janet Lee, and Mabry Tyson.

References

[1] J. F. Allen. *Recognizing Intentions from Natural Language Utterances*, pages 107–166. MIT Press, Cambridge, MA, 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, 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.

[4] Mark Boddy and Thomas Dean. Solving time-dependent planning problems. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 979–984, Detroit, 1989.

[5] Michael E. Bratman, David J. Israel, and Martha E. Pollack. Plans and resource-bounded practical reasoning. *Computational Intelligence*, 4(4):349–355, 1988.

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

[7] 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.

[8] T. Garvey and K.L. Myers. The intelligent information manager. Final Report SRI Project 8005, Artificial Intelligence Center, SRI International, March 1993.

[9] 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.

[10] Michael P. Georgeff and François Félix Ingrand. Research on procedural reasoning systems. Final Report Phase 1, Artificial Intelligence Center, SRI International, Menlo Park, CA, October 1988.

[11] 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.

[12] 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.

[13] Eric J. Horvitz, Gregory F. Cooper, and David E. Heckerman. Reflection and action under scarce resources: Theoretical principles and empiricial study. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 1121–1127, Detroit, 1989.

[14] A. E. Howe and P. R. Cohen. Failure recovery: A model and experiments. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, 1991.

[15] François Félix Ingrand, Jack Goldberg, and Janet D. Lee. SRI/GRUMMAN Crew Members' Associate Program: Development of an authority manager. Final Report SRI Project 7025, Artificial Intelligence Center, SRI International, Menlo Park, CA, March 1989.

[16] 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.

[17] S. Kambhampati and J. Hendler. A validation-structure-based theory of plan modification and reuse. *Artificial Intelligence*, 55(2):192–258, 1992.

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

[19] J.E. Laird. Integrating planning and execution in Soar. In *Proceedings of the AAAI Spring Symposium on Planning in Uncertain, Unpredicatable, or Changing Environments*, Stanford, Ca, 1990.

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

[21] John D. Lowrance. *Evidential Reasoning with Gister-CL: A Manual.* Artificial Intelligence Center, SRI International, 333 Ravenswood Avenue, Menlo Park, CA, Febuary 1994.

[22] John D. Lowrance, Thomas D. Garvey, and Thomas M. Strat. A framework for evidential-reasoning systems. In Ronald J. Brachman and Hector J. Levesque, editors,

Uncertain Reasoning, pages 611–618. Morgan Kaufmann Publishers Inc., San Mateo, CA, 1990.

[23] John D. Lowrance, Thomas M. Strat, and Thomas D. Garvey. Application of artificial intelligence techniques to naval intelligence analysis. Final Report SRI Contract 6486, SRI International Artificial Intelligence Center, Menlo Park, CA, June 1986.

[24] John D. Lowrance, Thomas M. Strat, Leonard P. Wesley, Thomas D. Garvey, Enrique H. Ruspini, and David E. Wilkins. The theory, implementation, and practice of evidential reasoning. Final Report SRI Contract 5701, Artificial Intelligence Center, SRI International, 333 Ravenswood Avenue, Menlo Park, CA, June 1991.

[25] John D. Lowrance and David E. Wilkins. Plan evaluation under uncertainity. In Katia P. Sycara, editor, *Proceedings of the Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 439–449. Morgan Kaufmann Publishers Inc., San Mateo, CA, November 1990.

[26] 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.

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

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

[29] Bernhard Nebel and Jana Koehler. Plan modification versus plan generation: A complexity-theoretic perspective. In *Proceedings of the 1993 International Joint Conference on Artificial Intelligence*, pages 1436–1441, American Association for Artificial Intelligence, Menlo Park, CA, 1993.

[30] E. D. Sacerdoti. *A Structure for Plans and Behaviour*. Elsevier, North Holland, New York City, NY, 1977.

[31] M. J. Schoppers. Universal plans for reactive robots in unpredictable environments. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 1039–1046, Milan, Italy, 1987.

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

[33] A. Tate. Project planning using a hierarchical nonlinear planner. Department of Artificial Intelligence Report 25, Edinburgh University, 1976.

[34] S. Vere. Planning in time: Windows and durations for activities and goals. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 5(3):246–267, 1983.

[35] R. M. Washington. *Abstraction Planning in Real Time*. PhD thesis, Stanford University, 1994.

[36] R. M. Washington and B. Hayes-Roth. Practical real-time planning. Technical Report KSL-92-80, Knowledge Systems Laboratory, Stanford University, 1992.

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

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

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

[40] 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, 1994.

[41] David E. Wilkins and Karen L. Myers. A common knowledge representation for plan generation and reactive execution. *Journal of Logic and Computation*, to appear, 1994-5.