RL-TR-94-39
Final Technical Report
May 1994

PLANNING IN DYNAMIC AND UNCERTAIN ENVIRONMENTS

SRI International

Sponsored by
Advanced Research Projects Agency
ARPA Order No. 7513

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government.

Rome Laboratory
Air Force Materiel Command
Griffiss Air Force Base, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-94-39 has been reviewed and is approved for publication.

APPROVED: 

LOUIS J. HOEBEL
Project Engineer

FOR THE COMMANDER: 

JOHN A. GRANIERO
Chief Scientist
Command, Control & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL (C3CA) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

PLANNING IN DYNAMIC AND UNCERTAIN ENVIRONMENTS

David Wilkins
Karen Myers
Leonard Wesley
John Lowrance

Contractor: SRI International
Contract Number: F30602-90-C-0086
Effective Date of Contract: 30 August 1990
Contract Expiration Date: 30 August 1993
Short Title of Work: Planning in Dynamic & Uncertain Environments
Program Code Number: 3E20
Period of Work Covered: Aug 90 - Nov 93

Principal Investigator: David Wilkins
Phone: (415) 859-2057

RL Project Engineer: Louis J. Hoebel
Phone: (315) 330-3655

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by Louis J. Hoebel, RL (C3CA), 525 Brooks Road, Griffiss AFB NY 13441-4505 under Contract F30602-90-C-0086.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	May 1994	Final Aug 90 - Nov 93

4. TITLE AND SUBTITLE
PLANNING IN DYNAMIC AND UNCERTAIN ENVIRONMENTS

5. FUNDING NUMBERS
C - F30602-90-C-0086
PE - 62301E
PR - G513
TA - 00
WU - 01

6. AUTHOR(S)
David Wilkins, Karen Myers, Leonard Wesley, and John Lowrance

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
SRI International
333 Ravenswood Ave
Menlo Park CA 94025

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington VA 22203-1714

Rome Laboratory (C3CA)
525 Brooks Road
Griffiss AFB NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
RL-TR-94-39

11. SUPPLEMENTARY NOTES
Rome Laboratory Project Engineer: Louis J. Hoebel/C3CA/(315) 330-3655

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)
SRI has built a domain-independent planning and execution system called CYPRESS which satisfies the following requirements:

- imperfect input information
- rapid response during execution
- minimal allocated computing resources during execution
- incorporate preplanned plans
- integrate planning and execution
- special technique for optimization of schedules.

CYPRESS contains SIPE-2, a classical hierarchical AI planning system; PRS-CL, a reactive plan execution system; Gister-CL, a suite of evidential reasoning tools and the Grasper-CL system for interface and graphics. CYPRESS also runs in the ARPA/Rome Laboratory Planning Initiatives Common Prototyping Environment (CPE) and uses that system for communication between subsystems.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Planning, Uncertainty, Evidential Reasoning, Execution Monitoring	152
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Contents

List of Figures

Chapter 1

Introduction

SRI International (SRI) is pleased to present this final report under contract F30602-90-C-0086, Planning in Dynamic and Uncertain Environments, to Rome Laboratory (RL) of the USAF and the Advanced Research Projects Agency (ARPA). The research on this project was conducted by SRI's Artificial Intelligence Center (AIC), and hereafter the project will be referred to as UP, for "uncertain planning." This report describes accomplishments made under this contract in our effort to develop a flexible, integrated planning and execution system based on several high-performance artificial intelligence (AI) technologies. More detailed descriptions than those provided in this report can be found in the two previous annual reports on this project, and in papers we have written as described in Chapter 11.

1.1 Relationship with the Planning Initiative

The UP project was part of the ARPA/RL Knowledge-Based Planning and Scheduling Initiative (ARPI), and played an important role in the second Integrated Feasibility Demonstration (IFD-2). Because the needs of ARPI have, in part, directed this research, it is necessary to explain the relationship of this project to other projects. The software developed under UP runs in the Common Prototyping Environment (CPE) developed as part of ARPI, and can use systems developed by other projects in its problem solving process. The communication software developed by other ARPI projects is used both to interact with software of other projects and within our own system.

SRI's Information and Telecommunications Sciences Center (ITSC) also has a project that is part of ARPI. ITSC is applying currently existing AI technologies developed by the AIC to a planning problem from the U.S. Central Command. Their application system, SOCAP (System for Operations and Crises Action Planning), was a major component of IFD-2 in January 1992. The AIC, on the UP project, developed a new prototype technology

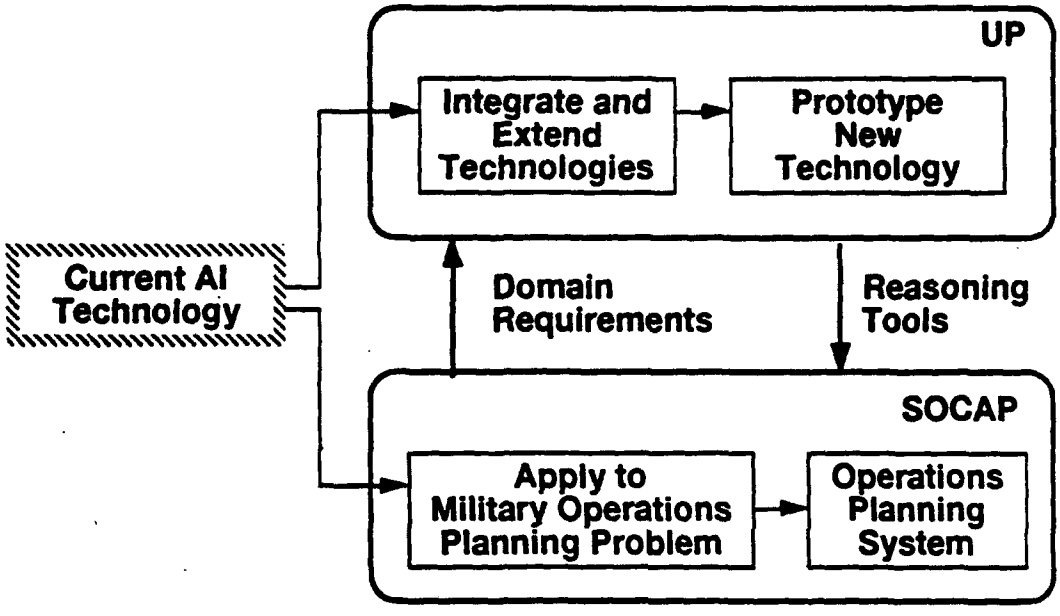

Figure 1.1: Overview of SRI projects

by extending and integrating several existing AI technologies, including those being used on the ITSC project. Figure 1.1 depicts the relationship between these two projects. It is important to note the cooperation: the domain requirements of the ITSC project drove some of the research in the AIC project, and the tools and techniques developed by the AIC project were transferred to the ITSC project rapidly.

1.2 Overview of CYPRESS: A New Technology

Planning in real-world domains is a complex and dynamic process. Automated planning systems must satisfy a number of requirements, including the ability to handle uncertain information, real-time response, intelligent application of standard operating procedures, dynamic plan modification, and interaction with a human planner.

There are several major shortcomings of current AI planning technology for addressing problems in military operations planning:

- Systems require perfect information.
- Systems cannot respond quickly during execution.
- Computing resources cannot be allocated during execution.
- Off-the-shelf plans are not incorporated into new plans.

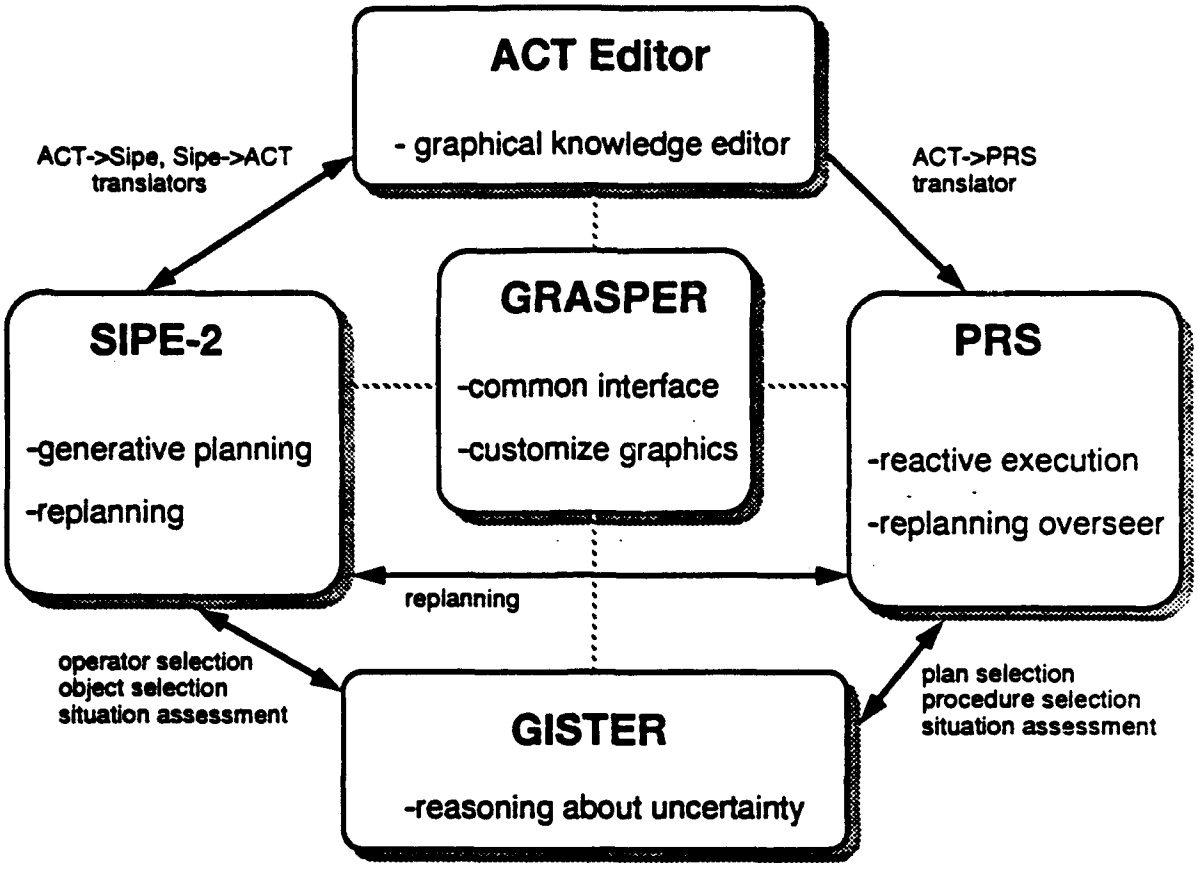

Figure 1.2: The Architecture of CYPRESS

- Replanning and execution are not integrated.
- Systems do not have special techniques for optimization of schedules.

As part of the UP project, the AIC has built a domain-independent planning and execution system called CYPRESS which has been designed to satisfy the above requirements. CYPRESS supports a full range of planning capabilities including action specification, generative planning, reactive plan execution, and dynamic replanning.

CYPRESS is built on top of four AI systems previously developed at SRI, namely SIPE-2, PRS-CL, Gister-CL, and Grasper-CL, along with a system developed specifically for CYPRESS called the ACT EDITOR.[1] Figure 1.2 depicts the architecture of CYPRESS.

SIPE-2 and PRS-CL constitute the two fundamental planning technologies in the system. SIPE-2 is a classical AI planning system capable of generating plans in a hierarchical fashion [33, 36]. PRS-CL is a reactive plan-execution system that integrates goal-oriented and event-driven activity in a flexible, uniform framework [11]. Although these two systems were

[1]SIPE-2, PRS-CL, Gister-CL, Grasper-CL, and ACT EDITOR are trademarks of SRI International.

developed independently, they have been made to work together within CYPRESS: PRS-CL can execute plans produced by SIPE-2 and can invoke SIPE-2 in situations where run-time replanning is required. 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 [20, 28]. SIPE-2 can use Gister-CL to aid in choosing among operators that apply for a given goal. Similarly, PRS-CL can employ Gister-CL to choose among suitable plans for execution at run-time. SIPE-2, PRS-CL, and Gister-CL are not theoretical exercises, but rather implemented, tested systems that have been applied to numerous problem domains. They are described in more detail in Appendices A, B, and C.

PRS-CL and SIPE-2 employ their own internal representations for plans and actions. For this reason, we developed an *interlingua* called the ACT representation [37] that enables these systems to share information. ACT provides a language for specifying the actions and plans employed by both plan-generation and plan-execution systems. At the heart of the formalism is the ACT structure, which corresponds to both an *operator* in the terminology of generative planners, and a *plan fragment* in the terminology of plan execution systems. ACT structures (also referred to as ACTs) provide a common representation language that bridges the gap between these two types of plan-based systems. The ACT EDITOR subsystem supports the creation, display and manipulation of ACTs. In addition, CYPRESS includes translators that can automatically map ACTs onto SIPE-2 operators and PRS-CL *Knowledge Areas*, as well as a translator that can map SIPE-2 operators and plans into ACTs.

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 [14]. 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.

CYPRESS runs in the CPE developed as part of ARPI. Our software has cooperated with ARPI systems from other sites in developing a military operations plan (in particular, General Electric's (GE) Tachyon system [2]), and uses the communication software provided in the CPE (in particular, the Cronus and Knet systems) for this purpose as well as for communication between subsystems of CYPRESS that are running on different machines.

ACTs have enabled SIPE-2 and PRS-CL to cooperate on the same problem for the first time — SIPE-2 generating plans while PRS-CL responds to events, executes the plans produced, and controls the execution of the lower-level actions that have not been planned. Replanning

during execution occurs whenever PRS-CL decides it is necessary, and execution continues during planning with the new plan eventually being merged with the current execution state. This is the first time that a planner and execution system of such complexity have cooperated in this fashion.

The ACT EDITOR provides a graphical knowledge-editor for both SIPE-2 and PRS-CL, since ACTs can be translated into either system. The integrated planning and execution we have demonstrated shows that ACT representational constructs have reasonable computational properties as well as being expressive enough for this domain. The ability to represent all the constructs in PRS and SIPE-2 implies the ACT formalism is sufficient for a wide range of interesting problems, since both of these systems have been applied to several practical domains.

Chapter 2

Summary of Accomplishments

CYPRESS includes the techniques developed to address the major tasks of the UP project. These were to (1) introduce evidential-reasoning methods into an integrated planning system; (2) design a uniform representation for operating procedures based on the knowledge areas (KAs) of PRS-CL, and on representational constructs to support representation and use of expert planning knowledge in SIPE-2; (3) construct a graphical man-machine interface for manipulating this uniform representation that also supports interaction with SIPE-2, PRS-CL, and Gister-CL; (4) integrate the resulting system with other ARPI systems in the CPE; (5) develop techniques to control planning and plan execution that will enable the system to react quickly when necessary, and to consider more alternatives when time permits; and (6) validate the technical developments through a proof-of-concept demonstration in the domain of military operations planning.

The most important of the following accomplishments are described in more detail later in this report. Figure 2.1 depicts graphically how our implementations interact with each other, and indicates the personnel involved in the implementations. The shaded boxes were supported partly by the ITSC project. The following sections mention each of the boxes in the figure as well as several accomplishments not relating to implementation. They are grouped by topic, although many items relate to more than one topic.

2.1 Integrating and Extending Generative and Reactive Planning

- We developed an *interlingua* called the ACT representation [37] that enables our systems to share information. ACT provides a language for specifying the actions and plans employed by both plan-generation and plan-execution systems. Its syntax is formally specified in Appendix E of this report.

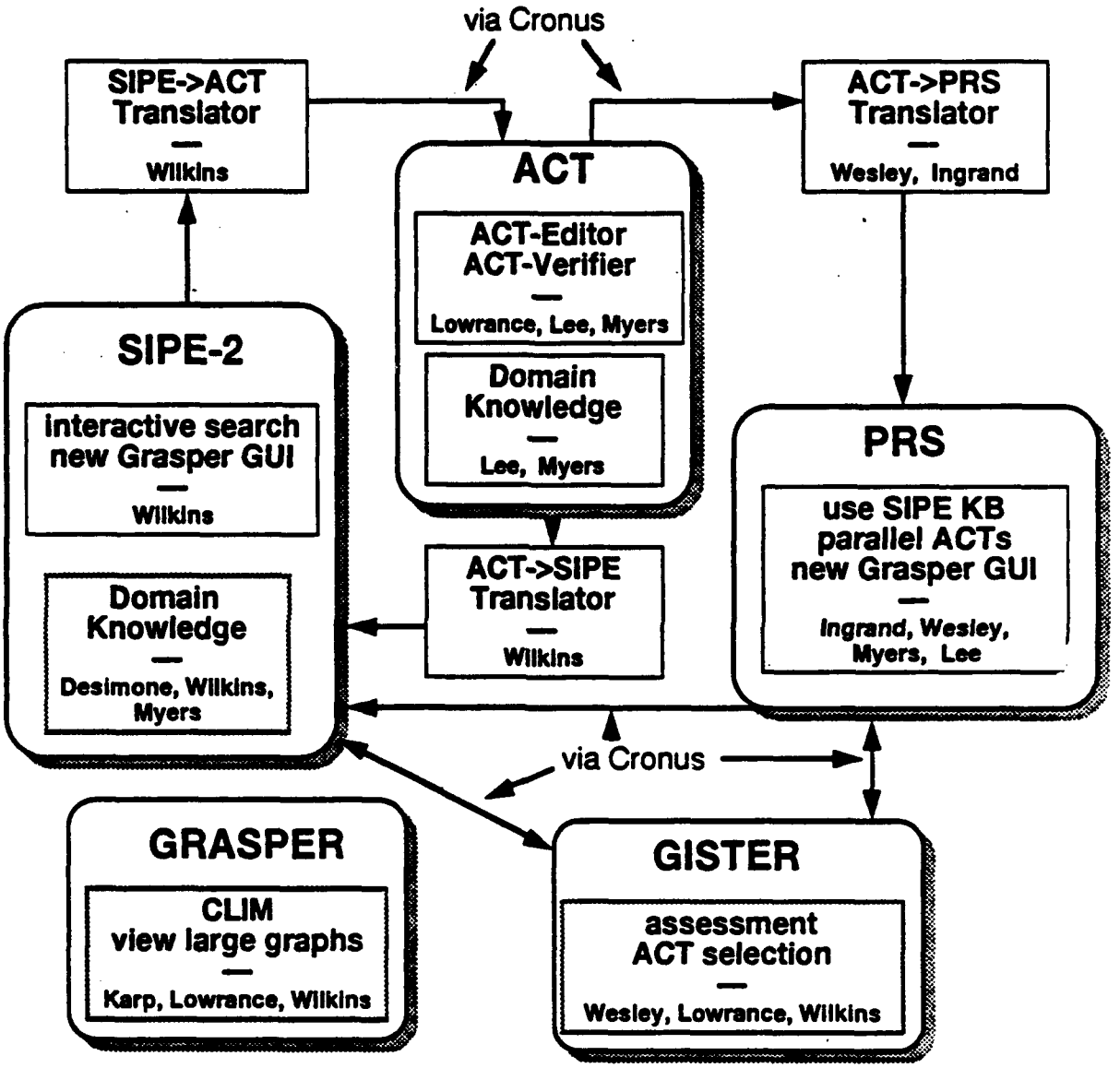

Figure 2.1: Organization and personnel for the software implementation done under UP project. Shaded boxes indicate partial ITSC support.

- We have enabled SIPE-2 and PRS-CL to cooperate on the same problem for the first time so that a generated plan can be executed, and replanning during execution can occur when necessary. A planner and execution system of such complexity have never before cooperated in this fashion.

- We developed a theory of reasoning about locations that is adequate from both the theoretical and the practical perspectives. Our theory applies to technological frameworks ranging from generative planners to schedulers to reactive systems. We extended the IFD-2 domain knowledge to implement this theory. Our effort was inspired by several inaccuracies in the IFD-2 representation that needed to be corrected before the knowledge could support more extended reasoning, such as that done in our demonstration.

- We made several important extensions to SIPE-2 to facilitate its use in military operations planning, to implement the ACT formalism, and to interact with PRS-CL, both by sending plans to PRS-CL and replanning during execution when PRS-CL so requests. These extensions provide a significant increase in capability and ease of use.

- We made several extensions to PRS-CL to handle the new aspects of the ACT representation, to provide a Grasper-based GUI, and to invoke SIPE-2 for replanning during execution. A major extension was the ability to execute parallel actions.

- For efficiency, PRS-CL and SIPE-2 use their own internal representations for plans and actions. We implemented translators from the ACT formalism into the internal representations of both systems, as well as a translator from SIPE-2 to ACT that translates both SIPE-2 operators and plans into the ACT formalism.

- We performed an analysis of the SIPE-2 planning for IFD-2 which resulted in efficiency gains of an order of magnitude.

2.2 Handling Uncertainty and Military Knowledge

- Together with ITSC, we encoded a significant amount of domain knowledge for joint military operations planning. This is an appropriate military problem for our proof-of-concept demonstration and for driving our technology development. Domain knowledge has been encoded in ACTs for use by SIPE-2 for planning and by PRS-CL during execution, and in Gister-CL for reasoning about uncertain information.

- We developed a theory and implemented it (using SIPE-2 and Gister-CL) for evaluating the likelihood that plans will accomplish their intended goals given both an uncertain description of the initial state of the world and the use of probabilistically reliable operators.

- We identified several opportunities for using uncertain information in our design. We developed Gister-CL analyses for the case of selecting an appropriate military unit during planning. To reason about uncertainty while executing within a PRS-CL environment, we developed several methods and techniques for calling and receiving the results of Gister-CL analyses.

2.3 Common Prototyping Environment

- We developed code and a set of conventions for building major software systems, that should provide a standard for specification of all software systems within ARPI. The conventions are described in Appendix D. These standards specify how to load and run a system as well as providing a patch facility. We extended the Cronus and Knet systems of the CPE to conform to these system specifications.

- We installed the Cronus and Knet systems from the CPE and used them in our final demonstration for communication between Gister-CL, SIPE-2, and PRS-CL, both of which support interaction through Knet. For example, SIPE-2 sends plans to PRS-CL and gets back information about the world and replanning requests.

- We extended SIPE-2 to interact with GE's Tachyon system in a loosely coupled manner. Tachyon is able to process extended temporal constraints for SIPE-2 during planning. They communicate by using the Cronus system in the CPE.

- We made progress on accessing multiple and common knowledge bases across all systems. PRS-CL has been extended to access multiple data bases; in particular, it now uses SIPE-2 knowledge structures. SIPE-2 was extended to make it easy to use a uniform frame-access language (designed by the AIC on a concurrent ARPA project). The IFD-2 plan has been generated with all knowledge base references going to a LOOM knowledge base.

- We ported all of our software systems, SIPE-2, Gister-CL, Grasper-CL, and PRS-CL, to the CPE (with support from the ITSC project). This involved conversion from SYMBOLICS Lisp to Lucid COMMON LISP, and from SYMBOLICS windows to CLIM.

2.4 Human Computer Interaction

- We implemented the ACT EDITOR which allows graphical input and editing of plans and operating procedures represented as ACTs. It can appropriately display large, complex ACTs. The ACT EDITOR includes a simplifier to simplify ACTs, and a verifier to check for legal syntax, legal topography, and consistent use of logical predicates.

- We designed and implemented graphical user interfaces (GUIs) based on Grasper-CL for SIPE-2, PRS-CL, and the ACT EDITOR. These GUIs provide a significant increase in capability and ease of use over the original interfaces.

2.5 Documentation and Technology Transfer

- We wrote several papers and documents (see Chapter 11), including a paper describing the SOCAP application, extensive manuals for all systems, a paper on the ACT interlingua, a paper on our theory of locations, and a paper on preliminary investigations of evidential measures for choosing among alternative actions. Some of these papers are in Appendix G.

- We devoted a considerable amount of effort to cooperating with the other participants in ARPI. This cooperation included participating in the design of the KRSL plan language being developed at ISX, attending quarterly ARPI meetings, revamping the Technology Roadmap of ARPI, making our software available to ARPI researchers, and using CPE software in our systems.

- We distributed our software to many ARPI sites during this project, as well as to several non-ARPI sites. This demonstrated the usefulness of our system specification standards, and enabled sharing and evaluation of software in the research community.

Chapter 3

A Common Knowledge Representation for Plan Generation and Reactive Execution

Our research concerned taking appropriate actions in a dynamic and uncertain environment. The dynamic environment requires that the execution system be able to respond to stimuli and make decisions in "real time." Furthermore, we were interested in applications complex enough that the ability to synthesize plans and use these plans to guide the execution system is critical. We developed a representation, called the ACT formalism, for representing the knowledge necessary to both generate plans and execute them while responding to external events. This chapter describes the ACT formalism and its use in CYPRESS.

Two features distinguish our approach: (1) the development of an heuristically adequate system that will be useful in practical applications, and (2) the need to generate complex plans with parallel actions of multiple agents. To achieve these requirements, we started with AI technologies in planning and reactive control (SIPE-2 and PRS-CL) that had already been implemented and shown to be useful in practical applications. However, these technologies did different types of reasoning using different knowledge representations — Sections 3.1 and 3.2.3 describe the difficulties in using a common representation for these tasks. In this chapter, we describe the ACT formalism, which is a common representation that planning and reactive control systems can use for representing and reasoning about plans and the knowledge necessary to generate and execute them. SIPE-2 and PRS-CL were extended and merged to fully implement the ACT formalism.

Developing an interlingua so that different reasoning systems can cooperate on different aspects of the same problem is a central challenge for the field of artificial intelligence. This chapter presents an attempt to do this for a narrowly focused problem, namely getting planning and execution systems to use a common language. We view this narrow focus as critical to achieving success in terms of near-term use in software tools, since a common representation that tries to cover too broad an area runs into serious problems that have been described elsewhere [12].

Our integration of planning and control focuses on domains with the following properties (among others):

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

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

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

The complexity of the plans is what differentiates our approach from previous approaches to the integration of planning and execution. Lyons and Hendricks [22] describe an approach in which the planner monitors the execution of the reactive system and specifies *adaptations* to the reactive system that will improve its performance relative to achievement of the given

goals. They require these incremental adaptations to *improve* system performance before they are added to the reactor. The RAP system [7] also uses simple plans to modify the reactive control of a robot. Such approaches work in the robot problem, where plans are relatively simple, but in the military planning problem, the planner must exercise more control over the execution system. In our system, the planner produces much more complex plans than in either of the above systems, and does not modify the reactor to improve its performance — rather, the planner generates a plan that will be the principal guidance for the reactor. The reactor will generally have no idea how to accomplish the top-level goals appropriately until the planner has generated a plan. The reactor will implement appropriate lower-level behaviors while it is waiting for a plan, and may pose additional problems to the planner during execution.

The development of the ACT formalism is similar in motivation to the development of the KRSL language [15] that is being done as part of ARPI. However, the ACT formalism is more focused, trying only to get planning and execution systems to speak a common language, while the KRSL effort is more ambitious, trying to develop a common language for many types of systems, including systems for planning, execution, scheduling, simulation, temporal reasoning, database management, and other tasks. The ACT formalism has been one of the formalisms driving the design of the KRSL specification for plans.

3.1 Developing a Common Representation

Our uniform representation refers to knowledge provided by the user, e.g., plan fragments and standard operating procedures (SOPs), as *ACTs*. ACTs are used by both the planning and execution systems. This allows both systems to cooperate on the same problem, and allows software tools for graphically editing and creating ACTs to enhance both systems simultaneously and uniformly.

In our implementation, ACTs represent operators in SIPE-2, and KAs in PRS-CL, and are the recommended method for encoding knowledge. We have implemented translators from the ACT formalism to the internal representations of both SIPE-2 and PRS-CL, as described in Section 3.2.5. The plans generated by the planner are translated to instantiated ACTs. The execution system uses procedures encoded as ACTs to respond to minor problems during execution, and may also reinvoke the planner when appropriate. We have implemented an ACT EDITOR (see Chapter 6) to support graphical input and modification of ACTs.

Obviously, the new representation must be capable of supporting the types of reasoning done by both planning systems and reactive control systems. This is no small task, since each system stresses a different type of knowledge and reasoning process. For example, PRS-CL frequently uses conditionals and loops in its procedures, and can monitor the world to

determine what is true. Before extensions were made to support ACTs, PRS-CL did not support parallel execution of separate actions. SIPE-2, on the other hand, stresses parallel actions and uses the planner's reasoning to predict what is true (since sensors cannot sense future states). The ACT representation supports all these types of reasoning, and PRS-CL and SIPE-2 have been extended to support certain aspects of the ACT formalism. These extensions are described in detail elsewhere [35].

A central issue in designing the ACT formalism is the difference between planning and execution. While executing a plan and generating a plan require much of the same knowledge, there are still some important differences between the two activities. The planner is attempting to look ahead to the future consequences of particular courses of action, while the execution system is sensing the world and reacting to incoming information. This difference manifests itself in several ways, including system response to violations of requirements, the triggering of ACTs, and the representation of knowledge about the world. These manifestations result in different interpretations of the ACT formalism during planning and execution. How these differing interpretations are implemented in SIPE-2 and PRS-CL is described in Section 3.2.3.

One important difference is that the planner can permit requirements to be violated, because its plan-critic algorithms can then be used to patch the plan to prevent the violations from occurring. The execution system, however, must fail as soon as a requirement is violated since the world is actually in a state that violates the requirement. It is generally difficult to determine the most appropriate response to an execution failure; thus, the execution system will need knowledge, unnecessary to the planner, about how to respond to failures.

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

Representation of the world state is generally different in planning and execution systems. For example, to be efficient enough to be reactive, PRS-CL maintains only one world model that reflects the current state of the world. By contrast, SIPE-2 must be able to reason about the world in many different future states, which necessarily adds overhead to the representation and reasoning. Most planning systems, including SIPE-2, encode a new world state by recording changes since a prior world state. Thus, the planner requires knowledge, unnecessary to the execution system, relating to modeling how the world state changes as

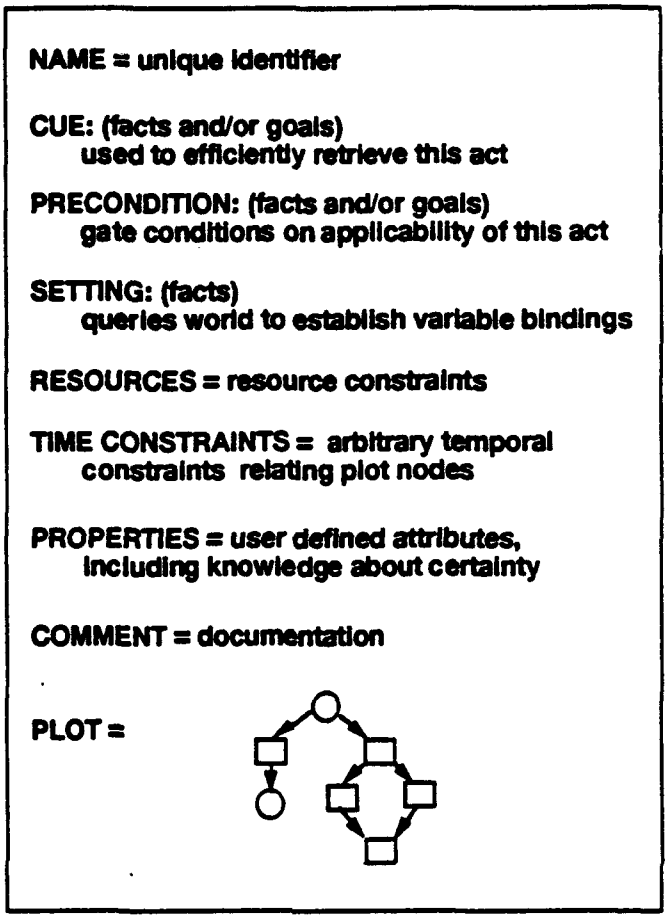

Figure 3.1: ACT Slots

actions are taken. Another aspect of this difference is that the execution system can use the world as an information source and use sensors to query some property in the world. Because the planner reasons about world states that may exist in the future, it must model every proposition in which it is interested, again requiring knowledge that may be unnecessary to the execution system.

A final issue of practical importance is that the ACT representation must be clear enough to enable users to understand plans and SOPs represented in it, and to allow knowledge engineers to encode domain knowledge as ACTs. Thus, it was necessary to balance the power of the representation with its perspicuity. Section 3.2 describes the decisions we made in this regard.

3.2 The ACT Formalism

Each ACT represents a plan or a piece of domain knowledge about an action or set of actions that can be taken. Each ACT consists of a number of predefined slots and a set of nodes representing actions. Each slot and node uses a combination of a set of predefined metapredicates and logical formulae to represent its knowledge. The predefined slots occur in every ACT, and are shown in Figure 3.1, which describes briefly the idea behind each. The slots *Cue*, *Precondition*, and *Setting* have as values logical formulae describing goals and/or facts, while the other slots represent information in other forms.

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

While the metapredicates that may appear on the plot nodes and slots are described in detail in Section 3.2.3, a brief summary of the metapredicates, listed in Figure 3.2, is necessary before explaining the slots. The `Test` metapredicate is used to determine whether a formula is true in the database or world without taking actions to achieve it. The `Use-Resource` metapredicate makes a declaration of resources that will be used by the ACT. Three metapredicates can be thought of as specifying actions: `Achieve`, `Achieve-By`, and `Wait-Until`. `Achieve` directs the system to accomplish a goal by any means possible, `Achieve-By` is similar but directs the system on how to accomplish the goal, and `Wait-Until` directs the system to wait until some other process, agent, or action has achieved a particular condition. The `Require-Until` metapredicate sets up conditions that must be maintained over a specified interval, and the `Conclude` metapredicate specifies any effects that are accomplished by an action (in addition to the predicates mentioned in the action-metapredicate specifying the action).

3.2.1 ACT Slots

The Cue, Precondition, and Setting slots, as well as plot nodes, are filled in with logical formulae. Each slot can have an entry for zero or more metapredicates. The formulae used as the values for each metapredicate are built from predicates specified in first-order logic, connectives, and the names of ACTs. The metapredicates recognized in the ACT formalism are shown in Figure 3.2, which hints at the syntax accepted by each where P stands for a formula composed of first-order predicates and connectives. The syntax for metapredicates is described fully in Appendix E. The predicates will have three truth values: true, false,

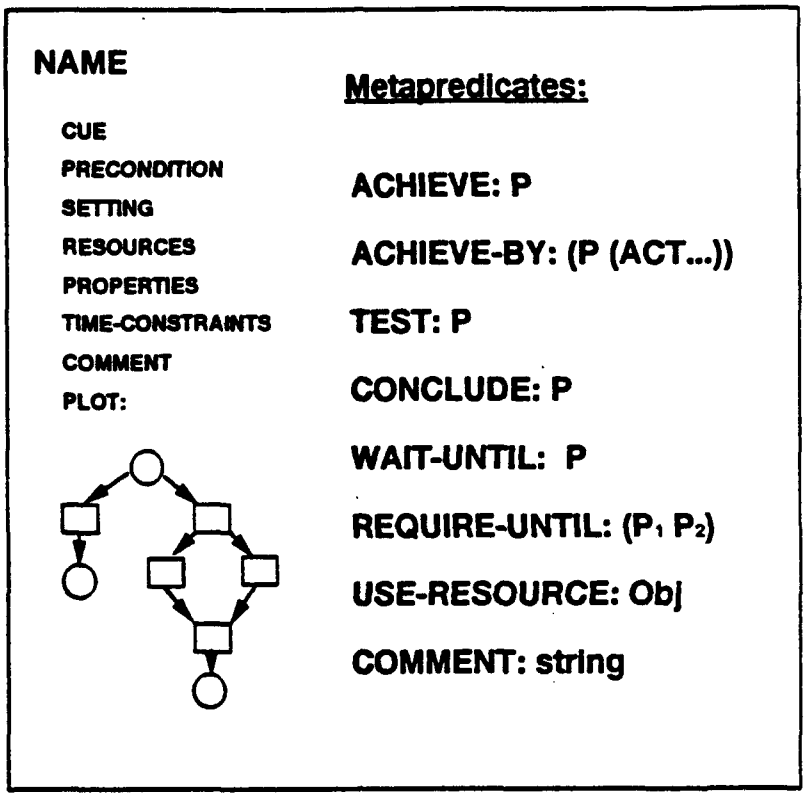

Figure 3.2: ACT Metapredicates

and unknown. Predicate names can be declared as open or closed, to instruct the system as to whether the closed world assumption applies. In our implementation, PRS-CL currently restricts the use of unknown truth values to open predicates, while SIPE-2 implements it for all predicates. If necessary, PRS-CL could be extended to handle unknown truth values in all cases.

Static information represented as binary predicates can be stored in a sort hierarchy, and the variables in our formalism are typed, with the types corresponding to classes in the sort hierarchy.[1] The sort hierarchy should be used whenever possible for efficiency [33].

All plot nodes and all slots except *Name* allow arbitrary user properties (which can be used by users and their programs but are ignored by the system) and the Comment metapredicate. Table 3.1 lists the metapredicates the system recognizes for each slot. The slots with no metapredicates require further explanation. Name is a symbol that names the ACT, and Comment provides documentation (generally, just a string). Plot is a set of plot nodes, and all metapredicates are valid for a plot node.

[1]SIPE-2 has such a sort hierarchy implemented and this is currently used in both PRS-CL and SIPE-2. This may someday be replaced by a more expressive frame system, e.g., LOOM [23].

Slot	Metapredicates
Cue	Achieve, Test, Conclude
Preconditions	Achieve, Test
Setting	Test
Resources	Use-Resource
Time-Constraints	none
Properties	none
Comment	none
Name	none

Table 3.1: Metapredicates Allowed on Slots

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

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

The Cue is used to index and retrieve an ACT for possible execution. Cue specifications should be short so that the system can rapidly identify a subset of potentially executable ACTs. This is important in PRS-CL to maintain real-time response. Complicated conditions should be put in the Precondition or Setting. Restrictions on the use of metapredicates in the Cue are given in Section 3.2.3. Generally, a `Test` metapredicate in the Cue indicates an event-driven ACT that responds to some new fact becoming true, while an `Achieve` metapredicate indicates a goal-driven ACT that will be used to solve system goals. A `Conclude` metapredicate is used to deduce further consequences of conclusions. The Cue corresponds to the "purpose" in a SIPE-2 operator, the "trigger" in a deductive operator, and the "invocation condition" in a PRS-CL KA.

The Precondition specifies the gate conditions (in addition to the Cue) on applicability of the ACT. The ACT is not invoked unless its Cue and Precondition are true. The Setting specifies conditions that are expected to be true in the world, and is used to instantiate variables; O-Plan refers to such conditions as "query conditions" [4]. Although both are gating conditions, the Setting is separated from the Precondition to make the ACT more easily

understood. Generally, the Precondition and Setting will consist of a `Test` metapredicate (see Section 3.2.3).

The *Time-Constraints* slot is used to specify time constraints between plot nodes that cannot be represented by ordering arcs. Many dependencies between different military actions require such constraints. For instance, cargo offload teams should arrive at the same time as the first air transport arrives at the airport. The ACT syntax for Time-Constraints allows specification of any of the 13 Allen relations [1]. Rather than implementing time constraints displayed as text, we plan to have the option of specifying them by different color arcs between nodes (perhaps also with a textual representation in the properties slot).

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

In our ACT implementation, the temporal constraints in the Time-Constraints slot are translated for processing by GE's Tachyon system. Tachyon [2] was chosen as the external software module for propagating the temporal constraints because it had the required capabilities, was readily available, and was known to be efficient. Tachyon is an efficient implementation of a constraint-based model for representing and maintaining qualitative and quantitative temporal information. SIPE-2 calls Tachyon, which propagates these constraints and combines them with the "commonsense" constraints that it represents internally, returning an updated set of time windows on the plan nodes which are inserted into the plan. The temporal constraints are currently ignored by PRS-CL, although PRS-CL could be extended to support them.

Figure 3.3 is an example ACT (not all of it fits on the screen) taken from the screen of the ACT EDITOR, a system we implemented for viewing and editing ACTs (see Chapter 6). It is an ACT that was originally written as a SIPE-2 operator in the military operations domain and was translated by our ACT-to-Sipe translator. The slots are displayed on the left side of the screen and the first few plot nodes are on the right. This ACT is a planning action designed to deploy an air force to a particular location. The Cue is used to invoke the ACT when the system has the goal of achieving such a deployment. The Preconditions enforce various constraints on the intermediate locations to be used in the deployment. The Setting essentially looks up the cargo that must go by air and sea for this deployment. The plot is

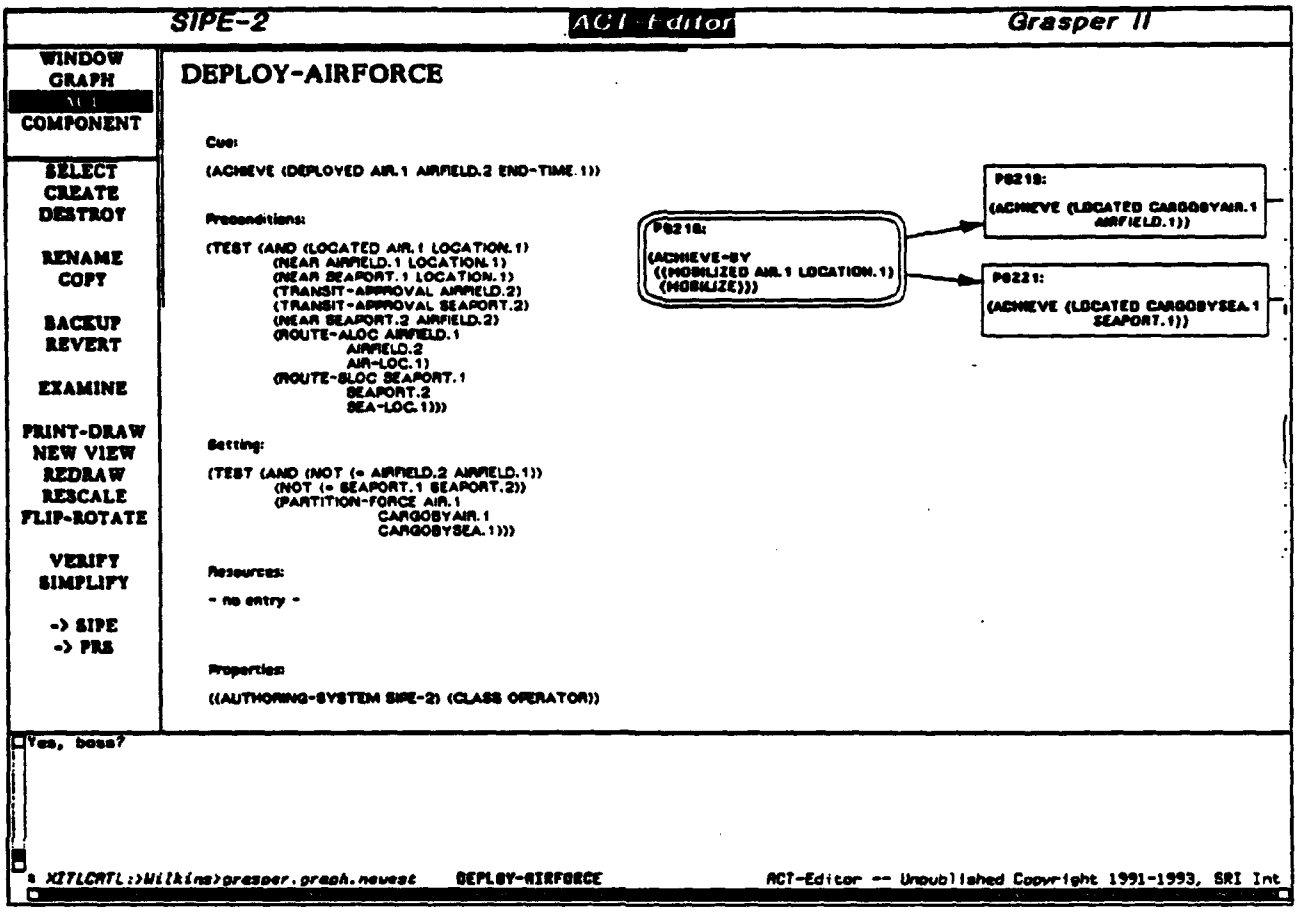

Figure 3.3: Deploy Airforce Operator as an ACT

described in Section 3.2.2.

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

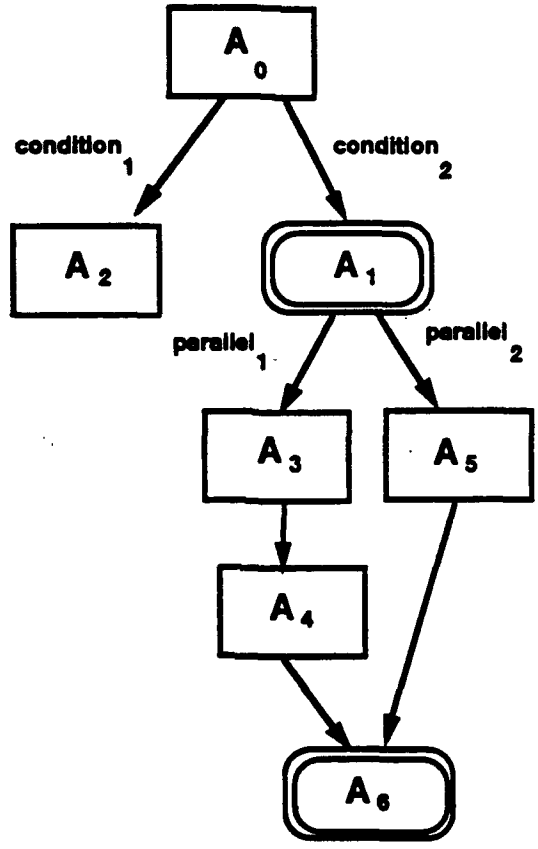

Figure 3.4: ACT with Graphical Display of Plot

3.2.2 Plots

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

We investigated several alternative representations for plots that combine the conditionals and loops typically used in execution systems with the parallel actions typical of planning systems. The problem with most alternatives is that the complexity of the representations makes them hard to understand when nodes have various combinations of disjunctive and conjunctive arcs either incoming or outgoing. Our solution, which is reasonably perspicuous and still general, is to define two types of nodes, *conditional* (or disjunctive) and *parallel* (or conjunctive). All arcs coming into and going out of a conditional node (drawn as a single-

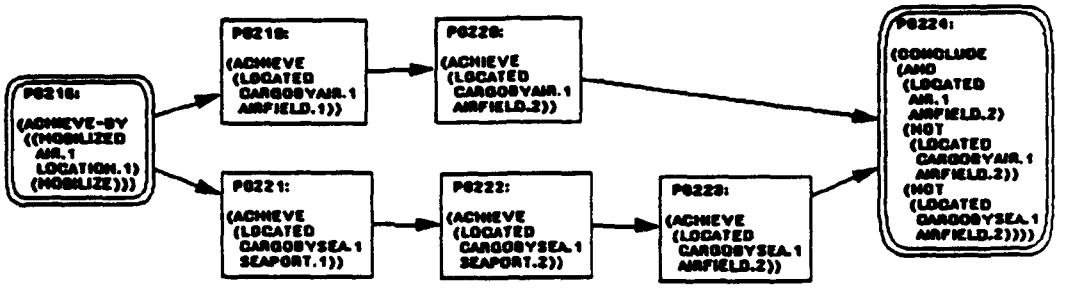

Figure 3.5: Plot of Deploy Airforce ACT

border rectangle) are disjunctive, while all arcs coming into and going out of a parallel node (drawn as a double-border oval) are conjunctive. This is depicted in Figure 3.4.

Two consequences of this handling of multiple edges are important: (1) If a node has zero or one incoming edges and zero or one outgoing edges, it is irrelevant whether it is a conditional or a parallel node — they are equivalent. (2) If one action is to be activated by only one of its incoming edges and must activate all of its outgoing edges, then it must be represented by a conditional node that collects the incoming edges followed by a parallel node that collects the outgoing edges. The metapredicates may occur on either of these two nodes, while the other node need not have any metapredicates.

Figure 3.5 shows the plot of the Deploy Airforce ACT previously discussed. The first plot node specifies that the air force is to be mobilized. It is a parallel node, so its successors will be invoked in either order or at the same time. The successors specify that the air cargo will be moved to the final destination in parallel with moving the sea cargo to two intermediate ports and finally to the final destination. The final node joins the parallel actions and posts conclusions about the locations of the air force and its subparts.

3.2.3 Metapredicates

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

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

Table 3.2: Metapredicate Grouping for Execution in Plot

differently, and execution systems with architectures different from PRS-CL may handle failures of `Require-Until` metapredicates differently.

Figure 3.2 lists the metapredicates that may appear on each plot node. For purposes of ordering their execution, these metapredicates are partitioned into three groups, as shown in Table 3.2. The environment-metapredicates, `Test` and `Use-Resource`, are executed first by the execution system. During planning, `Use-Resource` makes a declaration that will be used by the planner (in SIPE-2, the plan critic algorithms satisfy resource requirements), and a `Test` is generally ignored while expanding a plan with an ACT except in Cues and Preconditions. The action-metapredicates, `Achieve`, `Achieve-By`, and `Wait-Until`, are executed next, and there should be only one of these on a node. The effects-metapredicates, `Require-Until` and `Conclude`, are executed last by the execution system. During planning, `Require-Until` sets up a protection interval that the planner must maintain, and `Conclude` specifies any effects that are additional to the predicates mentioned in the action-metapredicate of the node.

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

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

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

TEST: This metapredicate specifies facts (formulae containing predicates) that can be queried in the database and/or, for the execution system, by sensing the world. When a `Test` is in the Precondition or Setting slots, it is used to determine if the ACT should be executed after it becomes relevant. A `Test` in a Cue is used to determine whether the ACT is eligible for execution, and indicates the system is testing a condition. During execution, if one or more of the specified facts in the Cue has just been posted in the database and the remaining facts are true in the database, then the ACT will be considered relevant but not yet ready for execution.

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

CONCLUDE: This metapredicate appears only in plot nodes, and specifies the facts (predicates) to be concluded in addition to the predicates mentioned in the action-metapredicate of the node. Disjunctions are not allowed. During planning, `Conclude` is used to reason about a different possible future world, and during execution, `Conclude` formulas are posted to the database, possibly triggering the execution of event-driven ACTs. A `Conclude` in a Cue triggers an event-driven ACT that responds to some new fact becoming true. During planning, this is used to trigger deductive operators to deduce context-dependent effects of an action.

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

ACHIEVE-BY: This metapredicate appears only in plot nodes, and specifies the goals to be achieved as well as a restricted set of ACTs, one of which must be used to achieve

the goals. This provides a means for guiding the system on how to achieve a goal. In SIPE-2, this is translated to a `process` node for singleton ACTs and a `choiceprocess` node when more that one ACT is given.

REQUIRE-UNTIL: This metapredicate appears only in plot nodes, and specifies a condition to be maintained until another indicated condition for terminating this requirement occurs (i.e., a protection interval). The accepted syntax for `Require-Until` has two options. The general one is (`req-wff term-wff`). Here, `req-wff` is a formula to be maintained until the termination condition `term-wff` becomes satisfied. The shorter option is simply `term-wff`, where the `req-wff` is assumed to be the goal formula specified for either the `Achieve` or `Achieve-By` metapredicate in the same node. An error arises if no such goal can be identified. In SIPE-2, a `Require-Until` is specified by the `protect-until` construct, and the plan critics will modify any partial plan that violates a `Require-Until` so that the final plan will satisfy all `Require-Until` instances.

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

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

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

USE-RESOURCE: This metapredicate in the Resources slot means each of its arguments is a resource throughout the plot. Our interpretaton of `Use-Resource` is based on the

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

3.2.4 Using ACTs

To show how the various constructs in the ACT formalism are used to represent plan generation and execution knowledge, we will look at three example ACTs from the military domain. A description of this domain can be found elsewhere [38], but the ACTs shown are self-explanatory. By using ACTs, we have demonstrated SIPE-2 and PRS-CL cooperating on the same problem for the first time. SIPE-2 generates large military operations plans while PRS-CL responds to events, executes the plans produced, and performs lower-level actions that have not been planned.

First, we briefly describe how the interaction between SIPE-2 and PRS-CL currently takes place in our implementation, although this interaction is still an area of ongoing research. While both the planning and execution systems can use ACTs at all levels of detail, it is necessary in realistic domains to fix a level of detail below which the planner will not plan. Planning to the lowest level of detail is often not desirable, and the combinatorics can be overwhelming. For example, it is not desirable to plan large military operations down to the minutest detail. Similarly, it is often not desirable for the execution system to respond to certain high-level goals without a plan. For example, a reactive system should not start executing a Desert Storm-sized operation without first having a plan. ACTs at interim levels of detail may be advantageously used by both planning and execution systems.

In the military domain, we fixed a level of detail to which SIPE-2 will plan. The more detailed actions are left to PRS-CL. This is accomplished by using the *class* property to specify which ACTs are to be used by SIPE-2 and/or PRS-CL. SIPE-2 produces complex plans at this level that are translated to one large instantiated ACT for execution by PRS-CL. (We are developing techniques for splitting such an ACT into a set of smaller ACTs.) This ACT

will be the only one that responds to the top-level goal (in the military domain), so that PRS-CL will simply react to events until the planner generates a plan and the top-level goal is posted in PRS-CL. This plan will be the principal guidance for the reactor. The reactor will implement appropriate lower-level behaviors while it is waiting for a plan, and may pose additional problems to the planner during execution.

We will discuss three example ACTs. Figures 3.3 and 3.5 show the ACT for deploying an air force that can be used by both SIPE-2 and PRS-CL. We then discuss a lower-level ACT used only by PRS-CL, and finally describe a deductive ACT used by both SIPE-2 and PRS-CL to deduce consequences of actions.

The Cue tells the system to use the Deploy Airforce ACT to achieve a goal of deploying an air force to a particular air field by a certain time. The Precondition and Setting must be true in the world state before the operator can be applied. The Precondition in Figure 3.3 requires the initial position of the air force to be known, and finds intermediate seaports and airports that are on known routes to the destination and which have transit approval. The Setting constrains the two airports and seaports to be different and partitions the air force into two subparts (by matching a predicate in the database). Planning variables are constrained to be in a certain class based on their name; for example, seaport1 and airfield1 are variables that are constrained to be in the classes seaport and airfield respectively.

The Plot, shown in Figure 3.5, is used either to expand a plan by inserting the plot as a subplan or to begin execution of deployment. The Plot begins with a parallel node that uses an **Achieve-By** metapredicate to force the mobilization of the air force by using the Mobilize ACT. The nodes following this can then be executed in any order or simultaneously. These nodes begin two sequences of conditional nodes that use **Achieve** metapredicates to cause the subparts of the air force to travel in parallel by air and by sea via different locations to the destination. Finally, there is a parallel node, which joins these two parallel threads and aggregates the subparts together when they have all reached the destination.

Figure 3.6 shows the Lookout-Red ACT for establishing lookouts. In our current implementation of the military domain, the planner does not plan down to the level of establishing lookouts, so this ACT is used only by PRS-CL during execution, although it could be used by SIPE-2 if the planning was extended to a lower level of detail. Lookout-Red is one alternative way to achieve a lookout, and the Precondition specifies it should be used only when the terrain is dangerous (represented here by the **code-red** predicate). The Setting finds the correct site for the lookout and the correct sector for the supporting air cover. There is a **Use-Resource** metapredicate in the Resources slot which requires that a **tfighter** resource be available before this ACT can be executed.

The plot of Lookout-Red first uses an **Achieve** metapredicate to get the fighters to the correct location. The second node in the plot achieves an air cover of a certain land sector and

LOOKOUT-RED

Cue:

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

Preconditions:

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

Setting:

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

Resources:

(USE-RESOURCE TFIGHTER.1)

Properties:

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

Comment:

INSTALL A LOOKOUT USING AIR COVER

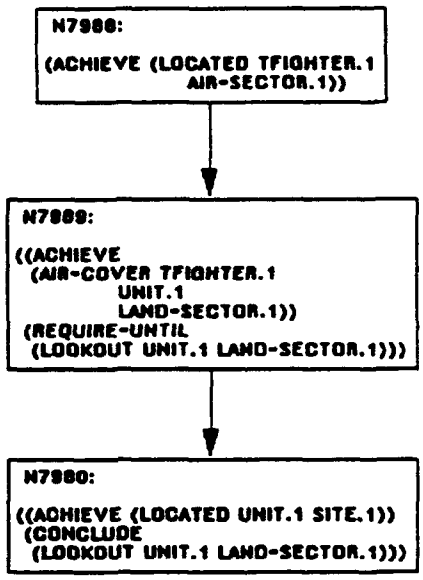

Figure 3.6: An ACT for Establishing a Lookout

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

Another type of knowledge used by some plan generation and execution systems is knowledge for deducing the context-dependent effects of actions. Such a capability greatly enhances the expressive power of a planning system, and is a feature of SIPE-2. The ACT formalism supports this type of knowledge. For example, the Located-sector-up ACT in Figure 3.7 is used to deduce the new region in which a movable object is located after that object has just moved to a new sector (a region is at a higher level of abstraction and may contain several sectors). This ACT is used by both SIPE-2 and PRS-CL to update the world model whenever an object is moved. The Cue of Located-sector-up specifies an event-driven ACT that responds

LOCATED-SECTOR-UP

Cue:

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

Preconditions:

- no entry -

Setting:

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

Resources:

- no entry -

Properties:

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

N8143:

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

Figure 3.7: An ACT for Deducing Locations

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

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

3.2.5 Translators

SIPE-2 and PRS-CL run their own internal representations for efficiency reasons. Thus, it is necessary to translate ACTs into these representations and back. Section 3.2.3 described some of the details of translating metapredicates.

The SIPE-to-ACT translator can translate all SIPE-2 operators to the ACT formalism, as well as translating fully instantiated plans. Thus, all operators originally written for SIPE-2 can be viewed and modified with the ACT EDITOR (as shown by the example in

Figure 3.3), and new operators can be created using the ACT EDITOR. The system cannot yet translate a partial plan with uninstantiated variables into ACT, since not all possible constraints on variables generated by SIPE-2 have a corresponding formulation in ACT. This could be handled by using the Properties slot to store an internal SIPE-2 representation, but we have not found it necessary to translate partially developed plans.

The ACT-to-SIPE translator translates all ACTs into either SIPE-2 operators or plans. This enables SIPE-2 to plan with ACTs that have been input by the user in the ACT EDITOR. The temporal constraints in the Time-Constraints slot are translated for processing by Tachyon. The ACT-to-PRS translator converts ACTs into a representation that can be interpreted and executed in real time under the control of PRS-CL. This makes all SIPE-2 operators accessible to PRS-CL by running SIPE-to-ACT and ACT-to-PRS. Section 3.2.3 described this process. The constraints in the Time-Constraints slot are currently ignored when translating to PRS-CL, although it is clear that PRS-CL could be extended to support them. We have not implemented a PRS-to-ACT translator, as PRS-CL does not have structures that will be passed to SIPE-2.

Chapter 4

Using CYPRESS

We successfully implemented all aspects of CYPRESS as described in Chapter 1.2. This involved porting all our existing systems to the CPE, and extending them as described in this report. Grasper-CL, SIPE-2, PRS-CL, the ACT EDITOR, and Gister-CL all now run in Lucid CommonLisp and CLIM 1.1, and are available for distribution to the ARPI community. New code was first developed and tested in the SYMBOLICS environment since its debugging capabilities allowed us to develop code at a much faster rate the could be done in the Lucid environment. This worked well in practice. Our code runs on both SUN and SYMBOLICS since it uses macros with conditional compilation flags for all functions that are implemented differently in the two environments. A significant amount of time was spent doing system debugging of the CLIM code.

In this chapter, we describe how to use CYPRESS and describe the final demonstration on this project, which is an example of one such use.

4.1 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 activities with interactions among the subsystems. Below, we describe how a user might exercise the full extent of CYPRESS during a plan-generation/plan-execution session. This description follows the use of CYPRESS in our final demonstration.

The first step for any application is to represent the possible actions for the domain. In CYPRESS, these actions would be defined as a collection of ACTs using the ACT EDITOR. These ACTs would next be translated, by clicking an ACT EDITOR command, into the

internal operator representation of SIPE-2 for plan generation. PRS-CL can execute ACTs directly as the translation to internal representations is done automatically by PRS-CL.

Based on the operators translated from the ACT representation, SIPE-2 could then be used to generate plans in response to user-specified goals. The user can instruct SIPE-2 to use Gister-CL to make certain decisions in the planning process either by setting appropriate parameters or by making an interactive request (details can be found in the SIPE-2 manual). If the user indicates that he or she wishes SIPE-2 to use Gister-CL, SIPE-2 will automatically call Gister-CL during the planning process to reason about uncertain information, and, if requested, trace the Gister-CL analysis in another window. In particular, Gister-CL 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. The user must supply the appropriate evidential analysis for the given problem domain and uncertain information.

The plan generated by SIPE-2 would be at some fixed level of abstraction deemed appropriate (by the domain programmer) for execution by PRS-CL. This is done by specifying whether or not an ACT should be used by SIPE-2 (see the ACT EDITOR manual). SIPE-2 should be used to plan only to a level that can be reasoned about ahead of time. The dynamic nature of many domains makes it unreasonable to try to plan everything to the lowest detail. Rather, it is the responsibility of the plan execution system to further refine and adapt the plan to the actual state of the world during the execution process.

Once a plan is generated using SIPE-2, the user can click on a SIPE-2 command to translate it into an ACT. If desired, the user could then make modifications or enhancements to the plan ACT using the ACT EDITOR. The final ACT can be executed directly by PRS-CL (internally, PRS-CL will first translate it to a KA for efficiency reasons).

During execution, PRS-CL will use additional domain-specific ACTs, not used by the planner, to flesh out the sub-goals of the plan (as appropriate). Gister-CL could be called by PRS-CL during plan execution to aid in choosing among multiple KAs that may apply in a given situation. Calls to Gister-CL would be made by the domain-specific meta-ACTs being executed by PRS-CL. Thus, when writing such ACTs, the domain programmer will indicate those situations in which PRS-CL should call Gister-CL.

CYPRESS supports a certain amount of replanning. Should an unexpected problem arise during execution of the plan, PRS-CL would automatically recognize the problem and call SIPE-2 to produce a new plan. PRS-CL would continue executing those portions of the plan that were not affected by the problem, until receiving a new plan from SIPE-2. At that point, PRS-CL would reconcile the new plan with its current state and continue execution.

CYPRESS has recently been applied to joint military operations planning, using the subsystems as described in this section to generate and execute a plan. SIPE-2 used knowledge

provided in the ACT EDITOR to generate employment and deployment plans for dealing with specific enemy threats, using Gister-CL to choose military units based on uncertain information about both friendly and enemy units. The plan was then translated to an ACT for execution by PRS-CL. PRS-CL employed a suite of lower-level ACTs (which could again be graphically modified by the user) that define the execution steps for the actions that the planner assumes as primitive. PRS-CL responded to unexpected events with ACTs that encode standard operating procedures. When the plan became compromised, PRS-CL invoked SIPE-2 to replan while continuing execution.

4.2 Final Demonstration

We developed a proof-of-concept demonstration of the use of CYPRESS to generate and execute plans, making use of uncertain intelligence reports, for an extended version of the defense scenario that was used in IFD-2. This application is described in the paper "Applying an AI Planner to Military Operations Planning" in Appendix G. This demonstration has been given (at least in part) several times, as documented in Chapter 11. Detailed instructions for running this demonstration are given in Appendix F. In this section, we give an overview of the demonstration.

The AIC participated in the encoding of the domain knowledge of the defense scenario which was done primarily by ITSC. The AIC used knowledge and plans from this domain during development and testing of all our software. SIPE-2 successfully generated employment plans for dealing with specific enemy COAs, 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. PRS-CL successfully executed these plans and applied lower-level standard operating procedures. Most of the input came from military databases; around 100 plan operators in SIPE-2's input language were developed to describe military operations that can achieve specific employment or deployment goals. The final plans contain approximately 200 primitive actions. The input has around 250 classes and 500 objects, 15-20 properties per object, and around 2200 predicate instances. The resulting representations and plans have been validated by military planners at the U.S. Central Command in Tampa, Florida, and at the Armed Forces Staff College at Norfolk, Virginia. A more detailed description of this domain is given elsewhere [38].

The demonstration uses a scenario in which a show of force is made to deter an enemy threat. In particular, ground patrols are established in the region where the threat has been

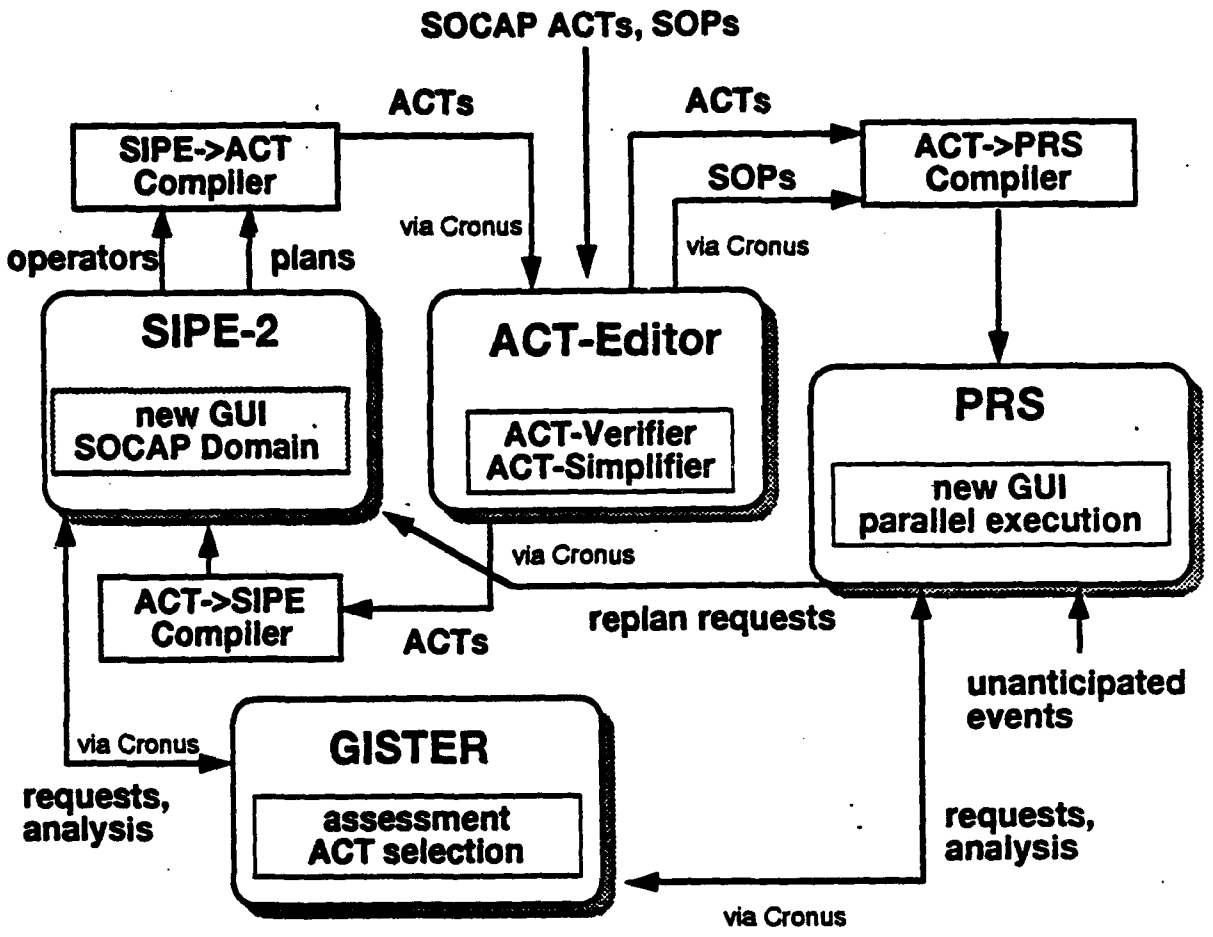

Figure 4.1: Final demonstration

noted. We can also show the planner generating a variety of different plans if so desired. The interaction between the subsystems of CYPRESS in the demonstration is shown in Figure 4.1.

The user uses the ACT EDITOR, the graphical knowledge editor developed on this project, to graphically input or modify ACTs that encode information about how to accomplish certain goals. These ACTs will be translated into both SIPE-2 and PRS-CL. The new SIPE-2 GUI is then used to interactively generate a plan, taking into account the newly provided knowledge. Our evidential reasoning system, Gister-CL, is used to help the planner make appropriate decisions based on its analysis of uncertain information.

After SIPE-2 and Gister-CL generate a plan, it is translated to an ACT, at which point the user can again graphically modify the plan with the ACT EDITOR. The plan is then executed by PRS-CL. PRS-CL uses a suite of lower-level ACTs (which can again be graphically modified by the user) that define the execution steps for the actions that the planner assumes as primitive. The lower-level ACTs emphasize the specific primitive planning action of establishing ground patrols. In the demonstration, ground patrols require constructing a camp,

securing the camp through the use of local patrols both around the camp and in nearby cities, and establishing a strategic lookout force. Lookouts can be established in different ways, depending on the perceived level of danger accorded to the threatening forces.

The scenario illustrates the reactivity of PRS-CL through its ability to adapt plan execution to certain unexpected events. In the demonstration, a force is sent to establish a lookout under safe conditions; however, when the enemy threat becomes sufficiently dangerous an alternative approach to establishing the lookout is adopted that entails air cover while lookout forces are in transit. During execution, PRS-CL responds to unexpected events with standard operating procedures. When the plan becomes compromised, PRS-CL will invoke SIPE-2 to replan while PRS-CL continues execution. PRS-CL will accept the new plan produced by SIPE-2 and incorporate it into its execution.

Chapter 5

Run-time Replanning

The ACT formalism is the interlingua that supports the integration of generative and reactive planning. ACTs have enabled SIPE-2 and PRS-CL to cooperate on the same problem for the first time. SIPE-2 generates plans and translates them to ACTs. PRS-CL then executes these plan ACTs, responds to events, and controls the execution of the lower-level actions that have not been planned. This process is described in Chapter 3.

Unexpected events during execution that cannot be adequately handled by the standard operating procedures encoded as ACTs may require run-time replanning. The ability for PRS-CL to make replanning requests of SIPE-2 completes the integration loop in CYPRESS and is the subject of this chapter. Replanning during execution is invoked whenever PRS-CL decides it is necessary, and execution continues during planning with the new plan (translated to an ACT) eventually being merged with the current execution state.

This is the first time that a planner and execution system of such complexity have cooperated in this fashion. As described in Chapter 3, the complexity of the plans is what differentiates our approach from previous approaches to the integration of planning and execution. Previous approaches work in a single-agent robot domain, where plans are relatively simple. In the military planning problem, many agents are executing in parallel and the planner must exercise more control over the execution system. The reactor will generally have no idea how to accomplish the top-level goals appropriately until the planner has generated a plan, and the reactor will implement appropriate lower-level behaviors while it is waiting for a plan.

5.1 Overview of Replanning

Applying CYPRESS to a particular domain involves using SIPE-2 to plan down to a fixed level of abstraction, then having PRS-CL execute the resultant plan by applying lower-level ACTs

to further expand the goals in the plan. 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. For 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 example, the repair facility described in Chapter 8 is used to recover from failures to maintain some specified condition over a designated time interval (as specified using the `Require-Until` metapredicate) through the use of domain-specific repair routines. The basic SOCAP demonstration described in Appendix F.1 provides an example of this kind of repair.

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. One of the main efforts in the final year of this project was to add a domain-independent *run-time replanning* capability to CYPRESS, which would allow PRS-CL to invoke SIPE-2 to perform replanning for failures that cannot be remediated locally. In particular, we have developed a replanning framework that supports the following behavior:

1. PRS-CL detects an irrecoverable failure during execution of a plan generated by SIPE-2.
2. PRS-CL communicates information about the current state of the 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 (including military operations), since it is infeasible to halt execution while replanning occurs for some parts of the plan.

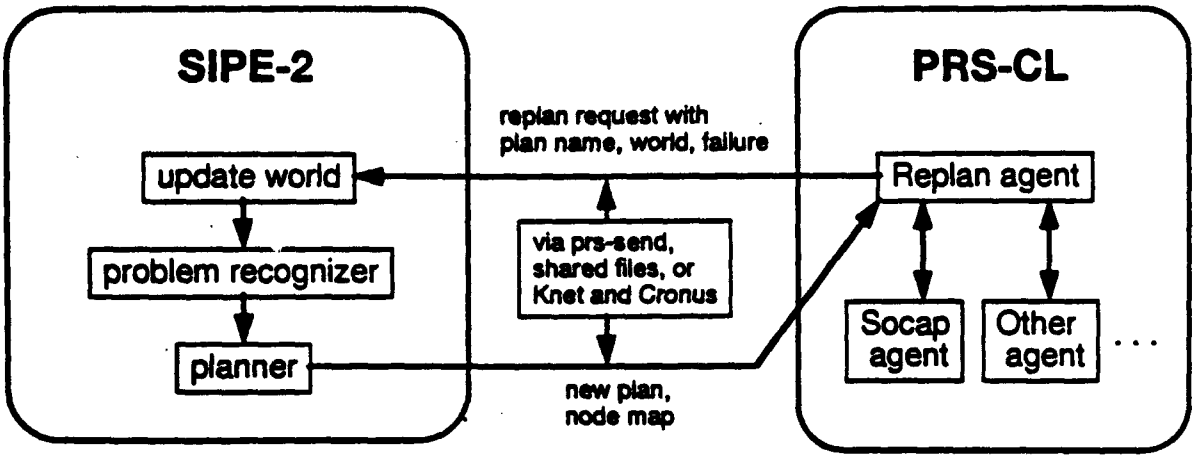

Figure 5.1: Replanning Architecture

5.2 Architecture

Our basic model for replanning is bottom-up in nature, being driven by the activities of the execution system. The execution system is responsible for recognizing failure situations, requesting that the planner produce a new plan, and finally implementing the new plan. In essence, the planner acts as a server to the execution system, providing new plans in response to new world situations as requested by the execution system.

The overall architecture for the replanning framework is depicted in Figure 5.1. An individual PRS-CL application agent (such as the SOCAP agent in the demonstration described in Appendix F) initiates the replanning process upon detection of a failure that it recognizes as being replannable. (The criteria for replannability of failures is discussed further in the next section.) All requests for replanning by application agents are forwarded to a special PRS-CL agent named REPLANNER using the internal PRS-CL message-passing facilities. This special agent performs all necessary communication with SIPE-2, thus enabling any domain agent that requested replanning to be able to continue execution of those parts of the plan that are unaffected by the failure that initiated the replanning request. The REPLANNER agent is identical to other PRS-CL agents; its specialized behavior results from the set of ACTs that it executes.

The message sent to the REPLANNER agent indicates the name of the application agent, a unique identifier for this particular failure episode, the plan ACT being executed, the failed goal in the ACT, and the *execution front*. The plan ACT is the ACT generated by the planner at the level of abstraction shared by PRS-CL and SIPE-2. The execution front is a list of the nodes last successfully executed on each parallel execution thread of the plan ACT. As will be seen, SIPE-2 makes use of this information as part of its replanning procedure.

The REPLANNER agent also sends SIPE-2 relevant information from its database that characterizes the current world state (since the planner is not monitoring the world during execution). Without updates from the REPLANNER agent, SIPE-2 could only generate plans for the original state, rather than the state in which the failure occurred.

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 application agent that initiated the replanning, who then integrates it with its current activities. Under certain circumstances, it is possible that no new plan will be found. In such a case, SIPE-2 notifies the REPLANNER agent of the failure, who in turn notifies the original agent that requested the replanning. This agent will then terminate its execution of the original plan.

Three different forms of communication between the REPLANNER agent and SIPE-2 are supported in CYPRESS. These communication protocols are discussed in Section 5.5.

5.3 Failure Recognition

Failure recognition is built into the plan execution process. Individual PRS-CL agents have been modified to identify situations that should trigger replanning, and respond accordingly. Not all failures in PRS-CL warrant replanning. Certain types of failures are a normal part of plan execution, or can be repaired by using the various local recovery techniques described above. For this reason, only failures at the highest level of PRS-CL execution (that is, in the plan ACT produced by SIPE-2) are considered as candidates 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 system focuses on failures for the most basic types of goals: goals of having some condition being true in the world (corresponding to the ACT metapredicate `Test`) and goals of achievement (corresponding to the ACT metapredicate `Achieve`).

A test goal is considered to have failed by the plan execution system when the condition being tested cannot be established as true. From the perspective of planning, a failed test goal corresponds to the failure of preconditions for application of an operator. Such failures arise because the world has changed in some unexpected way since the original plan was generated.

An achievement goal is considered to have failed by the plan execution system when the condition to be achieved does not hold and there are no ACTs available that can be used to achieve the condition. From the planning perspective, a failed achievement goal corresponds to a lack of suitable operators that could be used to satisfy the goal.

5.4 The Replanning Process

As noted above, SIPE-2 acts as a "planning server" to PRS-CL in that it responds to requests from the **REPLANNER** agent to fix plans that have failed. Two major extensions to SIPE-2 were implemented to meet the replanning requirements. One involved enhancing the existing replanning algorithms of SIPE-2; the other was adding the communication protocols for processing replanning requests from PRS-CL.

The most important change in the replanning algorithms was to provide the option of producing a plan that is different from the original. SIPE-2 does so by preventing the application of the operator that originally inserted the failed action into the plan. This modification to SIPE-2 was necessary to enable replanning of failed actions that the planner treats as primitive (that is, achievement goals). Since all the preconditions of such actions would remain true, the planner would have no reason to believe that the original plan wouldn't work. Without this modification, the planner might simply return the same plan.

When a replannable failure is detected, the **REPLANNER** agent passes SIPE-2 the name of the plan being executed, the action within the plan that failed, the execution front, a unique identifier for this replanning episode, and a description of the current state of the world. The latter can be either a file that completely describes the world, or a list of predicates that are incrementally added to SIPE-2's model of the world. SIPE-2 was extended to process these inputs and find a revised plan.

After finding a revised plan, SIPE-2 communicates two pieces of data to PRS-CL: the plan, and a *node map*. The node map is a partial mapping from nodes in the original plan to nodes in the new plan. The node map contains an entry for every node in the original plan that (1) has changed in some way or has been removed plan, and (2) might possibly be executing. (The latter condition simply means that actions before the execution front or that follow the failed action need not be included in the node map.) In a given node map entry, the node from the original plan maps to a *back-up node* in the new plan. The back-up nodes are used by PRS-CL when it is passed the new plan: if PRS-CL is executing any node in the original plan that has been modified or eliminated (and so has a node map entry), it continues execution in the new plan from the corresponding back-up node.

One of the key technical difficulties in developing the replanning capability was to provide the means to integrate the revised plan with the current activities being undertaken by PRS-CL. This problem is further complicated for asynchronous planning, since execution of the original plan continues while the new plan is generated. We developed a *transformational approach* to integration, in which the execution structures of PRS-CL are modified in accord with the new plan produced by SIPE-2. When presented with a new plan that overcomes some failure, PRS-CL transfers control from the original plan ACT to the new plan ACT.

Actions/subgoals being executed in response to nodes that are not in the node map simply continue execution. For nodes in the node map, PRS-CL aborts their associated activities, and begins execution at the corresponding node in the new plan as specified by the node map. While the transformational approach required the development of a number of complex routines for manipulating the PRS-CL data structures used to track current activities, it has the advantage of preserving undisturbed those activities in progress from the old plan that remain part of the new plan.

5.5 Communication

The communication between the execution and planning systems that underlies the replanning framework was implemented using three different protocols. The individual protocols are useful under various circumstances:

Message-passing: The message-passing protocol is based on the PRS-CL message-passing facilities. It works only when both systems are running in the same LISP image, but is very fast as a result (since the information already exists in the Lisp image and does not need to be transferred).

Shared-files: The shared file protocol involves writing files to disk that can be read by both systems. This form of communication works in any configuration but is the slowest method.

Knet-Cronus: This protocol employs the Knet and Cronus systems of the CPE. It is used when the systems are running on different machines (or in different images). The internal representation of an ACT is sent through Knet servers, directly from the planner to the execution system. This approach is much quicker than writing and reading files: large plans with hundreds of nodes can be transmitted in only a few seconds (perhaps even less than a second – it is not possible to get exact timings).

The communication protocol to be used for any particular run is determined by the binding for the variable `sipe::*AGENT-PROTOCOL*`, as described in the next section.

5.6 Replanning Example

Here, we describe briefly an implemented scenario that illustrates how the replanning capabilities can be used within the SOCAP military operations domain. For more details on running this example, consult Appendix F.2.

The original plan produced in this domain involves four main threads of execution for deterring a set of known threats. Two of these threads correspond to deterrence using ground forces, one corresponds to deterrence using naval forces, and the fourth using air forces. As part of the air deterrence operations, aircraft are moved among various air bases. The use of any individual air base requires explicit transit approval; such approval is granted for all air bases used in the plan as part of the original domain knowledge.

Execution failure can be triggered by rescinding transit approval for any of these air bases. Doing so simply involves deleting the appropriate fact of the form

`(transit-approval <airbase>)`

from the SOCAP agent's database. Removal of such facts lead to plan failure when execution reaches the stage where this approval is required. Without replanning, execution would completely fail at this point. When replanning is enabled, the SOCAP agent will notify the REPLANNER agent, which in turn will issue a replanning request to SIPE-2. Meanwhile, execution of the remaining branches of the original plan continues.

Replanning for this situation produces a modified plan in which an alternative mobilization strategy is employed. In particular, alternative air bases that have transit approval are used in the new plan. This plan is sent to the REPLANNER agent, who forwards it to the SOCAP agent. The SOCAP agent then integrates its current activities with the new plan and continues execution. CYPRESS responds effectively to an unexpected removal of a transit approval during execution of the operation. This is done without disrupting operations unaffected by the transit approval. In addition, the response will be approriate since the planner has checked the future consequences of the new mobilization plan. In this manner, the integration of planning and reactive control provide a system more robust in the face of unexpected events than was previously possible.

5.7 Future Work

The replanning facility described in this chapter provides a basis for flexible plan execution that is capable of adapting to significant unexpected changes in the world. This versatility is critical when operating in dynamic domains where unexpected events may invalidate significant portions of predefined plans. However, there are many interesting extensions to the replanning work described here that would provide a more powerful replanning facility. We describe a number of these extensions here.

One extension involves a more tightly-coupled approach to replanning. In the approach adopted on this project, the interactions between PRS-CL and SIPE-2 are the minimum required for the replanning to succeed. More complex methods would support further communication between the two systems during both execution and replanning, leading 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 plan failures, rather than simply responding to them 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.

Failure recognition could be extended both to support other types of failures, such as the unavailability of resources and failure of maintenance goals (represented using the ACT metapredicate `Require-Until`), and to anticipate failures before getting too deep in the execution.

Chapter 6

The ACT Editor

In military domains, the complexity of the plans and knowledge requires a graphical user interface to input ACTs, and to understand ACTs generated by the planner. We have implemented an ACT EDITOR for displaying, editing, and inputing ACTs. It effectively provides a graphical knowledge-editor for both SIPE-2 and PRS-CL.

The ACT EDITOR displays ACTs graphically as shown in Figures 3.3 and 6.1. Grasper-CL allows great flexibility for the user to customize the display of ACTs. As with all CYPRESS interfaces, command menus are used for interaction. The ACT EDITOR has four command menus: GRAPH, ACT, COMPONENT, and WINDOW. Figure 6.2 shows the command menus associated with each of the four noun modes, hinting at the functionality of the system. Some of the more important capabilities are mentioned below.

Figure 3.3 showed the commands available for acting on a whole ACT. The EXAMINE command was used to change the print width of the ACT to produce the drawing of the plot in Figure 3.5. At the bottom of the command menu is a command to translate this ACT to SIPE-2. Figure 6.1 shows the commands available for acting on individual slots and plot nodes. In this figure, the user has clicked the EXAMINE command and then clicked the first node of the plot. The resulting pop-up window allows the user to edit any of the metapredicates on that node — currently `Achieve-By` is the only metapredicate on node P0216.

A user can input ACTs graphically using the menu-based interface. A user can create or modify an ACT by selecting the appropriate actions from the menu; the ACT EDITOR will prompt for all necessary information. For example, in Figure 6.1 the pop-up window allows editing of the metapredicates, e.g., additional effects of the mobilize action could be added under the Conclude metapredicate. A user who has finished creating or modifying an ACT can use the VERIFY command to invoke the ACT-Verifier to check the syntax and topology.

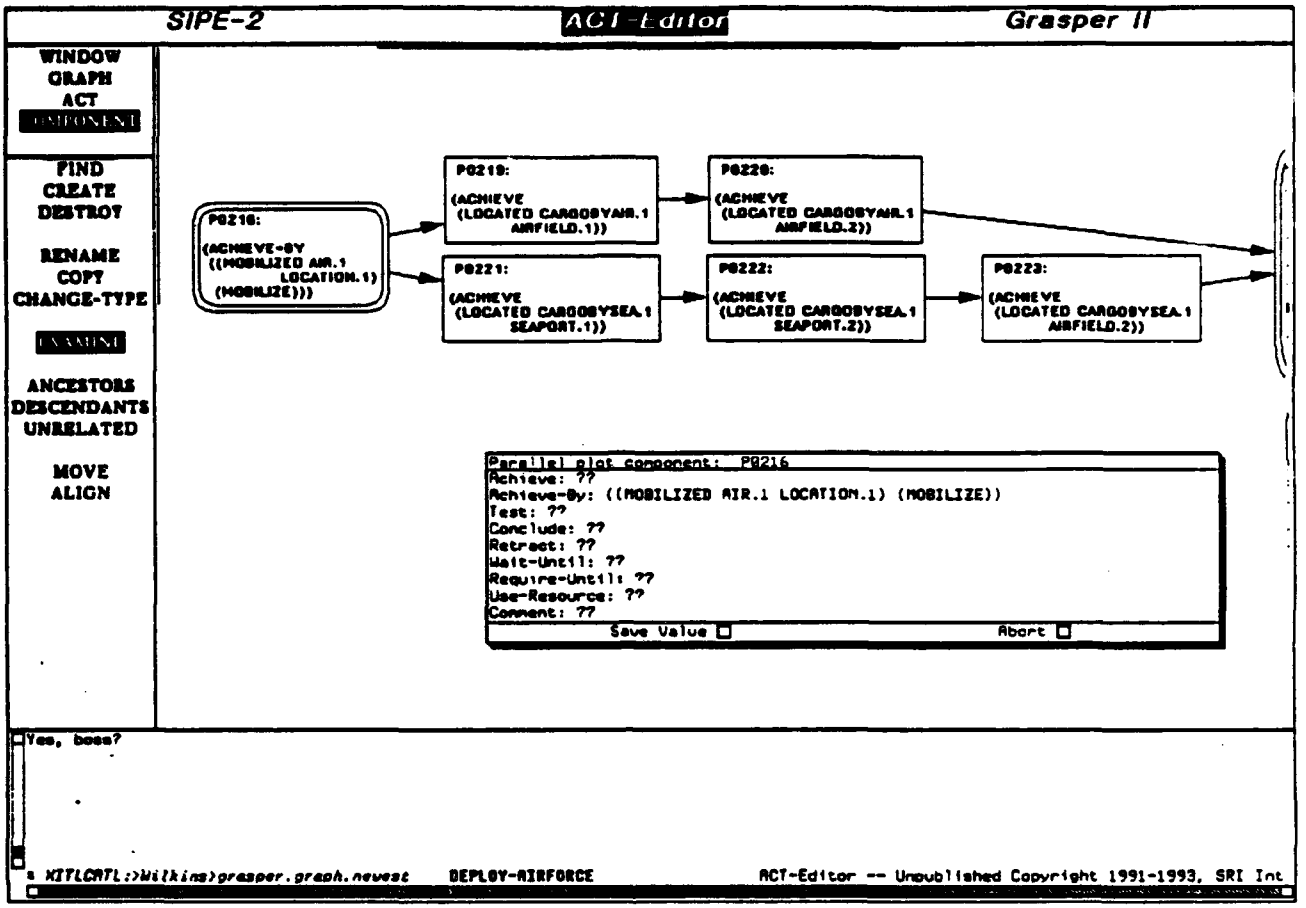

Figure 6.1: Plot of Deploy Airforce ACT

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

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

The NEW VIEW command does not modify the structure of an ACT but rather modifies

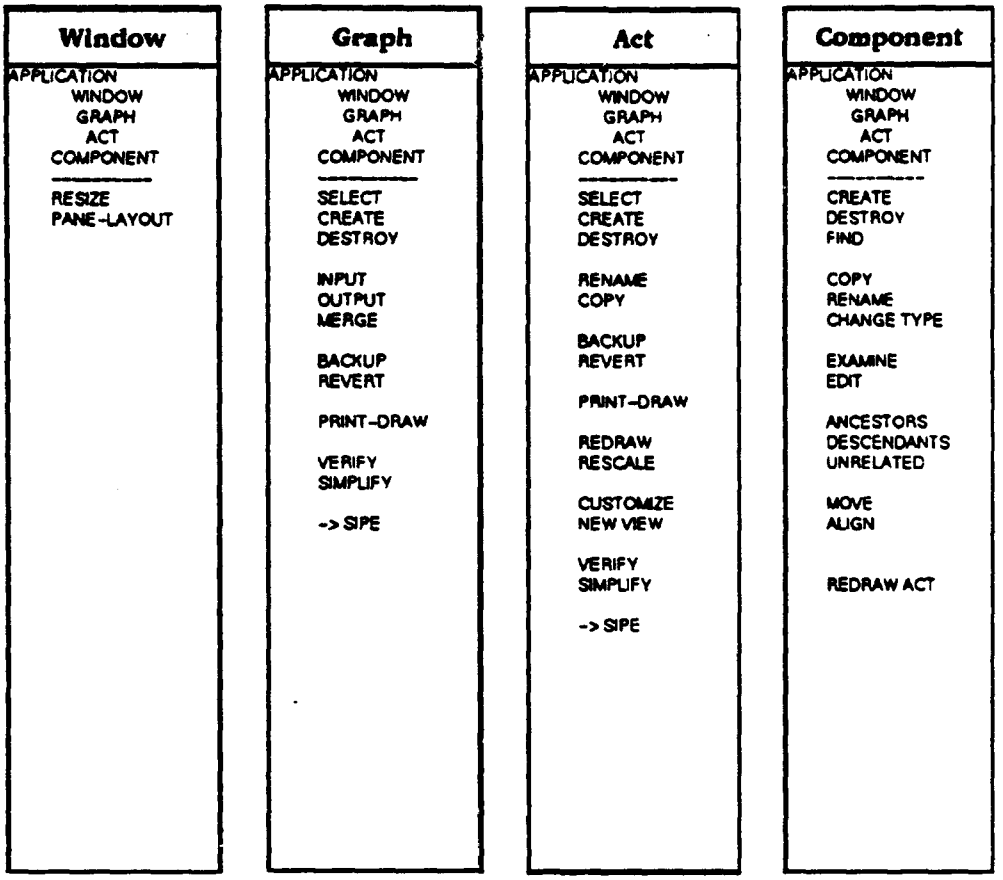

Figure 6.2: ACT Editor Command Menus

its presentation on the screen. While the underlying ACT remains unchanged, a user can vary the amount of detail to be presented on each plot node. Having the ability to modify the display characteristics in this manner is of value most notably when examining ACTs generated by SIPE-2 for complex problems, since their size can render them unwieldy.

Chapter 7

Generative Planning

This chapter describes extensions to generative planning technology made during the course of this project, concentrating on the advances made during the latter part of the project. (Earlier advances are described in our previous annual reports.) We developed a theory of reasoning about location that is adequate for solving many classes of problems of practical interest, and implemented it in SIPE-2 for the military operations planning problem as described in Section 7.1. This theory applies to technological frameworks ranging from generative planners to schedulers to reactive systems. The integration of SIPE-2 and PRS-CL during the final year of this project, which broke new ground in the cooperation of planning and execution systems, is described in Chapter 5.

SIPE-2 runs in the CPE and has been distributed to many ARPI sites (see Chapter 11 for a list). The SIPE-2 implementation of the IFD-2 domain resulted in several important extensions. These extensions, described in the remaining sections of this chapter, provide a significant increase in capability and ease of use and include a redesigned graphical interface that can graphically display plans and domain knowledge, a more powerful interactive planning mode, and several new features. We also describe how performance on the IFD-2 problem was significantly improved.

Several minor extensions are not described here; examples include extended numerical reasoning capabilities, the entering of new information after each planning level, the removal of redundant actions from a plan, and taking advantage of the capabilities of the Grasper-CL-based interface (for example, flashing the nodes as they are executed) in the execution monitor and replanner.

7.1 Reasoning about Locations

While much progress has been made in the development of knowledge-based problem-solving and planning systems in recent years, relatively little effort has been devoted to the task of formulating the knowledge required by such systems to solve practical problems. Correct and appropriate representations are critical to successful deployment of the systems for nontrivial applications; nevertheless, representational issues remain mostly unexplored.

While extending the IFD-2 domain knowledge to support more complex reasoning (such as that being done by PRS-CL), we developed an analysis of one particular ontological problem, namely *locations*, from both the theoretical and the practical perspectives. Our theory is both heuristically and epistemically adequate for solving many classes of problems of practical interest. Locational information is important for any system that must reason about objects whose positions change over time. A broad range of AI applications fits this description, including directed navigation for autonomous vehicles (such as mobile robots), transportation planning, military operations, process planning, production line scheduling, and real-time tracking. For this reason, our theory applies to technological frameworks ranging from generative planners to schedulers to reactive systems.

To implement our theory, two major changes were made in the domain knowledge. While the original version of this knowledge worked for the problems solved in IFD-2, there were several inaccuracies in the representation that needed to be corrected before the knowledge could support more extended reasoning, such as that being done by PRS-CL when it uses ACTs to control execution at a lower level of detail than that planned for by SIPE-2. First, we rewrote the subclass structure of the sort hierarchy to correctly reflect that fact that subclass links are interpreted as "IS-A" links. Second, we reformulated the ontology for predicates involved in reasoning about locations, and added about 25 deductive operators to correctly reason about objects changing location and to deduce all commutative predicates.

Our theory of reasoning about location will not be described in detail here, since it is described in a paper (included in Appendix G), by Dr. Myers and Dr. Wilkins, entitled "Reasoning about Locations and Movement." This paper presents our theory and describes the practical experience of implementing this theory in SIPE-2 and SOCAP. It was submitted to the Artificial Intelligence Journal special issue on planning in June 1993, and we have been notified that they want to publish it after revisions have been made and approved.

The journal paper begins with an analysis of the simplest characterization of location, progressively enriching it to include notions such as multiple abstraction levels and aggregation. For each case we present a formal theory and discuss the situations in which it applies. We define a belief revision framework along with a provably correct set of belief revision rules that maintains a STRIPS-style database in accordance with the constraints of each theory.

The final part of the paper describes how to operationalize the belief revision rules to support efficient reasoning about locations. While this analysis is applicable to many types of reasoning systems, we concentrate specifically on generative planners. In particular, we describe operationalization of the belief revision rules for SIPE-2. However, the same basic methods could be applied in many other planning frameworks. Experimental results show that a direct translation of the belief revision rules leads to computational inefficiency. To address this problem, several techniques are presented that provide greatly improved performance in locative reasoning.

7.2 Replanning for PRS

As described in Chapter 5, two major extensions to SIPE-2 were implemented to enable cooperation with PRS-CL. One involved enhancing the existing replanning algorithms of SIPE-2, and the other was the communication protocols used to receive requests from PRS-CL and respond with new plans and other information needed by PRS-CL to coordinate its continued execution with the new plan.

The changes to the replanning algorithms were numerous, but mostly too small and detailed to describe in this report. The most important change is that one mode of failure in PRS-CL is that some action that is primitive to the planner cannot be executed in the actual world for some (probably unknown) reason. However, all the preconditions of the action are still true, so that the planner would again find the same plan. SIPE-2 now provides the option of producing a plan that is different from the original plan. It does this by preventing the application of the operator that originally inserted the failed action into the plan.

Two of the smaller extensions are briefly mentioned. First, the replanning action of reinstantiating a planning variable was rewritten to analyze a proposed new instantiation in much more depth than was done previously. This should keep poor choices from being made. Second, the problem-recognizing algorithm was extended to appropriately handle problems with numerical information.

In our implemented scheme (described in Chapter 5), PRS-CL passes to SIPE-2 the name of the plan being executed, the action within the plan that failed, an execution front, a unique identifier for this request, and a description of the current state of the world. The latter can be either a file that completely describes the world or a list of predicates that are incrementally added to SIPE-2's model of the world. Processing this input to invoke the replanner required writing code to accept the world model and update the internal world model and to process the execution front correctly.

After finding a revised plan, SIPE-2 must communicate two things to PRS-CL: the plan, and a *node map*. The plan is translated to an ACT and sent to PRS-CL. The node map

includes a mapping for every changed or deleted action that PRS-CL might possibly have executed during the replanning process. Each such action is mapped to the appropriate node in the revised plan where execution should start. The node map required additional bookkeeping facilities in the replanning algorithyms.

This two-way communication is supported by SIPE-2 in any of three protocols: a send-message function in PRS-CL, writing to files, or using the Knet and Cronus systems of the CPE. The protocol is selected by binding the variable `sipe::*AGENT-PROTOCOL*` to either `:file`, `:prs-send`, or `:knet`. The file option works by writing files to disk that are read by the other system. Files work in any configuration, but is slower than the other options. The prs-send option will work only when both systems are running in the same Lisp image, and is very fast since the information already exists in the Lisp image and does not need to be transferred. The Knet option will work when the systems are running on different machines. The internal representation of an ACT is sent; through servers we wrote in Knet, directly from the planner to the execution system. This option is much quicker than writing files and then later reading them. Large plans with hundreds of nodes seem to transmit in only a few seconds (perhaps even less than a second – it is not possible to get exact timings).

7.3 New Features

The new features added to SIPE-2 during this project are too numerous to mention. Our previous annual reports describe many of them. This section briefly describes the most important new features implemented during the last year of the project.

7.3.1 Common Knowledge Bases

Another ARPA project at the AIC has been making progress on accessing multiple and common knowledge bases across all systems. This effort has resulted in a uniform frame-access language so that our systems can easily interface with any of several (non-SRI) database systems that may be used to store domain knowledge. SIPE-2 was extended to make it easy to use this frame-access language by simply redefining some simple access functions. Using this language, the IFD-2 plan has been generated with all knowledge base references going to a LOOM knowledge base [23]. However, using LOOM slows the planning process by one to two orders of magnitude.

7.3.2 CPE Interactions

With the help of Bolt, Berenek, and Newman (BBN), we installed the Cronus and Knet systems on our machines. We converted them to use our software specification conventions

which provide version control, a patch facility, and the ability to run these systems without each user having a considerable amount of code in his personal initialization files (as BBN requires). We use these systems in our final demonstration for communication between SIPE-2 and PRS-CL, both of which were extended to interact by using the Cronus system. SIPE-2 sends plans to PRS-CL and gets back information about the world and replanning requests.

We extended SIPE-2 to interact with GE's Tachyon system in a loosely coupled manner. Tachyon is able to process extended temporal constraints for SIPE-2 during planning. They communicate by using the Cronus system in the CPE.

7.3.3 Additive Numerical Goals

One important new feature in SIPE-2 (partially supported by this project) is the ability to solve a numerical goal by adding links to several parallel actions that together achieve it. This capability will prove to be important in military planning since with this facility a given unit can respond to several threats, without taking on too many responses.

7.4 Graphical User Interface

We completely redesigned the SIPE-2 GUI as part of this project, taking into account its shortcomings during development of SOCAP. The new design is both more powerful and easier to understand. The interface is built on Grasper-CL, as are the interfaces of all subsystems in CYPRESS, where selecting a noun in the command menu brings up a menu of commands appropriate to that noun.

Figure 7.1 shows the command menus in the new GUI. Important additions are the commands Print, List, and Draw, each of which can now be applied to each of the following structures: operators, problems, plans, worlds, and objects. Several new capabilities were implemented to support these commands and better serve the user, including the ability to draw graphically the sort hierarchy, the world model, deductive operators, and normal planning operators. Improved layout algorithms for these drawings were developed.

Our previous annual reports describe many of the capabilities in the GUI. It 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 parallel links, such as military operations plans. For example, highlighting all the nodes that mention Fennario Port shows the schedule for that port as in Figure 7.2 (the user can pan over the entire plan), and highlighting all the nodes unordered

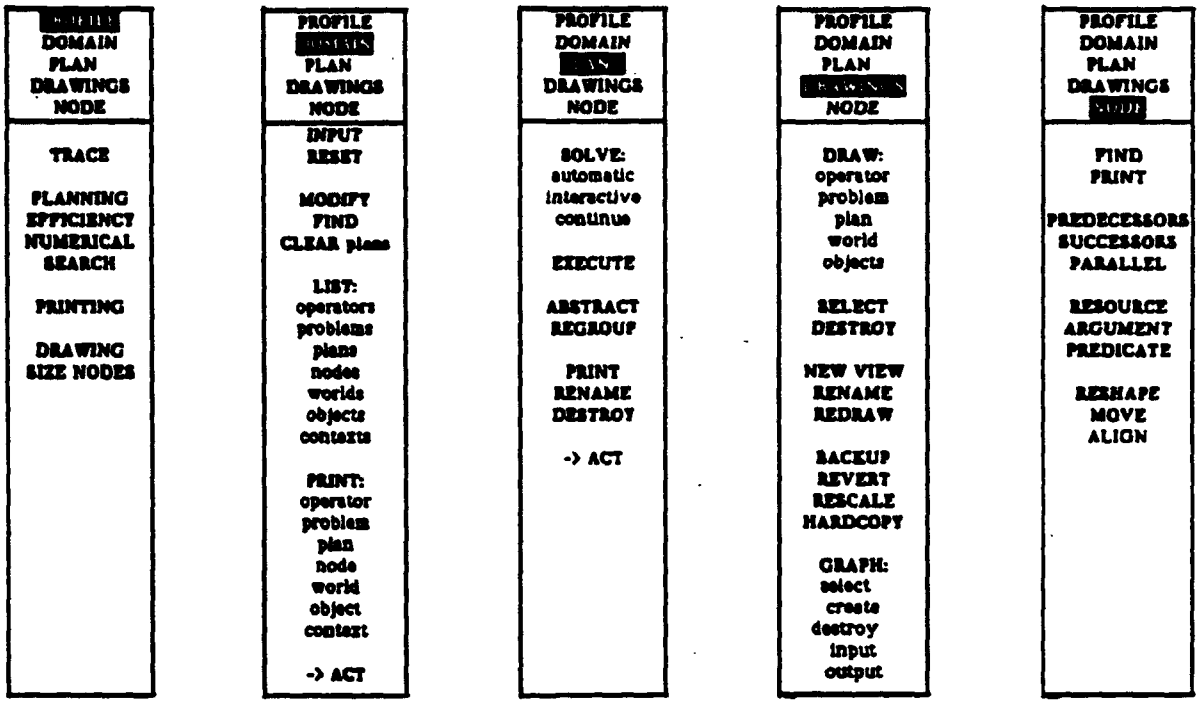

Figure 7.1: SIPE-2 GUI command menus

with respect to a certain node could show all actions in the current phase of the operation. One can also highlight all the actions that accomplish a certain condition. For example, to highlight all the actions that move troops in a plan, one can click the Predicate verb and specify the *moved* predicate. The user can move nodes around with the mouse to make the graph look exactly as desired. Other capabilities include the ability to hardcopy graphs, save them, find a node by moving it onto the screen and highlighting it, and execute plans while highlighting the nodes being executed. Similarly, the new interface supports the interactive planner, including highlighting nodes for which choices are being made.

Several capabilities were added to the GUI during the last year of the project. One of the most important was a new capability for viewing the planning process on the screen. SIPE-2 can now incrementally highlight and draw ordering links between actions as they are added during planning. In particular, it flashes the first node, then draws the ordering link, then flashes the second node. This gives an excellent visual depiction of how the plan is flowing. Other recent additions to the GUI are briefly mentioned below.

- a number of new window-layout options are available in the CLIM interface with a new LAYOUT command in the PROFILE menu to choose among them
- pull-down menus can be chosen instead of command menus in some layouts
- a permanent CHOICE window can be used for selecting operators, nodes, and units

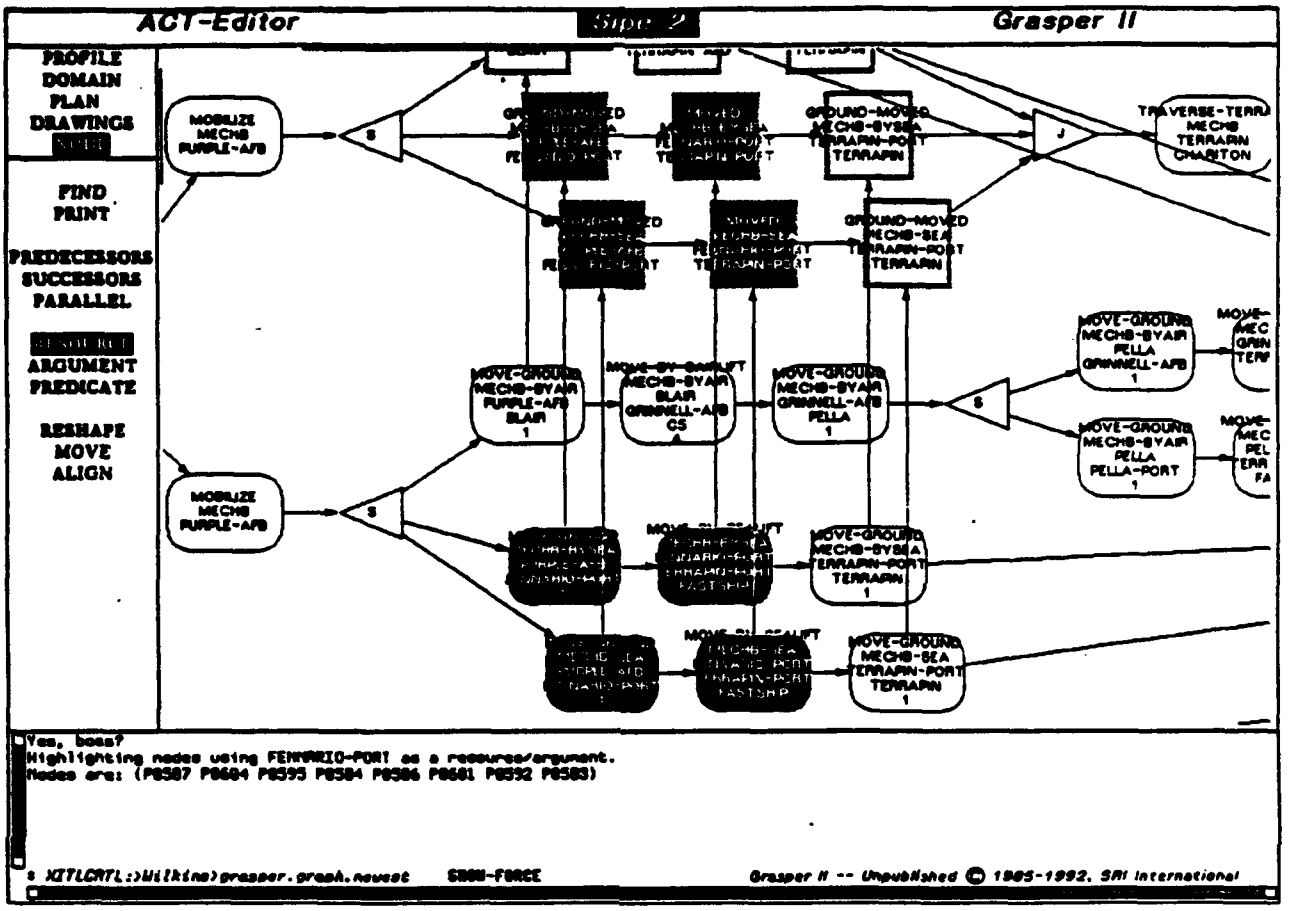

Figure 7.2: Plan highlighting resource: Fennario Port

- backtracking in the interactive search is now traced in the GUI by highlighting all node choices for backtracking
- four new commands for modifying plans were added: ADD GOAL, DELETE GOAL, ADD FACTS, and ADD+ANALYZE FACTS
- all node highlighting is now done in both the birds-eye and main windows
- printing can now be (optionally) done in pop-up windows which have their own process so they stay around while the user does other things (such as continue planning)
- meaningful names for plans and drawings are automatically generated.

7.5 Improved Interactive Planning

Several new capabilities were added to SIPE-2's Interactive Search. The most important of these are described below.

- The application of plan critics can be controlled with greater flexibility
- The user can choose to finish planning automatically at any point
- All possible replanning points are kept, even when another problem has been started or when the user aborts while generating an unfinished plan
- The user may now suspend the planning when desired and use the full power of the GUI and all its command menus to display data structures. The user can then exit and continue planning.
- The menus used by interactive planning were redesigned
- The interactive search now highlights each node on the screen whenever it is making a decision about that node, e.g., planning for the node, choosing an operator for the node, choosing instantiations for the variables at that node, or choosing a node for backtracking to make another choice.

7.6 Efficiency Gains

An analysis of the planning in IFD-2 was performed after IFD-2. This resulted in significant efficiency gains. The IFD-2 plan originally took 12.7 minutes to produce on our SYMBOLICS and 22.0 minutes to produce on a SUN Sparc 1+. Because much of the SUN time was garbage collecting, our SUN timings would vary significantly on a machine with a different memory configuration. These times have been reduced to 4.2 minutes on the SYMBOLICS and 1.0 minute on the same SUN by making the following changes.

- The values of four documented global variables were set differently than their defaults. This customized SIPE-2 to the domain and avoided unnecessary computation. This change cut the SYMBOLICS time in half, and should have a similar effect on SUN computation.
- To further improve the SUN time, it was necessary to compile every single scrap of code. The problem-specific functions for computing durations that are called during planning were not compiled. They were moved from the SIPE-2 input file to a Lisp file and compiled. This procedure did not affect time on the SYMBOLICS, but cut the SUN time in half.
- SIPE-2 was changed to not compute numerical bounds for a variable whose value is computed by a function when it is appropriate to wait for the arguments to the function to be instantiated. In particular, SOCAP was computing the duration of a move from

one place to another when the destination and embarkation were not yet instantiated. Each time, there are about 900 possible routes and a duration was calculated for each to compute the bounds. Without these calls, the speed increased by a factor of 3 on the SYMBOLICS, and by a factor of nearly 10 on the SUN (mostly because of less garbage collecting).

- SIPE-2 was compiled with the Lucid production mode compiler. This procedure had been tried earlier but compiler bugs prevented proper execution. These bugs appear to be fixed now. This reduced the Sparc 1+ time from 1.7 minutes to 1.0 minute.

Chapter 8

Reactive Control

This chapter describes extensions to reactive control technology made during the course of this project, concentrating on the advances made during the latter part of the project. (Earlier advances are described in our previous annual reports.) The integration of SIPE-2 and PRS-CL during the final year of this project, which broke new ground in the cooperation of planning and execution systems, is described in Chapter 5.

PRS-CL runs in the CPE and has been distributed to many non-SRI sites. PRS-CL was extended in numerous ways to provide the increased functionality required for the CYPRESS system. The most significant extensions, described below, were made to support the ACT formalism, to support data-sharing with SIPE-2, to support run-time replanning after execution failures, and to provide an improved user interface. We have prepared a new user's guide that documents the extended system in detail.

8.1 ACT Execution

Many of the modifications to PRS-CL were made to support the use of the ACT formalism for representing actions. Originally, actions were represented in PRS-CL using structures called *knowledge areas* (*KAs*). ACTs permit the representation of a number of important constructs not supported by KAs. For instance, the ACT formalism includes the metapredicates `Use-Resource`, `Require-Until`, and `Achieve-By`, which have no counterparts in KAs. Furthermore, ACT plots provide a more general graphical representation of actions than is allowed by KAs.

Extending PRS-CL to support the full generality of ACT plots involved accommodating parallel execution of actions. PRS-CL has always supported some parallelism, but only through the use of multiple agents or execution of multiple KAs simultaneously in response

to parallel goals. Parallelism was not supported *within* an individual KA, as it is within ACTs. This type of parallelism is valuable for representing many types of inherently parallel activities, as evidenced in the SOCAP domain. PRS-CL has been extended to execute directed, possibly cyclic, graphs of actions that support parallel topologies, including all SIPE-2 plans and operators. This was a complex task, as it involved a major overhaul of the run-time representation structures and the introduction of control mechanisms for activating/deactivating parallel execution threads.

A limited resource-handling capability was added to PRS-CL to support the ACT predicate **Use-Resource**. Resources can be allocated and released, but only for the entire scope of an ACT. This corresponds to the use of **Use-Resource** in the Resource environment node of an ACT; the use of this metapredicate in plot nodes is not supported. Resources are assigned on an *all-or-nothing* basis. All-or-nothing treatment is appropriate for reusable resources that can be used for only one purpose at a time. The current implementation must be extended to handle consumable resources, such as fuel, that can be allocated to multiple users simultaneously, provided that sufficient quantities are available.

Another ACT addition to PRS-CL was the **(REQUIRE-UNTIL (req-wff term-wff))** metapredicate, which can be used to specify that a required condition remain true until some termination condition is satisfied. (PRS-CL already included the operator # which could be used to specify conditions that needed to be preserved, but its semantics were quite different.) We have provided the **Require-Until** metapredicate with an *active maintenance* semantics, which allows the required condition to be temporarily violated without leading to a complete failure of the goal. Doing so enables the use of *repair procedures* that can applied in order to actively re-establish the required condition. Repair procedures are generally domain-specific ACTs designed for fixing individual conditions. The following describes the failure/success behavior for **Require-Until**:

- a **Require-Until** goal *fails* when either:
  - **req-wff** is unsatisfied at the point when **term-wff** becomes satisfied, OR
  - **req-wff** is unsatisfied and there is no means to repair it.
- a **Require-Until** goal *succeeds* when either:
  - **req-wff** is satisfied when **term-wff** becomes satisfied, OR
  - the ACT containing the **Require-Until** goal terminates without the goal having failed.

Note that the **req-wff** need not be satisfied when a **Require-Until** goal is posted initially. When the **req-wff** of a **Require-Until** goal is initially detected as unsatisfied, PRS-CL

will post a repair goal of the form (ACHIEVE (REPAIR req-wff)). If no repair ACTs have been specified or none succeed in re-establishing req-wff, the Require-Until goal is then considered to have failed. An example of the use of the repair facility is provided in our demonstration (see Section 4.2).

PRS-CL was also extended to support the Achieve-By metapredicate. Originally, PRS-CL allowed the specification of goals but not any information indicating which procedures should be used to accomplish those goals. In order to support the Achieve-By metapredicate, the PRS-CL interpreter loop was modified to filter candidate ACTs to be applied in order to select only from among those specified by the Achieve-By metapredicate.

Finally, the unification and database retrieval mechanisms were modified to use the typed variable syntax common to both the ACT formalism and SIPE-2.

8.2 Shared Knowledge

The ACT formalism serves as a language for communicating plans between the SIPE-2 and PRS systems. In addition to sharing plans, the two systems must also share knowledge about the domain of operation. SIPE-2 plans are based on *a priori* knowledge about the domain, much of which must be made accessible to PRS-CL in order for the plan to be executed. Similarly, when PRS-CL reaches an execution impasse, SIPE-2 will be invoked for replanning purposes and will require knowledge about what changes in the world have occurred during plan execution. These changes could be either direct consequences of plan execution or unexpected developments not predicted by the original model of the world.

Inputing domain knowledge independently for both systems both duplicates work unnecessarily and introduces the possibility of consistency problems. Instead, we have implemented a translation facility for exchanging domain knowledge between the two systems.

One part of the facility enables PRS-CL to access SIPE-2 domain information. The domain information includes both the SIPE-2 database and the SIPE-2 sort hierarchy. Initially, the SIPE-2 database is translated directly into the PRS-CL database. It is necessary to duplicate the SIPE-2 database in this manner since much of the database knowledge is dynamic in nature: execution of plans by PRS-CL will lead to changes in both the world and what is known about the world. The PRS-CL database is updated during plan execution to keep track of the system's overall knowledge about the world. In contrast, the information in the sort hierarchy describes static properties that are fixed for the scope of the application. Given the permanence of this information and the potential for hierarchies of substantial size, the hierarchy is not translated into the PRS-CL database. Instead, PRS-CL has been modified to consult not only its internal database but also an arbitrary number of external databases

(such as the SIPE-2 sort hierarchy) when evaluating the truth-value of a predicate or seeking bindings for logical variables. This generalized consultation mechanism could be used in the future to access additional sources of domain information, such as external military databases.

The second part of the translation facility enables PRS-CL to send SIPE-2 updates about the current state of the world. Since SIPE-2 is a planner rather than an execution system, it does not keep track of changes in the world that occur during execution. Thus, it requires updates from PRS-CL in order to keep its world model accurate. Although one could imagine having PRS-CL issue such updates to SIPE-2 on a routine basis, currently they are used only to update SIPE-2's world model when replanning is invoked.

8.3 User Interface

A Grasper-based graphical user interface was constructed for PRS-CL that employs the same basic 'look and feel' as the other component systems of CYPRESS. This interface extends and improves upon the previous PRS-CL interface in numerous ways. The main emphases of these modifications have been to simplify interactions with the system and to provide better run-time tracing facilities. The new user's manual for PRS-CL documents the interface.

8.4 Run-time Replanning

One of the most interesting aspects of the CYPRESS system is its ability to provide run-time replanning for goals when failures occur during execution. For replanning, PRS-CL is responsible for detecting replannable failure situations, making requests to SIPE-2 to produce an alternative plan, and then integrating the new plan into the current execution activities. The extensions to PRS-CL that were necessary to support replanning are described in detail in Chapter 5.

Chapter 9

Handling Uncertainty

Uncertain information is ubiquitous in military operations, as in most real-world problems. Most planning technologies are not capable of handling uncertain information because of the combinatorics involved. We have addressed this shortcoming on two fronts. First, we have developed a theory and implemented it (using SIPE-2 and Gister-CL) for evaluating the likelihood that plans will accomplish their intended goals given both an uncertain description of the initial state of the world and the use of probabilistically reliable operators. This method is described in Sections 9.1 through 9.3, and summarized in Section 9.4. Second, we have identified several opportunities for using uncertain information in CYPRESS.

One such opportunity is using Gister-CL for situation assessment to determine the facts to be inserted in the initial world model before planning. Another opportunity is choosing among competing ACTs in the presence of uncertainty about the situation and uncertainty about the effects of an ACT. This is a problem faced by both SIPE-2 and PRS-CL, and a paper published this year by Dr. Wesley describes preliminary investigations of evidential measures for choosing among alternative actions [31]. A third opportunity involves choosing of particular objects for use during planning: for example, choosing which unit to use in a particular operation, or which airport to use for landing them. Since each of these opportunities involves a time-consuming construction of analyses based on the domain-specific knowledge about the information sources, we have implemented analyses for one such opportunity as a proof-of-concept demonstration. In particular we have developed Gister-CL analyses for the case of selecting an appropriate military unit during planning, as described in Section 9.5.

In addition, we implemented techniques using the CPE for calling and receiving the results of Gister-CL analyses with both SIPE-2 and PRS-CL. These are described in Section 9.6. The evidential reasoning concepts used in this chapter are defined in Appendix B.

9.1 Evaluating Plans in Uncertain Worlds

The approach we have taken in our work is to combine our existing systems into a test bed suitable for exploring different technological solutions. By combining our existing systems, we are attempting to accumulate the benefits of each. Each system can support a common representation, namely first-order logic. In this section we describe our work on the problem of evaluation of a given plan in an uncertain environment. By *uncertain environment*, we mean a world where the initial state is not known with certainty and where the effects of actions are not known with certainty.

Since Gister-CL supports reasoning about uncertainty, but SIPE-2 and PRS-CL do not, our approach is to have Gister-CL evaluate a given plan in an uncertain environment. The plan will not incorporate uncertain information, rather it will be a plan produced by SIPE-2, or a standard operating procedure that has been selected by PRS-CL, or a plan created by the user. By *evaluate a plan*, we mean that Gister-CL will be able 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. Such evaluations could be utilized by PRS-CL's meta-level procedures to select those procedures for execution that are most likely to achieve their intended goals or by SIPE-2 to compare alternative plans.

9.1.1 Frame Logic

The first step in applying Gister-CL to a selected domain of application is to define the *frame logic*. Suppose that the answer to some question $\mathcal{A}$ is contained in a finite set Θ_A. That is, each element a_i of Θ_A corresponds to a distinct possible answer to the question $\mathcal{A}$, no two of which can be simultaneously true. For example, $\mathcal{A}$ might be a question concerning the configuration of a set of blocks. In this case, Θ_A would consist of all the possible configurations under consideration. Θ_A is called a *frame of discernment*. If there are exactly three blocks, labeled "A", "B", and "C", and each can rest on top of one other block or on a table, then Θ_A might be defined as follows:

$$\begin{aligned}\Theta_A \;=\; \{&\mathsf{ABC}, \mathsf{ACB}, \mathsf{AB\text{-}C}, \mathsf{AC\text{-}B}, \mathsf{BAC},\\ &\mathsf{BCA}, \mathsf{BC\text{-}A}, \mathsf{BA\text{-}C}, \mathsf{CAB},\\ &\mathsf{CBA}, \mathsf{CA\text{-}B}, \mathsf{CB\text{-}A}, \mathsf{A\text{-}B\text{-}C}\}\ ,\end{aligned}$$

where AB-C corresponds to block A resting on top of block B, and blocks B and C resting on the table.

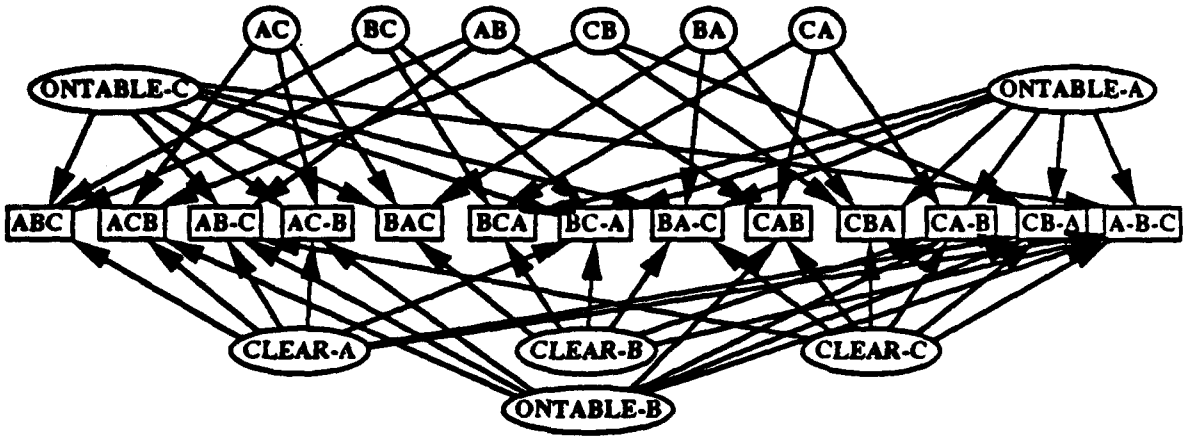

Figure 9.1: BLOCKS-WORLD frame.

In implementing this formal approach, we have found that frames, like the other formal elements in this theory, can be straightforwardly represented as graphs consisting of nodes connected by directed edges (i.e., arcs). Because they are graphs, these formal elements are easily understood, and they provide an intuitive basis for human-computer interaction. A frame is represented by a named graph that includes a node for each element of the frame and may include additional nodes representing aliases, i.e., named disjunctions of elements. Each of these additional nodes has edges pointing to elements of the frame (or other aliases) that make up the disjunction. Here, the possible configurations for three blocks are represented by a graph named BLOCKS-WORLD (Figure 9.1) that includes the thirteen different configurations as elements (e.g., ABC, ACB, AB-C) and aliases for each block being clear (i.e., CLEAR-A, CLEAR-B, CLEAR-C), for each block being on the table (i.e., ONTABLE-A, ONTABLE-B, ONTABLE-C), and for each block being immediately on top of each of the other two (i.e., AB, AC, BA, BC, CA, CB).

Propositions

Once a frame of discernment has been established for a given question, it formalizes a *variable* where each possible value for the variable is an element of the frame. A statement pertaining to the value of this variable is discerned by the frame, just in case the impact of the statement is to focus on some subset of the possible values in the frame as containing the true value. In other words, a propositional statement A_i about the answer to question $\mathcal{A}$ corresponds to a subset of Θ_A. For example, if the statement is "block A is on block B," then it corresponds to the set of block configuration in Θ_A where block A rests on block B.

$$AB = \{ABC, AB\text{-}C, CAB\} \subseteq \Theta_A \quad .$$

Other propositions related to this question can be similarly represented as subsets of Θ_A (i.e., as elements of the power set of Θ_A, denoted 2^{Θ_A}); the subset A_j might correspond to all those configurations in Θ_A that have no block on top of block C. Once this has been accomplished, logical questions involving multiple statements can be posed and resolved in terms of the frame. Given two propositions, A_i and A_j, and their corresponding sets, A_i and A_j, the following logical operations and relation can be resolved through the associated set operations and relation:

$$\begin{aligned} \neg A_i &\iff \Theta_A - A_i \\ A_i \wedge A_j &\iff A_i \cap A_j \\ A_i \vee A_j &\iff A_i \cup A_j \\ A_i \Rightarrow A_j &\iff A_i \subseteq A_j \quad . \end{aligned}$$

Thus, when two statements pertaining to the same question are available, and they are each represented as subsets of the same frame, their joint impact is calculated by intersecting those two subsets. Given "A is on B" (AB) and "the top of C is clear" (CLEAR-C), their joint impact is

$$\begin{aligned} \text{AB} &= \{\text{ABC}, \text{AB-C}, \text{CAB}\} \\ \text{CLEAR-C} &= \{\text{AB-C}, \text{BA-C}, \text{CAB}, \text{CBA}, \\ &\quad \text{CA-B}, \text{CB-A}, \text{A-B-C}\} \\ \text{AB} \cap \text{CLEAR-C} &= \{\text{AB-C}, \text{CAB}\} \quad . \end{aligned}$$

All other statements that correspond to supersets of this result in Θ_A, are implicitly true (e.g., "a block is on B"); all of those statements whose corresponding sets are disjoint from this result are implicitly false (e.g., "A is on C"); and all others statements' truthfulness are undetermined (e.g., "B is on the table"). As additional information becomes available, it can be combined with the current result in the same way. Since intersection is commutative and associative, the order that information enters is of no consequence.

Translating Propositions

Suppose that another question of interest B has been separately framed. For example, if A corresponds to the state of the blocks at time 1, then B might correspond to the state of the blocks at time 2. Its frame of discernment, Θ_B, is defined as the set of possible block configurations at time 2 (for this example, Θ_A and Θ_B are equivalent).

$$\Theta_B = \{b_1, b_2, \ldots, b_m\}$$
$$B_j \subseteq \Theta_B \ .$$

If something is known about the state of the blocks at time 1, we would like to take advantage of this information to narrow the possibilities at time 2. To do this, one must first define a *compatibility relation* between the two frames. A compatibility relation simply describes which elements from the two frames can be true simultaneously i.e., which elements are compatible. For this example, if at most one block can be moved in a single unit of time, then state AB-C from Θ_A is compatible with AB-C, CAB, and A-B-C from Θ_B, since these are the only states that could immediately follow AB-C. Thus, a compatibility relation between frames Θ_A and Θ_B is a subset of the cross product of the two frames. A pair (a_i, b_j) is included if and only if they are compatible. Typically, there is at least one pair (a_i, b_j) included for each a_i in Θ_A (the analogue is true for each b_j):

$$\Pi_{(A,B)} \subseteq \Theta_A \times \Theta_B \ .$$

Using the compatibility relation $\Pi_{(A,B)}$ we can define a *compatibility mapping* $\Gamma_{A \rightarrowtail B}$ for translating propositional statements expressed relative to Θ_A to statements relative to Θ_B. If a statement A_k is true, then the statement $\Gamma_{A \rightarrowtail B}(A_k)$ is also true:

$$\Gamma_{A \rightarrowtail B} : 2^{\Theta_A} \mapsto 2^{\Theta_B}$$
$$\Gamma_{A \rightarrowtail B}(A_k) = \{ b_j \mid (a_i, b_j) \in \Pi_{(A,B)}, a_i \in A_k \} \ .$$

In our example, the compatibility relation $\Pi_{(A,B)}$ delimits all possible state changes between time 1 and 2. However, when evaluating a plan, additional information is available, namely, the specific action or operation that is to be performed. One means of incorporating this information is to define a distinct compatibility relation corresponding to each operation. For example, the compatibility relation $\Pi_{\text{PUT-C-ON-A}}$ would have CAB as the only state in Θ_B compatible with AB-C in Θ_A; those states in Θ_A that already have block C on block A (e.g., CAB) or that have some block on C, thus preventing it from being moved, are compatible with the same state in Θ_B.

Within Gister-CL, each compatibility relation is represented by a graph that includes a node for each of the elements in its corresponding frame and edges connecting compatible elements. An edge connects two elements just in case the element at the arrow head can be reached in one temporal step from the other connected element (Figure 9.2).

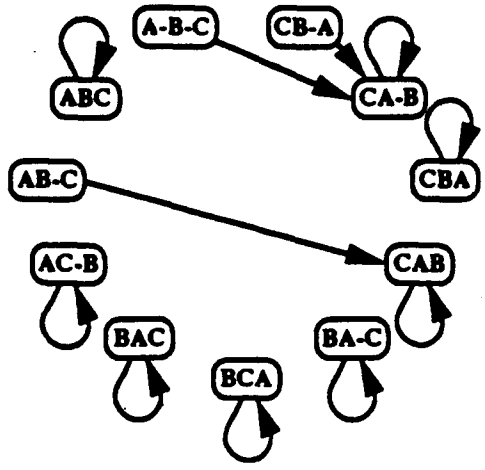

Figure 9.2: PUT-C-ON-A compatibility relation.

Given the propositions AB and CLEAR-C at time 1, we conclude that the initial state is either AB-C or CAB; if we know that *PUT-C-ON-A* is applied, then we conclude that the state at time 2 must be CAB. Presuming that the possible states for any time i are the same as those for times 1 and 2, and that the possible operations and their effects are the same in moving from any time i to $i+1$, these frames and compatibility relations can be used to calculate the effects of any planned sequence of actions.

The overall topology, formed by frames and compatibility relations, is represented as a graph, called a *gallery*. In a gallery the frames are represented as nodes, and the compatibility relations are represented as edges connecting the frames they relate. In Figure 9.3, the BLOCKS-WORLD frame is represented as a node, and the compatibility relations corresponding to each operator are represented by edges connecting the BLOCKS-WORLD frame to itself. Two other compatibility relations capture the null operation (i.e., the NO-OP compatibility relation has each element compatible with itself) and the random operation (i.e., the UNKNOWN compatibility relation that has every element connected to every element).

9.1.2 Framing Evidence

When information is inconclusive, partial beliefs replace certainty; probabilistic distributions over statements discerned by a frame replace Boolean valued propositions. These distributions are called *mass distributions*. Each body of evidence is represented as a mass distribution (e.g., m_A) that distributes a unit of belief over propositional statements discerned by a frame (e.g., Θ_A):

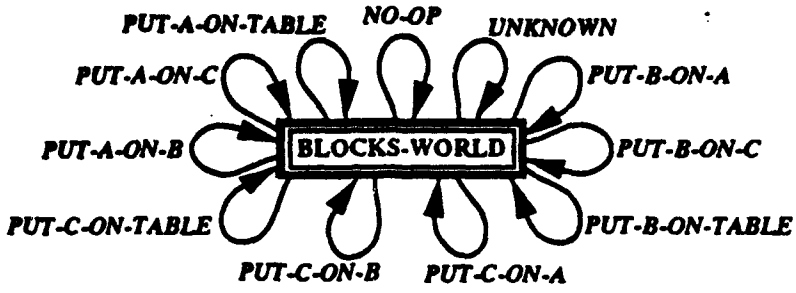

Figure 9.3: BLOCKS-WORLD gallery.

$$\begin{aligned} m_A : 2^{\Theta_A} &\mapsto [0,1] \\ \sum_{A_i \subseteq \Theta_A} m_A(A_i) &= 1 \\ m_A(\emptyset) &= 0 \ . \end{aligned}$$

For example, if we are told that there is an 80% chance that block A is on B and a 20% chance that block A is not on B, then this is represented by a mass distribution m_{AB} that attributes 0.8 to the set corresponding to AB, 0.2 to the complement of AB with respect to Θ_A, and 0.0 to all other subsets of Θ_A.

$$m_{\mathrm{AB}}(A_i) = \begin{cases} 0.8, & \{\mathrm{ABC, AB\text{-}C, CAB}\} \\ 0.2, & \{\mathrm{ACB\ AC\text{-}B, BAC, BCA, BC\text{-}A,} \\ & \ \mathrm{BA\text{-}C, CBA, CA\text{-}B, CB\text{-}A, A\text{-}B\text{-}C}\} \\ 0.0, & \text{otherwise} \ . \end{cases}$$

Interpreting Evidence

To *interpret* a body of evidence relative to the sta: A_j, we calculate its *support* and *plausibility* to derive its *evidential interval* as follows

$$\begin{aligned} Spt(A_j) &= \sum_{A_i \subseteq A_j} m_A(A_i) \\ Pls(A_j) &= 1 - Spt(\Theta_A - A_j) \\ [Spt(A_j), Pls(A_j)] &\subseteq [0,1] \ . \end{aligned}$$

Given the body of evidence represented by m_{AB}, the evidential interval for AB is $[0.8, 0.8]$, for CLEAR-C is $[0.0, 1.0]$, for {ABC} is $[0.0, 0.8]$, and for CLEAR-B is $[0.0, 0.2]$.

Propositional statements that are attributed nonzero mass are called the *focal elements* of the distribution. When a mass distribution's focal elements are all single element sets, the distribution corresponds to a classical *additive* probability distribution, and the evidential interval, for any proposition discerned by the frame, collapses to a point, i.e., support is equivalent to plausibility. For any other choice of focal elements, some propositional statement discerned by the frame will have an evidential interval with support strictly less than plausibility. This reflects the fact that mass attributed to a set consisting of more than one element represents an incomplete assessment; if additional information were available, the mass attributed to this set of elements would be distributed over its single element subsets. Thus, an evidential interval with support strictly less than plausibility is indicative of incomplete information relative to the frame.

For example, consider another point of evidence. A computer vision system reports that block C is clear. Based upon our previous experience with this system, we know that it always correctly determines if a block is clear or not, but 10% of the time it misidentifies the block. In other words, although we do not doubt that some block is clear, we are uncertain whether the block observed was C. Assuming that there is a 90% chance that the observed block was C and a 10% chance that it was not, then this evidence is represented by a mass distribution $m_{\text{CLEAR-C}}$ that attributes 0.9 to CLEAR-C and 0.1 to the set of all possible configurations, since every configuration has at least one clear block.

$$m_{\text{CLEAR-C}}(A_i) = \begin{cases} 0.9, & \{\text{AB-C, BA-C, CAB, CBA,} \\ & \text{CA-B, CB-A, A-B-C}\} \\ 0.1, & \{\text{ABC, ACB, AB-C, AC-B,} \\ & \text{BAC, BCA, BC-A, BA-C, CAB,} \\ & \text{CBA, CA-B, CB-A, A-B-C}\} \\ 0.0, & \text{otherwise} \quad . \end{cases}$$

Based upon this distribution, the evidential interval corresponding to the proposition that block C is clear is $[0.9, 1.0]$, that it is not clear $[0.0, 0.1]$, and that it is any particular configuration where C is clear is $[0.0, 1.0]$.

Fusing Evidence

When two mass distributions m_A^1 and m_A^2 representing independent opinions are expressed relative to the same frame of discernment, they can be *fused* (i.e., combined) using *Dempster's Rule of Combination* [26]. Dempster's rule pools mass distributions to produce a new mass distribution m_A^3 that represents the consensus of the original disparate opinions. That is, Dempster's rule produces a new mass distribution that leans towards points of agreement

between the original opinions and away from points of disagreement. Dempster's rule is defined as follows:

$$\begin{aligned} m_A^3(A_k) &= m_A^1 \oplus m_A^2(A_k) \\ &= \frac{1}{1-\kappa} \sum_{A_i \cap A_j = A_k} m_A^1(A_i) m_A^2(A_j) \\ \kappa &= \sum_{A_i \cap A_j = \emptyset} m_A^1(A_i) m_A^2(A_j) \\ &< 1 \ . \end{aligned}$$

Combining the two bodies of evidence m_{AB} and $m_{\text{CLEAR-C}}$ by Dempster's rule results in a mass distribution that attributes 0.72 to C being clear and A being on B, 0.18 to C being clear and A not on B, 0.08 to A on B, and 0.02 to A not on B.

$$m_{\text{AB}} \oplus m_{\text{CLEAR-C}}(A_i) = \begin{cases} 0.72, & \{\text{CAB, AB-C}\} \\ 0.18, & \{\text{A-B-C, CB-A, CA-B, CBA, BA-C}\} \\ 0.08, & \{\text{ABC, AB-C, CAB}\} \\ 0.02, & \{\text{ACB AC-B, BAC, BCA, BC-A,} \\ & \quad \text{BA-C, CBA, CA-B, CB-A, A-B-C}\} \\ 0.0, & \text{otherwise} \ . \end{cases}$$

This induces the following evidential intervals: [0.9, 1.0] for CLEAR-C, [0.72, 1.0] for CA, [0.8, 0.8] for AB, [0.0, 0.8] for {CAB} and {AB-C}, [0.0, 0.2] for {CBA}, and [0.0, 1.0] for CLEAR-A.

Since Dempster's rule is both commutative and associative, multiple (independent) bodies of evidence can be combined in any order without affecting the result. If the initial bodies of evidence are independent, then the derivative bodies of evidence are independent as long as they share no common ancestors.

The *conflict* (i.e., κ) generated during the application of Dempster's rule quantifies the degree to which the mass distributions being combined are incompatible, that is, the degree to which the two distributions are directly contradictory. When $\kappa = 1$, the distributions are in direct and complete contradiction to one another and no consensus exists (i.e., Dempster's rule is undefined); when $\kappa = 0$, there is no contradiction and the evidential intervals based upon the consensus distribution will be contained within the bounds of the evidential intervals based upon the component distributions, i.e., the combination is *monotonic*; otherwise, the component distribution are partially contradictory. In this case, Dempster's rule focuses the consensus on the compatible portions of the component distributions by eliminating

the contradictory portions and normalizing what remains; some evidential intervals based upon the consensus distributions will not fall within the bounds of intervals based upon the component distributions, i.e., the combination is *nonmonotonic*.

Translating Evidence

If a body of evidence is to be interpreted relative to a question expressed over a different frame from the one over which the evidence is expressed, a path of compatibility relations connecting the two frames is required. The mass distribution expressing the body of evidence is then repeatedly *translated* from frame to frame, via compatibility mappings, until it reaches the ultimate frame of the question. In our planning example, interpreting the effects of a body of evidence about time 1 on propositions at time 5 requires that the evidence be translated from frame to frame, for each planned action between time 1 and time 5.

In translating m_A from frame Θ_A to frame Θ_B via compatibility mapping $\Gamma_{A \rightarrow B}$, the following computation is applied to derive the translated mass distribution m_B:

$$\begin{aligned} m_B(B_j) &= \frac{1}{1-\kappa} \sum_{\Gamma_{A \rightarrow B}(A_i)=B_j} m_A(A_i) \\ \kappa &= \sum_{\Gamma_{A \rightarrow B}(A_i)=\emptyset} m_A(A_i) \\ &< 1 \ . \end{aligned}$$

Intuitively, if we (partially) believe A_i, and A_i implies B_j, then we should (partially) believe B_j; if some focal element A_i is incompatible with every element in Θ_B, then there is *conflict* (i.e., κ) between the evidence and the logic of the frames and compatibility relation. This is equivalent to the conflict in Dempster's rule.

In our example, to evaluate the effect of applying the *PUT-C-ON-A* operator, given the two independent bodies of evidence about the initial state, m_{AB} and $m_{CLEAR\text{-}C}$, we first combine these mass distributions using Dempster's rule and then translate the result via compatibility mapping $\Gamma_{PUT\text{-}C\text{-}ON\text{-}A}$ to frame Θ_B. The result is a mass distribution that attributes 0.72 to {CAB}, 0.18 to {CA-B, CBA, BA-C}, 0.08 to {CAB, ABC}, and 0.02 to the complement of {A-B-C}.

$$m_{\text{PUT-C-ON-A(AB⊕CLEAR-C)}}(A_i) = \begin{cases} 0.72, & \{\text{CAB}\} \\ 0.18, & \{\text{CA-B, CBA, BA-C}\} \\ 0.08, & \{\text{ABC, CAB}\} \\ 0.02, & \{\text{ABC, ACB, AB-C, AC-B,} \\ & \text{BAC, BCA, BC-A, BA-C,} \\ & \text{CAB, CBA, CA-B, CB-A}\} \\ 0.0, & \text{otherwise} \quad . \end{cases}$$

This induces the following evidential intervals: $[0.9, 1.0]$ for CLEAR-C, $[0.72, 1.0]$ for CA, $[0.8, 0.8]$ for AB, $[0.72, 0.8]$ for {CAB}, $[0.0, 0.2]$ for {CBA}, $[0.0, 0.1]$ for CLEAR-A, and $[0.0, 0.0]$ for {AB-C}. Given a sequence of operators to be applied after executing *PUT-C-ON-A*, we simply perform successive translations until the sequence is exhausted.

When multiple bodies of evidence are available over different frames, they must be translated to a common frame before they can be combined using Dempster's rule. They can all be translated to a single frame and combined, or subsets of the available evidence can be translated and combined at intermediate frames, and these intermediate results then can be translated and combined until the final destination frame is reached. If during plan execution, intermediate observations are made, resulting in additional bodies of evidence about the state of world at time i, these bodies of evidence can be combined with the body of evidence representing the presumed state of the world at time i, to refine the predicted outcome of the plan. So, in our example, if we had additional information about the state of the world at time 2, it could be combined with our result for that time before additional translations are performed.

9.1.3 Evidential Analyses

Once a gallery has been established, Gister-CL can analyze the available evidence. The goal of this analysis is to establish a line of reasoning, based upon both the possibilistic information in the gallery and the probabilistic information from the evidence, that determines the most likely answers to some questions. The gallery delimits the space of possible situations, and the evidential information establishes the likelihoods of these possibilities. Within an analysis, bodies of evidence are expressed relative to frames in the gallery, and paths are established for the bodies of evidence to move through the frames via the compatibility mappings. An analysis also specifies if other evidential operations are to be performed, including whether multiple bodies of evidence are to be combined when they arrive at common frames. Finally, an analysis specifies which frame and ultimate bodies of evidence are to be used to answer each target question. Thus, an analysis specifies a means for arguing from multiple bodies of

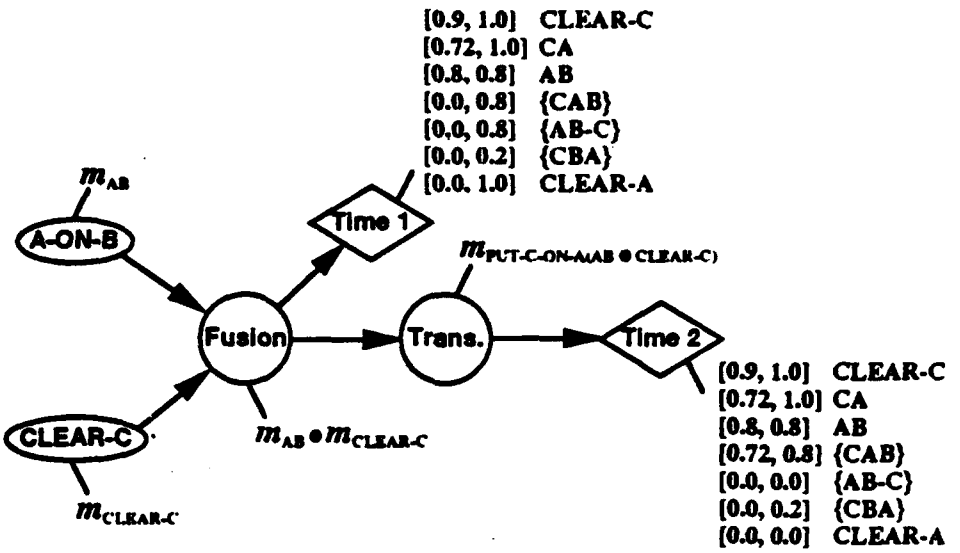

Figure 9.4: An analysis.

evidence toward a particular (probabilistic) conclusion. An analysis, in an evidential context, is the analogue of a proof tree in a logical context.

Analyses are represented as data-flow graphs where the data and the operations are evidential. Figure 9.4 is the analysis that fuses m_{AB} and $m_{CLEAR\text{-}C}$ and then applies the *PUT-C-ON-A* operator to the result, and finally interprets that result relative to some selected propositions. Primitive bodies of evidence are represented by elliptical nodes, and derivative bodies of evidence are represented by circular nodes. Diamond-shaped nodes represent interpretations of bodies of evidence. The values of these nodes are used as repositories for the information (i.e., data) that they represent. For bodies of evidence, this information includes a frame of discernment, a mass distribution, and other supporting information (e.g., the compatibility relation to use during translation).

9.1.4 Implementing Evidential Reasoning

In the preceding discussion, we have defined the frame logic in terms of set theoretic concepts. This is the way that it is most often presented, since the audience is usually more familiar with multivariate decision theory and statistics than with propositional logic. However, all of the evidential reasoning operations can be recast using propositional logic. These modified definitions follow.

Interpretation:

$$Spt(A_j) = \sum_{A_i \Rightarrow A_j} m_A(A_i)$$

$$
\begin{aligned}
Pls(A_j) &= 1 - Spt(\neg A_j) \\
[Spt(A_j), Pls(A_j)] &\subseteq [0,1] \ .
\end{aligned}
$$

Fusion:

$$
\begin{aligned}
m^3_A(A_i \wedge A_j) &= m^1_A \oplus m^2_A(A_i \wedge A_j) \\
&= \frac{1}{1-\kappa} \sum_{A_i \wedge A_j} m^1_A(A_i) m^2_A(A_j) \\
\kappa &= \sum_{\neg(A_i \wedge A_j)} m^1_A(A_i) m^2_A(A_j) \\
&< 1 \ .
\end{aligned}
$$

Translation:

$$
\begin{aligned}
m_B(B_j) &= \frac{1}{1-\kappa} \sum_{\Gamma_{A \to B}(A_i)=B_j} m_A(A_i) \\
\kappa &= \sum_{\Gamma_{A \to B}(A_i)=\mathsf{FALSE}} m_A(A_i) \\
&< 1 \ .
\end{aligned}
$$

Implementing evidential-reasoning systems can be divided into two independent subproblems: How to represent mass distributions and perform numeric calculations on them? How to represent propositions and perform logical inferences? Accordingly, Gister-CL's implementation of evidential reasoning consists of two distinct components: one that manipulates and interprets mass distributions and another that performs logical reasoning. As mass 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. The best-suited implementations depends upon the characteristics of the domain of application and upon the characteristics of the host computational environment. Most importantly, the numeric component places no constraints on the representation of propositions or the implementation of the logical operations, just so long as the logical questions posed by the numeric component are answered by the logical component.

Since different logical representations are better suited to different applications, Gister-CL allows a frame logic implementation to essentially be given as a parameter. A frame logic implementation is represented as a distinct object (using object-oriented programming techniques) capable of answering all of the logical questions required to support the numeric module's evidential operations. One such implementation is based on set theory, mirroring the set-theoretic presentation of the frame logic in this report.

However, it should be clear from the simple blocks world example in this report that this representation is not suitable for real-world planning. The representation is too cumbersome, since each possible world state must be enumerated, and the compatibility relations must specify all compatible states for every possible world state. Instead, we have developed a frame logic based upon SIPE-2's representation of plans and techniques for reasoning about them.

9.2 A SIPE-2 Logic for Gister-CL

The central idea behind the combination of SIPE-2 and Gister-CL is that the former can provide the logic used by the latter when the domain is planning, with several advantages. Briefly, these advantages are compactness of representation (one does not enumerate every possible world state nor elements of compatibility relations), SIPE-2's efficiency when determining the truth of a proposition in a world state, the use of nonlinear plans under conditions imposed by SIPE-2's heuristics, and the ability of the planner to generate plans automatically when Gister-CL eventually asks that goals be satisfied rather than that operators be applied.

We have implemented a SIPE-2 frame logic for Gister-CL as described below, and have tested it by evaluating and interactively constructing plans in a blocks world with uncertain states. When Gister-CL evaluates plans, the SIPE-2 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 performing translations using compact SIPE-2 operators.

9.2.1 World States and Propositions

The different possible initial world states are represented in Gister-CL as SIPE-2 plan nodes of type *planhead*. Planhead nodes explicitly list all predicates that are true at that node. All other world states, i.e., those generated by planned actions, are represented in Gister-CL by SIPE-2 plan nodes of type *process*. The planner creates these plan nodes in response to requests from Gister-CL to perform translations (see Section 9.2.2). Process nodes implicitly represent the world state since they list only predicates that have changed since the previous node. Gister-CL represents an incompletely specified world state as a set of plan nodes.

For example, given the propositions AB and CLEAR-C at time 1, we represent the resulting incompletely specified world state as {AB-C, CAB}, where AB-C and CAB are names for SIPE-2 planhead nodes. Suppose an operator is applied in this state. As described in the next section, this will cause SIPE-2 to produce plan networks for each element of this set, and the incompletely specified world state at time 2 would be represented as {P13, P19}, where P13

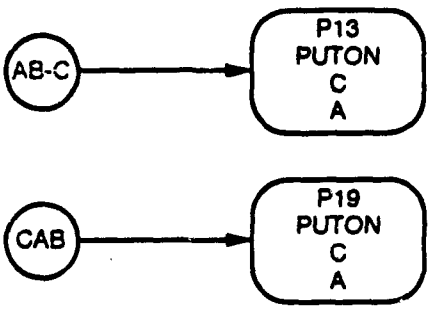

Figure 9.5: A plan network.

and P19 are names for SIPE-2 process nodes in a plan network. SIPE-2's graphical portrayal of this plan network is in Figure 9.5. From SIPE-2's perspective, this network consists of two distinct plans, each beginning in a different world state, and each applying a single operator.

Gister-CL needs to query the truth of propositions in specific world states. In our implementation, Gister-CL accepts propositions specified in SIPE-2 syntax, which are then passed on to the planner together with the plan node representing the desired world state. Checking a proposition in an incompletely specified state will result in several such calls to the planner, one for each element in the set of plan nodes representing the incomplete state information. For the answer to be true, SIPE-2 must conclude that the proposition is true at each plan node in the set; for the answer to be false, SIPE-2 must conclude that the proposition is false for each plan node; otherwise, the truth value of the proposition is indeterminate with respect to the incompletely specified state. SIPE-2 parses the propositions into its data structures and simply applies its truth criterion to the proposition at the plan node. The plan node is part of a plan network that the truth criterion uses to compute its result. This computation has been shown in practice to be efficient, even in realistic domains [34].

In the blocks world example, the SIPE-2 predicates, plans, and operators are exactly the same as they are in published examples [33]. The only limitation of this technique is that certain restrictions are placed by SIPE-2 on the form of the input propositions [33]. This did not prove to be problematic, and some restrictions could be easily relaxed. This implementation provides both the compactness of plan nodes as a representation and the efficiency of computing on them with the truth criterion.

9.2.2 Translations

As discussed earlier, Gister-CL can use a graph to represent a compatibility relation that captures the effects of an action. However, this representation is much too cumbersome in practice since an arc must be included from every possible pair of successive world states under that action.

The SIPE-2 logic does translations for Gister-CL by using its operators to generate plan networks. Gister-CL has a set of named compatibility mappings, but does not represent these mappings in any more detail. Gister-CL calls SIPE-2 to translate from one world state to another using the named compatibility mapping. The planner translates Gister-CL's name into a goal or process node in a plan network. For example, a translation request for the compatibility mapping PUT-A-ON-B causes the planner to add a process node to a plan network. The process node specifies that the standard PUTON operator be applied to the arguments A and B. Alternatively, a goal node might have been added instead of a process node. Such a goal node would specify (ON A B) as the goal. The current implementation creates process nodes; the use of goal nodes is discussed in Section 9.2.2.

The node is added to the plan at the point that represents the state from which we are translating. The planner then expands this plan in more detail to obtain the final representation of the new world state. This process makes use of SIPE-2's causal theory for deducing the effects of actions. The plan node returned to Gister-CL is the last node in the expansion of the added node. Since Gister-CL may make the same request several times, SIPE-2 uses its context mechanism to keep track of all expansions and returns an already constructed plan node whenever appropriate.

Suppose we apply PUT-A-ON-C in the incompletely specified state {AB-C, CAB} at time 1. SIPE-2 creates process nodes for applying PUTON to A and C after each of the two planhead nodes AB-C and CAB. The first one is expanded by the planner, and a process node, P13, is returned. As described later, an equivalence test in the SIPE-2 frame logic allows Gister-CL to merge P13 with AC-B. The second process node cannot be expanded because the precondition of the operator is not satisfied. SIPE-2 returns the previous world state, CAB in this case, on the assumption that the executing agent recognizes the unexecutable action and ignores it. Thus the resulting uncertain state at time 2 is {P13, CAB}. Domain-specific knowledge about the effects of attempting unexecutable actions could easily be incorporated. For example, if the agent would knock C off A while attempting to put A on C in CAB, an operator could be written to encode this knowledge, and the precondition of this operator would allow it to expand the node for putting A on C.

We enhanced Gister-CL so that it can associate parameters with compatibility relations that are being used in translations. Formerly, each domain process had to be represented by a distinct compatibility relation. This required that each possible parameterization of the PUTON operator be modeled as a distinct compatibility relation with a unique name, e.g., PUT-A-ON-C. This is impractical for all but trival planning problems. With this enhancement to Gister-CL, only a single PUTON compatibility relation is required. When it is used for translation, both it and the arguments to use have to be provided, e.g., PUTON(A, C).

One advantage of using plan networks in this way is that they might contain nonlin-

ear plans, yet the nonlinearity would be invisible to Gister-CL. The process node returned to Gister-CL would be after the unordered actions in the nonlinear plan, so only SIPE-2's truth criterion needs to address the question of nonlinearity. Thus, the restriction to linear sequences can be partially alleviated.

Generating Plans for Translations

One extension of this scheme is to allow the translation to be described as a goal node to be achieved. The planner could then build an arbitrary plan for achieving this goal and use it to represent the compatibility mapping. One complication is that this means the "compatibility mapping" might vary depending on the situation (since different plans might be generated). However, achieving a goal in an uncertain world state will require that the same plan be used for each world state in the mass distribution. While this complication does not appear to pose theoretical difficulties, it has not yet been implemented. This capability of translating via goals would be useful for letting the planner fill in the details of a more abstract plan that has been provided.

9.2.3 Equivalent States

It is important to notice when two world states are equivalent in Gister-CL, since this can significantly collapse the size of the sets that the system must reason about, which in turn significantly reduces the combinatorics. This is particularly useful in the blocks world because the simple states mean that all sorts of plan networks might result in the same world state. In more complex, realistic domains it may be rare for different sequences of actions to result in exactly the same state. However, even in these domains it will eventually be necessary to recognize states as equivalent in all relevant aspects so that the combinatorics can be reduced.

For this reason, we have not written code to determine the equality of two states in SIPE-2 (a possibly expensive computation). Instead we allow the user to specify the relevant aspects for dividing states into equivalence classes. While this puts more of a burden on the user, we view it as necessary for obtaining heuristic adequacy in complex domains. This is accomplished by defining an "equivalence" operator that is designated for this checking. This is a standard SIPE-2 operator with a list of arguments and a precondition, but nothing else. Matching the precondition in a particular world state will return a list of instantiations for variables in the operator that effectively specify its equivalence class. Thus, when Gister-CL asks whether two world states are equivalent, SIPE-2 simply calls its truth criterion on the precondition of the equivalence operator at each of the two states. If the result is failure in both cases, or success with the same variable instantiations in both cases, then the two states

are equivalent. Again, the efficiency of the truth criterion is used to significantly improve on an algorithm for determining the equality of any two states.

For example, our equivalence operator in the blocks world has a precondition of (ON A OBJECT1) ∧ (ON B OBJECT2) ∧ (ON C OBJECT3), where the OBJECTn are variables to be instantiated. In a world where A, B, and C are the only blocks, this condition distinguishes every state, effectively implementing a test for equality with its efficiency obtained through use of SIPE-2's equivalence-operator mechanism. In our previous example, P13 and AC-B were equivalent. This is easily determined by matching the equivalence condition at each of these two nodes, and getting C, TABLE, and TABLE as the instantiations for the OBJECTn in both cases.

9.3 Probabilistic Operators

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. Within the blocks world, one can imagine that if a robot is attempting to move the blocks as specified in a plan, that each operation will only probabilistically achieve the intended goal. For example, if the operation is to put block C on block A, the initial grasp for block C might fall short, leaving block C in its original position, or the placement of block C on top of block A might fail, causing block C to fall to the table. These probabilistically accurate operators can be incorporated into an evidential model as probabilistic translations.

Let's first examine this using the set-based logic introduced in Section 9.1. As previously discussed, given two frames, Θ_A and Θ_B, and a compatibility relation, $\Pi_{(A,B)}$, propositional statements can be translated between these two frames. Alternatively, instead of translating propositional statements between these two frames via $\Gamma_{A \rightarrow B}$ and $\Gamma_{B \rightarrow A}$, we might choose to translate these statements to a common frame that captures all of the information and then on to the target frame. This common frame, $\Theta_{(A,B)}$, is identical to the compatibility relation $\Pi_{(A,B)}$. Frames Θ_A and Θ_B are trivially related to frame $\Theta_{(A,B)}$ via the following compatibility relations and compatibility mappings:

$$
\begin{aligned}
\Theta_{(A,B)} &= \Pi_{(A,B)} \subseteq \Theta_A \times \Theta_B \\
\Pi_{(A,(A,B))} &= \{ (a_i, (a_i, b_j)) \mid (a_i, b_j) \in \Pi_{(A,B)} \} \\
\Pi_{((A,B),B)} &= \{ ((a_i, b_j), b_j) \mid (a_i, b_j) \in \Pi_{(A,B)} \} \\
\Gamma_{A \rightarrow (A,B)}(A_k) &= \{ (a_i, b_j) \mid (a_i, b_j) \in \Pi_{(A,B)}, a_i \in A_k \} \\
\Gamma_{(A,B) \rightarrow B}(X_k) &= \{ b_j \mid (a_i, b_j) \in \Pi_{(A,B)}, (a_i, b_j) \in X_k \} \quad .
\end{aligned}
$$

Given these three frames, Θ_A, $\Theta_{(A,B)}$, and Θ_B, and two compatibility mappings, $\Gamma_{A \rightarrow (A,B)}$ and $\Gamma_{(A,B) \rightarrow B}$, a mass distribution over Θ_A can be translated to $\Theta_{(A,B)}$ and then on to Θ_B; the result will be identical to that produced through a single translation from Θ_A to Θ_B via $\Gamma_{A \rightarrow B}$.

Once this intermediate frame has been introduced, probabilistic information about the relationship between Θ_A and Θ_B can be taken into account. This information, expressed as a mass distribution, $m_{(A,B)}$, over $\Theta_{(A,B)}$, provides a means of "weighting" translations to favor some elements of $\Theta_{(A,B)}$ over others. The probabilistic translation is accomplished by translating the mass distribution over Θ_A to $\Theta_{(A,B)}$, fusing the result with $m_{(A,B)}$, and translating the fused result to Θ_B.

In our blocks world example, if $\Pi_{(A,B)}$ (and consequently $\Theta_{(A,B)}$) delimits all possible state changes between time i and $i+1$, then each nonprobabilistic operator can be represented by a mass distribution that assigns all of its mass to a single set, the set consisting of paired states from Θ_A and Θ_B where the state from Θ_A is transformed into the state from Θ_B by applying that operator. For the operator *PUT-C-ON-A* this set assigned unit mass is designated PUT-C-ON-A. Given similarly constructed sets, PUT-C-ON-TABLE and NO-OP, corresponding to the operators for putting C on the table and doing nothing (i.e., no changes in state), we can represent a probabilistically accurate operator for putting C on A by a mass distribution $m_{\text{PUT-C-ON-A}}$. If this mass distribution is intended to represent the knowledge that 90% of the time this operator acts as intended, but 10% of the time it functions as if the intended action were to put C on the table or to do nothing, then it would be defined as follows:

$$m_{\text{PUT-C-ON-A}}(X_i) = \begin{cases} 0.9, & \text{PUT-C-ON-A} \\ 0.1, & \text{PUT-C-ON-TABLE} \cup \text{NO-OP} \\ 0.0, & \text{otherwise} \end{cases} .$$

To determine the probabilistic effect of this operator on $m_{\text{AB}} \oplus m_{\text{CLEAR-C}}$, we first translate $m_{\text{AB}} \oplus m_{\text{CLEAR-C}}$ to the equivalent of frame $\Theta_{(A,B)}$ via compatibility mapping $\Gamma_{A \rightarrow (A,B)}$, fuse this result with $m_{\text{PUT-C-ON-A}}$, and then translate this result via $\Gamma_{(A,B) \rightarrow B}$ to obtain the final result. Using this technique, we conclude [0.9, 1.0] for CLEAR-C, [0.65, 1.0] for CA, [0.8, 0.8] for AB, [0.65, 0.8] for {CAB}, [0.0, 0.2] for {CBA}, [0.0, 0.19] for CLEAR-A, and [0.0, 0.8] for {AB-C}. Comparing these results with previous ones obtained using a nonprobabilistic version of PUT-C-ON-A, we find that the support for CA and {CAB} has decreased, and the plausibility for CLEAR-A and {AB-C} has increased, while the evidential intervals for the others have remained unchanged. This reflects the fact that C is less likely to be on top of A and more likely to be elsewhere.

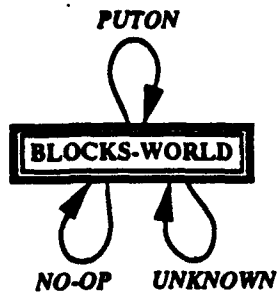

Figure 9.6: A gallery based on the Sipe-2 logic.

More specifically, we can define a propabilistic translation that transforms a mass distribution m_A^t, defined over frame Θ_A, via compatibility mappings to a frame $\Theta_{(A,A)}$ and a mass distribution $m_{(A,A)}$, to determine the result m_A^{t+1} over the same frame Θ_A, in a single step.

$$m_A^{t+1}(A_k) = \frac{1}{1-\kappa} \sum_{\Gamma_{(A,A)\leftarrow A}(\Gamma_{A\leftarrow(A,A)}(A_i))=A_k} m_A^t(A_i) m_{(A,A)}(X_j)$$

$$\kappa = \sum_{\Gamma_{(A,A)\leftarrow A}(\Gamma_{A\leftarrow(A,A)}(A_i))=\emptyset} m_A^t(A_i) m_{(A,A)}(X_j) \quad .$$

If we now turn to the SIPE–2 logic for Gister-CL, we can simplify even further. Within the SIPE–2 logic, a nondeterministic operator is captured by a mass distribution over parameterized compatibility relations where these compatibility relations correspond to SIPE–2 operators. For the blocks world, the gallery consists of a single frame with two compatibility relations: PUTON and NO-OP (Figure 9.6). Utilizing this gallery, $m_{\text{PUT-C-ON-A}}$ is represented as follows.

$$m_{\text{PUT-C-ON-A}}(X_i) = \begin{cases} 0.9, & \{\text{PUTON(C, A)}\} \\ 0.1, & \{\text{PUTON(C, TABLE), NO-OP()}\} \\ 0.0, & \text{otherwise} \quad . \end{cases}$$

If m_A^t expresses what is currently known about the state of frame Θ_A and m_X expresses a probabilistic SIPE–2 operator relative to Θ_A, then the following expression for m_A^{t+1} determines the probabilistic effect of that operator in that state.

$$m_A^{t+1}(A_k) = \sum_{\bigcup_{x_l \in X_j} x_l(A_i)=A_k} m_A^t(A_i) m_X(X_j) \quad .$$

In essence, the application of the parameterized SIPE–2 operators x_l replace the compatibility mappings and the normalizing factor is not needed since every SIPE–2 operator is defined to

return something for every state. Gister-CL was modified to directly support this operation as a single probabilistic translation. Importantly, this approach to probabilistic operators requires no changes to the SIPE-2 frame logic previously described.

Given this last addition to Gister-CL, it is fully capable of evaluating plans in the presence of nondeterministic operators and uncertain initial states.

9.4 Summary of Plan Evaluation Under Uncertainty

We have implemented a SIPE-2 logic within Gister-CL, and have tested it by evaluating and interactively constructing plans in a blocks world with uncertain world states. This work demonstrates some of the ways that our planning technology can be beneficially combined with evidential reasoning. Several advantages are obtained by this combination: compactness of representation, the efficiency of SIPE-2 operators to determine the effects of actions, SIPE-2's efficiency when determining the truth of a proposition in a world state, the use of nonlinear plans, and the ability of the planner to generate plans automatically while Gister-CL manages the uncertain aspects of the situation.

9.5 Selection of Military Forces During Planning

Military operations planning requires selecting particular military forces to carry out specific missions in support of larger military objectives. 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.

In this section, we describe how Gister-CL, which is based on the theory of evidential reasoning (ER), can be used during plan generation to help reason about selecting an appropriate Army unit to carry out a specified mission. We begin by describing how this is currently accomplished in the Army. Then we describe the problem Gister-CL is intended to address in this task and indicate how this task can be carried out within a Gister-CL framework. We present an example from the IFD-2 plan generation.

Current Unit Selection Method: Selecting a unit to carry out a particular mission is currently a highly manual process that typically involves assessing and then comparing the perceived strength of friendly and potential enemy forces. A unit is then chosen based on the

degree to which it meets or exceeds specified force-strength ratio requirements. For example, to carry out a deterrent 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 and ready complement of military personnel.
- **ARMAMENT:** The type and extent to which the troops are appropriately 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 equipment used by the unit.
- **TRANSPORT:** The type and readiness of transportation required to move the unit.
- **POSTURE:** The degree to which the unit is located where it needs to be deployed to carry out a mission.
- **MORALE:** The morale of the troops assigned to the unit.
- **TRAINING & EXPERIENCE:** The amount and type of training the unit has received for the mission at hand, and the amount and type of mission experience.
- **MOBILITY:** The degree and speed with which the unit can be relocated.

Friendly as well as enemy unit forces have an associated numerical "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 (i.e., power). Intuitively, deficiencies in one or more characteristics should be reflected in terms of a reduction in a units overall power. Expressing deficiencies is traditionally done in terms of a "force multiplier" (FM) that is associated with each characteristic. A FM is a numerical value in the range $[0, 1]$, where a value of 0 means a unit is totally deficient with respect to the corresponding characteristic, a value of 1 means a unit is not deficient with respect to the corresponding characteristic, and values decreasing from 1 to 0 mean the unit is increasingly deficient with respect to the corresponding characteristic.

The overall power of a unit should decrease as a unit becomes increasingly deficient in one or more characteristics. This intuitive behavior can be modeled in several ways. One

commonly used method involves summing the product of the FMs and the unit's base-strength value. Expressed in equation form, for $1 \leq i \leq n$ = the number of characteristics, the power of a unit u is defined as:

$$\text{Power}(u) = \text{base-strength}(u) * \prod_{i=1}^{n} \text{FM}(\text{characteristic}_i) .$$

If all FMs equal 1, then the overall power of the unit remains at its designed base-strength. A decrease in one or more FMs results in a reduction in the unit's overall power.

To select an appropriate unit to carry out a particular mission, the power of friendly (i.e., Power_f) as well as enemy (i.e., Power_e) force units must be determined. The force-strength ratio for a particular friendly and enemy unit pair is defined to be

$$\frac{\text{Power}_f(u)}{\text{Power}_e(u)} ,$$

where a ratio of $\frac{0}{0}$ is defined to be 1, a ratio of $\frac{0}{\text{Power}_e(u)>0}$ is defined to be 0, and a ratio of $\frac{\text{Power}_f(u)>0}{0}$ is defined to be ∞ which means that the power of the friendly force far exceeds the mission force-strength ratio requirements.

An appropriate unit to carry out a mission, therefore, is one which meets or exceeds the mission force-strength ratio requirements. For the case where multiple units meet or exceed such requirements, choosing any of these alternatives is said to be equally valid. It may also be the case that one or more units far exceed a force-strength ratio requirement. In this case, it may be wasteful to select such a unit if alternative units are available which meet or exceed such requirements.

Problem Statement: Under ideal circumstances, assessments of deficiencies in a unit's characteristics and determination of its overall power can be made with complete certainty. Unfortunately, we rarely have the luxury of operating under ideal situations. In practice, it may not be possible to know and assess characteristics of even friendly, much less enemy, forces. For example, assessing characteristics like morale, readiness, and tradition is a highly subjective process. Thus, it may be difficult to give a precise FM value, so a range is given instead. Furthermore, one may be uncertain about any assessment of a FM they might wish to express. In other words, the probability that a given FM is correct may be less than 1–that is, $0 \leq \text{Prob}(\text{FM}) \leq 1$. Consequently, evidential reasoning is important for selecting an appropriate unit.

In the following sections, we describe how to use Gister-CL to select an Army unit to carry out a prescribed mission.

9.5.1 Framing the Problem

Unit selection involves first comparing the perceived strength of friendly and potential enemy forces, then selecting a unit based on the degree to which a unit meets or exceeds prespecified strength-ratio requirements. Accomplishing this within an ER framework requires constructing frames of discernment and relationships between them for both friendly and enemy unit forces, a process commonly called framing the problem (see Appendix B). 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. Within Gister-CL, these frames are constructed in a gallery as shown in Figure 9.7. Frames corresponding to a friendly force begin with "F", and frames corresponding to an enemy force begin with an "E".

Relationships between the frames are represented by connecting arcs called compatibility relations (CRs). In some cases, frames such as MORALE, TRAINING & EXPERIENCE, and MOBILITY are related through CRs to form a FM associated with READINESS. Similarly, the TROOPS and ARMAMENT frames related through CRs to form a "RELATIVE FIREPOWER" FM frame. The READINESS FM values reflect the multiplicative result of combining the MORALE, TRAINING & EXPERIENCE, and MOBILITY FM values. For example, in Figure 9.8, the GUI is being used to interactively examine the FM for the morale of friendly forces, which is 0.7. (The value may have been entered interactively by a human or programmatically by a computer agent.) Similarly 0.7 was entered as a FM value for the mobility of friendly forces, and 1.0 was entered for the TRAINING & EXPERIENCE FM frame. After translating the entered FM values from their respective frames to the F.READINESS frame, the examination in Figure 9.9 shows a FM value= $0.7 * 0.7 = 0.49$ (computed by Gister-CL) that is the multiplicative result of the input FM frames.

Some relationships between FM frames are not multiplicative in nature. For example, the relative firepower of a unit depends on the relative values of its constituent TROOPS and ARMAMENT FM values. The RELATIVE FIREPOWER FM value is, in our implementation, the maximum as opposed to the product 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.

A separate frame, indicated by "B.VALS" represents the base-strength values for enemy and friendly units. A heavily armored unit, for example, would typically have a higher base-strength value than a light infantry unit. The FM frames for both friendly and enemy unit are related through CRs to frames which represents the result of computing their respective Power. These respective Power calculations are further translated through CRs to the

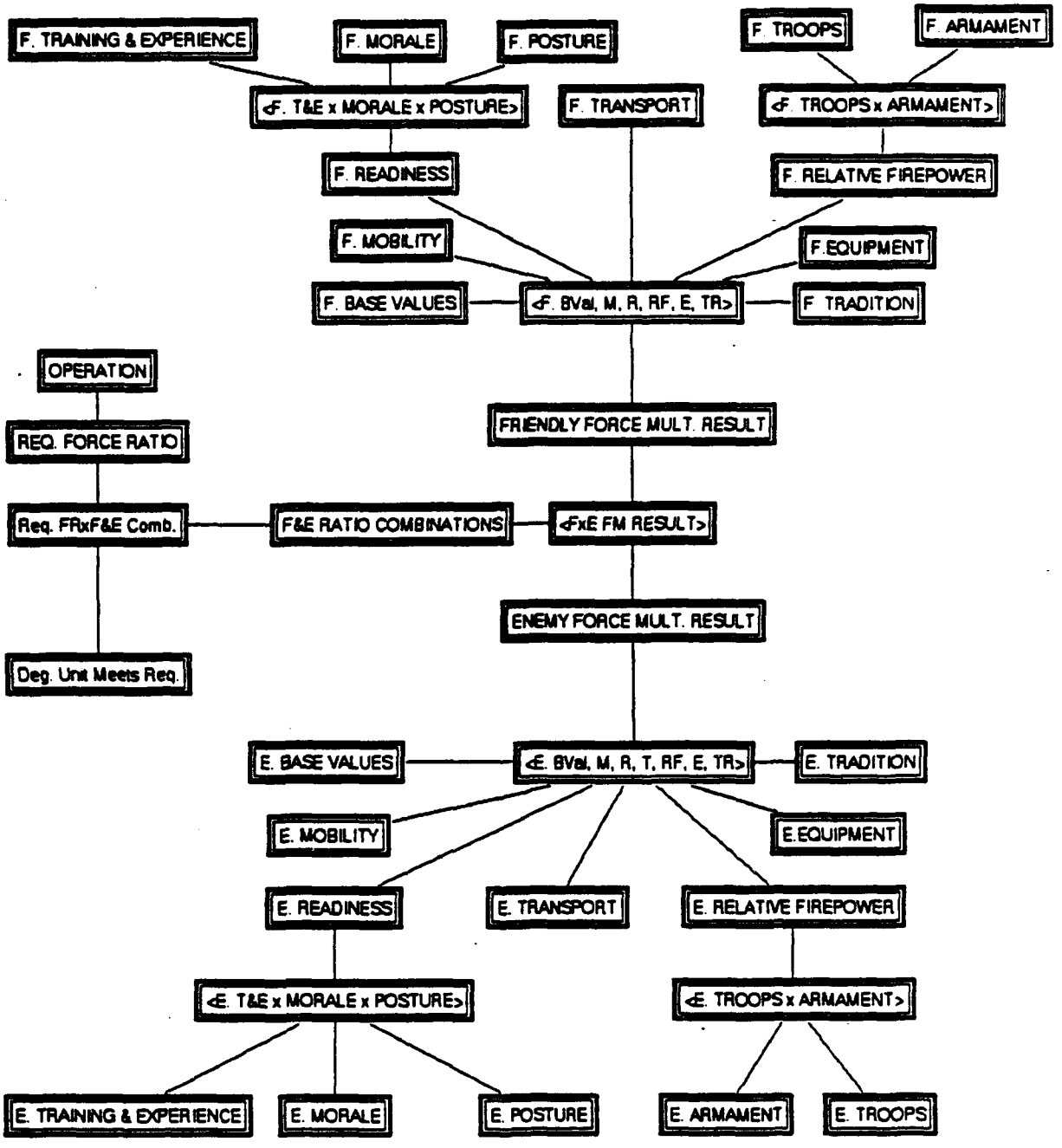

Figure 9.7: Gister gallery for Army unit selection

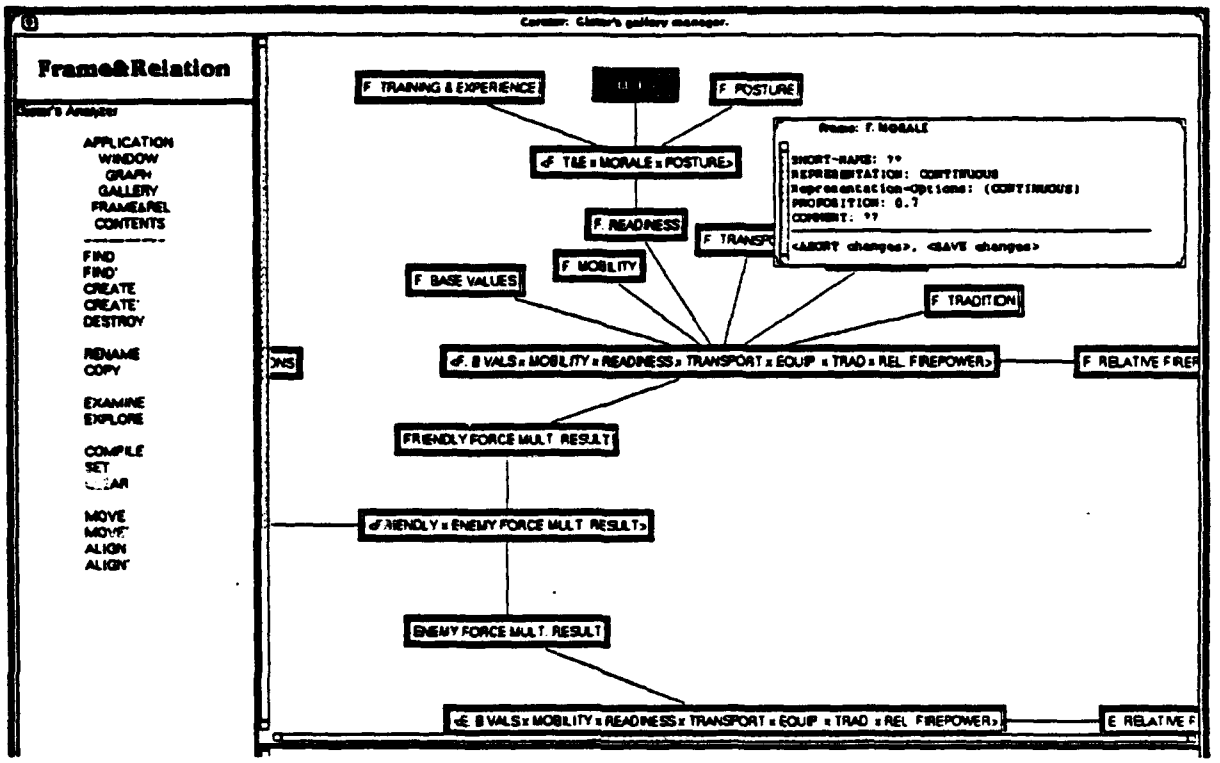

Figure 9.8: An example FM value for the F. MORALE frame.

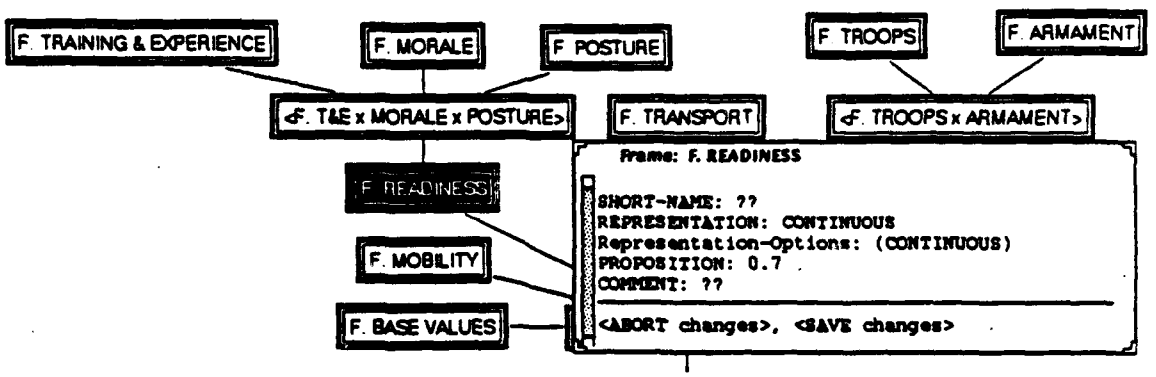

Figure 9.9: The resulting FM value for the F. READINESS frame.

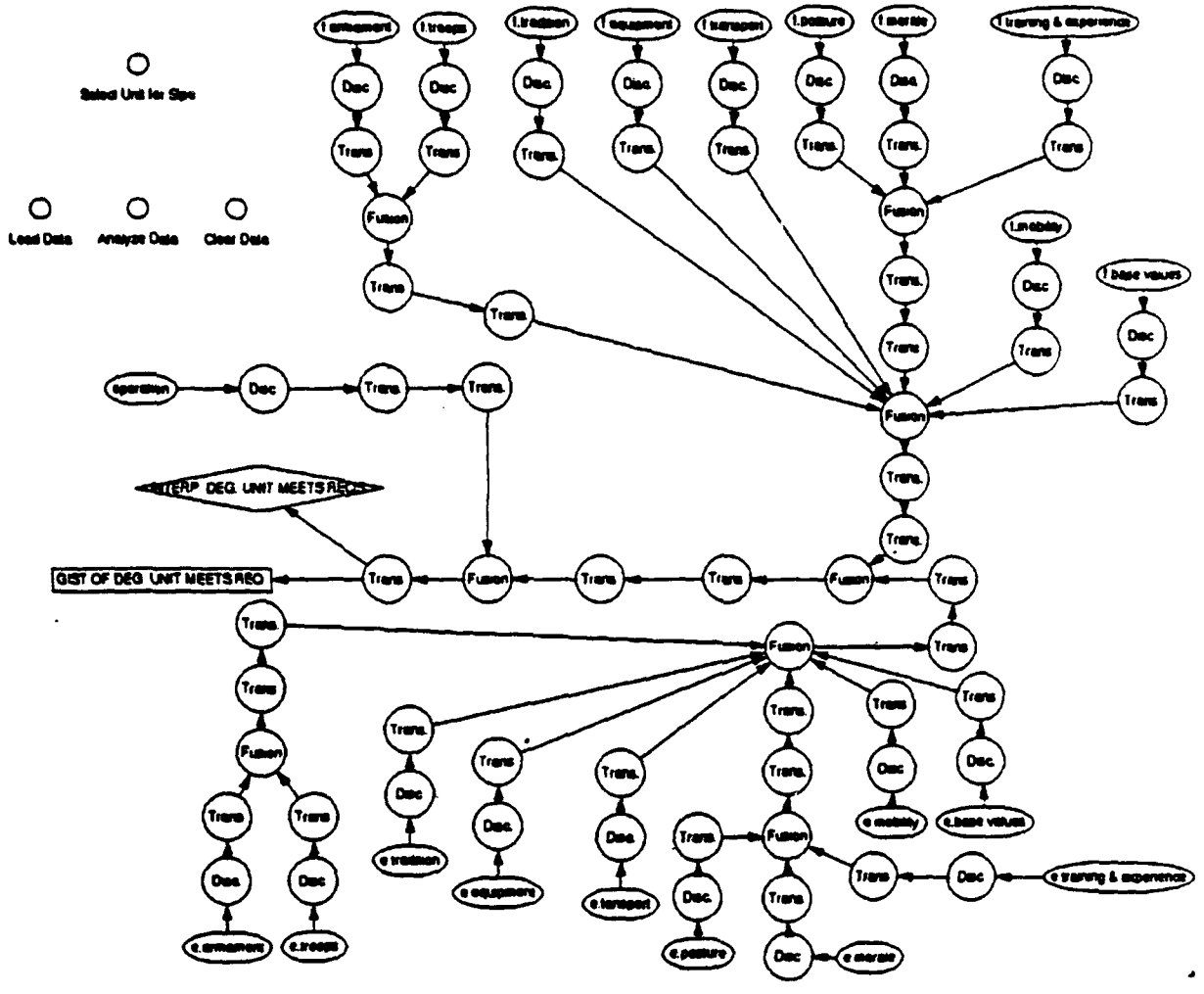

Figure 9.10: Gister analysis for interpreting information

"F&E RATIO COMBINATIONS" frame which represents their ratio. The computed ratio is then compared with the required force ratio to determine the final result for this unit. A separate frame that specifies the type of mission (e.g., deterrent or attack) is also represented in the gallery and contributes to the force ratio requirements.

9.5.2 Analyzing a Unit

Given a set of FMs for a subset of friendly and enemy characteristics, the analysis of Figure 9.10 is used to reason about the characteristics and come to a conclusion about each unit. The nodes represent the evidential operations of fusing, translating, and discounting (see Appendix B).

For example, an analysis of the characteristics of the `199th-IMB` for a deterrent mission against a particular enemy force results in the distribution shown in Figure 9.11. The distri-

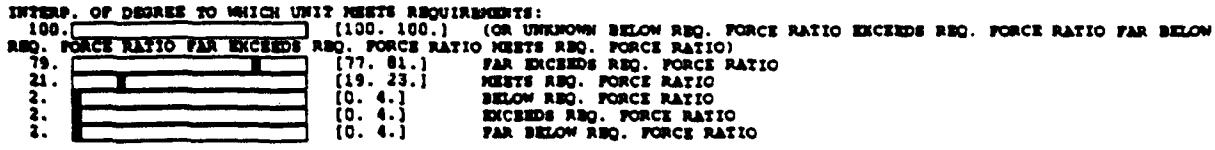

Figure 9.11: Result of degree to which unit meets force ratio requirements.

bution shown is obtained by examining the node "INTERP DEG UNIT MEETS REQS" in Figure 9.10.

9.6 Subsytem Communications

We implemented several methods and techniques for calling and receiving the results of Gister-CL analyses with both SIPE-2 and PRS-CL. We implemented the ability to graphically trace the Gister-CL analyses in a separate window while planning and/or executioin continues in the original window. It is also possible to run Gister-CL on a different machine in the CPE and use Knet and Cronus to request the execution of analyses by Gister-CL.

Chapter 10

Grasper-CL

Several of this project's accomplishments relate to Grasper-CL. The use of a Grasper-based GUI was essential in the military operations planning problem since it allowed graphical display of all plans and data structures. With such large amounts of complex knowledge, graphing the data structures proved essential to understanding and correcting the knowledge.

Grasper-CL, the system that supports the GUIs of all AIC systems, is a system for viewing and manipulating graph-structured information. Grasper-CL defines a graph as a set of labeled subgraphs; each subgraph consists of a set of labeled nodes and a set of labeled directed edges. Each edge connects two nodes; each node, edge, and subgraph have values that can be used as general repositories for information. Grasper-CL includes procedures for graph construction, modification, and queries as well as a menu-driven, interactive layout and drawing package that allows graphs to be constructed, modified, and viewed through direct pictorial manipulation. Nodes can appear in various shapes, including simple geometric figures (e.g., circles, rectangles, diamonds) and user-defined icons; edges can appear in various forms, including piecewise linear or arbitrarily curved arrows between nodes. User-definable actions are associated with every graphical object, providing complete control of mouse interactions with graphs. Grasper-CL consists of several different components: a core procedure library for programmatically manipulating the graph abstract datatype, a graph-display module for producing drawings of graphs, a graph editor that allows users to interactively draw and edit arbitrary graphs, and a suite of automatic graph-layout algorithms. Grasper-CL has proven to be an extremely flexible support system for work in expert systems and related AI topics, and serves as a general-purpose foundation for implementing graph-based user interfaces.

Work under this project accomplished the following major tasks:

- We ported a previous version of the system from the SYMBOLICS Lisp and Window environment to run under Lucid COMMON LISP and CLIM.

- We extended the capabilities of Grasper-CL in a number of respects; most significantly, we added more sophisticated automatic graph layout facilities and a birds-eye view mechanism that provides a low-resolution view of a complex graph.
- We fully documented Grasper-CL by writing a User's Guide, Programmer's Manual, and Installation Guide.
- We completed a manuscript describing Grasper-CL that was submitted to a journal for publication.

The CLIM window system is an emerging standard in Lisp windowing environments, and is part of the CPE. We have also ported Grasper-CL to run on DEC workstations under Lucid COMMON LISP and CLIM, and to run on SUN workstations under Allegro COMMON LISP and CLIM. Grasper-CL contains 27,000 lines of source code.

10.1 Extensions

We spent significant effort extending and fine tuning the Grasper-CL implementation. Our performance analysis work decreased graph-display time by a factor of 5, which is quite significant for large graphs.

We added a number of automatic graph layout capabilities to Grasper-CL. The system now has a suite of layout algorithms that will automatically position nodes in a variety of arrangements, such as in a tree, a circle, a two-dimensional array, and as a set of tiers. Each style will be preferable for graphs with different connectivities and applications with different stylistic requirements. SIPE–2 uses several of these layout algorithms to display information such as operators and the sort hierarchy. The layout algorithms can also be composed to capture hierarchical structure within a complete graph. For example, in a graph containing several cycles, we might lay out each cycle using the circular layout algorithm, and position each circular cluster of nodes relative to one another using the array layout algorithm.

We are also implementing new tools for navigating through large graphs such as SIPE–2 plans. Such graphs do not fit on the screen in their entirety, and are difficult to visualize. We have implemented a *birds-eye-view* window that shows a low-resolution view of the entire graph — nodes, edges, or both. This window can control the scrolling of the full-resolution window, and the user can query graph relationships in the birds-eye window. We also implemented 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. Such capabilities are particularly useful when viewing large military plans.

10.2 Impact

A wide variety of AI programs manipulate graphs and will benefit from a user interface that allows user interaction with a displayed graph. Grasper-CL is the most sophisticated Lisp system yet developed for manipulating and displaying graphs. Therefore, it is important to bring Grasper-CL and the innovations it is based upon to the attention of the Lisp community. Several research centers are now using Grasper-CL. Examples include researchers at NASA Ames who are using Grasper-CL to construct the user interface for a space-telescope scheduling system, and researchers at the University of Pittsburgh who are using Grasper-CL to build a computer-aided instruction application.

We have prepared 35-page manuscript for journal submission that provides an overview of Grasper-CL, focusing on the open architecture of the system (see Chapter 11). It includes the section "SIPE-2: The Anatomy of a Grasper-CL Application." A significant source of power within the system is that all the levels of its architecture are fully documented and accessible to the programmer. For example, the programmer can make use of the Grasper-CL functions that manipulate the Graph abstract datatype but have no drawing capabilities, or use high-level Grasper-CL functions to interactively alter drawn graph components (such as renaming or reshaping nodes). Each level of the Grasper-CL architecture thus provides support for different types of application programs. No previous Lisp grapher supports such a wide range of applications.

Chapter 11

Technology Transfer, Travel, Demonstrations, and Publications

This project supported a considerable amount of effort aimed at cooperating with the other participants in the ARPA/RL Planning Initiative. This chapter documents our trips, demonstrations, talks, and meetings as well as our documentation and publications, and summarizes our efforts toward technology transfer.

11.1 Technology Transfer

SRI has put considerable effort into helping to transfer the technology in CYPRESS to others. We have written several papers describing our work as detailed in Chapter 11. We have written extensive documentation for all subsystems of CYPRESS. Each system has its own manual (e.g., the manual for SIPE-2 is 168 pages long) covering use from both the user's and the programmer's perspective. Because of the complexity of Grasper-CL and its use in all the subsystems of CYPRESS, we documented it extensively, authoring a User's Guide, a Programmer's Manual, and an Installation Guide. These manuals provide in-depth, comprehensive descriptions of Grasper-CL from both the user's and the programmer's perspective.

To enhance technology transfer of CYPRESS, we have implemented a demonstration in the military operations planning domain. The demonstration is described in Chapter 4, and detailed instructions for running it are given in Appendix F. We have distributed the input data for IFD-2 to many ARPI participants, which has enabled evaluation and comparison of other technologies.

CYPRESS is available to other ARPI contractors and to the larger AI research community. Many sites are already using SIPE-2 and Grasper-CL, including Rome Laboratory,

BBN, the AI Applications Institute in Edinburgh, UCLA, GE, Rockwell, The MITRE Corporation, Paramax, ORA, Colorado State University, the University of Texas at Arlington, the University of Pittsburgh, and Stanford University. Researchers at NASA Ames are using Grasper-CL to construct the user interface for a space-telescope scheduling system, and researchers at the University of Pittsburgh are using Grasper-CL to build a computer-aided instruction application. We have made a number of improvements to our systems based on suggestions from these groups.

The AIC developed code and a set of conventions for building major software systems, that should provide a standard for specification of all software systems within ARPI. These standards specify how to load and run a system as well as providing a patch facility. They also make possible easy distribution of systems through either tape or ftp, thus overcoming a common stumbling block in technology transfer. The ease with which we have distributed and maintained our code shows the usefulness of these system specification standards.

11.2 Travel and Demonstrations

- Drs. Ingrand, Wilkins, and Lowrance attended the DARPA/RADC workshop in San Diego during November, 1990. We had discussions with Steve Cross and Nort Fowler about what they expected from this initiative and our project; and, we got a better understanding of the military transportation planning domain. Dr. Ingrand gave a talk entitled "Managing Deliberation and Reasoning in Real-Time AI Systems," and a paper of the same name appeared in the proceedings. A paper by Dr. Lowrance and Dr. Wilkins, discussed below, also appeared in the proceedings.

- Dr. Wilkins met with Mark Berstein of BBN and Mark Hoffman, Gary Edwards, and Phil Dodson of ISX, to discuss the CPE in December 1990.

- A significant amount of our effort went into preparing for and attending the Kickoff Meeting for this initiative in St. Louis, February 5-7, 1991. Both Dr. Lowrance and Dr. Wilkins attended the meeting, and Dr. Wilkins prepared and delivered a talk on the future of generative planning in this initiative.

- Dr. Wilkins participated in the knowledge representation meeting for initiative participants, held the last week of March 1991 at SRI. He described the needs of this project to the ISX and BBN team, helped design IFD-2, and started cooperating with ISX to get SIPE-2 released as a technology package.

- Dr. Wilkins traveled to Rome Labs to give a technology-transfer tutorial on SIPE-2 on May 23, 1991.

• Dr. Wilkins met with Jim Jacobs of ISX in June 1991 in an effort to release SIPE-2 as a technology package.

• In July 1991 at SRI, Dr. Wilkins and Dr. Lowrance met with ISX, BBN, and ITSC personnel to plan IFD-2. They discussed integration issues with the BBN team concerning the use of SRI technology in the prototyping environment. SIPE-2 and Grasper-CL were delivered to ISX at this meeting.

• Dr. Wilkins attended the IJCAI conference in August, 1991 where he appeared on a panel and discussed planning and execution issues with other researchers.

• In September 1991, we gave a demonstration and presentation of the results of our research to Lt. Col. Steve Cross, Dr. Gio Weiderhold, Nort Fowler, and others.

• Dr. Wilkins and Dr. Lowrance attended the ARPA/RL Planning Initiative workshop in Chicago in November 1991. They were named co-chairmen of the Generative Planning Technology Subgroup, which rewrote the technology roadmap.

• Lt. Skidmore visited SRI in November 1991, and we gave our first annual demonstration of the software so far implemented and described our future plans.

• Dr. Wilkins attended the quarterly review of the Planning Initiative in St. Louis in February 1992. He helped define seven Technology Integration Experiments; three include SIPE-2, and a fourth relating to the development of the KRSL plan language.

• Dr. Wilkins has been active in the definition of the KRSL plan language, and in February 1992 met with Nancy Lehrer of ISX and Dr. Bienkowski and Dr. Desimone of SRI to discuss this issue.

• Dr. Wesley attended the International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems, and delivered a paper entitled "An Entropy Formulation of Evidential Measures and Their Application to Real-World Problem Solving" [31].

• During the month of July 1992, several project personnel attended the AAAI conference in San Jose. The week after the conference, Dr. Wilkins and Dr. Lowrance meet for half a day with Lt. Skidmore and Mr. Hoebel to discuss this project.

• Dr. Wilkins attended the TIE demonstrations given at Rome Laboratory at the end of October 1992. SIPE-2 was demonstrated cooperating with GE's Tachyon system. Dr. Wilkins participated in discussions about future TIEs, and attempted to clarify the idea of a "common plan representation" since different people were using this term to mean different things.

• Dr. Lowrance gave a talk on evidential reasoning at Rome Laboratory and had discussions with RL personnel on November 19, 1992.

• On January 18, 1993 we gave our second annual demonstration to Mr. Hoebel at SRI. It consisted of our final demonstration, without the ability to invoke replanning during execution, and without using the CPE for communication. In a second demonstration, Dr. Lowrance and Dr. Wilkins demonstrated their combination of Gister-CL and SIPE-2 to do plan evaluation under uncertainty. This demonstration showed parameterized compatibility relations being used to model uncertain actions, and SIPE-2 being used to project the consequences of uncertain states and actions.

• Dr. Wilkins and Dr. Lowrance attended the ARPA/RL Planning Initiative annual meeting in San Antonio February 22-25, 1993. We demonstrated the CYPRESS system during the demonstrations sessions. It consisted of an improved version of our second annual demonstration. Dr. Wilkins participated in the panel on Perceptual Reasoning.

• In March 1993, Dr. Wilkins attended the 1993 Stanford Spring Symposium on *Foundations of Automatic Planning: The Classical Approach and Beyond*, and gave the introductory invited talk. This was the largest spring symposium in the series this year and probably the largest forum for AI-planning in 1993, with approximately sixty planning researchers attending. The talk was entitled, "How Many Domains Has Your Planner Planned In?", and appealed for less trivial theoretical work, and more work on real planning problems. The talk was well received.

• In July 1993, Dr. Wilkins attended the Planning Initiative meeting in Washington D.C., held prior to the AAAI conference. He also attended the conference, and had discussions with several PI participants and Rome Laboratory personnel. Plans for the SOTTE effort were made concrete.

• On September 7-8, 1993, Dr. Wilkins attended the PI Quarterly Review meeting at Rome Laboratory. Version control is a problem in the CPE, and we distributed documentation to the CPE developers on SRI's standard for system definition, version control, and patch definition/loading. These standards have proven to be useful and robust in systems like Grasper-CL, SIPE-2, PRS-CL, Gister-CL, and the ACT EDITOR. Dr. Wilkins participated in the SOTTE working group and wrote the final SOTTE white paper. The demonstration given at San Antonio was given at Rome Laboratory (and at ARPA later that month by Marie Bienkowski).

• On December 1-2, 1993, Dr. Wilkins attended the PI meeting at Yale University and participated in the working group on evaluation.

11.3 Publications

One of the major tasks of this project was to document our software. Previously, only inadequate and outdated documentation existed for SIPE-2, PRS-CL, Gister-CL, and Grasper-CL. On this project, we have written extensive documentation for all subsystems of CYPRESS. Each system has its own extensive manual covering use from both the user's and the programmer's perspective. In addition, we wrote an Installation Guide for all systems. As an example of the extent of our documentation, the original manual for SIPE-2 was a 40-page document, while the current manual is 170 pages long and contains numerous figures, commented input files for three different domains, and a formal specification of the input language in Backus-Naur Form. It should be invaluable to others in ARPI who plan to use SIPE-2.

Because of the complexity of Grasper-CL, its use in all the subsystems of CYPRESS, and its relevance to many AI applications, we documented it extensively, authoring both a 40-page User's Guide and a 170-page Programmer's Manual. These manuals provide in-depth, comprehensive descriptions of Grasper-CL from both the user's and the programmer's perspective. Extensive manuals have also been written for PRS-CL, Gister-CL, ACT EDITOR, and CYPRESS, and we have also written a Programmer's Maintenance Manual and Software Design Document as required by our contract.

Several papers written during this project have either been published, or submitted for publication. These are summarized below. In addition, we plan to complete a paper describing the application of CYPRESS in our demonstration and to submit it to the *Journal of Experimental and Theoretical AI*. An editor of the journal has invited this paper.

- A paper by Dr. Myers and Dr. Wilkins, entitled "Reasoning about Locations and Movement," was submitted to the it Artificial Intelligence Journal special issue on planning in June 1993. It presents a theory of locations and describes the practical experience of implementing this theory in SIPE-2 and SOCAP. We have been told they want to publish it after revisions have been made and approved. This paper is included in Appendix G.

- A paper, by Dr. Wilkins, entitled *A Common Knowledge Representation for Plan Generation and Reactive Execution*, was submitted to the *Journal of Logic and Computation* special issue on actions and processes in June 1993. It describes and documents the ACT formalism developed as part of this project.

- A paper, entitled "SOCAP: Lessons Learned in Automating Military Operations Planning", coauthored by Dr. Wilkins and people from ITSC was published in the *Proceedings of the Sixth International Conference on Industrial and Engineering Applications of AI and Expert Systems*, Edinburgh, Scotland, June, 1993.

- Dr. Wilkins was invited to contribute a chapter to a forthcoming Morgan Kaufmann book, *Intelligent Scheduling*, edited by Mark Fox and Monte Zweben. We wrote a paper

on the use of SIPE-2 in SOCAP as developed for IFD-2. The paper, coauthored by David Wilkins and Roberto Desimone, is entitled "Applying an AI Planner to Military Operations Planning." This paper describes the use of SIPE-2 for joint military operations planning in SOCAP, and makes clear the strengths and weaknesses of this approach. The book should appear in early 1994. This paper is included in Appendix G.

• We have prepared 35-page manuscript for journal submission that provides an overview of Grasper-CL, focusing on the open architecture of the system. It includes the section "SIPE-2: The Anatomy of a Grasper-CL Application." A wide variety of AI programs manipulate graphs and will benefit from a user interface that allows user interaction with a displayed graph. Grasper-CL is the most sophisticated Lisp system yet developed for manipulating and displaying graphs. Therefore, it is important to bring Grasper-CL and the innovations it is based upon to the attention of the Lisp community.

• The paper, "An Entropy Formulation of Evidential Measures and Their Application to Real-World Problem Solving", by Dr. Wesley was published in the proceedings of the International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems. It describes preliminary investigations of evidential measures for choosing among alternative actions.

• The paper, "Plan Evaluation Under Uncertainty," by Dr. Lowrance and Dr. Wilkins [21], appeared in the *Proceedings of the Workshop on Innovative Approaches to Planning, Scheduling, and Control*, November 1990. It describes our results on how to evaluate the likelihood that plans will accomplish their intended goals given both an uncertain description of the initial state of the world and the use of probabilistically reliable operators.

Bibliography

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

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

[3] Paul Cohen. *Heuristic Reasoning about Uncertainty: An Artificial Intelligence Approach.* Pitman Publishing, Inc., 1985.

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

[5] Arthur P. Dempster. A generalization of Bayesian inference. *Journal of the Royal Statistical Society*, 30:205–247, 1968.

[6] Jon Doyle. A truth maintenance system. In Bonnie Lynn Webber and Nils J. Nilsson, editors, *Readings in Artificial Intelligence*, pages 496–516. Tioga Publishing Company, Palo Alto, California, 1981.

[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] Thomas D. Garvey. Evidential reasoning for geographic evaluation for helicopter route planning. Technical Report 405, SRI International Artificial Intelligence Center, Menlo Park, CA, December 1986.

[9] Thomas D. Garvey and John D. Lowrance. Machine intelligence for electronic warfare applications. Final Report SRI Contract 1655, SRI International Artificial Intelligence Center, Menlo Park, CA, November 1983.

[10] Michael P. Georgeff and François Félix Ingrand. Research on procedural reasoning systems. Final Report Phase 1, SRI International Artificial Intelligence Center, 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, August 1989.

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

[13] 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, SRI International Artificial Intelligence Center, Menlo Park, CA, March 1989.

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

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

[16] John D. Lowrance. *Dependency-Graph Models of Evidential Support.* PhD thesis, Department of Computer and Information Science, University of Massachusetts, Amherst, MA, September 1982.

[17] John D. Lowrance. Automating argument construction. In *Proceeding of the Workshop on Assessing Uncertainty (November 13-14, 1986)*, Department of Statistics, Stanford University, Stanford, California, March 1987. Stanford University and the Navy Center for International Science and Technology.

[18] John D. Lowrance. *Evidential Reasoning with Gister: A Manual.* SRI International Artificial Intelligence Center, Menlo Park, CA, April 1987.

[19] John D. Lowrance, Thomas D. Garvey, and Thomas M. Strat. A framework for evidential-reasoning systems. In *Proceeding of the National Conference on Artificial Intelligence*, pages 896–903, American Association for Artificial Intelligence, Menlo Park, CA, August 1986.

[20] 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.

[21] 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.

[22] 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.

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

[24] Judea Pearl. *Probabilistic Reasoning in Intelligence Systems.* Morgan Kaufmann Publishers Inc., San Mateo, CA, 1988.

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

[26] Glenn Shafer. *A Mathematical Theory of Evidence.* Princeton University Press, Princeton, NJ, 1976.

[27] Thomas M. Strat. The generation of explanations within evidential reasoning systems. In *Proceedings of the 1987 International Joint Conference on Artificial Intelligence*, pages 1097–1104, American Association for Artificial Intelligence, Menlo Park, CA, August 1987.

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

[29] W. W. Wadge and E. A. Ashcroft. *Lucid, The Dataflow Programming Language.* Academic Press U. K., 1984.

[30] Leonard P. Wesley. *Evidential-Based Control in Knowledge-Based Systems.* PhD thesis, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts, 1988.

[31] Leonard P. Wesley. An entropy formulation of evidential measures and their application to real-world problem solving. In *Proceedings of the International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems*, pages 779–782, Servei de Publicacions i Intercanvi Cientific, Campus de la UIB. Cra. de Valldemossa, Palma, 1992.

[32] Leonard P. Wesley, John D. Lowrance, and Thomas D. Garvey. Reasoning about control: An evidential approach. Technical Report 324, SRI International Artificial Intelligence Center, Menlo Park, CA, July 1984.

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

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

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

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

[37] David E. Wilkins. A common knowledge representation for plan generation and reactive execution. Technical Report 532, SRI International Artificial Intelligence Center, Menlo Park, CA, June 1993.

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

Appendix A

SIPE-2: Interactive Planning and Execution

AI planning techniques that have been developed over the past few decades at the SRI AI Center, culminating in SIPE-2 [33, 36], have now reached the point where they can impact real problems. This is demonstrated by the recent application of SRI's SIPE-2 planning system to the problem of producing products from raw materials on process lines under production and resource constraints in a realistic factory setting [34]. SIPE-2 generates a plan that schedules dozens of orders (for possibly hundreds of products) on 6 production lines with approximately 20 separate product runs (with their corresponding needs for different raw materials) in about 4 minutes on a Symbolics 3645. To produce one such plan with no backtracking requires the generation of 1500 action and goal nodes (at all planning levels).

Because this technology is generic and domain-independent, it has the potential to impact a large variety of problems in both manufacturing and the military. For example, SIPE-2 has also been applied to planning the actions of a mobile robot, planning the movement of aircraft on a carrier deck, travel planning, and construction tasks. The manufacturing problem, i.e., moving raw materials to a production line at the correct time, has many similarities with the logistics problems that SRI is proposing to address.

SIPE-2 provides a domain-independent formalism for describing actions and utilizes the knowledge encoded in this formalism, together with its heuristics for handling the combinatorics of the problem, to plan to achieve given goals in diverse problem domains. SIPE-2 uses a hierarchical-planning paradigm, representing plans in procedural networks—as has been done in NOAH [25] and other systems. The plans include causal information so that the system can modify these plans in response to unanticipated events during plan execution. Unlike expert systems, SIPE-2 is capable of generating a novel sequence of actions that responds precisely to the situation at hand. This capability requires that the system reason about how

the state of the world changes as actions are performed, something that cannot be accomplished within the expert systems framework. SIPE-2 is a hierarchical, domain-independent, nonlinear planning system implemented in Symbolics Common Lisp. Unlike most AI planning research, the SIPE-2 development effort has had heuristic adequacy (efficiency) as one of its primary design goals.

SIPE-2 has implemented several extensions of previous planning systems, including the use of constraints for the partial description of objects, the incorporation of heuristics for reasoning about resources, the deduction context-dependent effects of actions, and replanning techniques. SIPE-2's formalism allows description of planning (and simple scheduling) problems in terms of the goals to be attained and the various activities that can be undertaken to achieve these goals. The system, either automatically or under interactive control, generates plans (possibly containing conditionals) to achieve the prescribed goals given an arbitrary initial situation. During plan execution, it can accept descriptions of arbitrary unexpected occurrences and modify its plans to take these into account.

A central problem in planning and simulation is that the effects an action has on the world depend on the exact situation in which it is executed. Thus, actions can be awkward or impossible to describe unless the system can deduce context-dependent effects. SIPE-2's domain rules are used to deduce the effects of an event that are conditional on the current situation. By allowing knowledge of cause-and-effect relations to be specified independently of the operators that describe actions, both the operators and the planning process are simplified. This makes it much easier for the user to express his domain knowledge as SIPE-2 operators. The system controls deduction by using heuristics and triggering mechanisms.

One of the primary goals during the development of SIPE-2 has been efficiency. SIPE-2 is currently the most efficient planner we know for solving combinatorially difficult planning problems. To achieve this efficiency, SIPE-2 incorporates special techniques for solving such problems as:

- The determination of the truth of a formula at a particular point in a plan
- Deduction of context-dependent effects
- The unification of two variables once they have constraints on them
- The handling of parallel interactions
- Resource conflicts
- Efficient searching through the space of all possible plans

The replanning and execution-monitoring problem being addressed is the following: Given a plan, a world description, and some appropriate description of an unexpected situation that occurs during execution of the plan, transform the plan–retaining as much of the old plan as is reasonable–into a plan that will still accomplish the original goal from the current situation. SIPE-2 divides this process into four steps (1) discovering or inputing the information about the current situation, (2) determining the problems this causes in the plan, if any, (similarly, determining shortcuts that could be taken in the plan after unexpected but helpful events), (3) creating "fixes" that change the old plan, possibly by deleting part of it and inserting some newly created subplan, and (4) determining whether any changes made by the above fixes will conflict with remaining parts of the old plan.

Because of the generic nature and wide applicability of the SIPE-2 software, domain encoding in past prototypes has taken only a few man-months. The primary advantages over conventional (non-AI) planning and scheduling software are the following:

- SIPE-2's representational language allows representation of some constraints that cannot be expressed by other techniques.
- Resources are allocated to each action with no violation of resource constraints.
- The system can be run interactively, letting a human make crucial, high-level decisions while the system ensures that all the details are correctly worked out.
- Because plans are produced in seconds or minutes, various "what-if" analyses can be run to produce and compare alternative plans.
- SIPE-2 can modify its plan in seconds in response to unexpected events. Surprises are ubiquitous in the real world and often quickly render useless the unmodifiable plans produced by linear programming techniques.

Appendix B

Gister: Evidential Reasoning

The Artificial Intelligence Center at SRI International has been developing new technology to address the problem of automated information management within real-world contexts [20, 19, 28]. The result of this work is a body of techniques for automated reasoning from evidence that we call *evidential reasoning*. The techniques are based upon the mathematics of belief functions developed by Dempster and Shafer [5, 26] and have been successfully applied to a variety of problem domains.

In this theory, the belief in a propositional statement A is summarized by an interval [Spt(A), Pls(A)], where this *evidential interval* is a subinterval of the closed real interval [0,1]. The lower bound Spt(A) represents the degree to which the evidence *supports* the proposition; the upper bound Pls(A) represents the degree to which the evidence fails to refute the proposition, i.e., the degree to which it remains *plausible*; and the difference between the two represents the residual *ignorance*. Applying this technique, complete ignorance is represented by the unit interval [0,1], while a precise-likelihood assignment is represented by the "interval" collapsed about that point, e.g., false [0,0], probability of .7 [.7,.7], true [1,1]. Other degrees of ignorance are captured by evidential intervals with widths greater than 0 and less than 1, e.g., [0,.4], [.6,.9], [.8,1]. The advantage of this representation of belief is that it directly accounts for what remains unknown. It represents exactly what is known, no more and no less.

In this technology, a body of evidence corresponds to an assignment of evidential intervals to the set of all relevant propositional statements. That is, a body of evidence induces a (large) set of partial beliefs. This representation is fine in theory, but in practice, over an expansive and dynamic real-world domain, it is computationally intractable. Therefore, an alternate, more compact, representation is required. Fortunately, the Dempster-Shafer theory provides just such a representation, called a *mass distribution*. Instead of representing a body of evidence by enumerating all of the corresponding partial beliefs, a mass distribution

associates a measure of belief with just those propositional statements that the evidence directly supports; it excludes other propositional statements that the evidence indirectly supports, or that are unsupported by the evidence.

We introduced the Dempster-Shafer theory of belief functions to the AI community in the context of image understanding [16]. Soon after, we adapted these techniques to the problem of multisensor integration for electronic warfare [9]. In both domains, we developed operable demonstration systems that drew conclusions from multiple bodies of evidence provided by disparate sources; and in both cases, the working systems remained true to the theory. This is significant, since other expert systems, based upon different uncertain reasoning theories, have had to introduce heuristic methods to compensate for mismatches between their theories and domains. Although we had successfully demonstrated the utility of Dempster-Shafer theory applied to real-world problems, we had not yet develop a general methodology for constructing evidential-reasoning systems.

More recently [19, 17], we developed both a formal basis and a framework for implementing automated reasoning systems based upon these 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* in a *gallery*); (3) representing bodies of evidence as belief distributions over these propositional spaces (i.e., *mass distributions*); and (4) establishing paths (i.e., *analyses*) for the bodies of evidence to move through these propositional spaces by means of evidential operations, eventually converging on spaces where the target questions can be answered. These steps specify a means for arguing from multiple bodies of evidence toward probabilistic conclusions.

This technology features the ability to reason from uncertain, incomplete, and occasionally inaccurate information (these being characteristics of the information available in real-world domains). It provides options for the representation of information: independent opinions are expressed by multiple (independent) bodies of evidence; dependent opinions can be expressed either by a single body of evidence or by a network (i.e., analysis) that describes the interrelationships among several bodies of evidence. These networks of bodies of evidence capture the genealogy of each body (similar in spirit to [3]) and are used as data-flow models [29] to automatically update interrelated beliefs whenever any given belief is revised (i.e., for belief revision [6]). The technology includes the following evidential operations, which are based in theory but have intuitive appeal as well:

- Fusion—This operation pools multiple bodies of evidence into a single body of evidence that emphasizes points of agreement and deemphasizes points of disagreement.
- Discounting—This operation adjusts a body of evidence to reflect its source's credibility.

If a source is completely reliable, discounting has no effect; if it is completely unreliable, discounting strips away all apparent information content; otherwise, discounting reduces the apparent information content in proportion to the source's unreliability.

- Translation—This operation moves a body of evidence away from its original context to a related one, to assess its impact on dependent hypotheses.

- Projection—This operation moves a body of evidence away from its original temporal context, to a related one. For example, a report might make direct statements that pertain to a ship's location at a particular time. Through projection, this evidence can be used to estimate the possible locations of this ship at other times, either future or past.

- Summarization—This operation eliminates extraneous details from a body of information. The resulting body of evidence is slightly less informative, but remains consistent with the original.

- Interpretation—This operation calculates the "truthfulness" of a given statement based upon a given body of evidence. It produces an estimate of both the positive and negative effects of the evidence on the truthfulness of the statement.

In addition, more recent work has incorporated concepts from sensitivity analysis [27, 28] and decision theory into the evidential-reasoning framework.

In implementing this formal approach, we have found that the gallery, frames, compatibility relations, and analyses can all be represented as graphs consisting of nodes connected by directed edges. To support the construction, modification, and interrogation of evidential structures, we have developed Gister-CL [18]. Gister-CL provides an interactive, menu-driven, graphical interface that allows these graphical structures to be easily manipulated. The user simply makes menu selections to add an evidential operation to an analysis, to modify operation parameters, or to change any portion of a gallery, including its frames and compatibility relations. In response, Gister-CL automatically updates the analyses.

Unlike other expert systems, 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. This differs markedly from other expert systems where a single line of reasoning is developed by an expert and then is instantiated over different situations by nonexperts. Using Gister-CL as the basis, we developed Navint [20], an integrated package of expert aids for

naval intelligence analysts. Gister-CL also has served as the basis for research in the areas of helicopter mission planning [8] and situation assessment for the battlefield [19].

For real-world applications where additional information might be available from sensors or other sources of knowledge, or where other actions can be taken, evidential reasoning can be applied directly to the control problem [32, 30]. Here information about the probable world state, goals, costs, etc. can be viewed as evidence, both pro and con, for taking selected actions. This work emphasizes the selection of one or more actions for immediate invocation. A natural extension would be to consider sequences of actions, i.e., evidential planning.

Recent work by Pearl [24] has focused on networks of interconnected processing elements where each element computes the probabilistic values of selected variables. In these Bayesian networks, probabilistic information flows bidirectionally along the edges in accordance with Bayes's Rule, propagating the effects of all bodies of evidence throughout the network. We recently have shown how such networks can be represented straightforwardly as special cases of our evidential analyses and how these networks provide a theoretically sound method of reasoning from probabilistically interconnected concepts, similar in spirit to "weighted" rule-based reasoning, but based upon the formal foundation of propositional logic and probability theory.

Thus, evidential reasoning supports true probabilistic reasoning, when the required probabilistic information is available, logical reasoning, when the available information is in the form of logical constraints, and interval probabilistic reasoning, in those situations where probabilities can be bounded. It has the flexibility and ease-of-use of rule-based systems while remaining consistent with logic and probability theory.

Appendix C

PRS: Procedural Reasoning

PRS-CL [11] is a reactive system for reasoning about and performing complex tasks in dynamic environments. It supports both multiagent architecture and real-time reasoning. Its architecture is shown in Figure C.1.

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

The system interacts with its environment, including other systems, through its database (which acquires new beliefs in response to changes in the environment) and through the actions that it performs as it carries out its intentions. The features of PRS-CL that contribute the most to our proposed research are its reactivity, its use of procedural knowledge, and its meta-level (reflective) capabilities.

A more detailed description of the features of PRS-CL follows.

Procedural knowledge: Procedural knowledge is a very powerful way to express plans and sequences of actions that are executed when goals have to be fulfilled or events cause activation conditions to be true. These procedures, called KAs, are PRS-CL representations of plans. It is essential that actions be described in terms of their *behaviors* rather than in terms of arbitrarily named procedures. Because the *purpose* of every step in a plan is explicitly represented, individual processes can independently decide how to achieve their own goals without thwarting that plan — indeed, they may even decide to assist. Similarly, because the *effect* of any step in the plan is explicitly represented, individual processes can determine whether or not their own proposed plans could be affected.

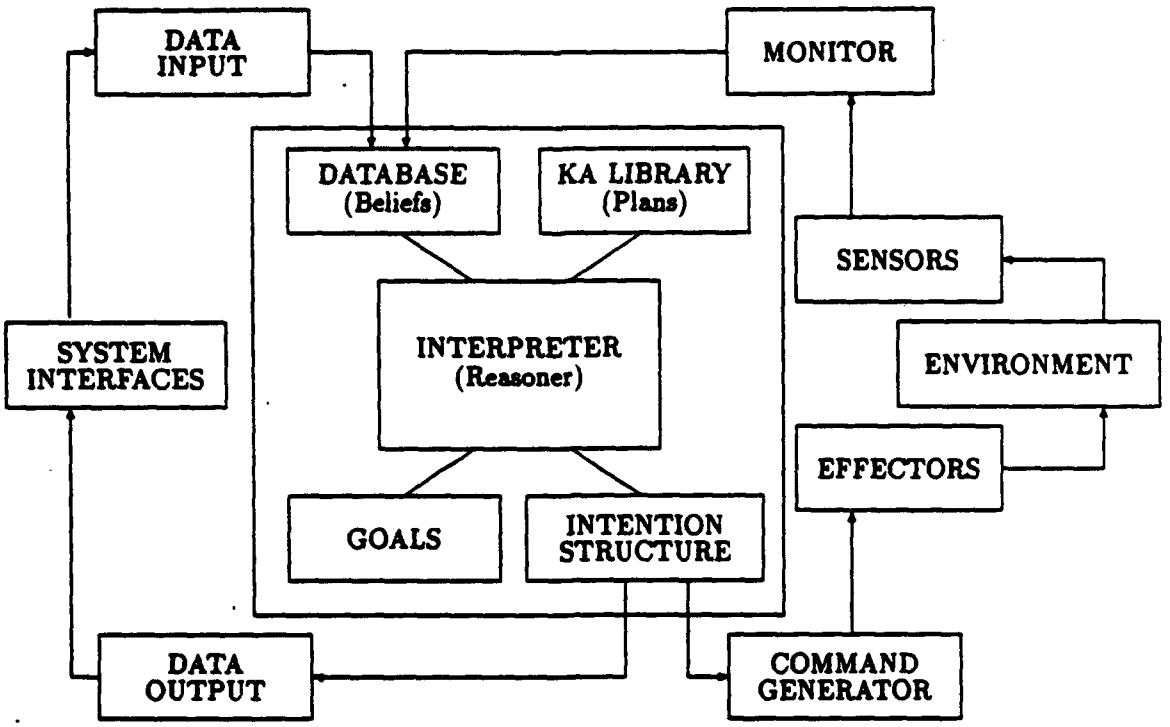

Figure C.1: Structure of the Procedural Reasoning System

Reactive *and* goal-driven system: The capability of being simultaneously data- and goal-driven is an interesting feature of PRS-CL. It provides goal-driven reasoning when explicit goals must be achieved in the world, and at the same time it enables some implicit goals to be fulfilled using data-driven reasoning. This capability of reacting to new important events makes the system highly adaptive to situation changes — any plan can be interrupted and reconsidered at the light of new incoming information.

Real-time reasoning: The PRS-CL architecture provides a framework to define real-time systems. Although we are executing procedures — which are usually complex conditional plans — the inference mechanism used in PRS-CL guarantees that any new event is noticed in a bounded time. While the system is executing any procedure, it monitors new incoming events and goals. Given that the real-time behavior of the meta-level KAs used in a PRS-CL application can be analyzed, the user can prove that his application can operate in real time — any new event is taken care of in a bounded time.

Intention graph: The intention graph enables one PRS-CL agent to follow more than one intention at a time — in other words, the system can be simultaneously working on more than one task at a time. The flexibility provided by PRS-CL to manipulate the intention graph

enables the user to specify any kind of priority or scheduling mechanism for the execution of these different intentions.

Meta reasoning: Part or all of the control of the system itself is made using meta-level KAs. These meta level KAs follow the same syntax and semantics as application KAs, except that they deal with the control of the execution of PRS-CL itself. Thus one can write meta-level KAs that can reason effectively about the problem-solving process itself, and about the utility of various strategies.

Multiagent architecture: PRS-CL exhibits a multiagent architecture where different instances of PRS-CL can be used in any application which requires the cooperation of more than one agent. Because the different PRS-CL agents run asynchronously, their activity is not constrained a priori by that of their colleagues.

Message-passing mechanism: A message-passing mechanism is provided to make possible communication between the different PRS-CL agents as well as with external modules such as simulators or monitors. The messages that are sent are not synchronized on any process, therefore enabling asynchronous interaction between agents and external modules.

The PRS-CL architecture has proven useful in developing real applications such as a monitoring and control system for the Reaction Control System of the NASA Space Shuttle (a real-time complex dynamic system) [10], and also a control system for naval battle management aboard a Grumman E-2C [13].

Appendix D

AIC Software Specifications

Authors: Peter D. Karp, David E. Wilkins, and John D. Lowrance

D.1 Introduction

This document lays out a set of conventions for building major software systems in use at the Artificial Intelligence Center (AIC). The main reason for needing such conventions is that many of these software systems are used by a number of different people. People may wish to simply run the system, or to use the system as a foundation for building another system, or to recompile the system. Until now, every system has required the user to employ a completely different procedure for performing these tasks, leading to great confusion. This document defines a uniform set of Lisp functions, UNIX shell commands, and UNIX Make operations that should be implemented for every AIC software system. The conventions pertain to system directory structures, and operations for loading, compiling, running, and distributing (to sites outside SRI) the system.

Examples of systems which conform to all the conventions in this document are CYPRESS, Grasper-CL, SIPE-2, Gister-CL, ACT EDITOR, and PRS. Some of these systems are subsystems of the others. When users build systems on top of one or more of these subsystems, they are encouraged to follow these conventions.[1]

One possible point of confusion in this specification is that it refers at different times to three different languages and interpreters: COMMON LISP, The UNIX Shell, and the UNIX Make utility. Whenever we refer to a function, we mean a function defined within COMMON LISP. Whenever we refer to "loading" a file, we mean an invocation of the COMMON LISP

[1]The relevant files in the released versions of one of these systems can be used as an example for defining a new system. Grasper-CL does not have subsystems, so use one of the other systems. If you wish to use the *defsys* facility for defining systems and logical pathnames for loading files, follow the example of SIPE-2; the other systems include their own code for compiling and loading files.

function called load. When we refer to shell commands, we mean command names that have been defined within the UNIX Shell. When we refer to shell commands with names of the form "make XXX," we mean operations defined within a file called Makefile that are processed by the UNIX Make utility. We use the word "executable" to mean a file that was created by saving the virtual memory of a COMMON LISP process to disk (also known as a disksave, sysout, or band), and that can be run as a UNIX command.

Each system is in its own directory within the aic root directory; different versions (releases) are stored in distinct subdirectories. Within each such subdirectory, Makefile commands are defined that support system recompilation, the creation of new executables, and the writing of distribution tapes and files. From Lisp, loading the appropriate system.lisp file is all that needs to be done to define procedures for loading, running, and compiling a selected version of any given system. The management of executables with patches is supported by the EXE system (see its documentation, within aic/library/sri/exe.lisp, for details).

Throughout this discussion, strings in all caps represent variables. For example, the name AICSYS represents the name of an AIC software system (e.g., CYPRESS, Grasper-CL, SIPE-2, Gister-CL, ACT EDITOR, or PRS).

D.2 Directory Structure Conventions

Note: Below, the tilde simply stands for the pathname of the directory where these systems are stored on the local file system.

```
~AICSYS  - main directory for the system

~AICSYS/V1/ - sudirectories corresponding to different versions of
~AICSYS/V2/   the system, named according to the version naming
                          scheme for that system

~AICSYS/released/ - a pointer to the released (public) version of the system

~AICSYS/beta/ - a pointer to the test version that will become the
                        next released version

-------------------------------------------------------------------------
```

```
Each version subdirectory, V, has the following structure and contents; all such
files need to be included in each sudirectory, i.e., no pointers

~AICSYS/V/Makefile - Makefile for building executables and distributions

~AICSYS/V/doc/ - subdirectory for documentation

~AICSYS/V/lisp/ - subdirectory for lisp source and binary files

~AICSYS/V/EXE1 - executable disksave files corresponding to version V,
~AICSYS/V/EXE2   named according to the naming scheme for different
  subsystems of AICSYS

~AICSYS/V/patches - subdirectory for executable disksave patches

--------------------------------------------------------------------------------

~AICSYS/V/lisp/translations.lisp
      - contains all system-related logical pathname definitions

~AICSYS/V/lisp/package.lisp
      - defines all system related packages

~AICSYS/V/lisp/system.lisp
      - loads translations.lisp and package.lisp, and defines the following
procedures in both the main system package and :user.
```

D.3 Lisp Functions Defined for each System

load-AICSYS &key *(subsystem :all) (reload :changed)* [*Function*]

This function defines all packages and translations and loads the subsystem files (and any other systems on which it depends). The subsystem files actually loaded depend upon the value of reload and whether or not subsystems of AICSYS are already loaded. If reload is :none then no files are loaded if subsystems of AICSYS are loaded (as determined by loaded-AICSYS), else all subsystem files of AICSYS are loaded. If reload is :changed then only AICSYS files modified they were last loaded are reloaded (files not previously loaded are loaded). If reload is :all, then all subsystem files of AICSYS are loaded.

loaded-AICSYS &key *(subsystem :all)* [*Function*]

This is a predicate that tests if the given subsystem is currently loaded.

compile-AICSYS &key *(subsystem :all) (recompile? nil) (reload :changed)* [*Function*]

This function compiles lisp source files for subsystem of AICSYS and then loads the subsystem files; unloaded systems on which subsystem of AICSYS depends are loaded; the subsystem files actually compiled and loaded depend upon the value of recompile? and reload.

If recompile? is t, compile and load all subsystem files.

If recompile? is nil and reload is :none, compile files with outdated or nonexistant binaries (no loading). (This option is not available for all systems.)

If recompile? is nil and reload is:changed compile and load files with outdated or nonexistant binaries.

If recompile? is nil and reload is :all, compile files with outdated or nonexistant binaries and load all subsystem files.

run-AICSYS &key *(subsystem :all)* [*Function*]

The function launches the system, first loading any subsystems not already loaded.

save-AICSYS &key *(subsystem :all) (resave? nil)* [*Function*]

This function creates an executable file for AICSYS using the EXE system for managing executables; this procedure is only to be called from a Makefile; the executable is constructed according to the value of resave?.

If resave? is nil, it loads subsystem of AICSYS and all systems on which it depends (by calling load-AICSYS) and then saves the lisp image as an execuatable.

If resave? is t, it loads the previously saved executable for subsystem of AICSYS, loads patches for that subsystem executable, and then saves the lisp image as an executable.

D.4 Shell Commands Defined for each System

All these shell commands should be given while connected to the directory AICSYS/V/:

AICSYS	- runs (launches) the full executable disksave from ~AICSYS/released/AICSYS and loads applicable patches
make compile	- compiles version V of AICSYS, creating new binaries for sources dated later than their corresponding binaries
make recompile	- recompiles version V of AICSYS, creating new binaries for all sources
make save	- makes a new executable disksave out of the most recent binaries for version V of AICSYS, and makes the resulting executable the one loaded by shell command AICSYS; more than one executable file may be defined for any given system; substitute the name of the executable for EXE
make save-EXE	- makes a new executable disksave (named EXE) from the most recent binaries for version V of AICSYS; substitute the name of the specific executable for EXE
make resave	- launches the executable disksave AICSYS, loads all appropriate patches, and then recreates the patched executable file for AICSYS
make resave-EXE	- launches the executable (named EXE) of AICSYS, loads all appropriate patches, and then recreates the patched executable file EXE for AICSYS; substitute the name of the specific executable for EXE
make tape	- writes a tape of the binaries, selected sources, and documentation for version V of AICSYS
make ftp	- writes a file suitable for ftping including the binaries, selected sources, and documentation for version V of AICSYS

D.5 Examples

To execute the latest (disksaved) version of Grasper-CL from the shell:

```
% grasper
```

To load the latest system definition of Grasper-CL into a LISP job:

```
> (load "~grasper/released/lisp/system")
```

To load the latest version of Grasper-CL into a Lisp job:

```
> (load "~grasper/released/lisp/system")
> (load-grasper)
```

To load and run Grasper-CL in a LISP job:

```
> (load "~grasper/released/lisp/system")
> (run-grasper)
```

To compile all Grasper-CL files whose sources are newer than their corresponding binaries and to load all newly compiled or unloaded Grasper-CL files into a LISP job:

```
> (load "~grasper/released/lisp/system")
> (compile-grasper)
```

To recompile all Grasper-CL files, do the following from the shell:

```
% cd ~grasper/released
% make recompile
```

To create a new executable for (the :all subsystem of) Grasper-CL , do one of the following from the shell.

```
% cd ~grasper/released
% make save
```

or

```
% cd ~grasper/released
% make save-grasper
```

Appendix E

ACT Syntax

This appendix presents the syntax for defining ACTs and documents the restrictions on ACTs that are to be translated to SIPE-2 operators and PRS KAs. The restrictions are necessary because currently neither PRS nor SIPE-2 supports the full generality of the ACT formalism.

E.1 ACT Specification

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

Logical Formulas

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

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

Metapredicates

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

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

Plot Nodes

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

Environment Slots

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

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

Properties Slot

The Properties slot is filled with a property list, PRS does not process any properties, but SIPE-2 looks for two properties for declaring variables and for specifying temporal constraints. The ACT->SIPE translator recognizes the property Variables, provided that its value is a list of quantifier-pairs. Each quantifier-pair whose first element is `existential` or `universal` is processed. The translator also recognizes the property Time-Constraints. Time constraint information is used to specify ordering constraints between plot nodes that cannot be represented by the precedence arcs of the plots. Two types of constraints are used: time windows on nodes and inter-node constraints. The syntax for these properties are summarized below.

Value of Time-Constraints Property

TC-value ::= ({time-constraint}$^+$)
time-constraint ::= (allen-reln node node [{integer}-{integer}]) |
(qual-relation (point node) (point node))
allen-reln ::= **starts** | **overlaps** | **before** | **meets** | **during** |
finish | **finishes** | **equals**
qual-relation ::= **later** | **later-eq** | **earlier** | **earlier-eq** | **equals**
point ::= **start** | **end**
node ::= pred-name[.integer] | act[.integer] | plotnode
plotnode ::= *a symbol*

Value of Variables Property

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

Comment Slot

The Comment slot is generally filled with a string.

E.2 PRS Restrictions

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

- The `Unknown` operator for construction of literals is not supported.
- `Use-Resource` metapredicates on plot nodes are ignored.
- The `Conclude` metapredicate cannot be applied to either an explicit disjunction such as (OR $P_1 \ldots P_k$), or an explicit disjunction embedded within a conjunction such as (AND (Q) (OR $P_1 \ldots P_k$)). This restriction is necessary because (a) PRS does not support the insertion of disjunctive facts into its database, and (b) when given a conjunctive fact to be concluded, PRS adds the individual conjuncts into the database.

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

In addition, the following conventions should be noted:

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

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

  are equivalent for plot nodes. For environment nodes, the latter is translated to (FACT* (AND $\text{wff}_1 \ldots \text{wff}_k$)) while the former is translated to (& (FACT* wff_1) ...(FACT* wff_k)). The analogous convention holds for the translation of metapredicates of the form (TEST ($\text{wff}_1 \ldots \text{wff}_k$)).

E.3 SIPE Restrictions

The following are restrictions placed on ACTs that are to be translated to SIPE-2 operators. The ACT-to-SIPE translator issues a warning message whenever a restriction is violated.

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

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

Appendix F

Demonstrations in the SOCAP Domain

This chapter provides instructions for running two demonstrations of the CYPRESS system within the SOCAP domain. Section F.1 describes the basic demonstration, in which a deterence plan is created using SIPE-2 and Gister-CL, translated into the Act representation, then executed by PRS-CL. Section F.2 presents extensions to the basic demo that highlight the replanning capabilities of CYPRESS.

F.1 SOCAP Demonstration

The first step is to load the CYPRESS system. If an executable containing SIPE-2, PRS-CL, Gister-CL and Grasper-CL is available, invoke that executable. Otherwise, the CYPRESS system is loaded by executing the following steps:

- start a Lisp process
- execute `(load "/homedir/cypress/released/lisp/system.lisp")`
- load CYPRESS, using the appropriate call to the function `load-cypress`

Note that /homedir/ refers to the directory on your local file system where the CYPRESS system is stored. At the AIC, /homedir/ can replaced by the tilde character.

The CYPRESS system is now ready for use. The next three steps load and initialize the SOCAP demonstration files, as well as starting up the CLIM frame for the system.

- execute `(load "/homedir/cypress/released/demos/aic-socap/up-demo.lisp")`
- execute `(run-sipe)`
- execute `(load-demo :all)` from within the CLIM lisp listener pane

The command (load-demo :all) initializes the CYPRESS system for the entire SOCAP demo. The function load-demo also accepts the arguments :prs and :gister to initialize the demonstrations for those two subsystems only (SIPE-2 is initialized as part of initializing either PRS-CL and Gister-CL).

The demo is ready to run at this point. The remainder of this section describes how to interact with the various CYPRESS subsystems to control the demonstration.

To restart the demonstration after it has been run previously, call the function (prs::init-demo).

ACT EDITOR (Part 1)

To demonstrate the capabilities of the ACT EDITOR, first select this system from the application menu (click on APPLICATION, then select ACT EDITOR from the pop-up window that appears). Next, select the graph socap-acts.graph and the Act Deter-border-incursion-by-ground-patrol. (These steps can be done prior to the start of the demonstration.) This Act is used by SIPE-2 in generating the deterence plan. Make sure that before the demo starts, the Resources slot is empty.

During the actual demonstration, execute the following steps:

- Select the APPLICATION button to show the various systems that have been loaded. Explain the architecture of CYPRESS.
- Use the EXAMINE command in the NODE menu to add (army.1) as an entry for the Use-Resource slot of the node Resources (make sure that you enter a list!). This change is made to ensure that different armies are used for parallel deterence patrols.
- Invoke the ->SIPE command in the ACT menu to translate the ACT to a SIPE-2 operator. (You will be prompted for a filename in which to store the translated operator.)

SIPE-2 (Part 1)

Switch to the SIPE-2 system by clicking APPLICATION. Use the INPUT command in the DOMAIN menu to load the file containing the new version of the ACT just edited.

If desired, the user can show off various features of SIPE-2's interface, such as its ability to interact with the database and to display operators and problems. In particular, you may wish to display the planning problem to be used in the demonstration. To do so, use the DRAW PROBLEM command in the DRAWINGS menu. A list of problems will be displayed; choose the problem SOCAP-1 (problem SOCAP-2 is the same and can also be used). The problem will be graphically displayed, showing the top-level goal (protected-ti coa-1). Note that all drawing operations within SIPE-2 will be done within the current graph. For example, drawing a problem or generating a new plan will cause changes to be made to the current graph. For this reason, the user is advised to create a new graph to contain these drawings. To do so, select the DRAWINGS menu; then choose the CREATE option under the GRAPH: heading. You will be prompted for a name for the graph being created.

Optional: use the LAYOUT command in the PROFILE menu to switch to the layout with a choice window. This is a nice feature for making choices in the interactive search in a permanent pane instead of in pop-up menus.

- To begin planning, select the PLAN menu. Click on INTERACTIVE to initiate interactive planning.
- In the pop-up menu for the interactive planning options, reset the following values:
  - Under the heading "Choosing Operators", set *User chooses operator at each node* to YES.
  - Under the heading "Choosing Instantiations, Ordering", set *User chooses required instantiations* to YES.

SIPE-2 will now enter its main interactive planning loop. At the start of each cycle, the user is prompted with a pop-up menu listing the top-level interactive choices. Three of these choices are relevant for the demo. When PLAN NEW LEVEL is selected, SIPE-2 will refine the current plan one level deeper. CONTINUE PLANNING AUTOMATICALLY commands SIPE-2 to finish the planning process on its own. QUIT will terminate the interactive planning for the current problem.

- Click PLAN NEW LEVEL to start the first planning cycle.
- At this first planning level, SIPE-2 has a choice of operators to apply. You can show the interface capabilities of SIPE-2 by drawing the possible choices. The choices can be drawn by clicking on the name of the operator to be drawn (here, either `Joint-show-of-force` or `Joint-ops-defense`) then setting the flag for *Draw operator and redisplay plan after final choice* to YES.
- After you have finished drawing the operators, select `Joint-show-of-force` as the desired operator.
- After finishing with this planning level, SIPE-2 will pop-up a menu asking you to choose a name and whether to display the current plan. Set the USE-DEFAULTS flag under the heading *Draw plan graphically* by clicking on it. SIPE-2 will display the first level of the plan.
- The top-level planning menu will appear again. Continue the cycle of selecting PLAN NEW LEVEL to generate the next planning level, then drawing the resultant plan.
- During the third planning cycle, the user will again be prompted to choose among a set of operators (there will be four `deter-border-incursion` operators listed as possibilities). At this point, change the second search option (labeled *Continue search, choosing operators automatically at all nodes*) to force SIPE-2 to perform all future operator selections.
- SIPE-2 will next prompt for help in instantiating the variable `army.1`. Choose the option to use Gister-CL to analyze the situation. A menu will ask for the particular armies to consider. Here, pick `199th-imb` and `107th-acr` and YES for *Switch to Gister*.

At this point, control is switched to Gister-CL. The demonstration of SIPE-2 will continue after the use of Gister-CL.

GISTER-CL

A new window for Gister-CL will appear on the screen. Begin by explaining how Gister-CL is to be used. Basically, a probabilistic analysis of both the friendly and enemy forces is done. The analysis results in the calculation of a required force ratio for deterring the enemy, along with a probability that the ratio is satisfied by the individual friendly force. The birds-eye window can be used to show the over-all structure of the analysis graph.

- Click on the upper left circle to begin the Gister-CL analysis.
- When finished, note how the `199th-imb` is best.
- Click on EVIDENCE to bring up the evidence-level menu, then select EXAMINE. Click on the last node in the analysis (the diamond-shaped node) to show probabilistic intervals for the current force (the `199th-imb`).
- When finished with Gister-CL, select the APPLICATION menu, then click on EXIT.

At this point, control returns to SIPE-2.

SIPE-2 (Part 2)

- SIPE-2 will automatically use the `199th-imb`, as recommended by Gister-CL. It will continue to ask about other instantiations for this planning level; simply use the default value (which is highlighted) for each. Note that after the `199th-imb` has been used for the army on the first branch of the ground deterence, it is not eligible for use in the second branch. This exclusion is a result of the resource constraint added to the Act `Deter-border-incursion-by-ground-patrol` earlier in the demo.
- At the end of planning this level, the top-level interactive planning menu will appear again. The user can continue to plan each level interactively by selecting PLAN NEW LEVEL. Alternatively, the user can either:
  - choose CONTINUE PLANNING AUTOMATICALLY rather than PLAN NEW LEVEL to have SIPE-2 finish the plan on its own, OR
  - choose QUIT rather than PLAN NEW LEVEL.

  The number of levels of SIPE-2 planning that should be performed will vary, depending on the audience.
- After the final level has been completed, draw the resultant plan. This plan consists of four main threads of parallel activity for deterring the set of known threats. Two of these threads correspond to deterence using ground forces, one corresponds to deterence using naval forces, and the fourth using air forces. Use the birds-eye to show the over-all structure of the plan, pointing out the four parallel threads.

- Optional: The command ->ACT on the PLAN menu can be used to translate the final plan to the Act representation, and the ACT EDITOR can then be used to modify it before executing it in PRS-CL. Because a copy of the translated plan has been pre-loaded into PRS-CL to speed the demo, it isn't necessary to perform this step. Under normal circumstances though, this translation process would be required.

If you changed the pane layout as part of the SIPE-2 execution, use the LAYOUT command in the PROFILE menu to return to the default layout at this point. The default layout is preferable for both the ACT EDITOR and PRS-CL systems.

ACT EDITOR (Part 2)

Use the APPLICATION menu to choose the ACT EDITOR. We will display the ACT representation of the SIPE-2-generated plan and explain how PRS-CL executes this Act, using lower-level standard operating procedures that are also expressed as Acts.

Click SELECT in the GRAPH menu. In the pop-up window that is presented, choose the graph **show-of-force.graph**. Doing so will result in the plan ACT being displayed.

Now is a good time to explain the interaction between SIPE-2 and PRS-CL. The basic idea is that SIPE-2 should be used for planning down to a certain level of detail; PRS-CL can then be used to further flesh out the plan at run-time by applying appropriate ACTs at lower levels of abstraction. As an example:

- Point out the two **Ground-patrols** goals in the **Show-of-force** act.
- Choose the SELECT command and switch to the graph **ground-patrols-sops.graph**. When prompted for an Act, select **Ground-patrol**. Show how this ACT can be applied to satisfy the **Ground-patrols** goals. Also, point out the **Lookout** goal in this ACT.
- Click SELECT in the ACT menu. From the pop-up window, choose **Lookout-clear**. This act can be used to establish the **Lookout** goal in the **Ground-patrol** act in situations where there is no **Code-red** in effect (see the preconditions slot). Click on SELECT then choose **Lookout-dangerous** to show the operator to be used when a **Code-red** is in effect. The **Code-red** predicate is a good example of a run-time condition for which it is more appropriate to have PRS-CL respond then to have SIPE-2 plan.

Next, we show the ACTs used in the demonstration of PRS-CL's ability to repair failed goals.

- Click on SELECT in the GRAPH menu. From the pop-up window, choose **naval-patrol-sops.graph**. When prompted for an ACT, choose **Naval-patrols**.
- The second node in the plot of this ACT shows the goal of having **Ship-to-shore-communications** established and maintained for the duration of the ACT (this goal is specified with the **Require-Until** metapredicate). In the demonstration, PRS-CL will achieve this goal; the user will then intervene and remove the corresponding fact from the database. This retraction will trigger a repair ACT to reachieve the goal without causing the entire plan execution to fail.

- Click SELECT in the ACT menu and choose the ACT `Repair communications` and show how it can be used to re-establish the communication goals.

PRS-CL

We are now ready to switch to PRS-CL in order to execute the plan. Enter the APPLICATION menu and select PRS-CL. For initialization purposes, perform the following steps:

- Enter the AGENT menu then click on SELECT. This step will set the current agent to SOCAP if it is the only defined agent, otherwise select SOCAP from the menu presented.
- Select the EXECUTION menu to access the run-time knowledge-level operations. All remaining commands will be chosen from this menu.
- Select the TRACE command. In response to the pop-up menu, select PROCEDURE-GRAPHIC, DATABASE and RESOURCES. You will be prompted by a second pop-up window to choose the ACTs to be graphically traced. Choose among the following, as appropriate:

  Strongly Recommended:
  `Show-of-force` (the generated plan)

  To show Reactivity/Repair:
  `Naval-patrol`
  `Repair-communications`
  `Secure-sea-sector` (for extra detail, when speed is not an issue)
  `Ship-to-shore-communications` (for extra detail, when speed is not an issue)

  To demonstrate Lower-level Expansion, Resources (not generally traced):
  `Ground-patrols`
  `Lookout`
  `Secure-site`
  `Set-up-camp`

Execute the following steps to run the PRS-CL portion of the demo:

- Post the goal `(protected-ti coa-1)` using the POST GOAL menu item.
- Click PAUSE when the graphic trace switches from `Show-of-force` to `Naval-patrols`.
- Explain the naval patrol scenario and repair mechanism.
  Scenario: The Act `Naval-Patrol` requires that the fact `(ship-shore-communications 22nd-ssf)` be protected for the duration of the Act. We will interrupt the protection by removing this fact from the PRS-CL database, which will in turn invoke the Act `Repair Communications`, which was written specifically to fix this protection violation.

- Click RUN to continue execution.
- Click CONFIRM and select `(ok naval-patrols)`.
  (Do this step any time after seeing the message "Wait for `(ok naval-patrols)`".)
- Click RETRACT FACT, enter `(ship-shore-communications 22nd-ssf)`. This step causes the fact to be removed from the agent's database.
  (Do this retraction any time after the fact has been added to the database.)
- Click CONFIRM, choose `(ok repair)` after seeing the message "Wait for `(ok repair)`". (This confirmation allows the repair process to proceed.)
- Click CONFIRM, choose `(ok secure-sea-sector)`.
  (Do this step after seeing the message "Wait for `(ok secure-sea-sector)`")

No further user interaction is required at this point. PRS-CL will continue execution of the plan until it reaches a successful completion. If desired, the user can interact with PRS-CL during the remainder of the execution to perform operations such as changing the trace selections, accessing the database or pausing execution.

To redo the PRS-CL portion of the demo, it is necessary to restore the agent database to its original state. The restoration can be achieved by selecting the KNOWLEDGE menu, then clicking on RESTORE DB. The PRS-CL demo can now be repeated.

F.2 Replanning Demonstration

The replanning demonstration uses the basic SOCAP scenario described in the previous section. During plan execution, however, the user can makes modifications to the SOCAP agent database in order to cause an execution failure. The failure will in turn initiate a replanning process designed to produce an alternative plan that addresses the failure.

Overview

The original plan produced in this domain involves four main threads of execution for deterring a set of known threats. Two of these threads correspond to deterrence using ground forces, one corresponds to deterrence using naval forces, and the fourth using air forces. As part of the air deterrence operations aircraft are moved among various air bases. The use of any individual air base requires explicit transit approval; such approval is granted for all air bases used in the plan as part of the original domain knowledge.

Execution failure can be triggered by rescinding transit approval for any of these air bases. Doing so simply involves deleting the appropriate fact of the form

`(transit-approval <airbase>)`

from the SOCAP agent's database. Removal of such facts lead to plan failure when execution reaches the stage where this approval is required. Without replanning, execution would completely fail at this point. When replanning is enabled, the SOCAP agent will notify the

REPLANNER agent, which in turn will issue a replanning request to SIPE-2. Meanwhile, execution of the remaining branches of the original plan continues.

Replanning for this situation produces a modified plan in which an alternative mobilization strategy is employed. This plan is sent to the REPLANNER agent, who forwards it to the SOCAP agent who integrates the new plan into its intention structures and continues execution.

Initialization

Initialization for the replanning demonstration employs the same steps described in Section F.1, with one minor modification. Prior to executing the LISP function

```
(load-demo :all)
```

it is necessary to set the LISP variable

```
PRS::*LOAD-STORED-SOCAP-PLAN*
```

to nil. This variable controls whether or not to load a predefined Act plan for the SOCAP domain into the SOCAP agent. Pre-loading the plan is useful in situations such as the basic SOCAP demonstration from the previous section, where no modifications to the plan are needed. When replanning is to be used though, it is necessary to load a plan that has been generated by SIPE-2 during the current session. This requirement arises because the replanning process needs access to the internal SIPE-2 structures that were used to create the plan, not just the final Act version of the plan. Currently, those SIPE-2 structures are not saved with the Act representation of a plan.

Prior to the start of the demonstration it is necessary to enable replanning. The use of run-time replanning is controlled by the variable

```
PRS::*USE-REPLANNING*
```

When this variable is t, replanning is enabled. It should already be set to t by the domain initialization process, but you may wish to check before proceeding with the demonstration.

In addition to enabling replanning, it is also necessary to specify a communication protocol. As described in Chapter 5, CYPRESS replanning supports three different protocols: *message-passing*, *file-sharing*, and the use of *Knet-Cronus*. To specify a particular protocol, set the LISP variable

```
SIPE::*AGENT-PROTOCOL*
```

to either :prs-send for message-passing, :file for shared files, or :knet for KNET and CRONUS. The default value is :knet. The message-passing protocol works only when PRS-CL and SIPE-2 are running in the same LISP image. The shared-file protocol is slow. The Knet-Cronus protocol uses the Knet and Cronus systems of the CPE for fast communication when the two systems are running on different machines or in different processes. The replanning process itself is identical under any of the protocols.

The message-passing and shared-file protocols require no additional steps. To run the replanning demonstration using the Knet and Cronus protocol, the initialization process described above should be performed on the two machines to be used. In addition, it is

necessary to start the appropriate servers. On the machine that will be used for PRS-CL execution, execute the function

```
(knet:start-server :PRS-AGENT)
```

in its LISP image. On the machine for SIPE-2, execute

```
(knet:start-server :SIPE-AGENT).
```

Plan Generation using Sipe-2

The first step in the replanning demonstration is to generate a plan. To do so, follow the steps described in the previous section (remember to create a new graph in which to store the generated plan). Once the plan is generated, it needs to be translated to the Act representation for use by PRS-CL. The translation can be done using the ->ACT command on the PLAN menu. Now, switch to PRS-CL(click on APPLICATION).

Plan Execution and Replanning using PRS-CL

As described in Chapter 5, the replanning architecture makes use of a PRS-CL agent called **REPLANNER** that is dedicated to servicing replan requests from other PRS-CL agents. Thus, for the SOCAP domain there will be two active PRS-CL agents: **SOCAP** and **REPLANNER**.

Within PRS-CL, there is a notion of the *current agent* which denotes the single agent that is 'attached' to the interface. In particular, lower-level menu operations such as loading Acts or posting goals, are applied to the current agent. The current agent can be set using the SELECT command in the AGENT menu. For the replanning demonstration, a pop-up menu will appear with two agents from which to choose: **SOCAP** and **REPLANNER**.

The first step is to load the plan into the **SOCAP** agent. Set the current agent to be **SOCAP**, then select the KNOWLEDGE menu. Clicking on APPEND will bring up a menu of choices, from which you should select PROCEDURES. A menu of graph choices will appear, from which you should select the graph containing the Act version of the plan you have created. Doing so will cause the plan to be loaded into the **SOCAP** agent.

The system is ready to execute the plan. Before doing so, it is best to enable some run-time tracing for both the **SOCAP** and **REPLANNER** agents. PRS-CL permits tracing of multiple agents simultaneously. However, graphical tracing is activated only for the current agent. This restriction is necessary because the various agents operate in parallel, thus making it impossible to graphically trace the activities of more than one agent at a time. Textual trace and output windows will appear for all agents. These windows are created at the time of receipt of their first display messages.

To set the trace selections for an agent, first make that agent be the current agent. Then click the TRACE command in the EXECUTION menu. A pop-up menu will appear with the possible choices.

For the **SOCAP** agent, choose DATABASE, MESSAGES, RESOURCES (optional), PROCEDURE-GRAPHIC, and PROCEDURE-TEXT. Pop-up menus will appear that allow you to specify which Acts (procedures) should be traced graphically and textually. Activate text tracing for the Acts **Successful Replanning**, **Failed Replanning** and the

Act corresponding to the plan created using SIPE-2. Activate graphic tracing only for the Act corresponding to the plan. For the REPLANNER agent, choose only MESSAGES and PROCEDURE-TEXT for the Act REPLANNER.

To run the demonstration, set the current agent to be SOCAP. Then post the goal

```
(protected-ti coa-1)
```

to the SOCAP agent, as described in the previous section. Doing so will cause the agent to begin execution of the Act representing the plan generated using SIPE-2.

Shortly after execution commences (or prior to posting the goal), remove the fact

```
(transit-approval rota.naval.statio-afb)
```

from the SOCAP agent's database using the RETRACT FACT command. Eventually, the SOCAP agent will encounter an execution failure for a node with a Test metapredicate containing that fact.

The SOCAP agent will notify the REPLANNER agent of the failure via PRS-CL message-passing. The trace and output windows for the REPLANNER agent will provide information about the status of the replanning operation. These windows will appear as soon as the replanning process commences.

The REPLANNER agent will issue a replanning request to SIPE-2. A pop-up window will be displayed on the SIPE-2 host that documents its replanning process. Upon generation of a new plan, SIPE-2 notifies the REPLANNER agent, which in turn passes on the new plan to the SOCAP agent. Upon integration of this plan into the execution structures, the graphical tracing will switch to the newly generated Act plan. If you wish to examine this Act, click the PAUSE command to temporarily halt execution, then use the scroll bars in the interface to navigate through the Act display. To continue execution, click the RUN command.

Execution will proceed from this point without further interruptions, although the user is free to interact with the system as desired.

Controlling Replanning

Below is a list of LISP variables that can be set to control the replanning process. All variables are in the package :prs unless otherwise specified.

SIPE::*AGENT-PROTOCOL* This variable selects the communication protocol to be used. Its value can be :PRS-CL-send for message-passing, :file for file-sharing, or :Knet for Knet and Cronus.

USE-REPLANNING When this variable is nil, replanning is disabled; for all other values, replanning is enabled. The system default is nil, but the SOCAP domain files reset the value to t.

TRACE-REPLAN When set to t (the default), this variable enables both textual and graphical execution traces to be enabled automatically for any new plan generated by replanning.

PRS-ONLY-PREDICATES Contains a list of predicates that should not be forwarded to SIPE-2 when transferring the current world state for replanning. To improve efficiency, this variable should be set to a list of predicate names that are accessed only by PRS-CL. Correctness does not require that this variable be set.

REPLAN-IO-DIRECTORY Identifies the directory in which the files used under the shared-file protocol for communication should be stored. For our demonstration, this variable is set to "/homedir/cypress/beta/demos/aic-socap/".

Appendix G

Papers

This appendix contains the two papers written on this project that have not been covered in this report, or previously submitted under the documentation requirements of this contract.

The first paper, coauthored by David Wilkins and Roberto Desimone, is entitled "Applying an AI Planner to Military Operations Planning." This paper describes the SOCAP domain in some detail, the use of SIPE-2 for joint military operations planning in SOCAP, and the strengths and weaknesses of this approach. This paper will appear in early 1994 in a Morgan Kaufmann book, *Intelligent Scheduling*, edited by Mark Fox and Monte Zweben.

Following this is the paper by Dr. Myers and Dr. Wilkins entitled "Reasoning about Locations and Movement." This was submitted to the it Artificial Intelligence Journal special issue on planning in June 1993, and will be published after revisions are made and approved. It presents a theory of locations and describes the practical experience of implementing this theory in SIPE-2 and SOCAP.

DISTRIBUTION LIST

addresses	number of copies
MR. LOUIS J. HOEBEL RL/C3CA BLDG #3 525 BROOKS ROAD GRIFFISS AFB NY 13441-4505	5
SRI INTERNATIONAL 333 RAVENSWOOD AVE MENLO PARK CA 94025	5
RL/SUL TECHNICAL LIBRARY 26 ELECTRONIC PKY GRIFFISS AFB NY 13441-4514	1
ADMINISTRATOR DEFENSE TECHNICAL INFO CENTER DTIC-FDAC CAMERON STATION BUILDING 5 ALEXANDRIA VA 22304-6145	2
ADVANCED RESEARCH PROJECTS AGENCY 3701 NORTH FAIRFAX DRIVE ARLINGTON VA 22203-1714	1
NAVAL WARFARE ASSESSMENT CENTER GIDEP OPERATIONS CENTER/CODE QA-50 ATTN: E RICHARDS CORONA CA 91718-5000	1
HQ ACC/DRIY ATTN: MAJ. DIVINE LANGLEY AFB VA 23665-5575	1
WRIGHT LABORATORY/AAAI-4 WRIGHT-PATTERSON AFB OH 45433-6543	1

WRIGHT LABORATORY/AAAI-2 1
ATTN: MR FRANKLIN HUTSON
WRIGHT-PATTERSON AFB OH 45433-6543

AFIT/LDEE 1
BUILDING 642, AREA B
WRIGHT-PATTERSON AFB OH 45433-6583

WRIGHT LABORATORY/MTEL 1
WRIGHT-PATTERSON AFB OH 45433

AAMRL/HE 1
WRIGHT-PATTERSON AFB OH 45433-6573

AUL/LSE 1
BLDG 1405
MAXWELL AFB AL 36112-5564

US ARMY STRATEGIC DEF 1
CSSD-IM-PA
PO BOX 1500
HUNTSVILLE AL 35807-3801

COMMANDING OFFICER 1
NAVAL AVIONICS CENTER
LIBRARY D/765
INDIANAPOLIS IN 46219-2189

CMDR 1
NAVAL WEAPONS CENTER
TECHNICAL LIBRARY/C3431
CHINA LAKE CA 93555-6001

SPACE & NAVAL WARFARE SYSTEMS COMM
WASHINGTON DC 20363-5100 1

CDR, U.S. ARMY MISSILE COMMAND
REDSTONE SCIENTIFIC INFO CENTER
AMSMI-RD-CS-R/ILL DOCUMENTS
REDSTONE ARSENAL AL 35898-5241 2

ADVISORY GROUP ON ELECTRON DEVICES
ATTN: DOCUMENTS
2011 CRYSTAL DRIVE, SUITE 307
ARLINGTON VA 22202 2

LOS ALAMOS NATIONAL LABORATORY
REPORT LIBRARY
MS 5000
LOS ALAMOS NM 87544 1

AEDC LIBRARY
TECH FILES/MS-100
ARNOLD AFB TN 37389 1

COMMANDER/USAISC
ATTN: ASQP-DO-TL
BLDG 61801
FT HUACHUCA AZ 85613-5000 1

AIR WEATHER SERVICE TECHNICAL LIB
FL 4414
SCOTT AFB IL 62225-5458 1

AFIWC/MSO
102 HALL BLVD STE 315
SAN ANTONIO TX 78243-7016 1

SOFTWARE ENGINEERING INST (SEI)
TECHNICAL LIBRARY
5000 FORBES AVE
PITTSBURGH PA 15213 1

DIRECTOR NSA/CSS 1
W157
9800 SAVAGE ROAD
FORT MEADE MD 21055-6000

NSA 1
E323/HC
SAB2 DOOR 22
FORT MEADE MD 21055-6000

NSA 1
ATTN: D. ALLEY
DIV X911
800 SAVAGE ROAD
FT MEADE MD 20755-6000

DOD 1
R31
9800 SAVAGE ROAD
FT. MEADE MD 20755-6000

DIRNSA 1
R509
9800 SAVAGE ROAD
FT MEADE MD 20775

DIRECTOR 1
NSA/CSS
R08/R & E BLDG
FORT GEORGE G. MEADE MD 20755-6000

ESC/IC 1
50 GRIFFISS STREET
HANSCOM AFB MA 01731-1619

ESC/AV 1
20 SCHILLING CIRCLE
HANSCOM AFB MA 01731-2816

EL 2807/RESEARCH LIBRARY 1
DL AA/SULL
HANSCOM AFB MA 01731-5000

TECHNICAL REPORTS CENTER 1
MAIL DROP D130
BURLINGTON ROAD
BEDFORD MA 01731

DEFENSE TECHNOLOGY SEC ADMIN (DTSA) 1
ATTN: STTD/PATRICK SULLIVAN
400 ARMY NAVY DRIVE
SUITE 300
ARLINGTON VA 22202

MISSION

OF

ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

a. Conducts vigorous research, development and test programs in all applicable technologies;

b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;

c. Provides a full range of technical support to Air Force Materiel Command product centers and other Air Force organizations;

d. Promotes transfer of technology to the private sector;

e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.