Working Document:
Revisions as of October 9, 1998

Multiagent Planning Architecture

MPA Version 1.7

SRI Project 7150
Contract No. F30602-95-C-0235

By: **David E. Wilkins, Senior Computer Scientist**
Karen L. Myers, Senior Computer Scientist
Marie desJardins, Computer Scientist
Pauline M. Berry, Computer Scientist
SRI International

Contents

1 Introduction

The Multiagent Planning Architecture (MPA) is a framework for integrating diverse technologies into a system capable of solving complex planning problems. MPA has been designed for application to planning problems that cannot be solved by individual systems, but rather require the coordinated efforts of a diverse set of technologies and human experts. Software agents within MPA are expected to be sophisticated problem-solving systems in their own right, and may span a range of programming languages. MPA's open design facilitates rapid incorporation of tools and capabilities, which in turn encourages experimentation with new technologies.

MPA provides protocols to support the sharing of knowledge and capabilities among agents involved in cooperative planning and problem solving. MPA allows planning systems to capitalize on the benefits of distributed computing architectures for efficiency and robustness. For example, MPA is capable of generating multiple, alternative plans in parallel.

Agents within MPA share well-defined, uniform interface specifications, so that we can explore ways of reconfiguring, reimplementing, and adding new capabilities to the planning system. This paper describes at least two areas in which such exploration was undertaken. The first is exploring alternative strategies for integrating planning and scheduling. We decompose a DARPA/Rome Laboratory Planning Initiative (ARPI) planning system and an ARPI scheduling system into component modules, where each component becomes its own planning agent (PA). This decomposition allows these ARPI technologies to be more tightly and flexibly integrated.

A second area of exploration is the definition of organizational units for agents that permit flexible control policies in generating plans. Within MPA, notions of *baselevel planning cells* and *metalevel planning cells* have been defined, where the baselevel cells provide sequential solution generation and the metalevel cells employ baselevel cells to support parallel generation of qualitatively different solutions. Metalevel cells provide the ability to rapidly explore the space of solutions to a given planning problem. Meta-PAs (PAs that control other PAs) define different control strategies among the PAs under their control. For example, one meta-PA might tightly couple planning and scheduling while another might loosely couple them. At a higher level, meta-PAs might use different algorithms to accept a stream of planning requests and distribute them to PAs that are not already busy.

The MPA framework has been used to develop several large-scale problem-solving systems for the domain of Air Campaign Planning (ACP), which was first used in ARPI's IFD-4

(Fourth Integrated Feasibility Demonstration). MPA was used as the infrastructure for ARPI's Technology Integration Experiment (TIE) 97-1, providing plan generation and evaluation capabilities. This application integrated a set of technologies that spanned plan generation, scheduling, temporal reasoning, simulation, and visualization. These technologies cooperated in the development and evaluation of a complex plan containing over 4000 nodes. This integration has validated the utility of MPA for combining sophisticated stand-alone systems into a powerful integrated problem-solving framework.

Three demonstrations are described in this paper, as examples of the use of MPA. The interaction between the planner and the scheduler in one demonstration is described in some detail, as an example of how two MPA agents might interact.

This paper is a working document that describes MPA from the point of view of someone wishing to create an MPA agent. This project's home page can be found at
`http://www.ai.sri.com/~wilkins/mpa`

Disclaimer: Frequent modifications to this document should be expected. Please report errors and suggestions to wilkins@ai.sri.com.

2 The Multiagent Planning Architecture

MPA is organized around the concept of a *planning cell*, which consists of a collection of agents committed to a particular planning process. A planning cell contains two distinguished agents — the *planning-cell manager* and the *plan server*. The *planning-cell manager* composes a planning cell drawn from a community of agents and distributes the planning task among the selected agents. The *plan server* is the central repository for plans and plan-related information during the course of a planning task. It accepts incoming information from PAs, performs necessary processing, stores relevant information, and makes this information accessible to any PA through queries. An optional *knowledge server* stores application-specific information of relevance to the planning process, which may be shared among all cell agents. The knowledge server might store, for example, situation models, an object hierarchy, legacy databases, and the set of action descriptions to be used in planning.

MPA also supports the use of multiple planning cells to simultaneously produce alternative plans. Multiple planning cells are controlled by a *meta planning-cell manager*, which accepts a stream of asynchronous planning requests and distributes them to available planning cells. A meta planning-cell manager implemented in MPA is described in Section 5.1.

Supporting the MPA architecture requires a significant amount of infrastructure. One component is a *shared plan representation* that can be understood by all planning agents. MPA employs the Act formalism for this purpose, as described in Section 2.3. Additional components include a communication substrate to support multithreaded interagent message passing across networks, and tools to facilitate the construction of agents and planning cells. MPA provides both a message format and a message-handling protocol to support the sharing of knowledge among agents involved in cooperative plan generation. With MPA messages, software modules written in different programming languages can easily communicate. We have demonstrated agents written in C, C++, LISP, and Java communicating with each other.

In this section, we describe planning cells and the common plan representation. MPA intends to have PAs communicate using the common plan representation and the agent communication languages being developed by other ARPI projects, but other alternatives are being explored as well. Agent-construction tools and communication capabilities are described in Section 3, plan servers are described in Section 4, and planning-cell managers are described in Section 5.

We have reviewed emerging standards for distributed architectures and attempted to be consistent when possible with other agent frameworks, particularly the ARPA efforts in I^3 and AITS (formerly JTF-ATD). To better understand the capabilities our architecture ought to support and to help make MPA as useful as possible to the broader ARPI community, we surveyed the software tools and products that are available or under development by ARPI researchers. The results of this survey will identify technologies that are sufficiently far along in their development to be considered for inclusion as PAs.

2.1 What Is an agent?

The term *agent* has been used in many, often meaningless, ways. Petrie gives a good overview of various uses of this term and concludes [11]:

> ... "intelligent/autonomous agents" is a term that, for the moment, is not of obvious utility.

We take a systems-engineering approach to agents, and discuss here the types of agents that are useful for building planning systems. Within Petrie's classification of agents, MPA agents are *typed-message* agents, which are described as follows:

> [typed-message agents] take more of a systems-engineering approach to defining agents, which has the advantage that it more objectively distinguishes agents from other types of software. ... agents communicate using a shared outer language, inner (content) language, and ontology.

Within this context, MPA's notion of agent is quite broad. One of the goals is to organize a community of agents, mostly programs, to solve a problem. Thus, in addition to supporting agents with beliefs, desires, and intentions ("intelligent" agents) the architecture must provide support for lower-level activities. For now, we'll call these lower-level processes "agents", because they are defined and invoked in the same way as more intelligent agents.

Agenthood in the MPA is a fuzzy classification, made for pragmatic reasons. All agents should have

- A well-defined functionality
- The ability to communicate in the common language
- A precise interface specification, including the capabilities and accepted languages of the agent
- The ability to handle asynchronous incoming messages (this ability may be provided by a wrapper or facilitator and does not need to be encoded in each agent)

In the following discussion, we use the term *external* to refer to programs/agents that are not in the same program address space as a given agent. Examples include agents making requests over the net and a LISP process on one machine requesting execution of a C++ program on the same machine.

Several factors might warrant promoting a functionality to agenthood:

- The functionality would be used by external agents. A subroutine used only internally by different modules of the same system need not be specified as an agent. However, it could be desirable to do so because agent specifications encourage good documentation and good programming practice.
- There are multiple ways to provide the functionality, and thus multiple agents to respond to a request for the functionality.
- The functionality performs a significant amount of computation or applies a significant amount of knowledge, from the point of view of other agents.

- The functionality is not readily available in the programming environment.

It is useful to distinguish between certain types of agents. Possibilities include *facilitator agents* whose function is to handle asynchronous incoming messages for other agents or to distribute messages for other agents, *server agents* whose only function is to retrieve data and answer queries about the data, *dispatch agents* that determine the most efficient means for invoking a given agent or functionality, and *broker agents* that find an appropriate agent for fulfilling a given role. A standard subroutine could become an agent by combining with a facilitator-agent that handles incoming asynchronous messages. This would allow such subroutines to be invoked by a message, possibly from a remote site. We expect to develop a deeper classification of agent types as we further populate our architecture with agents.

Functionalities that only answer queries and retrieve data can be combined into one server agent that is able to answer a range of different queries. There are trade-offs here: a common query might be its own agent so as not to overload a server agent. If there are different ways to provide the information, then multiple agents should be used.

2.2 Planning Cells

MPA planning cells are hierarchically organized collections of planning agents that are committed to one particular planning process for a given period of time (see Figure 1). Each planning cell has a meta-PA that serves as the planning-cell manager (or, simply the *cell manager*), which decomposes a planning task and distributes it to the PAs of the planning cell.

Cell managers hierarchically decompose a planning task and distribute it to the PAs and meta-PAs of a planning cell, as shown in Figure 1. The planning cell is the closure of all PAs (and meta-PAs) registered by the cell manager and any PAs they invoke. Individual cell managers compose their planning cells to reflect their own planning approach. For example, a mixed-initiative meta-PA may include human agents in its cell. One cell manager might loosely couple planning and scheduling, while another might tightly couple them.

The cell manager maintains a Planning-Cell Designator (PCD) that defines a set of *roles* to be filled. A role constitutes a capacity or responsibility, such as plan generation or scheduling. Each role must be filled by an agent with the capabilities required by that role. The PCD records the name of the agent that fulfills each role in the current planning cell at any given point in time. (The agent playing a role may change during the planning process.) The cell manager stores and maintains the PCD, broadcasting changes to the cell agents as appropriate.

Figure 1: The composition of a planning cell. A meta-PA acting as the cell manager both distributes the planning task to both PAs and other meta-PAs, and oversees the overall planning process.

Local copies of the PCD are maintained by each agent to eliminate frequent queries to the cell manager for current role assignments (thus substantially reducing interagent message traffic).

Composing a planning cell involves establishing communication with an agent for each specified role in the PCD, where these agents are then committed to this particular planning process. The cell manager registers the agents that are part of the cell. Currently, our cell manager either uses a prespecified planning cell or permits a human user to select a planning cell from a set of specific agents. As MPA expands to include different technologies with similar or overlapping capabilities, it will be desirable to compose a planning cell by broadcasting for agents with the required capabilities. Note that if several agents can fill a role, the role may be filled by a broker agent that would then be responsible for routing any messages it receives to other agents. Most ARPI technologies currently have capabilities or input languages that differ from other similar technologies, so automatic construction of a planning cell by widely broadcasting a message is not yet feasible.

Developing a *capabilities model* for agents is not part of the charter of MPA. Nevertheless, each planning agent should describe inputs, outputs, side effects, and other attributes such as whether it reads and/or writes from/to the plan server, what answers it produces, whether it produces a single answer or multiple answers, whether all possible answers are returned, and whether the answer returned is optimal or just a heuristic selection. This enables one agent to replace another in a planning cell simply by meeting the stated specifications.

A key distinction made in the I^3 architecture is that between environments and configurations. An environment corresponds, in our architecture, to the entire family of available agents and the services they provide. A configuration corresponds to a specific planning cell that has been configured by a metaplanning agent to solve a specific planning problem. The I^3 architecture refers to templates that are used to generate configurations. In our architecture, we use the more general notion of metaplanning agents, who might in turn use a template to generate a planning cell (configuration).

2.3 Common Plan Representation

We reviewed ongoing efforts in a number of DARPA programs to develop common ontologies and plan representations, including KIF, Ontolingua, the JTF C^2 schema, and CORBA. Until an adequate common plan representation (CPR) at a low enough level of detail emerges as the ARPI standard, we are using the Act formalism [18] as the MPA plan representation. An extended version of Act may be suitable for fleshing out the JTF CPR in enough detail to make it usable.

Act has been used as an interlingua in a number of SRI projects, including both DARPA and NRaD projects, and has been used by the University of Michigan and Orincon in the Ships Systems Automation program of DARPA [2]. Act currently can represent planning and execution operators, as well as completely instantiated plans. As part of MPA, Act has been extended to support hierarchical plans at multiple levels of detail, using the concepts of task, plan, and action network, as described in Section 2.3.1. A separate document summarizes Act and defines its syntax [10].

We intend to modify MPA as required to be compatible with any ARPI-standard CPR. Act has shortcomings as a CPR, and will be extended as long as it remains the CPR of choice. We have extended it to incorporate the MPA plan model, and to support ASCII Acts. MPA records global data about a plan as annotations (see Section 4), and these may need to be considered as part of the CPR.

An Act is the basic structure used to represent a plan or plan fragment in the Act formalism. The Act-Editor system supports the graphical creation, manipulation, browsing and display of Acts, thus serving as a graphical knowledge editor for systems that use Act. The Act-Editor is in use at several sites and has a manual describing its capabilities [9].

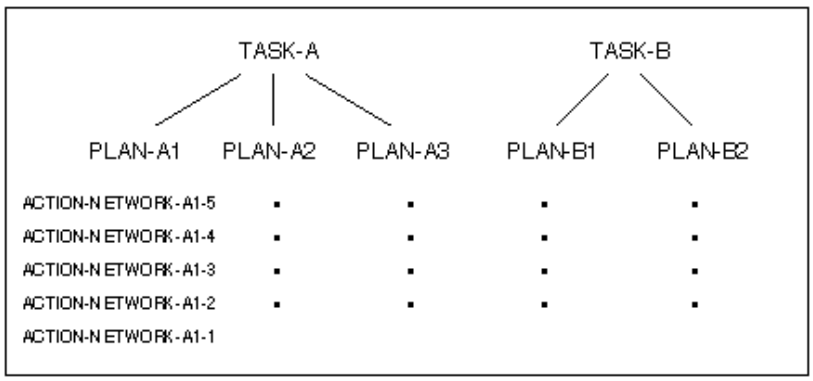

Figure 2: The structure of plan information: Each planning *task* can be composed of several alternative *plans* which in turn are composed of increasingly detailed expansions of the plan called *action networks.*

2.3.1 Plan Model

Planning cells can be generating different alternative plans for the same task, or working on different tasks. Thus, the plan server and all MPA messages must keep track of the task and the plan alternative as well as the actual action network.[1]

The MPA plan model distinguishes the concepts of task, plan, and action network, as shown in Figure 2. Most MPA messages that denote a plan require 3 arguments: the task, the plan, and the action network. Certain message are plan-level queries or updates, which do not need an action network argument.

A *task* is a set of objectives, the current situation, and other similar information. For example, a task may include actions that must be included in any plan, or advice, but such things may also be included as assumptions on a plan.

A number of alternative *plans* can be produced for a given task. A plan consists of the whole set of information about the course of action generated for this task, possibly including multiple views and multiple action networks. Each plan might have a set of *assumptions.*[2]

An *action network* is a description of the plan at some point it its development such that all agents querying that action network will get the same partial order of goals and actions. We will refer to the elements of an action network as *nodes*, where a node can represent an action

[1]We have included Austin Tate and Adam Pease in discussions of the MPA plan model to maintain compatibility with the JTF common plan representation.

[2]Whether a particular statement should be a part of the task or an assumption in the plan is a fuzzy distinction that can be made appropriately for a given domain. This will allow users to talk about tasks, plans, and assumptions in way that may correspond more naturally to their given domain.

or a goal (among other things). Action networks are intermediate results in some planning process and may be at multiple levels of abstraction.

2.3.2 ASCII Acts

Before MPA, Acts were represented as Grasper-based data structures [6, 10] to support their display display. [3] Because plans and plan fragments must be easily communicated between planning agents, we defined a syntax for ASCII specification of Acts, and documented it in BNF. We have extended the Act-Editor to print Grasper-based Acts and files of Acts in this ASCII specification, and to read this specification and create Grasper-based Acts, thus allowing graphical display and manipulation of ASCII Acts.

A separate document summarizes Act, describes functions for manipulating ASCII Acts, and specifies the current syntax for ASCII Acts [10] (superseding the appendix in our in-depth description of Act [18]).

3 MPA Infrastructure

An MPA planning cell requires low-level communication capabilities to support asynchronous agent communication across networks, particularly the Internet. Within MPA, software modules written in different programming languages can easily communicate. Utilities to facilitate the construction of agents and planning cells are also required. For example, a capabilities model will aid in planning cell construction.

In this section, we describe wrappers and agent libraries developed by SRI to facilitate the construction of agents and planning cells, and MPA's low-level communication capabilities.

3.1 Agent Construction and Wrappers

MPA provides wrappers and agent libraries that support interagent message passing, multi-threaded processing, tracing of messages on the screen, and logging of messages in a history. Eventually an agent construction tool would be desirable, such as the one that is part of the

[3] Grasper-CL is a programming-language extension to LISP that introduces graphs — arbitrarily connected networks — as a primitive data type. Grasper-CL is used as a uniform basis for the man-machine interface of many SRI systems, and is also used extensively outside of SRI.

Open Agent Architecture (OAA) [7], but such a tool is not possible under the current level of effort.

To allow more complex responses to incoming requests, such as interrupting certain responses to reprioritize the computation, it is possible to combine SRI's PRS (Procedural Reasoning System) [19] with the MPA LISP wrapper. The use of PRS wrappers is complicated, requires a knowledge of PRS and the Act formalism, and is facilitated by our examples. However, MPA provides several agents that can be used as examples — PRS is the basis for both our cell manager, plan server, and meta planning-cell manager. PRS is not a required part of any agent, and agent interface specifications are independent of PRS.

MPA agents have been implemented in C, C++, LISP, and Java. MPA provides an extensive wrapper for LISP, and some support for C, as described below.

3.1.1 LISP Wrapper

MPA provides an agent wrapper in LISP that supports asynchronous interagent communication, including functions for making, sending and replying to messages that log and trace the message exchange. There are also functions for starting and killing agents, and for starting and stopping tracing. This wrapper can easily be employed by any agent running in LISP. Other sites have been able to download our wrapper and get an existing LISP program communicating as an MPA agent in one day.

In SRI's software distribution, the wrapper is defined as the core of SRI's MPA system, which is loaded by all agents. Several wrapper functions are described in Sections 9 and 3.5.2. Documentation for wrapper functions is in Appendix A.

3.1.2 C Wrapper and Agent Library

MPA provides a library of C functions that can be used to implement a C-language wrapper for an existing "legacy" agent. This library provides an asynchronous message handler that invokes the executable image for the legacy agent as requested, and returns the responses. The handler must be programmed to translate specific messages into suitable invocations of the legacy agent (e.g., appropriate command line arguments).

The library includes a set of functions for parsing LISP s-expressions, which facilitates interaction between LISP-based and C-based agents, and allows the C agents to handle the

LISP-like MPA message syntax. Logging of messages is supported. The use of this library is documented in Appendix B.

3.2 Managing Agents

MPA makes use of a program named Startit from OAA to initiate, monitor, and terminate a set of agents. Using Startit greatly facilitates the process of starting and controlling agents. Several Startit configuration files have been written for various MPA demonstrations and can be used as templates for customized planning cells. These files are `mpa/released/oaa/*.config`

The `mpa/released/doc/` directory contains documentation on how to run Startit and how to run various demonstrations that use Startit configuration files.

3.3 How Will Agents Communicate?

Three software packages were considered for providing the low-level communication substrate for invoking procedures over a network:

- Knowledge Query and Manipulation Language (KQML) [4], developed with ARPI support by Loral and the University of Maryland Baltimore County
- Inter-Language Unification (ILU) [5], developed at Xerox-PARC
- Open Agent Architecture (OAA) [7], developed at SRI International

Currently, MPA is demonstrated on top of KQML, until such time as efficiency or robustness considerations force us to consider another approach. A preliminary implementation of MPA on top of OAA also exists, and an implementation based on ILU is being developed. The desired substrate can be selected using the variable `cl-user::*mpa-substrate*` (see Section A.1).

KQML is not CORBA-compliant, but has good high-level support for declaring agents and services and facilitating their use. It has been used fairly extensively (both within and outside of ARPI). The KQML developers stress their set of high-level performatives with semantics. While we view the semantics of the given performatives as weak, we believe the use of standard performatives is the best way to proceed. OAA provides more high-level agent facilitation than does KQML.

3.4 KQML

We found KQML to be easy to use initially, and have made several extensions to KQML's LISP interface to improve its robustness and efficiency. Information on KQML can be found at `http://www.cs.umbc.edu/kqml/`.

SRI extended the KQML system used within MPA with SRI's system-management code. This provides version control and a patch facility (for LISP agents), which is of crucial importance in maintaining working demonstrations. We have distributed our system code to other KQML users, and written clearer instructions for using KQML than previously existed. We have uncovered a few bugs that have been fixed in the SRI version of KQML. With these fixes, we have found KQML to be fast enough even for low-level messages (where a tenth of a second is adequate response), and its high-level facilitation to be useful.

In SRI's software distribution, KQML is automatically loaded as a subsystem by the MPA system. Thus, the following command will load both KQML, the MPA LISP wrapper functions, and all patches to these two systems: `(sri:load-system :mpa)`.

3.5 MPA Messages

MPA provides both a message format and a message-handling protocol to support the sharing of knowledge among agents involved in cooperative problem solving. The message facility is built on KQML, which provides a set of high-level performatives, called *communication performatives* in MPA. MPA further specifies its own set of *plan performatives*, which further specialize messages to MPA. Combinations of communication and plan performatives define the message protocols and the operations that agents may attempt to invoke by using other agents, thus providing a substrate on which to build higher-level models of agent interaction.

The same KQML-based messages are used for all communication substrates. In particular, they are used for the OAA implementation. With MPA messages, we have demonstrated agents written in C, C++, LISP, and Java communicating with each other. The following sections document the messages used to communicate between various MPA components.

3.5.1 Detailed Syntax of Messages

MPA messages are built using KQML messages, which are lists composed of a performative followed by a set of fields specified as keyword and value pairs. Of particular importance is

the :content field, which contains the real content of the message, and can be specified in any agreed upon language. The :reply-with field denotes whether or not a reply is expected, and must be non-nil to get a reply.

MPA uses a communication performative to begin a message. The content field of an MPA message begins with a *plan performative*, which is a keyword. Plan performatives are also included in the content of any replies. The general form of the content field of any MPA message (sending or reply) is

```
(:plan-performative {value} {keyword-and-value-pairs})
```

Throughout this document, braces {} show optional entities. The first value is the plan performative (a keyword), the second (optional) value is an arbitrary value, followed by optional fields specified as keyword and value pairs. The allowable values, both for the optional value and the fields, are dependent on the plan performative being used. The one field that may be used for all plan performatives is :error. When there is an error or unexpected condition, the replying agent adds an :error field to the message, whose value is generally a string explaining the problem, suitable for passing on to a user.

Below are two sample messages sent by the cell manager to the plan server. The first is used to verify that the receiving agent is alive; for this reason a reply is requested. A sample response is provided. The second message informs the receiving agent of an update to the PCD; it requires no response. These messages are handled by all agents.

```
(evaluate :sender pcm :reply-with ping
          :content (:ping))
(reply :sender "plan-server"
       :content (:ping :ok))

(tell :sender pcm :content
  (:pcd ((:pcm "pcm")(:planner "sipe")
         (:scheduler "opis")
         (:plan-server "ps1"))))
```

In the MPA release, there are logs of actual message traffic in subdirectories beginning with `log` in `/mpa/released/doc/`. For example, to see the messages during the planning for the TIE 97-1 demonstration, look at the files in `/mpa/released/doc/log-97-123/`.

Agents, particularly the plan server, will respond to messages that are not intended to have replies if the :reply-with field is non-nil. The standard content field of such a reply is `(P-PERF :ok)`, where `P-PERF` is the plan performative of the incoming message. This is referred to as an :ok message.

3.5.2 Wrapper Functions for Messages

The wrapper function for creating an MPA message to be sent is `mpa:make-msg`. The message is sent with the function `mpa:send-msg`, which sends the message, and (depending on the settings of global variables) both traces the message exchange in a window on the screen and logs the exchange in a history. Reply messages are generated using the following wrapper functions, which are documented in Section A:

```
mpa:make-response mpa:make-response-key  mpa:make-ok-response
mpa:always-reply  mpa:unknown-perf-response
```

For example, the standard reply for an unrecognized plan performative, `P-PERF`, is generated with this call to a MPA wrapper function:

```
  (mpa:unknown-perf-response P-PERF MESSAGE)
which generates the message:
  (error :content (P-PERF :unrecognized-plan-performative
            :error "Unknown plan performative P-PERF for C-PERF"))
```

where `MESSAGE` is the received message that contains the error, `C-PERF` is the communication performative beginning it, and `P-PERF` is the plan performative.

3.5.3 Pseudoagents

Currently, KQML is restricted to having one agent in each image. We have developed a protocol for allowing multiple *pseudoagents* in an image. Our protocol requires different pseudoagents to handle different plan performatives. When a distinction is required, we use "agent" to refer to a KQML agent, and "pseudoagent" to refer to agents inside one image that would be agents if KQML permitted multiple agents in an image.

For example, the planner image may have many pseudo-agents that respond to a given communication performative, e.g. Evaluate. We assume that each image has only one pseudo-agent for responding to a particular plan performative. For example, if the planner image has

a pseudoagent that uses Tachyon to respond to a :temporal-ok? plan performative, then all messages using the Evaluate and :temporal-ok? performatives will go to this pseudoagent. If either the communication or plan performative is different, then it is possible to direct the message to a different pseudoagent.

By assuming that each agent has only one pseudoagent that can handle a given plan performative, the message handlers can be coded to distribute messages based on their plan performatives. Without this assumption, we would have to register the pseudoagents within each agent to keep track of the pseudoagent currently being used for each request.

3.5.4 Reporting Errors

When there is an error or unexpected condition, the agent replying to a request adds an :error field to the message, whose value is generally a string explaining the problem, suitable for passing on to a user. Such error keywords can appear if three categories of messages:

1. replies with an Error communication performative, perhaps with the plan performative.
2. replies with a either an Error or Reply communication performative, and the normal plan performative.[4]
3. messages with the normal communication and plan performatives.

(1) is used for errors where the agent could not understand or start processing the request. An example is the unrecognized plan performative error described above.

The latter two are used when the request could be processed, but the processing failed or had an error after invocation. ((3) is used when the requestor is not waiting for a reply, and (2) is used when requestor is waiting.)

For example, if a meta planning-cell manager cannot recognize the syntax of a request, (1) would be used. If it recognizes the syntax, but cannot find the necessary planning cells, (2) would be used. If it started planning in a planning cell (causing a non-error reply to the initial request), but later fails to find a solution, (3) is used (with a :failed plan performative sent to the requestor). Such a meta planning-cell manager implemented in MPA is described in Section 5.1.

[4]This option is often identical to (1) if the Error communication performative is used.

The wrapper function `mpa:make-response-key` (see Appendix A) is generally used to construct a message for case (2) by specifying the :error keyword. For agents that control their own logging and tracing, the :log keyword should be specified as nil.[5]

4 Plan Servers

MPA can be assigned a planning task for which multiple plans may be developed depending on context, advice or other factors. For each alternative plan being developed, a cell of agents is committed to the planning process. The question arises of how to maintain and communicate the current plan state within a planning cell. Our architecture accomplishes this by having a *plan server* agent in each planning cell, although a single plan server agent can be shared by multiple planning cells. The plan server is a *passive* agent in that it responds to messages sent by other agents. It does not issue performatives to other agents on its own initiative.

The plan server provides the central repository for plan and plan-related information. A plan model based on the concepts of task, plan, and action network is described in Section 2.3.1. The plan server must be able to represent multiple tasks each with multiple alternative plans. This approach is useful when such alternatives share a lot of structure.

The plan server accepts incoming plan information from agents, performs necessary processing, and stores relevant information in its internal representation. This information is then accessible to any cell agent through queries. The plan server maintains all data structures associated with the evolving plan. Because a plan server must be able to stand alone, independent of the control structure or state of any planning agent, the information it contains must include not only the partially ordered actions of the plan, but also contexts, backtracking points, declarative information about the state of the plan (e.g., a list of flaws), and so forth.

A plan server should be able to record a history of all changes made to its data structures, including the name of the requesting agent. Queries could also be included in the history. Histories are useful for debugging, and this capability can be disabled for efficiency, if desired. A plan server should have declarative equivalents for its information that can be passed to other agents or written to files. Pedigree management (i.e., recording the source of plan derivation) is another desirable service.

[5] For no particular reason, our PRS-based agents were written to localize all reply construction and logging in the handler functions, so the MPA wrapper functions are only used there. The code for this agents either constructs error values explicitly or uses the function `prs::print-return-error`, which prints the error to the PRS interface, and returns the error value for later use in a call to `mpa:make-response`.

An MPA plan server is also responsible for storing relevant information about the task, plan, and the planning process, and notifying various agents of relevant planning events. These capabilities are implemented by *annotations* and *triggers*, which are each described in their own sections. It is also necessary to document the types of plan queries that can be serviced by the plan server.

An MPA plan server embodies the notion of different *views* of a given plan, a powerful feature of MPA not found in other architectures. A view constitutes some aspect of the plan that could be relevant to individual MPA agents. For example, the resource usage in a plan might be one view of the plan. Certain views are directly retrievable from the plan server (e.g., plain-text and graphical representations), while others must be computed by running algorithms over the basic plan representation (e.g., resource usage).

Views allow the plan server to provide better service to other agents. These agents can get information in forms they can use instead of having to include their own software to extract the desired information from the underlying representation (which can be excessively large). For example, the resource view of the plan might return a representation of the resources used without needing to return the entire plan structure from which the requesting agent would have to extract the resources.

Many other plan-server features are desirable, although not strictly necessary for plan generation. Our research is not concentrating on such features, which include providing persistent storage, controlling access to the plan or its parts, providing version control for plans, and developing graphical browsing capabilities.

In general, a large amount of data is static for a given domain, that is, it remains the same for each alternative plan. This information need not be stored in each plan server used in this domain. Instead, a knowledge server will handle requests for static knowledge (perhaps a plan server forwards such requests). For example, the knowledge server might keep dictionaries, the object hierarchy, legacy databases, the set of action descriptions, and customizations required by agents for use in this domain (e.g., what model of time to use).

In addition to describing triggers, annotations, and queries, this section describes a PRS-based plan server, referred to as the *Act Plan Server*, which we implemented for MPA. The Act Plan Server implements new MPA features, but has many shortcomings as an operational plan server.

Plan Performative	Communication Performatives
:annotation	Insert Delete Ask-All Ask-One
:trigger	Insert Delete Ask-All Ask-One
:update-task	Tell Delete
:query-task	Ask-All Ask-One
:update-plan	Tell Delete
:query-plan	Ask-All Ask-One
:query-node	Ask-All Ask-One
:ping	Evaluate
:pcd	Tell

Figure 3: Plan Server Plan and Communication Performatives

4.1 Communication and Plan Performatives

The currently allowed combinations of communication and plan performatives in messages handled by an MPA plan server are shown in Figure 3. Ask-One is the appropriate communication performative for plan queries when only one answer is desired, while Ask-All is used when all answers (i.e., a set of answers) are desired.

Annotations and triggers are enough like database objects that the plan server uses the database communication performatives for manipulating them. Updating and querying a plan is more complex than adding and deleting something from a database. Therefore, we have a richer plan performative language for manipulating plans. The performatives described here are an initial set that will be extended. The detailed syntax of these performatives is given in Section 9. Examples of the plan performatives are given in the following sections.

```
(ANNOTATION <basic-annotation>
   :task TASK :plan PLAN :action-network A-NETWORK)
(ANNOTATION <basic-annotation> :task TASK :plan PLAN)
(ANNOTATION <basic-annotation> :task TASK))
(ANNOTATION <basic-annotation>)
```

Figure 4: Variants of Annotations: these variants distinguish action-network-specific from plan-specific from task-specific from task-independent annotations.

4.2 Annotations

Annotations are declarations of high-level attributes of either a plan or the state of the planning process. Useful annotations for tasks, plans, and action-networks (*product annotations*) include flaws or problems to be repaired, plan quality information, pedigree (how and by whom parts of the plan were derived), and comparative relationships among alternative plans and plan fragments. Annotations related to the planning process (*process annotations*) could include time spent and the current state of development for a given plan (e.g., :ongoing, :completed, :failed).

Annotations can be posted to the plan server by any cell agent, including the plan server itself. Annotations are encoded as predicates and stored in the plan server's database. Product annotations are indexed by task/plan/action-network, thus allowing flexible retrieval (e.g., finding all plans or action networks that have a given annotation, or finding all annotations for a given plan, task or action network).

To distinguish action-network-specific from plan-specific from task-specific from task-independent annotations, four variants are used as shown in Figure 4. The syntax shown is used in query messages, but not in the returned answers, which omit the keywords. In Figure 4, `<basic-annotation>` corresponds to an arbitrary annotation predicate (which generally will not specify the plan to which it applies). In the Act Plan Server, the keywords in Figure 4 must be given in the order shown (inside an annotation form only).

The following are examples of basic annotations that might be used to coordinate PAs within a meta-PA:

(Time t) Posted at the completion of a action network, or planning level, to indicate the amount of time spent. Timing information could be used to modify the strategies used for planning. If the planning process is consuming too much time, the cell manager might switch to a more efficient strategy.

(Node-Expanded node) Posted when a goal has been completely expanded; a meta-PA might, for example, apply the scheduling critic after every expansion.

(Level-Complete) Posted when an action network, or planning level, has been completely expanded; a meta-PA might invoke the Critic-Manager.

(Plan-Complete) Posted when a final plan (action network) has been generated; a meta-PA might invoke the Critic-Manager in a different mode.

(Final-Plan) Indicates that the plan is final and has been approved by all critics.

The latter three annotations were used in our initial demonstration. Lower-level annotations are necessary when lower-level control of the planning process is desired. Details on the syntax for annotation-related performatives and example messages can be found in Section 9.

4.3 Triggers

A trigger is a form of event-driven rule whose function is to notify specified PAs of a designated plan server event. An individual trigger is activated by the occurrence of a designated event. For now, events are limited to the insertion of annotations in the plan server. Activation of a trigger results in the dispatch of a trigger-dependent message to a designated PA. The triggered message may simply inform the receiving agents of the triggering event, or request that some action be taken.

Triggers can be supplied by various sources. Certain of them may be built into the plan server, while others may be dynamically added and removed during the planning process. For instance, the cell manager may post triggers at various times to influence the overall planning process. Individual PAs may post and remove triggers to request notification of particular events. The operation of triggers are contained within a planning cell.

A trigger specification must include three fields, :event, :destination, and :msg. The :msg field specifies the message to be sent when a trigger is activated. The recipient of this message should be specified in the :destination field, and must be an *agent role*.

The :event field must be an annotation fact having one of the forms shown in Figure 4, except that each `<basic-annotation>` is replaced by an `<annotation-schema>`. In contrast to basic annotations, an annotation schema can contain variables. Variables may

appear in the :msg and :event specifications. The triggering event will bind the variables appropriately, so they can be used in the outgoing message. Details on the syntax for trigger-related performatives can be found in Section 9.10, including messages for deleting and querying triggers.

This use of annotations and triggers in MPA enables a variety of control modes, and enables the planning process to be controlled dynamically. For example, let's suppose temporal conflicts are arising frequently during the planning process, causing backtracking at the end of every level. The meta-PA with a planning cell can be aware of this by monitoring annotations posted to the plan server (triggers are useful for this). After noticing the temporal problems, the meta-PA decides to invoke the temporal reasoning agent after every node expansion, in order to catch temporal conflicts earlier and reduce backtracking. This change in control can be easily accomplished by the meta-PA inserting a trigger in the plan server that sends the appropriate message to the temporal reasoning agent whenever a Node-Expanded annotation is posted.

4.4 Queries

Queries to a plan server can be involved, and particularly complex and commonly used ones may want to be in their own agents to avoid overloading the plan server. Examples of queries to a plan server are:

- What are the legal bindings for a plan variable?
- What are all the unsolved goals?
- What resources are required by this plan?
- Who put this action in the plan?

The query language will be under continuous development as more PAs populate the architecture. A key feature of an MPA plan server is the generation of different views of the plan. The plan performatives and views currently supported for queries in the Act Plan Server are depicted in Figure 5. The query language could easily be extended — including all of the above queries is simply a matter of agreeing on a syntax for the query messages. Detailed explanations of each view, together with example queries and their responses are given in Section 9.8. If no view is given, plan queries assume :task-network as the default.

Plan Performative	Views	
for tasks:		
:query-task	:task-network	:assumptions
for plans:		
:query-plan	:task-network :ascii :monitors :parent	:subplans :ascii-filename :monitor-filename :available-resources
for action networks:		
:query-plan	:task-network :ascii :monitors :resource-allocations	:resource-constraints :ascii-filename :monitor-filename
:query-node	:predecessors :unrelated	:successors

Figure 5: Plan Performatives and Views for Queries

The :query-plan performative causes a particular view of the plan's entire action network to be retrieved from the plan server or computed from data stored in the plan server. For example, if :view is :resource-constraints, a function walks over the action network and collects the resource constraints in an interlingua. (The views that mention resources currently use an interlingua specific to the ACP domain.)

Similarly, the :query-task performative causes a particular view of the task's objectives to be retrieved. Tasks are currently represented as Acts. The :assumptions view returns a free text field. Additional views may be added later, such as retrieving the world situation.

A :query-plan message can apply to either a task, plan, or action network, while :query-node messages always apply to action networks. The special value :all can be used for any of those three keywords to find out which tasks, plans, and action networks are known to the plan server. When :all is used, a list of names is returned, and the :view keyword is ignored.

```
(:query-plan :task :all)
(:query-plan :task TASK :plan :all)
(:query-plan :task TASK :plan PLAN :action-network :all)
```

The first query returns a list of all known task names, the second returns a list of all known plan names for the given task, and the third returns a list of all known action-network names

for the given task and plan. The :query-task performative uses :all similarly for :task and :plan. Any value for :plan other than :all is ignored by :query-task.

In :query-plan, a query on plans is indicated by the :action-network keyword being omitted or having a value of :all or nil. When the value is nil or omitted, the views for plans shown in Figure 5 are applicable.

A query on an action network is indicated by the :action-network keyword being the name of a known action network, or the special value :last, which refers to the last action network defined for the given task and plan. In this case, the views for action networks shown in Figure 5 are applicable.

The substantive use of queries in the first-year demonstration is in the planner-scheduler interaction. In the TIE 97-1 demonstration, several agents queried the plan server for the plan and/or annotations, including the scheduler, the planner, the simulator, and the plan visualizer..

The first example is a task query that retrieves the ASCII Act representation of the objectives of the given task, and the second example extracts the resource constraints of the most recently expanded action network for the given task and plan.

```
(:query-task :task TASK :view :ascii)
(:query-plan :task TASK :plan PLAN :action-network :last
             :view :resource-constraints)
```

A message with the above content field invokes a plan server function that walks over the action network for the plan and collects and returns the necessary constraints as specified in Section 9.

4.5 Act Plan Server

We have implemented a specific plan server, named the Act Plan Server, which employs the Act formalism [18] for action representation. An Act is the basic structure used to represent a plan or action network in the Act formalism. Acts can be expressed either in a format with embedded graphical information or in plain-text format (to facilitate translation to other languages). The Act-Editor system [9, 19] supports the graphical creation, manipulation, browsing and display of Acts, thus serving as a graphical knowledge editor for systems that use Act.

The Act Plan Server is implemented as a PRS agent. PRS is responsible for maintaining annotations, handling and distributing incoming messages, and executing triggers appropriately. PRS was chosen as the basis for implementing the Act Plan Server because of the need to combine both declarative and procedural encodings of knowledge, as well as to support a mixture of event- and goal-driven processing as required for the maintenance of annotations, the handling and distributing of incoming messages, and the execution of triggers.

As with any PRS agent, the Act Plan Server includes a database for representing declarative knowledge, a set of Acts that encode procedural knowledge about how to accomplish goals, and LISP functions that implement the basic activities for the agent. The database storage and reasoning capabilities are used extensively to track the plan-related information sent to the Act Plan Server. The procedural knowledge capabilities are used for managing the activation of triggers. In addition, PRS uses knowledge encoded in the Act formalism, MPA's common plan representation, and is built on the Act-Editor which can display plans graphically.

4.5.1 Limitations of the Act Plan Server

Several other architectures, including the AITS, use plan servers. Often such plan servers are required to provide (among other features) persistence, access control, versioning, and browsing capabilities. While we view all these features as desirable in an MPA plan server, it is beyond the scope of MPA to implement a plan server with all of these properties.

Rather, we have implemented an Act Plan Server with several features not in other plan servers. MPA uses these new features (e.g., views, triggers and annotations) to control complex planning processes. The goal of MPA is to develop a new, more powerful architecture, and our Act Plan Server is a step in that direction. We view it as a straightforward effort to build the Act Plan Server on top of a plan server or database system that will provide the above AITS plan server features, although the task may require a significant amount of effort.

The Act Plan Server stores a plan as a series of action networks that together form one alternative plan for a given task. Action networks are stored as Acts (with links to other action networks in the same plan), and the only persistence provided is that Acts can be saved to files in their entirety. Alternative plans can be stored for the same task. No access control or versioning is provided (other than the ability to store alternative plans for the same task), and the only browsing capabilities are those provided by SRI's Act-Editor.

The Act Plan Server supports a reasonably broad set of queries, at the level of tasks, plans, and action networks. For example, queries can be used to extract the set of known tasks,

the set of plans for a given task, and the set of action networks for a given task or plan. Annotations and triggers can be queried with a fairly general query language. Seven different views on plans and action networks, and three different views on individual action-network nodes (actions) can be obtained through queries. The Act Plan Server could easily support more queries as needed, including access to individual plan and action-network components. The current MPA project will not transition the Act Plan Server to an existing plan server or database system, nor will it provide additional persistence, access control, versioning, and browsing capabilities.

4.5.2 Act Plan Server Interactions

The user can interact directly with the Act Plan Server through the PRS interface by posting appropriate goals and adding or retracting information from the Act Plan Server database. In addition, an MPA-specific **Plan Server** menu has been added to the PRS interface to enable simpler interactions with the Act Plan Server. The menu contains the following commands; we expect to extend this menu as the sophistication of the Act Plan Server increases.

Load Create the Act Plan Server agent, register it with the name server, create its trace windows, and initialize it with all relevant database entries, Acts, and functions.

Reset Return the Act Plan Server's database to its original state. All information about tasks, plans, and action networks is deleted, as are all annotations and triggers.

Show Triggers Display the current set of triggers.

Show Annotations Display the current set of annotations.

4.5.3 Use of the Act Plan Server in TIEs

SRI will provide the MPA Act Plan Server running in Allegro COMMON LISP, which will store plans as Acts, and support annotations and triggers. SRI will provide documentation and examples for writing code to extract information from Acts, and will assist those writing such code. Examples will include code that extracts the resources plan view from Acts in the MPA demonstration, and the code that translates Acts to the planner's representation. There are several issues:

Annotations SRI has already specified annotations that are used by the agents in its initial demonstration and in TIE 97-1. It is the responsibility of the producers and consumers of any new annotations to agree on the syntax and semantics of the new annotations.

Knowledge Server In general, a large amount of data is static for a given domain, that is, it remains the same for each alternative plan. In the MPA design, this information is not stored in each plan server used in the domain. Instead, a knowledge server agent handles requests for static knowledge. We have not yet implemented a separate knowledge server, as the planner currently stores the static data for our domains.

Alternative Representations To support alternative representations, the Act Plan Server could be extended to store plans that are arbitrary LISP expressions. Those responsible for this representation must provide LISP code to compute every query and view that is supported over this representation.

4.6 Future Issues for the Plan Server

Several important issues for future consideration were raised during the implementation of SRI's Act Plan Server:

- Efficiency of firing triggers (the assumption is that there will be relatively few triggers, efficiently indexed by annotations like Node-expanded and Level-expanded)
- Authorization (who can trigger what actions, and make what changes)
- Responsibility for managing annotations and triggers, keeping them up to date, and updating features when needed
- Synchronization of distributed activities and potentially inconsistent updates and requests
- Merging plans from local plan servers

A major problem as we move toward agents executing in parallel is to make sure changes of one agent do not interfere with other agents. (We may not have the resources to address this in the MPA project, but other SRI projects are interested.) One approach is to collect relevant plan changes without making the changes, and then, at some point, adding the combined changes. This would require some way of checking the consistency of constraints that came from different agents, a very difficult problem. A variation on this would be to "lock" part of the action network so certain agents could change certain parts of the action network, but other agents could count on the locked plan to not be corrupted. These techniques are part of our eventual vision, but are beyond the scope of the current MPA project.

5 Planning-Cell Managers

A cell manager is a persistent agent that can continuously accept tasks from other agents (human or software), decompose those tasks into subtasks to be performed by its cell agents, and then combine the results into solutions.

A planning cell can operate as a stand-alone problem-solving unit. Additionally, sets of planning cells can be aggregated into larger cells, which are in turn controlled by a *meta planning-cell manager*, which distributes and coordinates planning tasks among planning cells. This manager is given a set of tasks to solve and a set of planning cells as resources, and is responsible for overseeing the entire planning process. Different management schemes are appropriate for different applications. We have focused on generating multiple plans in parallel, using a pair of planning cells controlled by a meta planning-cell manager.

Here, we describe implementations of both a baselevel cell manager (the PCM), and a meta planning-cell manager (the meta-PCM). Their designs serve as templates for additional types of planning cells. In the long term, MPA will contain a library of such templates which users can adapt as appropriate for their applications.

We have implemented one particular cell manager, the PCM, with a small number of different planning styles. Instances of PCM are used to control each of several planning cells. In addition, we have implemented a particular instance of a meta planning-cell manager, the meta-PCM. These agents are described in more detail in Sections 5.1 and 5.2.

Both the meta-PCM and the PCM are implemented as PRS agents. As with any PRS agent, each agent includes a database for representing declarative knowledge, a set of Acts that encode procedural knowledge about how to accomplish goals, and LISP functions that implement the basic activities for the agent, such as sending an MPA message to another cell agent. SRI has made the meta-PCM and PCM Acts available to the ARPI community.

PRS provides several capabilities that make it a good framework for constructing such managers. Because cell managers direct the activities of multiple agents, they must be capable of smoothly interleaving responses to external requests with internal goal-driven activities. The uniform processing of goal- and event-directed behavior within PRS is ideal for supporting such behavior. PRS supports parallel processing within an agent, thus enabling multiple lines of activity to be pursued at any given time. The database facility within PRS enables declarative encodings of key characteristics, making them easy to access and modify. The Act language, used to represent procedural knowledge within PRS, provides a rich set of goal

Plan Performative	Communication Performative
:solution :failed	Tell
:multiple-solve :define-advice :new-agent :ping	Evaluate

Figure 6: Meta-PCM Plan and Communication Performatives

modalities for encoding activity, including notions of achievement, maintenance, testing, conclusion, and waiting. Finally, the extensive textual and graphical tracing in PRS provides valuable runtime insights into the operation of cell managers.

5.1 Meta Planning-Cell Manager

The meta-PCM is a persistent agent that can continuously accept planning requests and generate multiple plans for one or more tasks, given a set of planning cells as resources. Tasks can be specified by either a human user or another planning agent.

The meta-PCM accepts an asynchronous stream of planning request messages (from multiple agents), each of which can request that multiple plans be produced. For each requested plan, the meta-PCM will find a free planning cell if one exists. A planning cell can refuse a task if it is already occupied (our PCM agent will always do so). The meta-PCM distributes tasks to planning cells by sending messages to the cell manager of each cell. These messages take the form of planning tasks and advice about how to solve the task. The meta-PCM may distribute only a subset of the requests if planning cells are busy.

The meta-PCM also responds to any messages from planning cells that describe a solution or a failure for a planning request. (These are sent by cell managers immediately upon finding a solution or detecting a failure.) Such responses can be done simultaneously with the handling of new requests. The plans or failures are "forwarded" to the correct requestor by sending an MPA message. The meta-PCM also notices when all requests from a given incoming message have been serviced, and prints a summary of the solutions in its trace window. If desired, it could also send a summary message to the requestor or invoke a plan comparison agent.

The currently allowed combinations of communication and plan performatives in the messages handled by the meta-PCM are shown in Figure 6. Multiple-solve messages are the

incoming planning requests. Define-advice messages define new pieces of advice and are forwarded to all running planning-cell managers. Solution and Failed messages report results from planning cells. A New-agent message can be used to declare a new planning cell. Section 7.1 explains the role of the meta-PCM during our demonstration.

The current meta-PCM has several limitations, such as not maintaining queues of requests, and could easily be extended. The following features and limitations give a better idea of the capability of the meta-PCM:

- If there are more requests in one message than the total number of planning cells, then immediately send back an error saying there are not that many planning cells (without processing any requests).
- Otherwise, accept the request and generate a name for it. Send this name back to the requestor who can use it to match solution/failure messages received to requests.
- If all planning cells are found to be busy, send a message back to the requestor indicating the request is being refused because of busy planning cells.
- If some, but not all, requests are assigned to free planning cells, send a message back to the requestor indicating that only a subset of the requests will be serviced.
- For each solution or failure received, immediately forward the message to the correct requestor.
- When all accepted requests for a given message have been serviced, remove all information about that request from the database.

The user can interact directly with the meta-PCM through the PRS interface by posting appropriate goals and adding or retracting information from the plan server. In addition, an MPA-specific Meta-PCM menu has been added to the PRS interface to enable simpler interactions with the meta-PCM. The menu contains the following commands; we expect to extend this menu as the sophistication of the meta-PCM increases.

Load Create the meta-PCM agent, register it with the name server, create its trace windows, and initialize it with all relevant database entries, Acts, and functions.

Reset Return the meta-PCM's database to its original state. Resetting should not be necessary except in unforeseen error conditions.

Plan Initiate planning on a user-specified task by an available planning cell. A window appears in which the user enters the name of the task to be solved; entering :none (or NIL) aborts the request. The user is then asked for the number of alternative plans desired.

Plan Performative	Communication Performative
:annotation :advice :pcd	Tell
:solve :define-advice :ping	Evaluate

Figure 7: PCM Plan and Communication Performatives

5.2 Overview of the PCM

The PCM is a persistent agent that can continuously solve planning tasks. Tasks can be specified by either a human user or another planning agent, such as the meta-PCM. Planning requests are serviced sequentially rather than concurrently; thus, the PCM can solve multiple tasks but only one at any given time. If the PCM gets a request to generate a plan while it is already busy, it replies with a message indicating that it is busy. Concurrent plan generation is accomplished by the meta-PCM, which can invoke multiple PCM agents controlling multiple planning cells.

A cell manager is responsible for overseeing the entire planning process for a given task. One key responsibility is the management of the PCD. The agents with assigned roles in the PCD are referred to as the *cell agents.* In addition, the cell manager must determine the appropriate problem-solving strategy to apply to a given task, and manage the distribution of subtasks to agents within the cell. Finally, the cell manager must coordinate all results and activities of the cell agents.

Section 7.1 describes the different strategies used by the PCM for monitoring and controlling the planning process during our initial demonstration. The PCM reconfigured the planning cell during planning and demonstrated dynamic strategy adaptation.

The currently allowed combinations of communication and plan performatives in messages handled by the PCM are shown in Figure 7. Annotation messages advise the PCM of annotations that have been posted in the plan server. Such messages are sent by triggers posted by the PCM itself. Advice messages select (predefined) problem-solving advice [8] to be used by plan generation agents within the cell; the PCM will pass along specified advice to appropriate agents when making planning requests. Define-advice messages are forwarded to appropriate cell agents when new advice is being defined. Solve messages request the PCM to

PCD Role	Agent Names
:manager	pcm
:plan-server	plan-server
:planner :search-manager :critic-manager	sipe
:scheduler	opis
:temporal-reasoner	opis, tachyon
:executor	executor
:requestor	meta-pcm

Figure 8: Plan Cell Descriptor Roles and Their Possible Fillers

generate a plan, and Ping and PCD messages are handled by all agents. Sections 9.1 and 9.2 describe the message traffic of the PCM in more detail.

5.3 Planning Cell Characteristics

The PCM's planning cell is characterized by two key elements: the PCD, and the *plan style* to be used for problem solving. The PCD is encoded declaratively within the PRS database of the PCM. In particular, the PCD is composed of several entries of the form

```
(cell-agent <role> <agent>)
```

These facts declare both the active *roles* in the cell, and the *agents* that fill them. Each agent name in a PCD should be the registered name for the agent.

The roles in the current PCM and their possible agent fillers are listed in Figure 8. Each role is filled by zero, one, or a set of agents, depending on the nature of the problem-solving process. Within the current PCM, the :scheduler and :temporal-reasoner roles are optional but all others are required. Because of the KQML restriction of one agent per image (and in some cases for efficiency), the "sipe" KQML agent services messages for several pseudoagents in the initial demonstration. These pseudoagents include the Search Manager, Critic Manager, Schedule Critic, and Temporal Critic.

Planning may commence either in response to a request by another agent, the meta-PCM in our demo, or directly from the user. When planning commences, the PCM first determines whether the cell agents are alive and ready to receive messages. After verifying the readiness of the cell agents, the PCM distributes a copy of the PCD to each. Each cell agent consults

its local copy of the PCD to ascertain where to send intracell messages. The PCD can be reconfigured dynamically during the planning process. When such reconfigurations occur, updated versions of the PCD are sent to all cell agents.

Plan-style declarations are represented using the predicate `(use-plan-style <method>)`. This predicate is single-valued; that is, only one style can be active at one time. The PCM database explicitly manages the single-valuedness; thus, to change planning styles, simply assert a use-plan-style fact into the PCM's database.

5.3.1 Planning Styles

The PCM supports a small number of planning styles, all of which assume a level-by-level plan generation model, derived from the hierarchical task network (HTN) approach to planning [3]. For each level, a more refined plan is first generated, then critiqued. This model may not apply to all planners, but is reasonably general. In particular, nothing is assumed about what comprises a level, thus enabling a range of different level refinement methods (for example, expansion of a single goal or all goals).

A planner can expand a plan to some arbitrary extent, as appropriate to its planning technology. All that is required is that the plan critics can be applied after each such expansion. If critic roles are left unfilled, critiquing is skipped. In addition to HTN planners, causal-link planners [13] fit naturally into this scheme, with goal selection, operator selection, and subgoal generation viewed as forms of plan expansion. Causal link protection and checking constraint consistency correspond to plan critiquing.

The PCM planning styles vary in their choice of agent to perform the plan refinement, the selection of critic agents for the critique phase, and the frequency of critic invocation. Currently, the PCM supports two plan styles: one where the Planner is responsible for critiques, and one where the PCM is responsible. The PCM takes different actions based on the information returned by the Planner about the status of the expansion process.

Planner-Expand-and-Critique This method performs a level-by-level development of the plan, where each level is generating by sending an :expand-plan-and-critique performative to the Planner. Thus, at each level, the Planner both expands the plan and performs any necessary critiques. The role of the PCM is simply to oversee the search process. The following keywords can be returned by the Planner:

:final-plan The returned action network is a complete and validated solution. Receipt of such a response terminates the PCM planning operations for the current task.

:level-complete The returned action network is a successful refinement of the previous level's action network. The PCM continues with this new level.

:no-expansion No expansion was found for this level. The PCM initiates backtracking by setting expansion parameters and then reinvoking the Planner.

:plan-failure No plan was found and no backtracking is possible. The PCM abandons the task.

PCM-Expand-and-Critique This method performs a level-by-level development of the plan. Each level is generating by sending an :expand-plan performative to the Planner, but the resulting action network is not critiqued by the planner. The PCM follows this plan expansion phase with a separate plan critique phase under PCM control. Thus, the PCM explicitly manages both plan expansion and critiquing. This method is the default and has been used in all MPA demonstrations, including TIE 97-1.

In response to an :expand-plan performative, the Planner can return the latter three keywords listed above (and the PCM response is the same). However, the :final-plan keyword can never be returned, as critiquing has not been performed by the Planner. Instead, the following keyword may be returned:

:plan-complete The returned action network is complete but still needs to be verified by the appropriate critics.

The PCM-controlled critique phase involves the application of all critics that are currently *declared* for the Plan Cell. Critic declarations are encoded in the PCM database as facts of the form: `(cell-critic <critic-performative> <role>)`. Each declaration defines a critic performative and the cell role that is responsible for managing the corresponding critic. A critic performative is simply a plan performative that is handled by an agent that is considered to be a plan critic. The PCM sends a message to invoke each declared critic. Such messages use the Evaluate communication performative, and the content is:

```
(P-PERF :task TASK :plan PLAN :action-network A-NETWORK)
```

Currently, only the critic performatives :plan-ok?, :temporal-ok?, and :schedule-ok? are supported by the PCM, all of which are managed by the Planner. The critic returns a value indicating whether its critique revealed any problems. Any such problem causes the PCM to backtrack or terminate prematurely.

5.3.2 Database

Additional predicates are used in the PCM database to control cell activities. Currently, the only predicate of significance to users is

`(use-params <plan-perf> <param> <value>)`

which controls the inclusion of optional keyword arguments for plan performatives. These parameters control low-level processing instructions for the cell agents that service the specified plan performative (for example, whether to check phantoms when doing a plan expansion). When a performative is to be sent, the PCM checks its database for any declarations of keyword arguments for the performative and includes those that it finds in the content of the message sent.

5.4 PCM Invocation

The user can invoke the PCM through the PRS interface by posting appropriate goals and adding or retracting facts from the PCM's database. This is made possible by the declarative approach used to encode relevant PCM characteristics. In addition, an MPA-specific Plan-Cell menu has been added to the PRS interface for interactive control of the PCM. The menu contains the following commands; we expect to extend this menu significantly as the sophistication of the PCM increases.

Load Create the PCM agent, register it with the name server, create its trace windows, and initialize it with all relevant database entries, Acts, and functions.

Reset Return the PCM's database to its original state. Resetting should not be necessary except in unforeseen error conditions. Resetting the PCM does not change the state of any other PA in the cell.

Define Cell Using a menu, the user can interactively specify the composition of the planning cell and relevant cell parameters.

Plan Initiate planning on a user-specified task. A window appears in which the user enters the name of the task to be solved; entering :none (or NIL) aborts the request.

Plan-with-CTEM For ACP planning domains only, initiates planning on a user-specified task using an Act for three-phase planning: pre-CTEM planning, a CTEM run, and post-CTEM planning of support missions.

Alternatively the operation of the PCM can be controlled by another agent. We have implemented a meta-PCM agent which can coordinate the operation of multiple planning cells to produce multiple plans simultaneously. The meta-PCM, described in Section 5.1, distributes tasks to planning cells by sending messages to the cell manager of each cell, and handles messages about completed or failed plans.

5.5 Use of The PCM in TIEs

SRI wrote Acts for its PCM to incorporate the demonstration flow of TIE 97-1. For agents not already part of MPA, it is the responsibility of the agent implementor to determine the correct performatives and message syntax for invoking the agent, write handlers for such messages, and determine the syntax for any messages returned or sent (e.g., to the plan server) by the agent. The agent implementor must also document the I/O specifications for each agent, including any side effects on the plan server, and ensure that all users and consumers of the services provided by this agent are aware of (and agree to) the protocols and documentation. SRI may require certain messages to be sent by agents to the PCM at certain times.

SRI has made the meta-PCM and PCM Acts available to the ARPI community. The PCM does not currently support the user interacting with an agenda, and SRI does not plan to implement a GUI for such a purpose under its current contract.

6 Integrating Planning, Scheduling and Execution

The previous sections described agents that were written specifically for MPA, the meta-PCM, the PCM, and the Act Plan Server. Here we describe the modularization of legacy software systems into MPA agents. We first describe the use of executor and plan manager agents for continuous planning and execution, and then the use of an existing planner and an existing scheduler in MPA.

6.1 Continuous Planning and Execution

If plans are to be executed in dynamic environments, there must be agents able to deal with unpredictable changes in the world. As such, agents must be able to react to unanticipated

events by taking appropriate actions in a timely manner, while continuing activities that support current goals. Agents must have the ability to recover from failures by adapting their activities to the new situation, and/or repairing the plan. We will refer to an agent with these capabilites as an *executor* agent, because the MPA protocols described here support the role of an agent executing a plan and doing plan repair. (As described below, the executor capabilites are only some of the many desireable functionalities.)

Two different demonstrations show the use of MPA for continuous plan repair. The first demonstration is a distributed, multiagent version of SRI's Cypress system in the domain of joint military operations planning [17]. The second demonstration is a distributed, multiagent version of SRI's CPEF system (Continuous Planning and Execution Framework),[6] in the domain of air campaign planning (using an extended version of the TIE 97-1 planning knowledge base, but a different scenario).

The executor is always active, constantly monitoring the world for goals to be achieved or events that require immediate action. In accord with its current beliefs and goals, the executor takes actions in response to these goals and events. Appropriate responses include applying standard operating procedures, invoking the planner to produce a new plan for achieving a goal, or requesting that the planner modify the plan being executed. The planner plans only to a certain level of detail, with the executor taking that plan and expanding it at run time by applying appropriate library actions at lower levels of abstraction.

CPEF significantly extends the ability of Cypress to support the continuous development, monitoring, and adaptation of plans. CPEF has a Plan Manager agent that controls the overall life cycle of a plan, spanning plan generation, plan repairs, and plan execution. Thus, the Plan Manager performs the duties of the executor agent mentioned above, among other things. The executor agent could be a separate agent that is invoked by the Plan Manager. The creation and management of *monitors* is crucial to plan management. CPEF defines a monitor to be an event-response rule for which detection of the specified event leads to execution of the corresponding response. The MPA protocols support the planner automatically generating monitors appropriate to a specific plan, and sending these monitors to the Plan Manager.

[6]CPEF development was supported by DARPA Contract No. F30602-97-C-0067 as part of the JFACC program.

Plan Performative	Communication Performative
:install :execute :ping	Evaluate
:pcd :solution :failed	Tell

Figure 9: Executor Plan and Communication Performatives

6.1.1 Executor Agent

The MPA protocols described here support the role of an agent executing a plan, possibly doing plan repair. This is one of the many functionalities that a plan manager might provide (e.g., a plan manager might control planning agents in a manner similar to the PCM.) The currently allowed combinations of communication and plan performatives in messages handled by the executor are shown in Figure 9.

An Install message tells the executor to retrieve the plan from the plan server and get it ready for possible execution. Installation may involve creating and/or installing monitors. An Execute message tells the executor to being plan execution. Solution and Failed messages will be received after Revise or Solve requests are sent to the planner. The executor may also send Create-monitors requests to the planner.

6.2 Planning and Scheduling

One objective of the MPA project is to decompose and integrate planning and scheduling capabilities within the MPA architecture. Work on integrating planning and scheduling has proceeded via an SRI subcontract with CMU under the direction of Dr. Steve Smith, aimed specifically at adapting and integrating scheduling functionality contained in CMU's OPIS scheduler [12].[7]

The interaction of the planner and scheduler provides a good example of multiple agents cooperatively solving a problem in MPA, so is described in some detail (Section 6.2.3) after first describing the planner and scheduler agents. For more detail, Section 7.4 describes the actual messages sent during one of our demonstrations. The use of the scheduler is tied closely

[7] Steve Smith and Marcel Becker contributed to this section.

to the ACP domain used in both IFD-4 and TIE 97-1 — we describe only the limited use of scheduling currently in MPA.

One of the primary shortcomings of IFD-4 is the inability of the planner to do a capacity analysis early in the planning process. For example, when there are 75 F-16s and the plan requires 83 F-16s, the "capacity" of our F-16 assets is inadequate. Part of the MPA decomposition of the CMU scheduler includes a Scheduler agent for capacity analysis. We have implemented such an agent and developed its constraint and capacity models. Once we converged on definitions of resource capacity and relevant resource utilization constraints, we were able to adapt CMU's capacity analysis knowledge source straightforwardly. We believe that our demonstration showed significant value added to IFD-4 stemming from the resource utilization provided by CMU's scheduler.

6.2.1 Planner Agents: Search Manager and Critic Manager

Currently, the specification for the planner agent assumes a level-by-level plan generation process with a critique of the plan after each expansion, as described in Section 5.3. The planner can expand the plan at each level to some arbitrary extent, as appropriate to the planning technology being used. The critique can post annotations to the plan server and can cause a failure of the planning process if unresolvable conflicts are found. The expansion phase is accomplished by the Search Manager agent, while the critique of the plan is accomplished by the Critic Manager agent. Currently, these two agents are pseudoagents within a single planner agent, although we expect future planning cells will have them as separate agents.

The currently allowed combinations of communication and plan performatives in messages handled by the Search Manager are shown in Figure 10. The first group of Evaluate messages are used to generate or modify plans. An Init-problem message translates a given task to the planner's representation, and initializes the planner for planning (:start-problem is a synonym for :init-problem). An Expand-plan message causes the plan to be expanded to the next level, while an Expand-plan-and-critique message causes the plan to be expanded and also calls the plan critics on the plan. A Generate-plan message causes all levels of a plan to be generated for a given task (using the planner as the only plan-generation agent and bypassing PCM control). A Revise message causes an existing plan to be modified as required by a set of new facts given in the message.

The second group of Evaluate messages provide support functions, such as drawing plans, and updating and querying information. Define-advice messages specify new advice defi-

Plan Performative	Communication Performative
:init-problem :expand-plan :expand-plan-and-critique :generate-plan :revise	Evaluate
:define-advice :draw-plan :create-monitors :reset-problems :reset-domain :ping	Evaluate
:pcd :advice	Tell
:query-advice	Ask-All

Figure 10: Search Manager Plan and Communication Performatives

nitions. A Draw-plan message causes the given plan to be drawn in the planner's GUI. A Create-monitors message will create a set of monitors for the given plan (see Section 6.1).

The next two Evaluate plan performatives are not normally used but can be used to reset the agent after unexpected errors. A Reset-problems message causes the planner to redefine its predefined set of named problems. A Reset-domain message causes the planner to delete its entire knowledge base about the current planning domain and reload it The Ping and PCD messages are handled by all agents.

The Advice message tell the planner to activate and/or deactivate already defined pieces of advice, and the Query-advice message asks the planner to return the names and descriptions of defined and/or active advice.

The currently allowed combinations of communication and plan performatives in messages handled by the Critic Manager are shown in Figure 11. A Plan-ok? message causes the planner to apply all critics known to it. A Schedule-ok? message causes the agent to invoke the scheduler agent of the planning cell. A Temporal-ok? message causes the agent to invoke the temporal-reasoner agent of the planning cell. A Schedule-ctem? message is specific to the ACP planning domain and causes the planner to translate the current plan to CTEM, run CTEM, and translate the results back. The Ping and PCD messages are handled by all agents.

The domain-independent SIPE–2 planning system [14, 15] and the Advisable Planner [8]

Plan Performative	Communication Performative
:plan-ok? :schedule-ok? :temporal-ok? :schedule-ctem? :ping	Evaluate
:pcd	Tell

Figure 11: Critic Manager Plan and Communication Performatives

have been used as the basis for the planner agents in all our demonstrations. SIPE–2 has a precise notion of a planning level, and plan critics that fit naturally into the above scheme. To serve as an MPA planner agent, SIPE–2 had to be modularized, separating out its search control algorithm into the Search Manager pseudoagent, its plan critics algorithm into the Critic Manager pseudoagent, and its temporal reasoning critic into the Temporal Critic pseudoagent, which was extended to use any temporal reasoner in the planning cell. In addition, a new critic was created using the scheduler agent to do capacity analysis and resource allocation. The Schedule Critic pseudoagent was written to interact with whatever scheduler agent is in the planning cell.

6.2.2 Scheduler Agent

The use of the scheduler in our current implementation is tied closely to the ACP domain. This domain has been modeled within OPIS to produce a scheduling agent for the planning cell. This agent currently provides two types of service to support the planning process:

- Capacity analysis - this service provides a profile of the resource demand for airframe capacity of different types at various air bases over time, and identifies those periods where the "availability level" of allocated resources indicates that available assets are oversubscribed.
- Resource allocation - this service commits specific assets (i.e., airframe capacity of a particular type from a particular airbase) to specific missions and hence declares these resources unavailable for any subsequent missions that might be planned.

The currently allowed combinations of communication and plan performatives in messages handled by the scheduler are shown in Figure 12. Available-resources and Resource-

Plan Performative	Communication Performative
:available-resources :resource-constraints	Ask-all
:allocate-resources :ping	Evaluate
:pcd :solution	Tell

Figure 12: Scheduler Plan and Communication Performatives

constraints messages are simply "forwarded" on to the plan server using the appropriate view of the given plan, and the plan server's answer is returned. The primary invocation of the scheduler is done with an Allocate-resources message which causes the scheduler to analyze capacity and allocate resources. The OPIS agent accepts Solution messages, but does nothing with them, and Ping and PCD messages are handled by all agents.

From a scheduling perspective, a mission (or plan node) generated by the planner in developing an air campaign plan represents a request for a specific type and quantity of resource capacity and, at the same time, imposes some constraints and conditions on the utilization of this capacity. At the level of detail currently considered, the resource capacity required by a given mission corresponds to some number of airframes of a given type. Specific numbers of airframes of various types and availability levels are preassigned to different air bases in the theater. Thus, air bases can be seen as providing airframe (or more specifically sortie) "capacity" to support missions, and the scheduling problem is to allocate the airframe capacity available at specific bases to the missions in the plan. A given mission has a specific target (location), and it is desirable to support missions from nearby air bases. Hence, all air bases with available resource capacity are not equally preferable.

In theory, the two services provided by this agent could be performed at different levels of detail and precision, and typically this is the case. Conceptually, capacity analysis is usually seen as a means of providing look-ahead guidance to a more detailed resource allocation process. Currently, both capabilities are defined from the same core scheduling procedure and both are based on the abstract model of air frame capacity we have outlined. This agent significantly generalizes and improves on the simple conjunctive asset allocation decision procedure implemented originally in IFD-4, and, even at this high level of modeling abstraction, provides a more flexible framework for balancing conflicting preferences.

6.2.3 Planner-Scheduler Interaction in the Demonstration

The planner and scheduler agents in the current version of MPA are both implemented as LISP agents. After receiving its first request to allocate resources, the scheduler queries the Act Plan Server for the available resources in the scenario (given by airbase, airframe type, arrival day, number of sorties available, and availability level). The same available resources are used for future requests, unless the optional argument `:load-resources t` appears in the :allocate-resources message.

The availability levels for resources are as follows:

- (Deployed dayX) means that the resource will definitely be there on day X. Generally, this means that the the resource is already at its base at planning time.
- (En-Route dayX) means resource is scheduled to arrive and be available on day X, but is not at its base at planning time.
- (Requested dayX) means the planner has requested an additional type/number of resources, but resources have not yet been identified to fulfill the request, and perhaps the request has not even been acknowledged.
- (Hypothetical dayX) means the planner is supplementing available resources as part of a "what-if" analysis.
- (Overrun dayX) is an auxiliary availability level used to count overallocations. If no resources from the above availability levels are found, the scheduler starts "assigning" Overrun resources.

During the planning process, the planner periodically requests that the scheduler perform a capacity analysis and resource allocation of the resources in the plan. The scheduler is invoked by a message with the Evaluate communication performative, the :allocate-resources plan performative, and keyword arguments for the task, plan, and action network. The scheduler then queries the Act Plan Server for the :resource-constraints view of the named plan, and tries to allocate the resources required by the plan. Finally, the scheduler responds to the :allocate-resources request with either:

```
    (:allocate-resources :schedule-ok :task TASK :plan PLAN
          :action-network A-NETWORK)
or  (:allocate-resources :schedule-failure)
```

If scheduling was successful, the scheduler also posts several annotations and a :resource-allocations view of the plan's action network to the Plan Server.

The :resource-constraints plan view specifies all the resources that are used by missions plan's action network. The syntax is

```
(:nodes
 ((:node-id NAME
   :latitude LAT
   :longitude LON
   :resource-requirements
     ((MISSION-CATEGORY ((AIRFRAME SORTIES BASE AVAILABILITY)
                         ...))  ...)
   :start-time value  ;mission start day
   :end-time value)     ;mission end day + 1
  ...)
 :temporal-constraints (...))
```

The :resource-requirements slot contains a list of alternative airframe/quantity/base triples in order of preference for each mission-category. The base is optional. If given, it provides a preference for where the airframes should be based; otherwise, the nearest base with available resources is assigned.

The annotations sent by the scheduler include resource-profile, resources-overutilized, and resources-near-capacity. Resources are considered to be overutilized if they include Requested, Hypothetical, or Overrun resources. Resources are near capacity if they include En-Route resources, or more than 80 percent of the Deployed resources available at a given base. These annotations are used by the meta-PA to change the behavior of the planner. If the refueling resources are overutilized or near capacity, the meta-PA tells the planner to use the advice :fuel-low in its planning process. This in turn causes the planner to prefer operators that have lower fueling requirements.

The resource-profile annotation gives an overview of the resource utilization in the overall plan, specifying the utilization by airframe, day, supply (number of sorties available), demand (number of sorties required), and the plan nodes that are involved in each demand. The syntax of this profile is

```
((AIRFRAME1
   (:day DAY :supply SUPPLY :demand DEMAND :nodes NODE-LIST)
   (:day DAY :supply SUPPLY :demand DEMAND :nodes NODE-LIST) ...)
 (AIRFRAME2 ...) ...)
```

The :resource-allocations plan view is a subset of the :resource-constraints plan view and provides resource allocations. In the final plan, these resource allocations are used to bind the resource variables. The minimal format for :resource-allocations is

```
(:nodes
 ((:node-id NAME
   :resource-requirements
    ((MISSION-CATEGORY (AIRFRAME SORTIES BASE AVAILABILITY))))))
```

A more detailed description of the messages used is given in Section 7.4.

7 Single Planning Cell Configuration

We use the term *configuration* to refer to a particular organization of MPA agents and problem-solving strategies. Here, we describe a single-cell MPA configuration for generating individual solutions to a planning task, and the next section describes multiple-cell configurations for generating alternative solutions in parallel.

The initial demonstration was given in September 1996, using the configuration depicted in Figure 13. The demonstration showed a multiagent planner and scheduler, together with a temporal reasoning agent, accomplishing planning/scheduling in the Air Campaign Planning domain. To demonstrate the capabilities of MPA, we showed multiple asynchronous agents cooperatively generating a plan, the cell manager reconfiguring the planning cell during planning, agents implemented in different programming languages, and agents running on different machines both locally and over the Internet. The June 1997 demonstration, described in Section 8, includes multiple instances of the planning cell from the 1996 demonstration, coordinated by a meta planning-cell manager, producing multiple alternative plans in parallel.

The PCM is a meta-PA that controls the entire process, including initialization of the planning cell, and specification of a Planning-Cell Descriptor. Planning agents include GUI/Advice-Manager (not implemented until TIE 97-1), Act Plan Server, Search Manager, Critic Manager, Schedule Critic, Temporal Critic, Scheduler, and Temporal Reasoner (provided by OPIS, Tachyon, or both).

The PCM and the Act Plan Server are implemented in PRS, which allows complex real-time control of the processing of messages. Eventually, the Critic Manager may also be a PRS-based meta-PA that controls agents for individual critics. Currently, SIPE–2's critic manager

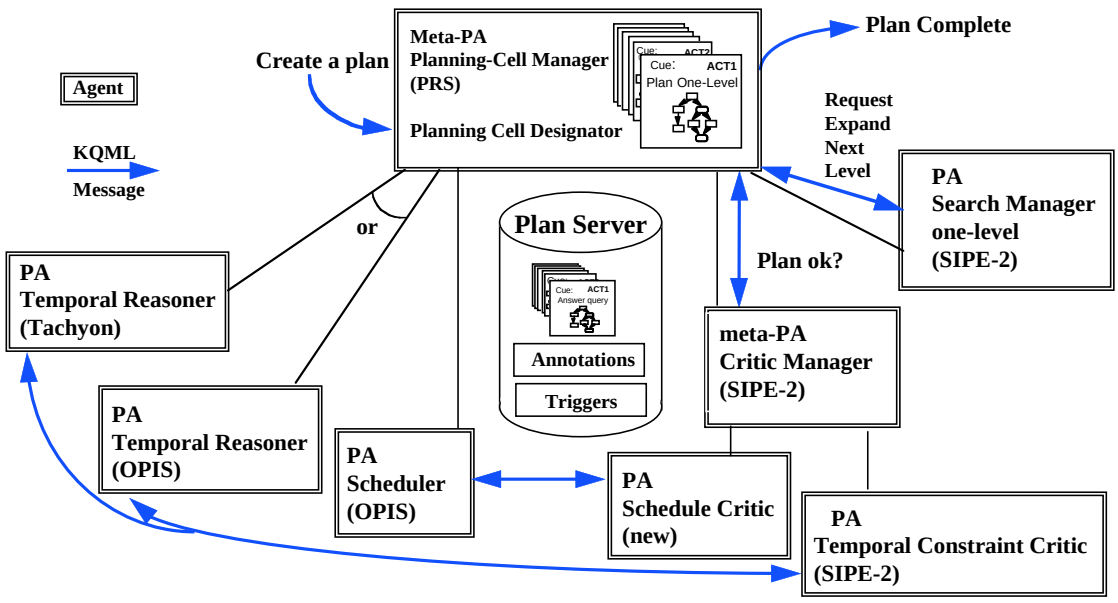

Figure 13: MPA Single Cell Configuration. Arrows represent message traffic, except that all agents communicate with the (Act) Plan Server so those arrows are omitted. The lines without arrowheads show planning cell composition, and the shaded boxes were not implemented. Agents insides dashed boxes were implemented within the same image.

is the basis for the Critic Manager, although it has been modified to send messages to temporal reasoning and scheduling agents. The Tachyon agent is in C and employs a C wrapper, while the other agents have LISP wrappers.

All agents send messages to and from the Act Plan Server, so arrows for these messages are omitted. The Act Plan Server supports annotations and triggers which are used to record features of the plan and notify agents of the posting of those features. The plan is written to the plan server in the Act formalism, which can be understood by the scheduler and the planner. The Act Plan Server answers queries about the plan, and handles the annotations and triggers.

The Search Manager is based on SIPE–2's interactive search routine, which has been extended to record its backtracking points and other information in the plan server. Another extension permits starting the search at various backtracking points. The backtracking points include (lowest level) individual nodes/operators, (mid-level) backtrack to a particular level without choosing a node, and (highest-level) continue searching (the Search Manager handles the backtracking). The cell manager can then exercise control at any of these levels as desired.

7.1 Control of the Planning Process

The demonstration involves strategy-to-task planning in which the PCM controls the interaction among the various components. The demonstration proceeds through the three phases described here. Because of the specialized nature of the task, a special Act (*Multiphase Planning with CTEM*) is included in the procedural knowledge base of the PCM to enable generation of plans that require CTEM invocation.[8]

Phase I: Targeting Plans The first phase produces the initial targeting plans for achieving air superiority. For this phase, no scheduler capabilities are used, as there is nothing to schedule in this phase. The plan is generated using the PCM-Expand-and-Critique method with the single critic :plan-ok?.

Phase II: CTEM Analysis The second phase consists of CTEM scheduling the Phase I plan. This request is made by sending a :schedule-ctem performative to the cell's Planner.

Phase III: Support Plan In the final phase, the support missions required for the CTEM-generated schedule are generated by a second planning process. This portion of the task requires scheduling, so the PCM modifies the PCD to include "opis" in the role of scheduler and redistributes the new PCD to all cell agents. The same planning style is used, this time with both the :plan-ok? and :schedule-ok? critics. In addition, phantom checking is deactivated to enable more rapid generation of plans.

7.2 Demonstration Scenario

The demonstration generates a two-day plan to achieve air superiority over two unnamed countries. We extended IFD-4 by adding some Intelligence Surveillance Reconnaissance (ISR) operators with temporal constraints (the ISR must be sometimes be done several days before other actions). The PCM invokes the Temporal Reasoner to check these constraints.

The PCM generates the Phase I plan using a planning cell including Tachyon as the Temporal Reasoner but no Scheduler. In Phase II, the PCM sends a message to the Critic Manager to invoke CTEM, and then reconfigures the planning cell to include OPIS as the Scheduler and to exclude the Temporal Reasoner (the temporal constraints are already satisfied).

[8] CTEM is a force requirements estimator and scheduler developed at AEM Services.

During generation of the Phase III plan for support missions, OPIS is called periodically to check the resource allocations (this was done by a simple, greedy algorithm in IFD-4). This period can be once per node, once per level, or once per plan. We did it once per level in the demonstration. OPIS may recommend new resource assignments, which causes the Schedule Critic to modify the plan. OPIS posts annotations describing which resources are overutilized and which are near capacity. If resource constraints are sufficiently unsatisfiable, OPIS reports a schedule failure.

The demonstration develops a plan in which fuel tankers are overutilized. When the Schedule Critic notices the Resources-Overutilized annotation posted by the Scheduler for specific tankers, it inserts a Resource-Class-Overutilized annotation for the class of tankers. The PCM has posted a trigger on such an annotation and is immediately notified. It responds with two different tactics to produce a better plan:

1. The PCM sends an :advice plan performative to the Planner, which causes the Planner to choose options requiring less fuel for the remainder of the plan expansion. This capability employs SRI's Advisable Planner. The plan will still have flaws because resources were already overutilized before the PCM issued the advice.
2. The PCM (after either finishing tactic (1) or aborting the planning process) invokes a second search for another plan, this time using advice from the start. This produces a fuel-economic plan in which tankers are not overutilized.

This scenario shows the flexibility provided by the MPA. Separate software systems (OPIS, Tachyon, and SIPE–2, using KQML and PRS for support), cooperatively generate a plan. They are distributed on different machines, implemented in different programming languages, and combined in multiple ways because of the flexible architecture. The Act Plan Server allows flexible communication of the plan among agents. The PCM encodes different strategies for monitoring and controlling the planning process, thus demonstrating dynamic strategy adaptation in response to partial results. Soon, we will show two different systems providing the same functionality with OPIS and Tachyon both providing temporal reasoning.

7.3 Demo Visuals

To communicate the multiagent, asynchronous nature of the planning, we use three monitors with three graphical displays in addition to a dozen or more windows that log message traffic for each agent. The agents all run in parallel, most on different machines. The three graphical displays are used as follows, each drawing asynchronously:

- The PCM screen shows the PRS graphic trace of the Acts from the meta-PA library as they are executing.
- The Search Manager screen draws the plan in the planner's internal representation after each expansion.
- The Act Plan Server screen draws each new plan in the common plan representation. Thus, the plan is drawn in two different representations. The Search Manager and Act Plan Server update their displays simultaneously, because the former sends an :update-plan message to the Act Plan Server before it sends itself a :draw-plan message.

The extensive textual and graphical tracing in the PCM and the tracing and logging capabilities of the MPA wrapper provide valuable runtime insights into the operation of the planning cell. With many agents operating simultaneously, it can be difficult to grasp what the cell is doing.

7.4 Message Traffic for Planning and Scheduling

This summary of the interactions between the Search Manager, the Act Plan Server, and the Scheduler in the demonstration can be skipped by those not intending to use MPA. The interaction of planning and scheduling and low-level details of KQML messages are stressed in this summary.

The PCM sets up the planning cell, including :ping and :pcd messages to the scheduler.

The PCM loops through a cycle of expanding and critiquing the plan until a final plan is produced. The :expand-plan performative is sent to the Search Manager and the :plan-ok? performative is sent to the Critic Manager. These agents send plan updates and annotations to the plan server. When :expand-plan returns `(:plan-complete TASK PLAN A-NETWORK)`, the PCM invokes the Critic Manager by using the :final optional argument. If the plan is correct, a `(:final-plan)` annotation is posted to the plan server for this plan. The PCM has posted a trigger that cause it to be immediately notified of any final plans.

There were three options in MPA for invoking the Scheduler:

1. Have the Scheduler invoked by use of triggers on the appropriate annotation in the Act Plan Server,
2. Have the Critic Manager or Schedule Manager invoke the Scheduler with a :schedule-ok? message.

3. Have the PCM invoke the Scheduler with a :schedule-ok? message.

We implemented Option 2: the PCM keeps a list, customizable by the user, of the messages it should send to the Critic Manager. In particular, the PCM sends :plan-ok? and :schedule-ok? messages to the Critic Manager. The Critic Manager and Schedule Critic in turn send messages to the Temporal Reasoner and the Scheduler. The messages for some of these lower-level interactions are explained here (these are not complete KQML messages, only the KQML performative and the content field).

When the Scheduler is added to the planning cell by the PCM, it queries the Act Plan Server for the available resources:

```
(ask-all :content (:query-plan :task TASK :plan PLAN
    :action-network A-NETWORK :view :available-resources))
```

The Search Manager writes the plan to the Act Plan Server after each planning level with a message of this form:

```
(tell :content (:update-plan :task TASK :plan PLAN
                        :action-network A-NETWORK
                        :view :task-network :content GRAPH-FILE))
```

The Critic Manager invokes the Temporal Critic (which in turn invokes the Temporal Reasoner specified in the PCD):

```
(evaluate :content (:temporal-ok? :task TASK :plan PLAN
                              :action-network A-NETWORK))
```

The Critic Manager invokes the Schedule Critic:

```
(evaluate :content (:schedule-ok? :task TASK :plan PLAN
                           :action-network A-NETWORK))
```

The Schedule Critic in turn invokes the Scheduler:

```
(evaluate :content (:allocate-resources :task TASK :plan PLAN
                             :action-network A-NETWORK
                            {:load-resources t}))
```

The Scheduler queries the Act Plan Server for the resource requirements:

```
(ask-all :content (:query-plan :task TASK :plan PLAN
                      :action-network A-NETWORK
                      :view :resource-constraints))
```

The Act Plan Server extracts the necessary information from the Act representation of the plan's relevant action network, and returns an expression representing the :resource-constraints view of the plan. The Scheduler asks for available resources only once. After that, it sets a global variable and does not query the plan server again. To force the Scheduler to get a new :available-resources view, use the optional argument `:load-resources t` in the :allocate-resources message.

The Scheduler posts annotations to the Act Plan Server indicating potential resource problems:

```
(insert :content
 (:annotation ((sched-complete)
               (resources-overutilized :resources RESOURCE-LIST)
               (resources-near-capacity :resources RESOURCE-LIST))
         :task TASK :plan PLAN :action-network A-NETWORK))
```

The Schedule Critic checks for the latter two annotations whenever it invokes the Scheduler. It abstracts from the individual resources mentioned by the Scheduler to the correct resource classes, and may post annotations of this form:

```
(insert :content (:annotation
                  ((resource-class-overutilized CLASS-NAME)
                  (resource-class-near-capacity CLASS-NAME))
           :task TASK :plan PLAN :action-network A-NETWORK))
```

The above abstraction could have been done by the Scheduler, the Act Plan Server, the PCM, or the Planner. While all these options are reasonable, we choose to do this analysis in the Planner because all necessary information was in the Planner, and we did not want the PCM interacting with the plan at the low level of individual resources. The PCM posted triggers for these annotations, so it is immediately notified when such annotations are posted. The PCM responds by sending an :advice message to the planner that tells it to activate a new piece of advice:

```
(tell :content (:advice :economical-defense :mode :add))
```

This message advises the Planner to use options that require fewer resources (economical-defense is the name of one of many prespecified pieces of advice). Other possible responses include backtracking in the case of overutilized resources, and running the Scheduler more often in the case of resources near capacity.

The Scheduler updates the plan's action network by posting its resource allocations:

```
(tell :content (:update-plan :task TASK
                             :plan PLAN
                             :action-network A-NETWORK
                             :view :resource-allocations
                             :content ALLOCATION-LIST))
```

The Scheduler replies to the :allocate-resources request with the following:

```
(reply :content (:schedule-ok? [:schedule-ok | :schedule-failure]))
```

At the end of the planning process, the Search Manager queries the Act Plan Server for the :resource-allocations view of the plan's action network:

```
(ask-all :content (:query-plan :task TASK
                               :plan PLAN
                               :action-network A-NETWORK
                               :view :resource-allocations))
```

The Search Manager uses this information to instantiate planning variables (e.g., bases, resources). A low-level protocol is used by the Temporal Critic to call the Temporal Reasoner. The Temporal Reasoner merely invokes Tachyon appropriately and returns the output, without looking at the results:

```
(evaluate (:temporal-ok? :constraint-file "FILENAME.tcn"
                         :temporal-output-file "FILENAME.out"))
```

We chose to use files for communication because IFD-4 Tachyon files could be larger than a quarter of a megabyte. This requires the Temporal Critic and Temporal Reasoner to have access to the same file system. MPA has not been optimized for direct communication of large amounts of data (known improvements exist).

8 Multiple Planning Cell Demonstrations

In June 1997, we demonstrated a configuration with two planning cells producing alternative plans for the same task in parallel. The TIE 97-1 demonstration built on this configuration, but used a new and more extensive knowledge base of planning operators and advice, and integrated additional agents.

The meta-PCM controls the entire process, including initialization of planning cells, distribution of tasks and advice, and reporting of solutions. The planning cells operate exactly as described for the single planning cell demonstration (Section 7), except that they are invoked by the meta-PCM instead of the user, and they refuse requests if they are already busy.

8.1 June 1997 Demonstration

This demonstration followed the same sequence as that for the single planning cell demonstration, except two alternate plans are constructed in parallel. Figure 14 depicts a multicell configuration similar to the one used in this demonstration (not all of the shared agents shown were used). Because the planning cells are given different advice initially, the two plans are different. The planning cells shared the same Act Plan Server, Temporal Reasoner and Scheduler agents. Sharing agents requires fewer running jobs, but is not a requirement. For example, each planning cell could have had its own plan server.

As with the single planning cell demonstration, described in Section 7, the demonstration generates two-day plans to achieve air superiority over two unnamed countries. Each PCM generates a plan in three phases as described in Section 7.1.

This time two different plans are generated. One planning cell is given the advice `:economical-defense` by the meta-PCM and the other is given the advice `:F-14D-intercept`. (This advice could have been selected by a human user.) The latter plan is a "gas-guzzling" plan that overutilized tankers, but the former plan is economical in its use of resources, and does not overutilize tankers. There may still be over-utilized reconnaissance assets in either plan, depending on the choices that are made and the availability of those assets.

Section 9.1 describes the message traffic between the meta-PCM and the planning cells.

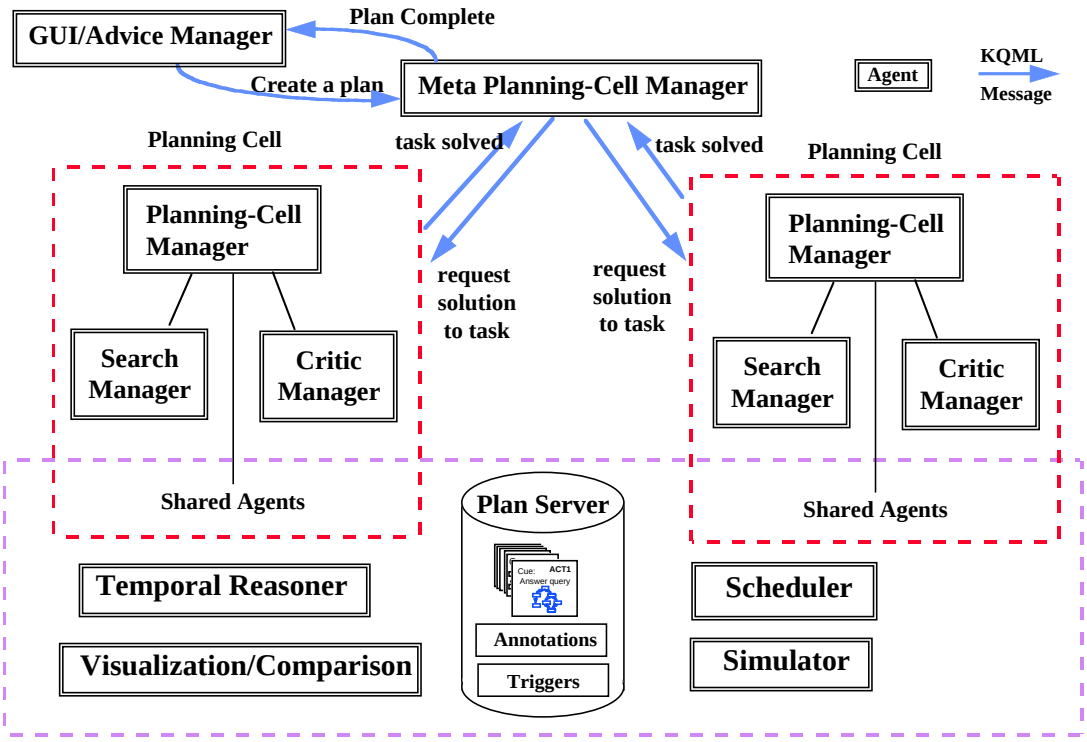

Figure 14: MPA Configuration for Multiple Planning Cells. Arrows represent cell-manager message traffic. The lines without arrowheads show planning cell composition.

8.2 TIE 97-1 Demonstration

The TIE 97-1 demonstration, first shown in November 1997, built directly on the multiple planning-cell demonstration just described. Significant extensions include the use of a new and more extensive knowledge base, and the integration of additional agents for plan evaluation, user interaction and plan visualization. Figure 14 depicts the configuration used in this demonstration.

Several new MPA agents were created. The ARPI Plan Authoring Tool (APAT) from ISX, a legacy system written in Java, was integrated as an MPA agent. It fills two roles, the first being the role of GUI/Advice Manager depicted in Figure 14, and the second being the role of plan visualization (a service also provided in the demonstration by the Visage system from MAYA). Another new agent was the Air Campaign Simulator (ACS) [1] from the University of Massachusetts, written in LISP, whose role was to run Monte Carlo simulations of plans,

and feed the resulting data to the VISAGE system from MAYA for plan visualization and evaluation. Both of these agents can read Acts from the Act Plan Server and translate them to their internal representation.

It took the University of Massachusetts only a day to download MPA and have ACS sending and receiving MPA messages. A few more days were needed for translating from the Act representation into the ACS representation. (This translation is easy because ACS can ignore much of the information in an Act.) It took APAT a week to be sending and receiving MPA messages, primarily because APAT was using the new Java API of KQML and there were bugs in it. It took several weeks to write and debug the Act translators. This translation was more difficult because it went both directions, and because APAT needs to translate many specific relationships, such as the parents of actions and objectives.

This experience indicates that MPA does indeed facilitate the integration of new technologies. To have a technology like ACS be invoked by APAT and retrieve the plan from the agents producing it took much longer in IFD-4 (before MPA).

8.2.1 ACP Knowledge Base

We built a new knowledge base (ACP KB) for the air campaign planning domain. The ACP KB is intended to support air superiority planning, both offensive and defensive, down to activity and support mission level. The ACP KB now models target networks. Each network provides a capability and requires capabilities from other networks in order to function. Network effectiveness is modeled quantitatively, allowing incremental degradation.

The network/capability model is used primarily to model threats to air superiority, though the model is rich enough to model production networks, lines of communication, and other networks of targets. The new KB allows more sophisticated and realistic effects-based planning than was done in IFD-4. For example, it keeps track of target networks (not present in IFD-4) and calculates degradation of the network as targets are attacked. The ACP KB adds more realism and provides a more substantive integration with the simulator, the Advisable Planner, and plan critique and evaluation modules than the old knowledge base.

8.2.2 Demonstration Flow

Here we describe the demonstration flow within MPA, although not all aspects of this scenario were done in real time during the live demonstration. All aspects of plan generation except

the CTEM run are done in real time, but translation of plans to APAT and multiple simulation runs are not (although one simulation run can be done in real time). The first plan is generated before the demonstration is begun, and the second and third plans are generated during the demonstration.

The user interacts with APAT, specifying part of the plan and selecting advice to control the automatic generation of the air superiority plan. As of September 1998, the user can easily create new advice using the Mastermind tool[9] to generate legal advice definitions which are then sent to all appropriate agents.

The APAT-authored plan is written as an Act to the plan server. When requested by a human using APAT's GUI, APAT sends a message to the meta-PCM with the user-selected advice requesting a plan to be generated. The meta-PCM invokes a planning cell on the request, using all the agents of our earlier demonstrations except the temporal reasoner. This planning process uses both the MPA-generated plan and the APAT-authored plan as input to CTEM, and then generates a support plan for the CTEM output.

When the post-CTEM plan is finished, APAT retrieves the plan by sending a message to the plan server, and translates it to APAT's internal representation. The user can use APAT's GUI to inspect the plan. Interacting with APAT, the user elects to simulate the plan, and a message is sent to invoke ACS on the generated plan. ACS retrieves the plan by sending a message to the plan server, and generates data by simulating the plan. VISAGE is then invoked to visualize the results of the simulation.

The user does not like certain aspects of the plan, so decides to generate two different alternatives, and selects two sets of advice in APAT. A message is sent to the meta-PCM requesting two plans. Two planning cells are invoked in parallel and two plans are generated. Again, simulation is invoked, and this time VISAGE can display graphs and charts comparing all three plans along various dimensions.

9 Agent Interface Specifications

The interface specifications for the MPA agents in the June 1997 demonstration are described here. These specifications allow the cell manager to interact with other PAs in the cell, and allow agents to plug and play if they conform to the interface specification of some agent. For

[9] from Information Sciences Institute (ISI).

a new agent, its implementor must determine the message syntax for invoking the agent, write handlers for such messages, determine the syntax for any messages returned or sent, document the I/O specifications, including any side effects on the plan server, and ensure that all users and consumers of the services provided by this agent are aware of (and agree to) the protocols and documentation.

This section, which also describes wrapper support for certain messages, can be skipped by those not intending to use MPA. The messages in this section are not complete messages, only the communication performative and the content field are shown. Certain delimiters are used below as they are in Backus-Naur Form (BNF). The | symbol represents 'or', the square brackets are grouping operators for |, and braces {} designate optional elements.

In the MPA release, there are logs of actual message traffic in subdirectories beginning with `log` in `/mpa/released/doc/`. For example, to see the messages during the planning for the TIE 97-1 demonstration, look at the files in `/mpa/released/doc/log-97-123/`. Message traffic for demonstrations using an executor can be found in `/mpa/released/doc/log-cpef/`. There is one file for each agent. Thus, `meta-pcm.text` logs the messages of the meta-PCM.

9.1 Meta Planning-Cell Manager Messages

To control the operation of multiple planning cells in parallel and distribute and coordinate tasks amongst those planning cells a meta planning-cell manager requires the message passing capabilities described here.

Initially, the meta-PCM is waiting for an incoming planning request in the form of a Multiple-solve message. In the TIE 97-1 demonstration, requests to the meta-PCM are sent by APAT, at the request of a human using APAT's GUI.

```
invoke:   (evaluate :content (:multiple-solve :task TASK
                        [ {:count N} |
                          {:advice-contexts ADVICE-CONTEXTS}]))
response: (reply :content (:multiple-solve (:ok :id REQUEST-1))) |
          (reply :content (:multiple-solve :error ERROR-STRING))
```

There are also several possible error responses that begin with an :error keyword and a string explaining why the request was refused.

One of :count or :advice-contexts must be specified; if both are given, :count is ignored. Advice-contexts are generally used as they allow naming of the plans generated, and specification of different advice for each plan so that the plans will be different. The argument to :count is an integer, and the argument to :advice-contexts is a list of sets of advice, or labeled sets of advice. If labels are provided, the meta-PCM will label the generated plans accordingly. Here is an example of a request using :advice-contexts:

```
invoke: (evaluate :content (:multiple-solve :task "AS-A"
         :advice-contexts
         (("advice1"  (:USE-TOMCATS-OVER-CARRIERS :DENY-AIR-PICTURE))
          ("advice2" (:MASS-VS-SAMS :PREEMPT-ANTISHIP-THREATS)))))
response: (reply :content (:MULTIPLE-SOLVE (:OK :ID REQUEST-2)))
```

Then the first request arrives (and only on the first request), the meta-PCM first checks the status of available planning cells by pinging their PCM agents. Every agent should be able to respond to the :ping performative, which verifies that the agent is alive and well.

This initialization message is sent to each PCM under the control of the meta-PCM:

```
invoke:   (evaluate :content (:ping) ...)
response: (reply :content (:ping :ok))
```

The :ping performative determines whether an agent is up and running. If the pinged agent does not respond, the pinging agent times out in KQML. Eventually, we should do something more sophisticated, such as removing the planning cell that does not respond from the list of available planning cells.

The meta-PCM then loops through each request. For each request, it loops through the planning cells, sending the request until some free planning cell accepts it. Alternatively, the meta-PCM could keep track of which planning cells are free, but the scheme we have implemented is more robust over resets and restarts of agents.

This is an example of a message sent to one of the PCMs:

```
invoke:   (evaluate :content (:solve :task "AS-A"
             :plan "advice1"
             :plan-option plan-with-ctem
             :requestor "meta-pcm"
             :advice (use-tomcats-over-carriers deny-air-picture)
             :id request-1))
response: (reply :content (:solve (goal
```

```
            (achieve (plan-with-ctem as-a plan-advice1))))) |
        (reply :content (:solve :error
   "PCM busy on request from meta-pcm for
   (use-advice as-a (use-tomcats-over-carriers deny-air-picture))
    Try again later."))
```

Here, :plan is the label from the entry in :advice-contexts, :advice is the list of advice from that same entry, and :id is the name generated for the request. If a PRS goal formula is returned, the meta-PCM is finished with this request until solutions or failures are reported. If the busy error message is returned, the request must be sent to other PCMs.

Whenever a cell manager finishes a requested task, it sends either a Solution for a Failed message back to the requestor. The syntax of these messages are described in Section 9.2. The meta-PCM handles these messages and passes their content on to the appropriate requestor. Note that Solution messages return the name of the plan server agent which is storing the plan. Without this argument, the meta-PCM would have to know in advance (i.e., before invoking a planning cell) what plan server would be used in each planning cell.

The meta-PCM will send some number of the following messages back to its requestor. Barring errors, there should be one Solution or Failed message for each request accepted. These are not replies to the initial request, but messages initiated by the meta-PCM. LABEL is the label from :advice-contexts.

```
invoke: (tell :content (:solution :task TASK :label LABEL
                          :plan PLAN :id REQUEST-ID
                          {:plan-server PLAN-SERVER}))
response: No reply is expected.

invoke: (tell :content (:subset-accepted :task TASK :count N
                          :id REQUEST-ID))
response: No reply is expected.

invoke: (tell :content (:busy :task TASK :error ERROR
                          :id REQUEST-ID))
response: No reply is expected.

invoke: (tell :content (:failed :task TASK :plan PLAN
                          :label LABEL :error ERROR
                          :id REQUEST-ID))
response: No reply is expected.
```

Finally, the meta-PCM forards Define-advice messages to all running planning-cell managers. The syntax is given in Section 9.3. The meta-PCM distributes Define-advice messages to all PCMs and waits for them to return. (KQML will return immediately if a PCM cannot be contacted.) The meta-PCM returns the value from the first successful reply. If there are no successful replies, it returns the first error reply, else NIL.

9.2 Planning-Cell Manager Messages

The cell manager (PCM) is invoked either by the user posting a goal in PRS (through the GUI), or by some agent sending a Solve message, as the meta-PCM does. The PCM responds immediately with a reply indicating it is busy or giving the PRS goal that was posted, indicating acceptance of the request. An example of a Solve message and the reply were given in Section 9.1. Here is the syntax for a Solve message:

```
invoke:   (evaluate :content (:solve :task TASK :plan PLAN
                                    {:plan-option PLAN-OPTION}
                                     :requestor AGENT-ID
                                     :advice ADVICE-LIST
                                     :id REQUEST-ID))
response: (reply :content (:solve GOAL-WFF)) |
          (reply :content (:solve :error ERROR-STRING))
```

Here, PLAN is a label provided by the requestor (in the meta-PCM, this is taken from the entry in :advice-contexts), and ADVICE-LIST is a list of advice. REQUEST-ID is a name for the request provided by the requestor. The requestor is included in the message because it may not always be the sender of the message, e.g., it may have been requested by a human user interacting with the meta-PCM. The optional `:plan-option` argument used for our demonstrations was `plan-with-ctem` which invokes Acts in the PCM for planning using CTEM, which differs from the normal planning process. Examples of GOAL-WFF and ERROR-STRING were given in Section 9.1.

When establishing a planning cell, the cell manager sends two messages to all agents. Every agent should be able to respond to the :ping performative, which verifies that the agent is alive and well, and to store the relevant PCD upon receipt of a :pcd performative.

```
invoke:   (tell :content (:pcd ((ROLE AGENT) ... (ROLE AGENT)) ))
response: No reply is expected.
```

The :ping performative is described in Section 9.1. The :pcd performative is used to inform all agents in the cell of the current PCD. The LISP wrapper for MPA that is available to all LISP-based agents defines handlers for both :ping and :pcd. The function `mpa:agent-for-role` is used by all cell agents to send messages using the PCD as follows: `(mpa:send-msg msg (mpa:agent-for-role :plan-server))`

The PCM announces a solution with a message of the following form, sent to the agent requesting the solution.

```
invoke: (tell :content (:solution :task TASK :plan PLAN
                          {:plan-server PLAN-SERVER}
                          :id REQUEST-ID))
response: No reply is expected.
```

where the optional `:plan-server` argument is given when the responding cell manager and the meta-PCM are not using the same plan server agent. The value of this argument is the name of the plan server agent from which the plan can be retrieved. Solution messages (and Failed messages) fill the :id field with the value in the :id field of the request. Note that the planner may send Solution messages with additional keyword arguments (see Section 9.7).

Alternatively, the PCM announces a failure with a message of the following form, sent to the agent requesting the solution.

```
invoke: (tell :content (:failed :task TASK :plan PLAN
                          :id REQUEST-ID))
response: No reply is expected.
```

In addition, the PCM handles Advice and Define-advice messages which it forwards to the planner. Advice messages name predefined advice and the issued advice is recorded in the PRS database. The syntax for Define-advice messages is given in Section 9.3. (The PCM may change any Error communication performative in a reply to Reply communication performative, but the :error keyword will remain intact.)

```
invoke:  (tell :content (:advice ADVICE {:mode MODE}))
response: No reply is expected.
```

The optional argument `:mode` can be set to either `:add` or `:replace`, to indicate whether whether the advice should be added to the current set of advice, or should replace it (the default is `:add`). ADVICE is either a symbol or a list of symbols.

9.3 Planner Messages: Search Manager

As described in Section 5.3.1, the Search Manager generates plans level by level when used with the PCM. For each level, a more detailed plan is first generated, then critiqued. This is fairly general, but other planners may require lower-level control of the planning process.

Alternatively, the Generate-plan message can be used to expand all levels and generate a final plan. In this case, the plan is generated completely inside the planner without using other agents (bypassing PCM control).

Any of the messages described below can return an error reply, which may also have an :error keyword and a string explaining why the request was refused. For example, all messages can generate replies of this form:

```
error response:  (error :content (P-PERF :unrecognized-plan
                                  :error ERROR-STRING ))
```

Some messages (e.g., Generate-plan and Init-problem) can also return an :unrecognized-task error. The :unrecognized-plan error may indicate problems either with the task, plan, or action network, and the value of :error will give a more detailed explanation.

The Search Manager accepts the following messages. Only non-error replies are documented below. In the MPA release, there are logs of actual message traffic in subdirectories beginning with `log` in `/mpa/released/doc/`. The first message below causes a plan to be drawn on the GUI of the planner.

```
invoke:   (evaluate :content (:draw-plan :task TASK :plan PLAN
                                  :action-network A-NETWORK))
response: (reply :content (:draw-plan :plan-drawn))
```

9.3.1 Invoking Planning on a Task

Planning for a new task can be invoked with any of these messages: Generate-plan and Init-problem. An Init-problem message is used to initialize planning for distributed, multiagent plan generation, and causes the planner to create an initial action network to represent a task.

```
invoke:   (evaluate :content (:init-problem {:task TASK :plan PLAN
                                        :act ASCII-ACT}))
response: (reply :content (:init-problem (TASK PLAN A-NETWORK)))
:start-problem is a synonym for :init-problem
```

All of the invocation messages mentioned above take the keywords shown above and use the same algorithm for determining which task to solve. In all cases, the `:plan` argument provides a name to use for the generated plan, and the task to solve is specified by trying these options in order until one finds the task:

1. If the `:act` argument is given, it's value is the ASCII Act for the task (in either string or s-expression form, see example in next section).
2. If the `:task` argument is given, the task is retrieved from the plan server with a Query-task message. Querying the plan server for the task can be disabled by setting `sipe::get-task-from-plan-server` to NIL.
3. If the `:task` argument is given, the task is any problem defined in the planner with that task name.
4. If neither the `:task` or `:act` argument is given, the task is the first unsolved problem found in the planner.
5. Finally, an :unrecognized-task error is returned.

9.3.2 Customizations

The Search Manager can be customized in a number of ways. We describe these first, as the messages to invoke planning allow specification of some of the customizations described here. When messages to the planner selects a customization, the new selection affects only the processing of this one message.

The default values described are for MPA when no domain is loaded. Many of our demonstrations (such as TIE 97-1) change these defaults when they are loaded. For example, in TIE 97-1, the plans are usually drawn after each expansion, and both the :ascii-filename and :task-network views are sent to the plan server for all levels of the final plan.

The planner generally sends an action network to the plan server for each level of a level-by-level expansion. For a Generate-plan request, nothing is sent to the plan server is complete, and then the options below control whether the final action network or all action networks are sent.

- The variable `mpa:*plan-to-server*` serves two purposes: when it is nil, it disables all sending of information to the plan server; else, it is the name of a view and the planner sends that view of the plan to the plan server server. The default value is :task-network, which has the advantages of persistently storing the action networks, and of

permitting the drawing of the action networks on the Act Plan Server GUI. Here are the allowable values:

  - :TASK-NETWORK – sends the :task-network view.
  - :ASCII – sends the :ascii view.
  - :ASCII-FILENAME – sends both the :task-network and :ascii-filename views. (since both must be computed anyway, and the plan server must be on the same file system as the planner)
  - :TASK-NETWORK-AND-ASCII – sends both the :task-network and :ascii views, but not the :ascii-filename view.
  - NIL – do no translation, never contact the plan server.

- When the plan is completed, there is an option for sending all the action networks of a plan to the plan server as a single plan-level update. The variable `mpa:*all-levels-to-server*` is the name of a view to use for such an update. The allowable values are the same as for `mpa:*plan-to-server*` above. The default value is nil, which means no all-levels view is sent, but only the view for the final action network (as determined by `mpa:*plan-to-server*`). The final action network is always sent in a level-by-level expansion, but for a Generate-plan request, it is sent only if `mpa:*all-levels-to-server*` is nil.

- The variable `mpa:*monitors-to-server*` determines how the monitors are sent to the plan server after a final plan is found. A value of nil disables sending monitors (as does a NIL value for `mpa:*plan-to-server*`). The other allowable values are the two views for monitors, :monitors and :monitor-filename. If monitors are not sent to the plan server, the filename of the monitors is sent in the :monitor-filename argument of Install and Solution messages. By default, the :monitor-filename argument is also included even when the plan server is being used, but this can be disabled by setting the variable `sipe::monitor-filename-with-notify` to NIL.

- If `mpa:*draw-plan-after-reply*` is T, the Search Manager will draw each action network it generates in a level-by-level expansion, or the final action network for a Generate-plan request. The default is nil, because drawing can slow down the planning process for large plans. The Search Manager draws a plan by sending a :draw-plan message to itself just before replying to an Expand-plan, Expand-plan-and-critique or Generate-plan message. This technique allows the requestor to receive a reply and the plan to be put in the plan server before drawing. Thus, other agents can be processing the plan in parallel with the drawing process.

- Querying the plan server for the task can be disabled by setting the variable `sipe::get-task-from-plan-server` to NIL. In this case, the task name should be internally defined in the planner as a problem to be solved.

9.3.3 Plan Expansion and Critique

The Generate-plan message causes all levels of a plan to be generated for a given task, using only the planner. This message is not used in the MPA demonstrations, but provides agent-based access to the capabilities of our planner, and is used by SRI's CPEF demonstrations. An example of a Generate-plan request is given at the end of this section.

```
invoke:   (evaluate :content '(:generate-plan
              :task TASK :plan PLAN
             {:act ASCII-ACT}
             {:facts FACTS}
             {:create-monitors BOOLEAN}
             {:plan-to-server VIEW}
             {:all-levels-to-server VIEW}
             {:id ID}))
response: (reply :content (:generate-plan :ok))
```

The requestor may not wish to wait for a reply to a Generate-plan message or the Revise message described below. Therefore, the reply for these messages is an :ok message whenever the task is valid. Like the PCM, the planner announces solutions and failures to its requestor with Solution and Failed messages, respectively. The syntax for the Failed message is given in Section 9.2. However, the Solution message returned by the planner has some additional fields (beyond those shown in Section 9.2) to support plan repair by an executor. The planner's Solution message is described in Section 9.7.

The arguments to Generate-plan are:

:task :plan :act used as described in Section 9.3.1.

:facts specifies a list of predicates to be changed in the world before plan generation. The value is NIL (the default) or a list of predicates (e.g., `((on c d) (not (on a b)))`).

:create-monitors is a boolean. If T, the planner creates a set of monitors for the generated plan, just as if a Create-monitors message had been sent. In our current planner agent, the CPEF system must be loaded to use this option.

:plan-to-server determines what view, if any, is sent to the plan server. The value will override the value of `mpa:*plan-to-server*` during the processing of this message. If this keyword is omitted, the default value of `mpa:*plan-to-server*` is used.

:all-levels-to-server determines if all action networks are sent to the plan server (and in what view). The value will override the value of `mpa:*all-levels-to-server*` during the processing of this message. If this keyword is omitted, the default value of `mpa:*all-levels-to-server*` is used. If both `mpa:*all-levels-to-server*` and `mpa:*plan-to-server*` are non-nil, then only the all-levels view is sent, else the final action network is sent as directed by `mpa:*plan-to-server*`.

:id is used for identification in Solution and Failed messages.

Here is an example of a Generate-plan message sent to the planner:

```
(evaluate :content (:generate-plan :task "task-gain-oas"
             :plan "plan-gain-oas"
             :act (solve-request-110
            (environment (properties (class problem)))
            (plot (goal-1 (achieve (breach-iads d+0 low)))))
             :plan-to-server :ascii-filename
             :id R110))
```

The next two messages cause the planner to expand an existing action network to the next level of detail. They are used by the PCM in our demonstrations.

```
invoke:   (evaluate :content (:expand-plan :task TASK :plan PLAN
                              :action-network A-NETWORK
                             {:plan-to-server VIEW}
                             {:node n :opr o :level D}))
response: (reply :content (:expand-plan
            [ (:level-complete TASK PLAN A-NETWORK) |
              (:no-expansion TASK PLAN A-NETWORK) |
              (:no-expansion) |
              (:plan-complete TASK PLAN A-NETWORK) |
              (:plan-failure) ]  ))

invoke:   (evaluate :content '(:expand-plan-and-critique
                               :task TASK :plan PLAN
                               :action-network A-NETWORK
                              {:plan-to-server VIEW}
                              {:all-levels-to-server VIEW}
                              {:node n :opr o :level D}))
response: (reply :content (:expand-plan-and-critique
             [ (:level-complete TASK PLAN A-NETWORK) |
               (:no-expansion TASK PLAN A-NETWORK) |
```

```
(:no-expansion) |
(:final-plan  TASK PLAN A-NETWORK) |
(:plan-failure) ] ))
```

The arguments to these two messages are:

:task :plan :action-network are used to determine which action network to expand.

:plan-to-server determines what view, if any, is sent to the plan server of the action network that is generated. The value will override the value of `mpa:*plan-to-server*` during the processing of this message. Generally, this keyword is omitted, and the default value of `mpa:*plan-to-server*` is used.

:all-levels-to-server only has an effect if a final plan is generated. The value will override the value of `mpa:*all-levels-to-server*` during the processing of this message. Generally, this keyword is omitted, and the default value of `mpa:*all-levels-to-server*` is used.

:node :opr :level are used invoking backtracking. This capability has not been tested.

The returned keywords of the plan performatives :expand-plan and :expand-plan-and-critique are posted as annotations in the plan server, and have the following meanings:

(:plan-failure) indicates no action network was found and no backtracking points exist.

(:no-expansion TASK PLAN A-NETWORK) indicates no action network was found but backtracking points exist. The A-NETWORK is the input action network name, unless it was not recognized as an action network, in which case (:no-expansion) is returned.

(:level-complete TASK PLAN A-NETWORK) indicates there is a action network at the next level. (In one case the action network will be the same action network level as input — only for :expand-plan-and-critique and only when the input action network had all goals solved, but the critics found a problem and modified the action network.)

(:plan-complete TASK PLAN A-NETWORK) indicates there is a action network with all goals solved, although it may contain flaws. (This is generally the same action network level as input for :expand-plan, but it could be a new action network level for :expand-plan-and-critique.)

(:final-plan TASK PLAN A-NETWORK) indicates there is an action network with all goals solved, and that it is correct. :Final-plan is returned only from :expand-plan-and-critique and may be a new action network level or the same level as input.

9.3.4 Plan Repair

A Revise message causes an existing plan to be modified as required by a set of new facts given in the message. In our current planner agent, the Revise message assumes the CPEF system is loaded and there is an executor agent in the PCD.

```
invoke:   (evaluate :content '(:revise
                                :task TASK :plan PLAN
                                :action-network A-NETWORK
                                {:facts FACTS}
                                {:replace-world BOOLEAN}
                                {:execution-front NODE-LIST}
                                {:node NODE}
                                {:reinstantiate BOOLEAN}
                                {:force-replan BOOLEAN}
                                {:replan-goal EXPRESSION}
                                {:id REQUEST-ID}))
response: (reply :content (:revise :ok))
```

The response to a Revise message is the same as the response to a Generate-plan message, as described above, including Solution and Failed messages that are sent. An example of a Revise request is given in Section 9.7. Here we describe the optional arguments to Revise.

:facts specified facts that are now true in the world. The value can either be a list of predicates (e.g., `((on c d) (not (on a b)))`), or the name of a file which contains a set of predicate entries.

:replace-world is a boolean. If T, the :facts argument completely replaces the existing world description. If NIL (the default), the new facts are appended to the existing world description.

:execution-front is a list of node names in the given action network that have been executed but whose immediate successors have not been executed (i.e., the last node executed on each parallel thread of activity). Alternatively, the value can be NIL or `:none`. The latter indicates that execution has not yet begun.

:node is the name of a node that failed in the given action network. This failed node should be in the execution front or an unexecuted node.

:id is used for identification in Solution and Failed messages.

The current planner invokes a specialized plan repair algorithm in the special case where the plan has not been executed (execution-front is :NONE), as the planner treats this case more like initial plan generation than like plan repair. In this case, the remaining arguments below are ignored as are the :node and :replace-world arguments. (The current Search Manager assumes :replace-world is NIL in this special case.)

The remaining three arguments are used to give direction and advice to the plan repair process. Their values are specialized to the current planner being used (SIPE–2), and may have to be modified for other planners.

:reinstantiate is a boolean. The default is T, but the default can be changed for a given domain. If T, the planner tries to reinstantiate planning variables to reestablish failed preconditions and protected conditions.

:force-replan is a boolean, the default is NIL. If T, the planner must produce a different plan that eliminates the failed node. In SIPE–2, the failed node should be executed or a precondition or process node. A different plan is produced even when there are no new facts and nothing is wrong with the current plan. :Force-replan is useful when a plan fails for unknown reasons, or a human requests a different plan for some particular goal.

:replan-goal can have many different values, and is used to direct or advise the plan repair. Values can vary from nodes to replan, to goal predicates to replan, to names of planner algorithms for selecting goals to replan. Details are given in the SIPE–2 manual [16] (see `*select-replan-goal*`).

During plan revision, nodes that are in parallel with the failed node are not changed, since they may be executing while the plan is being modified. Nodes that are after the failed node can be removed and/or replaced during plan revision. It may be desirable to have the executor and planner communicate about more complex schemes for determining which nodes may be executing and which can be replaced.

9.3.5 Advice

The Advice message was described in Section 9.2. The Define-advice message is used to allow users to dynamically create advice using ISI's Mastermind tool. Mastermind allows the user to easily generate legal advice definitions from an advice grammar. The value of :source is a Mastermind parse tree that can either be in a file, string, or list, depending on the value of :source-type. If :activate is non-nil, then the advice is defined and selected, else it is just defined. For the detailed syntax the parse tree, see the file `sipe/released/lisp/ap-mastermind.lisp`. The value returned is a list of the names of the newly defined advice.

```
invoke: (evaluate :content (:define-advice
              :source PARSE-TREE
              :source-type {:FILE | :STRING | :LIST}
              :activate BOOLEAN))
response: (reply :content (:define-advice (MM-ADVICE-1 ...)))
error response: (reply :content (:define-advice nil :error STRING))

invoke:  (tell :content (:advice ADVICE {:mode MODE})
response: No reply is expected. (return :ok message if requested)
error response:  (error :content (:advice
        [ :not-found | :some } :error ERROR-STRING))

invoke:  (ask-all :content (:query-advice {:active BOOLEAN})
response: (reply :content (:query-advice { NIL | ADVICE-TUPLES ]))
```

The Query-advice message asks the planner to return the names and descriptions of defined and/or active advice. If :active is nil (the default), all defined advice is returned, with active advice noted. If :active is non-nil, then only the active advice is returned. The returned value is a list of advice-tuples in both cases. An advice tuple is a list with four elements: a symbol naming the advice, a string describing the advice, the keyword :active, and a boolean.

An example return value for Query-advice is:

```
(:BREACH-AT-TWO-PLACES  "Breach IADS at two different sectors"
    :ACTIVE NIL)
(:MASS-VS-AIRBASES "When attacking airbases,
    mass forces against one target type" :ACTIVE T)
```

9.3.6 Miscellaneous

A Create-monitors message will create a set of monitors for the given plan (see Section 6.1). In our current planner agent, the Create-monitors message assumes the CPEF system is loaded.

```
create-monitors
invoke:   (evaluate :content (:create-monitors :task TASK :plan PLAN
                                :action-network A-NETWORK))
response: (reply :content (:create-monitors FILENAME))
```

The next two messages can be used to advise or reset the planner. A Reset-problems message causes the planner to redefine its predefined set of named problems. A Reset-domain message causes the planner to delete its entire knowledge base about the current planning domain and reload it.

```
invoke:   (evaluate :content '(:reset-problems))
response:  No reply is expected. (return :ok message if requested)

invoke:   (evaluate :content '(:reset-domain))
response:  No reply is expected. (return :ok message if requested)
```

9.4 Planner Messages: Critic Manager

Currently, the Critic Manager invokes most critics internally, but we plan to specify lower-level interfaces for individual plan critics. We have already done this for the Temporal Reasoner and Scheduler that were used in our demonstration. Error replies are the same as for the Search Manager (see Section 9.3).

The Critic Manager accepts the following messages:

```
invoke:   (evaluate :content '(:plan-ok? :task TASK :plan PLAN
                                 :action-network A-NETWORK
                                {:plan-to-server VIEW}
                                {:all-levels-to-server VIEW}
                                {:final F :interact I }))
response: (error :content (:plan-ok?
                      [:plan-ok |
                       :plan-rejected {:error ERROR-STRING} |
                       :final-plan]))

invoke:   (evaluate :content '(:schedule-ctem
```

```
                                    :task TASK  :plan PLAN
                                    :action-network A-NETWORK))
response: (reply :content (:schedule-ctem (TASK PLAN A-NETWORK)))
```

The values of :plan-to-server and :all-levels-to-server are as described for Generate-plan, but only have an effect when :final is T and :final-plan is returned. The returned keywords for :plan-ok? are posted as annotations in the plan server.

The plan identifiers returned by :schedule-ctem denote a planning problem for support missions and should be sent to the Planner in an :init-problem message.

9.5 Schedule Critic Messages

The Schedule Critic can be invoked by either the Critic Manager or the cell manager. Error replies are the same as for the Search Manager (see Section 9.3). The Schedule Critic accepts the following messages:

```
invoke:   (evaluate :content '(:schedule-ok?
                                :task TASK :plan PLAN
                                :action-network A-NETWORK))
response: (reply :content (:schedule-ok?
                   [:schedule-ok | :schedule-failure]) ...)
```

Before the schedule critic invokes the Scheduler, the Planner should have sent the appropriate :update-plan message to the plan server to post the action network. The Scheduler retrieves this information by querying the the plan server after receiving the request.

Before replying, the Scheduler may also send annotations and plan updates to the plan server. In our demonstration, it sends an :update-plan message to define the :resource-allocations view of the action network, and posts Sched-Complete, Resources-Overutilized, and Resources-Near-Capacity annotations in the plan server.

9.6 Temporal Critic Messages

The Temporal Critic can be invoked by either the Critic Manager or the cell manager. It accepts the following messages:

```
invoke:   (evaluate :content '(:temporal-ok?
                                :task TASK :plan PLAN
                                :action-network A-NETWORK))
response: (reply :content (:temporal-ok? [:temporal-ok |
                                          :temporal-failure |
                                          :error ERROR-STRING |
                                          :unrecognized-plan] ))
```

The Temporal Critic gets the action network name and extracts all the temporal constraints from the action network and sends them in a separate message to the Temporal Reasoner. The Temporal Critic then processes the values returned by the Temporal Reasoner and updates the plan server with any updated time windows.

The following low-level protocol is used by the Temporal Critic to call the Temporal Reasoner. The Temporal Reasoner merely invokes Tachyon appropriately and returns the output, without looking at the results.

```
invoke:   (evaluate (:temporal-ok? :constraint-file "FILENAME.tcn"
                            :temporal-output-file "FILENAME.out"))
response: (reply :content (:temporal-ok? :temporal :file OUTFILE))
response: (error :content (:temporal-ok? :temporal
                                :error ERROR-STRING))
```

We chose to use files for communication because IFD-4 Tachyon files could be larger than a quarter of a megabyte. This requires the Temporal Critic and Temporal Reasoner to have access to the same file system. Otherwise, it is trivial to include the contents of the file in the contents of the message, although the messages could get large.

9.7 Executor Messages

As described in Section 6.1, the executor executes (or simulates execution of) a plan, while constantly monitoring the world and taking actions in response to new goals and events. The following messages are a first attempt at supporting this capability. We expect this set of messages to expand significantly.

The executor can be told to ready a plan for execution with the Install message and begin execution of an already installed plan with the Execute message.

```
invoke:   (evaluate :content '(:install
                                :task TASK :plan PLAN
                                :action-network A-NETWORK
                                {:task-network FILENAME}
                                {:monitor-filename FILENAME}))
response: (reply :content (:install :ok))
```

Installation requires the executor to retrieve the plan from the plan server in whatever view the executor prefers. If the executor wants to use monitors created by the planner, it must also retrieve the :monitors view of the plan from the plan server. If the :monitor-filename argument is `:none`, then the executor does not need to retrieve the monitors from the plan server. The :task-network and :monitor-filename optional keywords are provided to support the option of not using the plan server when both the planner and executor have access to the same file system. If the :task-network argument is given, it is the :task-network view of the plan. Similarly, the :monitor-filename argument provides the :monitor-filename view.

Here is an example of an actual Install message sent to the SRI Executor:

```
(evaluate :content (:install :task "sipe-task":plan "sipe-plan"
  :action-network SOCAP1-6-PLAN-1
  :task-network "/cypress/act-socap1-6-plan-1.graph"
  :monitor-filename "/cypress/monitors-act-socap1-6-plan-1.graph"))
```

Like the Install message, the Execute message allows the :task-network view of the plan as an optional argument. The executor may not even need this information if it was stored during plan installation.

```
invoke:   (evaluate :content '(:execute
                                :task TASK :plan PLAN
                                :action-network A-NETWORK
                                {:task-network FILENAME}))
response: (reply :content (:execute :ok))
```

The executor may do many things in response to new events and goals. These responses can include, among other things, the sending of Revise and Generate-Plan messages (described in Section 9.3) to the planner. Here is an example of a Revise message sent by the executor to the planner:

```
(evaluate :content (:revise :task "task-gain-oas"
             :plan "plan-gain-oas"
             :action-network REQUEST-110-15-PLAN-1
             :node P3640 :execution-front (C3758 P3640)
             :facts "db-failure-22522.lisp"
             :replace-world NIL  :id "FAILURE-22522"))
```

The executor need not wait for a reply, as it is always an :ok message if the request is understood. Sometime later, the executor will receive Solution or Failed messages from the planner, which fill the :id field with the value in the :id field of the request. The syntax for the Failed message is the same as that for Failed messages from the PCM (see Section 9.2). However, the Solution message returned by the planner has some additional fields that support the use of solutions during plan repair by the executor.

```
invoke: (tell :content (:solution :task TASK :plan PLAN
                          {:plan-server PLAN-SERVER}
                          :id REQUEST-ID
                          :action-network A-NETWORK
                          {:graph FILE}
                          {:monitor-graph FILE}
                          {:node-map node-map}))
response: No reply is expected.
```

Here is an example of the Solution message received for the above Revise message.

```
invoke: (tell :content (:solution :task "task-gain-oas"
           :plan "plan-gain-oas"
           :action-network REQUEST-110-FAILURE-22522
           :graph NIL
           :monitor-graph "monitors-request-110-failure-22522.graph"
           :node-map ((P3640 C4463) (C3758 C4463) (P3636 C4463))
           :id "FAILURE-22522"))
response: No reply is expected.
```

9.8 Plan Server Messages

All types of plan server messages use task, plan, and action-network names. As of MPA version 1.3, these names can be either strings or symbols without string quotes (although these two are distinct), and both are case insensitive. Thus, `"FOO"` and `FOO` are distinct and do not name the same plan, while `"FOO"` and `"foo"` do name the same plan.

In general, plan server updates should not expect a response (only queries require a response). However, the plan server always replies if requested to do so. The plan server returns an :ok message if a reply is requested when none is expected.

9.8.1 Task Updates

Before a plan expansion is initiated, the task name and the plan name are posted to the plan server. The plan server creates the necessary structures required to represent them. If the task already exists the plan server creates the necessary plan structure and links it to the task.

```
invoke:   (tell :content (:update-task
                  :task TASK
                  {:view VIEW-NAME   :content VIEW
                   :plan PLAN
                   :parent PLAN      :subplans (PLAN ... )
                   :assumptions FREE :context FREE
                   :plan-assumptions FREE}))
response: No reply is expected. (return :ok message if requested)
```

The :task keyword is required, the others are optional. The :view and :content keywords, which should be used together, can be used to specify an Act representing the objectives (goals) of the task.

Whenever the :plan keyword is used, a new plan record for this task is created. If a plan record with than name already exists, the new plan record is pushed onto the task-plans, effectively masking the existence of the old plan by that name (it can be recovered by deleting the new plan). The :parent and :subplans keywords can be used to specify a subplan/parent relationship between the plan being created and an existing plan. For every parent link given, the corresponding subplan link is automatically filled in; similarly, parent links are generated from subplan links.

The final three keywords are there to allow additional information to be specified. Currently, their values are simply free text that is returned upon a query. These three keywords are not used in the current MPA release, but we expect to develop specifications for them in the future.

Two examples of :update-task messages are given below. The first creates task and plan records without specifying the objectives of the task. The second message gives an Act for the objectives, and will overwrite the Act stored in any existing task record for that task (or create a new task record if no such record exists).

```
invoke:   (tell :content (:update-task :task "fennario-task"
                                       :plan "terrapin-1"))
response: No reply is expected. (return :ok message if requested)

invoke:   (tell :content (:update-task :task "fennario-task"
                  :view :ascii
                  :content (Meet-Peggy
                   (plot (g1 (achieve (meet peggy fennario)))))))
response: No reply is expected. (return :ok message if requested)
```

9.8.2 Plan Updates

An Update-plan message can apply to either a plan or action network, and is used to create instances of those objects, or new views of action networks. The views currently supported in the Act Plan Server for Update-plan are the same as those supported for Query-plan (see Figure 5), except that :available-resources is not supported (it is assumed the plan server computes this from its database). If no view is given, Update-plan assumes :task-network as the default.

An update on plans is indicated by the :action-network keyword being omitted or having a value of `nil` or `:all` (for plan updates, these two are equivalent). In this case, the views for plans shown in Figure 5 are applicable. Otherwise, the request is an update on an action network, where the :action-network keyword gives the name of a known action network, a new action network, or the special value :last, which refers to the last action network defined for the given task and plan. The views for action networks shown in Figure 5 are applicable.

During planning, at the end of each level, the action network is posted to the plan server. This can be done in either ASCII or Grasper form. A plan update uses the following performatives and keywords:

```
invoke:   (tell :content (:update-plan :task TASK :plan PLAN
                               {:action-network A-NETWORK}
                               {:view VIEW-NAME}
                               :content VIEW))
response: No reply is expected. (return :ok message if requested)

Where :content gives the value for the given VIEW-NAME.  Examples:
  :task-network          (VIEW is a string for a Grasper filename)
  :ascii                 (VIEW is a string or s-expression
                             for an ASCII Act)
  :resource-allocations  (VIEW is an s-expression for resource
```

```
                                allocations, as specified below)
```

The :update-plan performative causes the plan server to load in the Grasper-file, parse the ASCII Act, or store the given view. The plan server associates the Act with the named task, plan, and action network, as appropriate. If an ASCII Act contains strings in its properties, then the Act must be specified without string quotes (as an s-expression).

The following example messages show the specification of two action networks. The task and plan have already been created by the example :update-task message shown above. The action networks are specified with different views. The first uses the :ascii view and provides an Act in ASCII representation (with strings in properties). The second action network uses the :task-network view. Any action-network name specified in the :action-network keyword must be the Act name specified in the ASCII Act, or an Act that is present in the given :task-network file.

```
invoke:   (tell :content (:update-plan
             :task "fennario-task"
             :plan "terrapin-1"
             :action-network  "level-3"
             :view :ascii
             :content
  (level-3
   (environment
    (cue (achieve (meet peggy fenarrio)))
    (preconditions)
    (properties (authoring-system sipe-2) (class plan)))
   (plot
    (a2 (type parallel) (parent c8) (achieve (go-to fennario))
     (orderings (next a3))
     (properties ((action-type "Move with rowboat"))))
    (a3 (type parallel) (parent c1) (achieve (find peggy)))))))
response: No reply is expected. (return :ok message if requested)

invoke:   (tell :content (:update-plan
             :task "fennario-task"
             :plan "terrapin-1"
             :action-network  "level-1"
             :view :task-network
             :content "/homedir/foo/level-1.graph"))
response: No reply is expected. (return :ok message if requested)
```

9.8.3 Deletions

The performatives and keywords for deleting an action network, plan or task from the plan server are described here. With the Delete communication performative, :update-task and :update-plan can be used interchangeably as the plan performative — the various combinations are chosen by the keyword arguments.

To delete all tasks or a specified task (plus all descendant plans and action networks), use the following message:

```
invoke:   (delete :content (:update-task :task [TASK | :all] ))
response: No reply is expected. (return :ok message if requested)
```

If :task is :all then all the tasks known to the plan server are deleted. This command is useful to free space for garbage collection and to prevent the plan server from confusing a new version of a plan with some previous version. Tasks can be deleted programmatically with `(prs:delete-ps :task :all).` Before sending a message to delete tasks to the plan server, an agent should consider asking a user for confirmation.

To delete a specific plan or all plans in a specified task, use the following message:

```
invoke:   (delete :content (:update-plan :task TASK
                                  :plan [PLAN | :all] ))
response: No reply is expected. (return :ok message if requested)
```

To delete a specified action network or all action networks in a specified plan, use the following message:

```
invoke:   (delete :content (:update-plan :task TASK
                    :plan PLAN
                    :action-network [A-NETWORK | :all | NIL ] ))
response: No reply is expected. (return :ok message if requested)
```

9.8.4 Task Queries

The current query language supports different views of the task's objectives and uses the :query-task plan performative. The plan performatives and views currently supported in the Act Plan Server are depicted in Figure 5. If no view is given, Query-plan assumes :ascii as the default.

Query messages use the following general syntax:

```
invoke:   (ask-all :content '(:query-task :task TASK {:plan :all}
                                          {:view VIEW-NAME}))
response: (reply :content (:query-task VIEW-DESCRIPTION))
```

Ask-one queries use the same syntax and generate the same responses for Query-plan because its views are considered to be single-valued, even when the value is a list. If Ask-one is used with other plan performatives, the value returned will be a list of length one, with the list member being the first thing in the list that would have been returned by the equivalent Ask-all query. If the equivalent Ask-all query does not return a list, then the value it does return is also returned as the value of the Ask-one.

The special value :all can be used for both the :task and :plan keywords to find out which tasks and plans are known to the plan server. When :all is used (it is the only allowable value for :plan), a list of names is returned, and the :view keyword is ignored. The following example message shows the use of the :all value to return a list of all known task names.

```
invoke:   (ask-all :content '(:query-task :task :all))
response: (reply :content (:query-task ("fennario-task" "foo"))
```

The :query-task performative causes a particular view of the task's objectives (expressed as an Act) to be retrieved from the plan server or computed from data stored in the plan server. For example, if :view is :ascii, the plan server will convert a Grasper representation to ASCII.

The allowable views for task queries are described here.

Task Network Task View (:task-network) The value is a string containing the name of a Grasper file. The file contains an Act representing the task's objectives, as well as information for drawing the Act attractively in Grasper.

ASCII Task View (:ascii) The value is an expression containing the ASCII representation of the Act representing the task's objectives.

ASCII Filename Plan View (:ascii-filename) The value is a string containing the name of a file. The file contains the value that would have been returned by the :ascii view. This example shows the use of this view:

```
invoke:   (ask-all :content '(:query-task :task "fennario-task"
                                          :view :ascii-filename))
response: (reply :content (:query-plan
             "/tie97/released/fennario.text"))
```

Assumptions Task View (:assumptions) The value is whatever free text was given as the task's assumptions.

9.8.5 Plan Queries

The current query language supports different views of the plan and uses two plan performatives, Query-plan and Query-node. The plan performatives and views currently supported in the Act Plan Server are depicted in Figure 5. If no view is given, Query-plan assumes :task-network as the default.

Query messages use the following general syntax:

```
invoke:   (ask-all :content '(:query-plan :task TASK {:plan PLAN}
                                          {:action-network A-NETWORK}
                                          {:view VIEW}))
response: (reply :content (:query-plan VIEW-DESCRIPTION))

invoke:   (ask-all :content '(:query-node :task TASK :plan PLAN
                                           {:action-network A-NETWORK}
                                           :view VIEW
                                           :node NODE))
response: (reply :content (:query-plan (LIST-OF-NODES)))
```

Ask-one queries use the same syntax and generate the same responses for Query-plan because its views are considered to be single-valued, even when the value is a list.

A Query-plan message can apply to either a task, plan or action network, while :query-node messages always apply to action networks. The special value :all can be used for any of the three keywords to find out which tasks, plans, and action networks are known to the plan server. When :all is used, a list of names is returned, and the :view keyword is ignored. To post different views at the plan level, :action-network should be NIL or omitted.

The following example messages show the use of the :all value in the three keywords that permit its use.

```
invoke:   (ask-all :content '(:query-plan :task :all))
response: (reply :content (:query-plan ("AS-A"))

invoke:   (ask-all :content '(:query-plan :task "AS-A" :plan :all))
response: (reply :content (:query-plan ("plan-advice1-phase3"
                                        "plan-advice1"))
```

```
invoke:   (ask-all :content '(:query-plan :task "AS-A"
                                          :plan "plan-advice1"
                                          :action-network  :all))
response: (reply :content (:query-plan (AS-A-18 AS-A-17 ...))
```

The first query returns a list of all known task names, the second returns a list of all known plan names for the given task, and the third returns a list of all known action-network names for the given task and plan. All keywords not shown above are ignored (when :all is used).

Currently, the only queries supported for tasks use the value :all for either the :task or :plan keyword. A query on plans is indicated by the :action-network keyword being omitted or having a value of :all or nil, and the views for plans shown in Figure 5 are applicable.

A query on an action network is indicated by the :action-network keyword being the name of a known action network, or the special value :last, which refers to the last action network defined for the given task and plan. In this case, the views for action networks shown in Figure 5 are applicable.

The :query-node performative always applies to an action network, although if the :action-network keyword is nil or not given, it defaults to :last (in :query-node queries only). The view must be given from those shown in Figure 5. A :query-node request returns a list of the names of nodes that precede, succeed, or are unordered with respect to the given node in the given plan (depending upon the view).

The Query-plan performative causes a particular view of the plan's entire action network to be retrieved from the plan server or computed from data stored in the plan server. For example, if :view is :resource-constraints, a function walks over the action network and collects the necessary constraints in an interlingua. The views that mention resources currently use an interlingua specific to the ACP domain. The allowable views for both plans and action networks are described here.

Task Network Plan View (:task-network) The value is a string containing the name of a Grasper file. For action network queries, the file contains the Act representation of that action network. For plan queries, the file contains an Act for all action networks in the plan. The file not only contains the action network, but information for drawing the network attractively in Grasper.

ASCII Plan View (:ascii) For action network queries, the value is an expression containing the ASCII Act representation of the action network. For plan queries, the file contains the ASCII Act representation of the plan, including all action networks in the plan.

ASCII Filename Plan View (:ascii-filename) The value is a string containing the name of a file. The file contains the value that would have been returned by the :ascii view. This example shows the use of this view:

```
invoke:   (ask-all :content '(:query-plan :task "AS-A"
                                          :plan "plan-advice1"
                                          :view :ascii-filename))
response: (reply :content (:query-plan
             "/tie97/released/temp/plan-advice1-all-levels.text"))
```

Subplans Plan View (:subplans) The Act Plan Server allows some plans to be denoted as subplans of other plans. This value is a list of the names of all plans that are considered subplans of the given plan. For example, in the TIE 97-1 demonstration, the post-CTEM support mission plan is considered a subplan of the pre-CTEM plan. This example shows the use of this view:

```
invoke:   (ask-all :content '(:query-plan :task "AS-A"
                                          :plan "plan-advice1"
                                          :view :subplans))
response: (reply :content (:query-plan ("plan-advice1-phase3")))
```

Monitors View (:monitors) The value is a list of the ASCII Act representations of the monitors for this plan (see Section 6.1). Currently, only the planner can compute the monitors (eventually, the Act Plan Server should do this), and then only when Cypress or CPEF code is loaded into the planner.

Monitor File View (:monitor-filename) The value is a string containing the name of a Grasper file, which in turn contains a number of Acts that represent the monitors for this plan (see Section 6.1).

Available-Resources ACP Plan View (:available-resources) In the future, the :available-resources view may become a :query-domain performative in the Knowledge Server. The :available-resources view specifies the available base and airframe objects, and the availability over time of airframes at bases, with availability levels. The syntax is as follows:

```
(:base-list BASE-CLASS-SPEC
 :airframe-list AIRFRAME-CLASS-SPEC
 :availability
  ((AIRBASE-NAME
    ((AIRFRAME1 ((ARRIVAL-DAY SORTIES AVAILABILITY-LEVEL)
                 (ARRIVAL-DAY SORTIES AVAILABILITY-LEVEL)
                  ...))
      (AIRFRAME1 ((ARRIVAL-DAY SORTIES AVAILABILITY-LEVEL)
                  ...))      ...)))
```

The BASE-CLASS-SPEC and AIRFRAME-CLASS-SPEC appear in the class/object hierarchy of the Planner (or Knowledge Server). From this data, the Scheduler can extract the necessary information about resource availability. Availability levels are described in Section 6.2.3.

Resource-Constraints ACP Plan View (:resource-constraints) The syntax is as follows:

```
(:nodes
 ((:node-id NAME
   :latitude LAT
   :longitude LON
   :resource-requirements
     ((MISSION-CATEGORY (AIRFRAME SORTIES BASE AVAILABILITY)
                             ...) ...))
   :start-time value ; mission start day
   :end-time value)    ; mission end day + 1
  ...)
 :temporal-constraints (...))
```

The :resource-requirements slot contains a list of alternative airframe/quantity/base tuples in order of preference for each mission-category. (The base is optional. If given, it provides a preference for where the airframes should be based; otherwise, the nearest base with available resources is assigned.)

Resource-Allocations ACP Plan View (:resource-allocations) The format for this view is the same as :resource-constraints, but all fields are optional except :resource-requirements and :base, which are the only things that are changed (allocated) by the Scheduler. Therefore, the minimal format is:

```
(:nodes
 ((:node-id NAME
   :resource-requirements
    ((MISSION-CATEGORY (AIRFRAME SORTIES BASE AVAILABILITY))
                           ...))))
```

Note that in this view, because allocations have been made, the base always appears, and only one airframe type is assigned for each mission category. Note that the airframes may be taken from different availability levels; the detailed information about how many airframes are taken from each level is not currently returned.

9.9 Annotations

All operations on annotations use the :annotation plan performative. The allowed communication performatives are shown in Figure 3. Annotations are added with the Insert communication performative:

```
invoke: (insert :content (:annotation
                           [ANNOTATION | LIST-OF-ANNOTATIONS]
                             {:plan PLAN
                              :task TASK
                              :action-network A-NETWORK} ))
response: (reply  :content (:annotation LIST-OF-ANNOTATION-RELNS))
```

Annotations are encoded as predicates and stored in the plan server's database. To distinguish action-network-specific from plan-specific from task-specific from task-independent annotations, four variants are used as shown in Figure 4. This approach of separating the task/plan/action-network from the relation used to represent the annotation enables more flexible retrieval of annotation information (e.g., finding all plans or action networks that have a given annotation, or finding all annotations for a given plan, task or action network).

Examples of messages that insert annotations were given in Section 7.4. The following LISP function call generates a valid message for adding two annotations:

```
(mpa:make-msg 'insert
              '(:annotation ((:level-complete) (:plan-ok?))
                     :task T-1 :plan P-3 :action-network AN-2))
```

Here is the message that posts backtracking points to the plan server; these points are relevant to the overall task, not just an individual action network. For this reason, they do not require a :plan argument:

```
(mpa:make-msg 'insert
              `(:annotation (:backtrack-choices ,choices)))
```

The return value is a list of annotation relations (`LIST-OF-ANNOTATION-RELNS`). An annotation relation has one of the forms in Figure 15. These forms are similar to those in Figure 4, except that keywords are not used.

```
(ANNOTATION <basic-annotation> TASK PLAN A-NETWORK)
(ANNOTATION <basic-annotation> TASK PLAN)
(ANNOTATION <basic-annotation> TASK))
(ANNOTATION <basic-annotation>)
```

Figure 15: Variants of Annotation relations: these variants distinguish action-network-specific from plan-specific from task-specific from task-independent annotations.

Deletion of annotations is achieved by the following message:

```
invoke:    (delete :content (:annotation [ANNOTATION | :any]
                             {:task [TASK |:all] :plan [PLAN | :all]
                             :action-network [A-NETWORK | :all} }))
response:  (reply  :content (:annotation LIST-OF-ANNOTATION-RELNS))
```

The :task :plan and :action-network arguments are optional, with their possible values interpreted as follows:

```
 :all    -  delete annotations associated with all such objects
 TASK    -  only delete annotations for the named task
 PLAN     -  only delete annotations for the named plan
 A-NETWORK-  only delete annotations for the named action network
 no value -  only delete annotations not tied to specific tasks
```

The value for the :annotation performative must be either an annotation or the special symbol :any. For the value :any, all annotations in the plan server for the given :task :plan or :action-network argument are deleted. For a given annotation, all annotations in the plan server that unify with it (modulo the cases indicated by the :plan argument) are deleted. The specified annotation can contain variables, in which case it acts as a kind of schema. Anything of the form ID.NUMBER is interpreted as a variable — for example, PLAN.1 matches anything. (PRS does not have a sorted logic, so the name of the variable is purely explanatory.)

The Delete-All performative for annotations returns a list of matches for the specified annotation, represented as a `LIST-OF-ANNOTATION-RELNS`.

Queries about annotations are performed using the following message:

```
invoke:   (ask-all :content (:annotation [ ANNOTATION | :any ]
                                {:task [ TASK | :all ]
                                 :plan [ PLAN | :all ]
                              :action-network [ A-NETWORK | :all] }))
response: (reply  :content (:annotation LIST-OF-ANNOTATION-RELNS))
```

The semantics for the Ask-All performative are similar to those of the Delete performative: the keyword arguments have the same meaning and they return the same values. The only difference is that annotations are not deleted by Ask-All. Ask-One queries use the same syntax and generate the same responses except that the list of annotations returned is of length one.

We would like to declare certain annotations as being functional in certain arguments within the plan server, so that every sender would not have to worry about first deleting old values of the annotation. This concern does not apply to "time-stamped" annotations, but annotations like the :backtrack-choices which apply to the whole process and not to a particular plan should replace the former values of the same annotation (i.e., :backtrack-choices is functional in its only argument). The Act Plan Server can do this by making the appropriate PRS declaration for functional predicates.

9.10 Triggers

All operations on triggers use the :trigger plan performative. The allowed communication performatives are shown in Figure 3. Triggers are added with the Insert performative:

```
invoke:   (insert :content (:trigger :event E :destination D
                                     :msg M :source S :id ID))
response: (reply :content (:trigger ID))
```

The first three keywords – :event, :destination, and :msg – are required. If :id is not provided, a unique identifier is supplied. The :msg field specifies the message to be sent when a trigger is activated. The recipient of this message should be specified in the :destination field, and must be an *agent*.

Currently, the :event field must be an annotation fact having one of the forms shown in Figure 4, except that each `<basic-annotation>` is replaced by an `<annotation-schema>`. In contrast to basic annotations, an annotation schema can contain variables. Variables may appear in the :msg and :event specifications. The triggering event will bind the variables appropriately, so they can be used in the outgoing message.

The origins of the trigger are documented using :source, which defaults to the name of the sending agent. If an agent wishes to provide other sources for certain triggers, the :source keyword can be used. Note that the source is specified as an agent name (e.g., "sipe") while the destination is specified as an agent role (e.g., :planner). It is necessary to document the identity of the specific agent from which the trigger originated (in case the role is later filled by a different agent), but the message should be sent to the agent filling a particular role, so that a different agent filling that role will not receive the triggered messages.

Trigger insertion is performed *only* if it is unique, meaning that one of the source, event, destination, or message differs from all previously defined triggers. Here, difference is up to variable renaming with respect to events and messages. If the trigger returned is unique, the ID of the inserted trigger is returned; if the inserted trigger matches some trigger already in the Plan Server, then the ID of the matched trigger is returned instead.

Triggers are deleted with the Delete communication performative:

```
invoke:   (delete :content (:trigger :id ID))
response: (reply  :content (:trigger ID))
```

If the plan server cannot find the trigger, nothing happens. MPA does not currently do any authority management – for example, it could require that only the person who adds a trigger can delete it.

Triggers are queried with the Ask-All communication performative:

```
invoke:   (ask-all :content (:trigger :event E))
response: (reply :content (:trigger LIST-OF-MATCHING-TRIGGERS))
```

Ask-One queries use the same syntax and generate the same responses except that the list of triggers returned is of length one.

Each trigger is represented by a fact in the database of the PRS agent that implements the plan server. Trigger facts have the form

```
(TRIGGER <id> <source> <event> <msg> <dest>)
```

The following is an example of a fact used to encode a trigger to inform that planner to activate certain advice when the :low-fuel annotation is posted for any action network.

```
(TRIGGER :fuel-advice-trigger       ;id
  "plan-server"                     ;source
  (ANNOTATION (:low-fuel) TASK.1 PLAN.1 A-NETWORK.1) ;event
  (TELL (:ADVICE :fuel-low))        ;message
  :destination :planner)            ;destination
```

10 Other Agents

In addition to the agents that were used in the demonstration, several other agents were either partially designed or fully implemented. This exercise has further driven the specification of our architecture and agent interface specification. Section 8.2 describes the agents for ACS and APAT used in the TIE 97-1 demonstration. Two agents that may be of use in TIEs are described here. The first has not been used to date, and the second is the lowest level agent used in the temporal reasoner.

10.1 Sort Hierarchy Agent

As larger problems are addressed, the planner must use legacy databases rather than requiring all data to be entered directly into the system. For this reason, we have begun making the SIPE–2 sort hierarchy into an agent, which allows SIPE–2 to access other databases and allow other agents to access any SIPE–2 sort hierarchy. This agent should be integrated into any knowledge server that is implemented.

We have developed a Sort Hierarchy Agent interface specification to provide modular access to static knowledge about classes and objects for SIPE–2. We have implemented two separate instantiations of this agent. The interface specification consists of a small set of functions (10) that provide the basic access operations required for planning. SIPE–2 can

be readily adapted to operate using any underlying frame representation system that provides appropriate definitions for this agent interface. The interface has been implemented for the original SIPE–2 hierarchy subsystem.

To further facilitate access to alternative representation systems, an instantiation of the Sort Hierarchy agent has been defined for the Generic Frame Protocol (GFP), jointly developed for DARPA at SRI International and Stanford University, that provides a uniform interface model for frame-representation systems. By using this agent, SIPE–2 can now be run with any frame representation system for which GFP has been implemented (e.g., USC/ISI's Loom, SRI's SIPE–2 sort hierarchy, CMU's Theo, Stanford's Ontolingua).

10.2 Temporal Critic and Tachyon Server

The modularization and interaction of SIPE–2 and Tachyon were already in place from an earlier ARPI TIE, but the communication had to be extended for KQML. We obtained the latest version of Tachyon from General Electric, and updated the SIPE-Tachyon interface to work with this version and to work within KQML. In fact, the SIPE-Tachyon interface was completely rewritten to more generally and accurately translate SIPE time constraints to Tachyon, to use Tachyon-generated time windows in more planning algorithms, to update to the newest version of Tachyon (which requires a different syntax), and to make use of new Tachyon features regarding hierarchical nodes. SIPE–2 lays out the Tachyon networks so that the plan displays nicely in the Tachyon GUI for easy debugging and viewing.

The interface for the Temporal Reasoner was described in Sections 9.6 and 7.4. We implemented two Temporal Reasoner agents (the KQML agent name is Tachyon-server), one using the LISP wrapper and one using the C wrapper. The Temporal Reasoner used in the MPA demonstration is built using the C-language agent library functions described in Appendix B. The Tachyon-server agent is invoked by the Temporal Critic using a lower-level protocol, described here, than the one used to invoke the Temporal Critic.

The Tachyon-server agent handles the Evaluate performative, and the plan performatives :ping and :temporal-ok?. The :temporal-ok? performative takes keyword arguments :constraint-file (giving the pathname of an input file for Tachyon) and :temporal-output-file (specifying the destination for Tachyon's output). If :temporal-output-file is not specified, the agent replaces the ".tcn" extension on the input file with ".out," or appends ".out" to the input file name if no ".tcn" extension is found.

If the procedure is successful (the message was parsed correctly, and Tachyon was invoked successfully), the Tachyon server returns a message with the content
`(:temporal-ok?  :temporal-ok :file OUTFILE)`
where OUTFILE is the pathname of the Tachyon output file. Note that in this case, it is still possible that the constraint network has not been solved (e.g., because there was a syntax error in the constraint file, or because the constraints are inconsistent). The Tachyon-server does not "know" anything about the contents of the input or output files, only whether the Tachyon program has run to completion. The Temporal Critic is responsible for parsing the information returned by Tachyon.

If an error occurs (for example, the message syntax is incorrect, or the Tachyon executable or input/output files cannot be accessed correctly), the server returns a message with the content `(:temporal-ok?  :temporal-failure :error ERROR-STRING)`
where `ERROR-STRING` is a description of the error.

11 Future Work

Promising directions for extending this work are numerous. They include the following:

- Experimenting with additional configurations and cooperative problem-solving methods.
- Broadcasting messages to fill planning cells with agents.
- Defining a broader range of cell-manager control strategies and planning styles.
- Incorporating additional technologies as new agents.
- Extending the plan server (possible extensions include the plan representation, access control to the plan, version control, and graphical browsing capabilities).

12 Summary

MPA is an open planning architecture that facilitates incorporation of new technologies and allows the planning system to capitalize on the benefits of distributed computing for efficiency and robustness. MPA provides protocols to support the sharing of knowledge and capabilities among agents involved in cooperative problem-solving. Within MPA, software modules written in different programming languages can easily interoperate.

MPA configurations show the flexibility provided by MPA. Separate software systems (OPIS, Tachyon, ACS, APAT, the Advisable Planner, and SIPE–2, using KQML, the Act-Editor, and PRS for support) cooperatively generate and evaluate plans, generating multiple, alternative plans in parallel. These systems are combined in multiple ways through the flexible architecture.

The MPA framework has been used to integrate several sophisticated stand-alone systems cooperating on a large-scale problem. The MPA configuration generated and evaluated complex plans (containing more than 4000 nodes) in the ACP domain, and included agents written in C, C++, LISP, and Java. MPA was used as the infrastructure for ARPI's TIE 97-1.

The Act Plan Server allows flexible communication of the plan among agents through the use of annotations, triggers, and views. The PCM encodes different strategies for controlling the planning process, demonstrating dynamic strategy adaptation in response to partial results. The planner and scheduler use legacy systems to provide a new integration of planning and scheduling technologies.

Other sites have been able to download the MPA wrapper and get an existing LISP program communicating as an MPA agent in one day. Our experience indicates that MPA does indeed facilitate the integration of new technologies, thus encouraging experimentation with and use of new technologies.

Acknowledgments

This research was supported by Contract F30602-95-C-0235 with the Defense Advanced Research Projects Agency, under the supervision of Air Force Research Lab – Rome. Tom Lee made significant contributions, and was responsible for the ACP KB.

References

[1] Paul Cohen, Scott Anderson, and David Westbrook. Simulation for ARPI and the Air Campaign Simulator. In A. Tate, editor, *Advanced Planning Technology: Technological Achievements of the ARPA/Rome Laboratory Planning Initiative*, pages 113–118, AAAI Press, Menlo Park, CA, 1996.

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

[3] Kutluhan Erol, James Hendler, and Dana S. Nau. Semantics for hierarchical task-network planning. Technical Report CS-TR-3239, Computer Science Department, University of Maryland, 1994.

[4] T. Finin, J. Weber, G. Wiederhold, M. Genesereth, R. Fritzson, D. McKay, and J. McGuire. Specification of the KQML Agent-Communication Language. Technical Report EIT T R92-04, Enterprise Integration Technologies, Palo Alto, CA, 1992.

[5] Bill Janssen, Mike Spreitzer, Dan Larner, and Chris Jacobi. ILU 2.0 reference manual. Technical report, Xerox PARC, December 1997.

[6] P.D. Karp, J.D. Lowrance, T.M. Strat, and D.E. Wilkins. The Grasper-CL graph management system. *LISP and Symbolic Computation*, 7:245–282, 1994.

[7] Douglas B. Moran, Adam J. Cheyer, Luc E. Julia, David L. Martin, and Sangkyu Park. Multimodal user interfaces in the Open Agent Architecture. In *Proc. of the 1997 International Conference on Intelligent User Interfaces (IUI97)*, Orlando, Florida, 6-9 January 1997.

[8] Karen L. Myers. Strategic advice for hierarchical planners. In L. C. Aiello, J. Doyle, and S. C. Shapiro, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifth International Conference (KR '96)*. Morgan Kaufmann Publishers, 1996.

[9] Karen L. Myers and David E. Wilkins. *The Act-Editor User's Guide: A Manual for Version 2.2*. Artificial Intelligence Center, SRI International, Menlo Park, CA, September 1997.

[10] Karen L. Myers and David E. Wilkins. *The Act Formalism*. Artificial Intelligence Center, SRI International, Menlo Park, CA, version 2.2 edition, September 1997.

[11] C. J. Petrie. Agent-based engineering, the web, and intelligence. *IEEE Expert*, 11(6):24–29, December 1996.

[12] Stephen Smith. Opis: A methodology and architecture for reactive scheduling. In M. Fox and M. Zweben, editors, *Intelligent Scheduling*. Morgan Kaufmann Publishers Inc., San Mateo, CA, 1994.

[13] D. S. Weld. An introduction to least committment planning. *AI Magazine*, 15(4):27–61, 1994.

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

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

[16] David E. Wilkins. *Using the SIPE-2 Planning System: A Manual for Version 4.21*. SRI International Artificial Intelligence Center, Menlo Park, CA, July 1998.

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

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

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

A LISP Wrapper

The LISP wrapper for MPA is easily incorporated in all LISP-based agents. It provides functions for making, sending and replying to messages that log and trace the message exchange. There are functions for the PCD, for starting and killing agents, and for starting and stopping tracing. There are also message handlers for both the :ping and :pcd plan performatives. The MPA LISP wrapper is loaded with `(sri:load-system :mpa)`, but generally a :subsystem keyword argument is given whose value is the subsystem for your agent. (All subsystems load the basic wrapper functions.)

The wrapper contains a file `handlers-mpa.lisp`, that defines default handlers for the :ping and :pcd plan performatives. This file is not loaded with the wrapper, because it would redefine any Tell and Evaluate handlers already defined in an agent. If an agent responds only to the above two plan performatives in its Tell and Evaluate handlers, the wrapper handlers can be used by loading `handlers-mpa.lisp` as part of the agent. If an agent responds to other plan performatives, then it must define its own Tell and/or Evaluate handler. Such handlers should be created by extending the default handlers, because every agent must respond to :ping and :pcd performatives. In addition, the default handlers show the proper use of several MPA wrapper functions.

The wrapper supports both logging and tracing. Tracing is the tracing of MPA messages in CLIM windows on the screen, generally there is one such window for each agent. Tracing is useful for demonstrations and following the flow of the system during execution, but the trace cannot be saved. Logging, on the other hand, produces a log that can be saved to a file or inserted in a mail message.

By default, the MPA wrapper functions translate the content field of messages to cl-user package to avoid package problems. All agents should have cl-user package defined, and the receiving agent must have any packages defined that occur in the content of a received message. Optional arguments allow this feature to be disabled in cases where the content is already in an appropriate package. Package translation produces copies of the content, so disabling it can save consing.

The following functions and variables compose the application programmer's interface (API) for the MPA wrapper. All symbols are in MPA package (unless otherwise specified).

A.1 Global Variables

`cl-user::*mpa-substrate*` [*Variable*]

This variable denotes which communication substrate should be loaded with MPA. The default value is `:kqml`, but `:oaa-kqml` also works with some restrictions.

`*log-stream*` [*Variable*]

This variable is the stream used for logging by the wrapper functions. Initially, this is `*debug-io*` which is the initial xterm or Emacs window in which the LISP job was run.

`*pcd*` [*Variable*]

This variable stores the PCD for the agent.

`*log-msgs*` [*Variable*]

This variable controls logging globally. When it is NIL no logging output is produced. When it is T (the default), logging is enabled and the `*unlogged-perfs*` variable and other agent-specific variables control the amount of logging.

`*unlogged-perfs*` [*Variable*]

The value of this variable can be a list of plan performatives which are not logged.

`*default-msg-trace-window*` [*Variable*]

This variable controls tracing globally. When it is NIL (the default), tracing is disabled. If there is only a single KQML agent in the image, this variable should be set by calling the function `start-tracing`. If there are multiple pseudoagents in the image, a `let` binding should bind this variable in each pseudoagent to a window produced by the function `create-msg-trace-window`.

`*abbrv-msg-trace-mode*` [*Variable*]

When this variable is nil, full messages are printed in the trace; when non-nil, abbreviated versions of messages are printed. The default value is T.

`*abbrv-max-string-content-length*` [*Variable*]

In abbreviation mode, this variable gives the maximum number of characters that will be traced for the content field; NIL means no limit. The default value is 50.

`*abbrv-print-length*` [*Variable*]

This variable specifies the value of `cl:*print-length*` used in abbreviation mode. The default value is 7.

`*abbrv-print-level*` [*Variable*]

This variable specifies the value of `cl:*print-level*` used in abbreviation mode. The default is 3, which traces one level deep into the content field.

The following variables are used to customize the behaviour of the planner for a particular domain (although other agents may want to use them). They are described in detail in Section 9.3.2, and can be overridden by messages sent to the planner.

`*plan-to-server*` [*Variable*]

This variable serves two purposes: when it is nil, it disables all sending of information to the plan server; else, it is the name of a view and the planner sends that view of the plan to the plan server server. The default value is :task-network (see Section 9.3.2).

`*all-levels-to-server*` [*Variable*]

When the plan is completed, there is an option for sending all the action networks of a plan to the plan server as a single plan-level update. This variable is the name of a view to use for such an update. The default value is nil, which means no all-levels view is sent, but only the view for the final action network (as determined by `mpa:*plan-to-server*`).

`*monitors-to-server*` [*Variable*]

This variable determines how and if the monitors are sent to the plan server after a final plan is found. A value of nil disables sending monitors (as does a nil value for `mpa:*plan-to-server*`). The other allowable values are the two views for monitors, :monitors and :monitor-filename.

`*draw-plan-after-reply*` [*Variable*]

If non-nil, this variable causes the Search Manager to draw each action network it generates in a level-by-level expansion, or the final action network for a Generate-plan request. The default is nil, because drawing can slow down the planning process for large plans.

A.2 Functions for Agents

`start-agent` *name &key trace log ans-host* [*Function*]

This function registers the current image as an agent with the given name. When non-nil, sexptrace enables tracing and gives the label to be used for the trace window that will be created by `start-tracing`. `Trace` defaults to "MPA Message Trace". `Log` specifies the value for `*log-msgs*`, and defaults to t. `Ans-host` specifies the internet address of the machine where the name server is running, and defaults to "wedge.ai.sri.com".

`kill-agent` [*Function*]

This function unregisters the currently running agent and calls `stop-tracing` to destroy its tracing window.

`agent-for-role` *role* [*Function*]

This function is used to retrieve the name of the specific agent that is fulfilling `role` in the PCD. For example, sending a message to the plan server is done as follows:
`(mpa:send-msg msg (mpa:agent-for-role :plan-server))`

A.3 Functions for Messages

`send-msg` *msg agent &optional log window* [*Function*]

This function sends the given message to the given agent, with tracing and logging (of both sending and any response) as directed by global variables and optional arguments. If `log` is t (the default), the message and the reply are logged. If `log` is :skip-reply, the message is logged, but not the reply. `Window` specifies the window to use for tracing, defaults to `*default-msg-trace-window*`, and can be used to disable a default trace or trace to a different window than the default.

`make-msg` *c-perf content &optional translate-pkg? reply-with* [*Function*]

This function makes an MPA message with the given communication performative and content field. If `translate-pkg?` is non-nil (the default), the content field is translated to cl-user package. If a reply is desired, `reply-with` should be non-nil (the default is NIL). Tracing is done as directed by the global variables.

`make-response` *val p-perf &optional error orig-msg*
cons-it translate-pkg log [*Function*]

This function makes an MPA message that is a response to an incoming message, with tracing and logging as directed by global variables and optional arguments. `Val` is the value computed for the response, and `p-perf` is the plan performative. The content field of the reply has `p-perf` as its car and `val` either as its cdr (if `cons-it` is non-nil) or as its cadr (the default).

If the reply represents an error, `error` should be non-nil, preferably a string explaining the error. In this case, the Error communication performative is used in the reply, and a :error keyword is added to the content field. The value of `orig-msg` is optional and is used to add information to the log. If `translate-pkg?` is non-nil (the default), the content field is translated to cl-user package. If `log` is non-nil (the default), the generated reply is logged. Tracing is done as directed by the global variables.

`make-response-key` *val p-perf &key error orig-msg cons-it translate-pkg log* [*Function*]
This function is a keyword version of `make-response`.

`make-ok-response` *msg &optional log-receipt* [*Function*]
This function generates a :ok response to an incoming message, with tracing and logging as directed by global variables and optional arguments. If `log-receipt` is non-nil (the default), the receipt of the incoming message is logged.

`unknown-perf-response` *p-perf &optional msg* [*Function*]
This function generates the standard reply message for an unknown plan performative error. If given, the incoming message, `msg`, is used to add its communication performative to the error message. The incoming message must be given it it comes from an agent written in Java.

`always-reply` *error plan-perf reply-content msg* [*Function*]
This function makes and logs a response to `msg` whenever a reply is requested, returning nil if no reply is requested. `Error` should be non-nil only for unknown performative errors, and causes `unknown-perf-response` to be called. Otherwise, a list of (`plan-perf reply-content`) is used as the content of a reply, except that if reply-content is a cons whose car is :error, then `plan-perf` is consed onto the `reply-content` list.

`translate-pkg` *s-exp pkg* [*Function*]
This function returns an s-expression that is equal to `s-exp` except for the packages of symbols. All non-keyword symbols in the returned value are in the package specified by `pkg`, which can be a package object, a symbol, or a string. For lists, this function returns a copy of `s-exp`.

`content-field` *msg* [*Function*]
This function returns the content field of the given message.

`reply-field` *msg* [*Function*]
This function returns the reply-with field of the given message.

`sender-field` *msg* [*Function*]
This function returns the sender field of the given message.

A.4 Functions for Tracing and Logging

The functions in this section are lower-level functions used by functions already described. Their use is only necessary when multiple pseudoagents exist in an image, or to customize the default tracing and logging. Functions whose names begin with `log` do both tracing and logging.

`create-msg-trace-window` *&optional label* [*Function*]

This function creates a CLIM window with the given label, which defaults to "MPA Message Trace".

`start-tracing` *&optional label* [*Function*]

This function creates a trace window (if necessary) with the given label, which defaults to "MPA Message Trace" , and binds it to `*default-msg-trace-window*`.

`stop-tracing` [*Function*]

This function can be called at anytime to stop tracing in the current agent. The default trace window is destroyed and `*default-msg-trace-window*` is set to nil.

`clear-tracing` *&optional (frame *default-msg-trace-window*)* [*Function*]

This function can be called at anytime to clear a trace window. This is useful for resetting agents. The default is to clear the `*default-msg-trace-window*` of the current agent.

`log-receipt` *msg &optional p-perf* [*Function*]

This function traces and logs a message that has been received, as directed by global variables. It is usually called inside a message handler. The plan performative `p-perf`, if given, is used to conditionalize logging and tracing using the function `log-for-perf?`. If `p-perf` is omitted, the message is logged.

`log-response` *msg &optional p-perf orig-msg* [*Function*]

This function traces and logs a message that is a response to an incoming message (`orig-msg`), as directed by global variables. This function should not be used when `make-response` is used. The plan performative `p-perf`, if given, is used to conditionalize logging and tracing using the function `log-for-perf?`. This function is useful when responses are computed in many different places and one wishes to log the response in a central place (e.g., the handler for the incoming message).

`log-for-perf?` *p-perf* [*Function*]

This function returns a non-nil value when the given plan performative should be logged, using the value of `*unlogged-perfs*`. A null performative is always logged.

A.5 Miscellaneous Functions and Examples

The LISP wrapper also includes some non-API functions that are useful either as functions or as examples of the use of API functions. The function `ping-msg` creates a :ping message, using the name of the current agent as the sender. Three functions can be used to generate messages with the required syntax for the :annotation plan performative:

`insert-annotation-msg query-annotation-msg delete-annotation-msg`

For example, the following call generates a message that, when sent to the plan server, will insert :level-complete and :plan-ok? annotations for task T-1, plan P-3, and action network AN-1:

```
(mpa:insert-annotation-msg
    '((:level-complete) (:plan-ok?))
    T-1 P-3 AN-1)
```

The function `basic-annotations` takes a complete message that is a reply from the plan server for a query on :annotation, and returns the basic annotations from the content field.

The function `make-msg` should be used to make responses without logging and tracing. Responses can then be logged in a central place (e.g., the handler for the incoming message) using `log-response`.

B C Wrapper and Agent Library Documentation

We developed a library of C functions that can be used to quickly implement a C-language wrapper for an existing "legacy" agent. This library provides an asynchronous MPA message handler that invokes the executable image for the legacy agent as requested, and returns the necessary responses. The handler must be programmed to translate the specific incoming messages into suitable invocations of the legacy agent (e.g., given appropriate command line arguments).

The library includes a set of functions for parsing LISP s-expressions, which facilitates interaction between LISP-based and C-based agents, and allows the C agents to handle the LISP-like MPA message syntax. A debug flag can be turned on or off by the developer to trace the parsing and message handling functions. (The hyphens in the function names are generally underscores in C.)

Implementing agents: To implement a C-based agent, the developer must write a C program that registers the agent with the KQML name server (using the KQML function `kqml-initialize`) and registers each handler (i.e., each type of performative the agent understands) (using the KQML function `kqml-register`). The handlers must be defined using the handler interface described in the KQML manual.

Implementing handlers: SRI has written a Tachyon-Server agent using this C wrapper. This agent serves as a template for handlers. The basic structure of a handler for an MPA agent is as follows:

- Trace the received message using `mpa-trace-msg` (this library function provides the same functionality as the LISP wrapper, printing a short description of a message that is being sent or received).
- Build a default reply message. Typically, this contains a Reply performative and the initial part of the content field for an error message.
- Initialize the parser by calling `init-lisplex`.
- Parse the content field of the received message. For LISP messages, this normally involves using `parse-token` to look for an open parenthesis, then repeatedly parsing and handling keywords and their associated values until a close parenthesis is reached (or the end of the stream is reached, which would be an error). The particular keywords and values depend on the message type being handled.

- Invoke the legacy agent using the `run-program` function with the appropriate arguments (which were extracted from the token stream in the previous step).
- Construct, trace, and return the reply message.
- If at any point an error is encountered, the library function `return-error` provides a simple interface for returning an error message in the correct format.

Parsing LISP expressions: We wrote a parser using the UNIX utility `lex` that parses the content of a message (which is received as a string by the C handler) into a token stream. Each token corresponds to a LISP symbol, keyword, number, variable name, quoted string, or parenthesis. In addition, the library includes a set of functions to make it easy to decompose this token stream into an expected sequence of keywords and values. The function `parse-token(type)` returns the next token in the stream if it is of the named type; otherwise, it returns an error. The `match-keyword(token, kwd)` function checks to see if the token matches a named keyword.

Invoking legacy agents: The function `run-program` mimics the LISP run-program function; it takes the pathname of the agent executable and a list of arguments, and invokes the agent using an external system call.