2

RADC-TR-90-404, Vol IX (of 18)
Final Technical Report
December 1990

A PLANNER SYSTEM FOR THE APPLICATION OF INDICATIONS AND WARNING

Northeast Artificial Intelligence Consortium (NAIC)

Sergei Nirenburg

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

This effort was funded partially by the Laboratory Director's fund.

Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

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

RADC-TR-90-404, Volume IX (of 18) has been reviewed and is approved for publication.

APPROVED: Sharon M. Walter

SHARON M. WALTER
Project Engineer

APPROVED: Raymond P. Urtz Jr

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER:

BILLY G. OAKS
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COES) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

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

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)

2. REPORT DATE
December 1990

3. REPORT TYPE AND DATES COVERED
Final Sep 84 - Dec 89

4. TITLE AND SUBTITLE
A PLANNER SYSTEM FOR THE APPLICATION OF
INDICATIONS AND WARNING

5. FUNDING NUMBERS
C - F30602-85-C-0008
PE - 62702F
PR - 5581
TA - 27
WU - 13
(See reverse)

6. AUTHOR(S)
Sergei Nirenburg

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Northeast Artificial Intelligence Consortium (NAIC)
Science & Technology Center, Rm 2-296
111 College Place, Syracuse University
Syracuse NY 13244-4100

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Rome Air Development Center (COES)
Griffiss AFB NY 13441-5700

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
RADC-TR-90-404, Vol IX (of 18)

11. SUPPLEMENTARY NOTES (See reverse)
RADC Project Engineer: Sharon M. Walter/COES/(315) 330-3577

This effort was funded partially by the Laboratory Director's fund.

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

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The Northeast Artificial Intelligence Consortium (NAIC) was created by the Air Force Systems Command, Rome Air Development Center, and the Office of Scientific Research. Its purpose was to conduct pertinent research in artificial intelligence and to perform activities ancillary to this research. This report describes progress during the existence of the NAIC on the technical research tasks undertaken at the member universities. The topics covered in general are: versatile expert system for equipment maintenance, distributed AI for communications system control, automatic photointerpretation, time-oriented problem solving, speech understanding systems, knowledge base maintenance, hardware architectures for very large systems, knowledge-based reasoning and planning, and a knowledge acquisition, assistance, and explanation system.

The specific topic for this volume is the design of a planning system for an Indications and Warning (I&W) application.

14. SUBJECT TERMS
Artificial Intelligence, Planning, Indications and Warning, I&W

15. NUMBER OF PAGES
98

16. PRICE CODE

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

Block 5 (Cont'd)   Funding Numbers

PE - 62702F	PE - 61102F	PE - 61102F	PE - 33126F	PE - 61101F
PR - 5581	PR - 2304	PR - 2304	PR - 2155	PR - LDFP
TA - 27	TA - J5	TA - J5	TA - 02	TA - 27
WU - 23	WU - 01	WU - 15	WU - 10	WU - 01

Block 11 (Cont'd)

This effort was performed as a subcontract by Colgate University to Syracuse University, Office of Sponsored Programs.

A-1

Planner System for the Application of Indications and Warning

Report Submitted by:
Sergei Nirenburg
Colgate University

Table of Contents

1. Introduction

The Colgate University project in the framework of the Northeast Artificial Intelligence Consortium (NAIC) was devoted to the design of a planner system for the application of I&W (Indications and Warning). The specification of the task evolved from the early direction of intelligent database management toward the emphasis on problem-solving activity. The task of the project was two-pronged:

A. Design of a system to
   1. obtain as input messages concerning events in a model of a real-life subworld;
   2. 'understand' these events by detecting what *plans* they are parts of and, whenever applicable, what *goals* are pursued by the instigators of these events;
   3. produce (suggestions for possible) plans of action necessary in connection with the situation in the world.

B. Implementation of this system for the world of I&W.

This general task included a large number of subtasks, many of which require significant research effort. We concentrated on designing the mechanisms and knowledge bases for the problems of plan recognition (a part of 'understanding' in 2. above) and plan production (in the framework of 3. above). We excluded from our consideration the problems of perception (speech, graphical, or visual); the problem of understanding natural language inputs (that is, understanding the contents of these messages), as well as actual performance of plans suggested by our system.

The conceptual background of this effort is described in some detail in Section 4. The historical background of the project and the way in which it merges with other efforts in the Consortium is briefly discussed here. Cooperation with other research teams within the Consortium, especially with the University of Massachusetts project led by Victor Lesser and Bruce Croft, led to the state of affairs where plan understanding and plan production has become the main thrust of the research effort at Colgate. It was decided, in consultations with the project monitors at RADC, that the natural language aspects of the task would be postponed for later consideration.

2. Strategy

We have taken a concentric approach to the task of designing and implementing the planning system. In other words, we decided to produce an implementation for every design version of our system (called, for historical reasons, POPLAR). As our study of the problem of knowledge-based automatic planning progresses, newer versions of the system will appear.

In what follows we describe in succession the design peculiarities and the implementation characteristics of the two versions of our system (POPLAR 1.3 and POPLAR 2.0) that were developed. Goals were set for the implementation of the next version of the system (POPLAR 3.0). This last version would have introduced very substantial changes to the overall design, and would have been addressed in the final year of the project if it had continued.

3. Technical Content

3.1. POPLAR 1.3

3.1.1. Introduction

This section presents an overview of a cognitive modeling system centered around a personality-oriented planner, and then describes in detail the types of knowledge it uses to make control decisions. POPLAR is a model of an intelligent actor capable of planning sequences of control and domain actions in a simulated world that exists independently of the planner. The world is a simplification of the 'Dungeon' computer game environment. The actor makes control decisions on the basis of situational knowledge as well as its personality characteristics (character traits, physical and mental states) and its belief about personality of other cognitive entities in the world. POPLAR is a step towards an AI system whose behavior is psychologically justified and can provide the basis for an experimental testbed in cognitive modelling.

3.1.2. Setting the Stage

The POPLAR planner is a component in a model of an intelligent actor. It is an approximation of the human actor in that:

i) like humans, it possesses multiple goals with associated plans;
ii) like humans, its control decisions depend upon multiple sources of information, e.g. input from the 'objective' world, its permanent character traits, its temporary physical and mental states, and past experience;
iii) like humans, it is immersed in an 'objective' world, changes in which can be introduced not only by the actor, but also by events beyond the actor's control, making it necessary to deal with non-monotonicity.

We believe that the essence of an intelligent actor's cognitive ability is best described in terms of the following loop:

1) perceive input stimuli (sensory, proprioceptive, or mental);
2) generate goals connected with these stimuli;
3) schedule the most important goal instance for the given period of time: the one to which the actor's cognitive resources are allocated;
4) choose (occasionally, create) and
5) execute plans to achieve this goal, including the performance of physical, verbal, or mental actions that are components of these plans. Executions of the loop provide continuous change and stimulation at several levels. Physical actions introduce continuous change and stimulation at several levels. Physical actions introduce changes in the objective world. Verbal actions can provide sensory input for other

intelligent actors in the world. Mental actions introduce changes in the world of the actor himself (his event memory and beliefs). So, the actions by the actor and other actors in the objective world change this world, and therefore, provide new inputs for the system.

POPLAR 1.3 offers a solution to above loop components 2), 3), the non-creative part of 4), and the mental action part of 5). The visual perception portion of 1) and the physical actions of 5) are **simulated** through interaction with the human user of POPLAR.

In the current implementation there is no natural language capability (i.e. the verbal behavior of 1) and 5) are not addressed). Nor do we tackle in any complete and principled manner the extremely complex problem of learning (one facet of which is the creative part of 4).

The central cognitive and architectural points that distinguish POPLAR 1.3 are, in addition to i) - iii) above, as follows:

A. The choice of the type(s) of knowledge for scheduling (cf. 3 above) and selecting (cf. 4 above) activities. We proceed from the assumption that in a non-trivial world these operations should be based on a psychologically justified model of human cognitive behavior. This property makes POPLAR 1.3 personality-oriented, i.e. provision is made in the present model for introducing personality factors that influence goal generation and plan selection.

B. Decisions concerning the organization of metaknowledge that monitors and directs the cognitive processes of goal generation and plan selection. POPLAR 1.3 represents such metaknowledge in the same framework as the domain plans (top-level, intermediate and primitive). This allows them to be processed by the same reasoning mechanism.

3.1.3. The Conceptual Architecture of POPLAR 1.3.

The conceptual architecture of POPLAR 1.3, as presented in Figure 1, consists of the following modules:

1) the objective **world**, information from which and from

2) the **regulatory system** of the actor (cf. Norman, 1981), where the non-cognitive knowledge about the actor's character and physical and mental states is stored, is obtained by

3) the **sensor**, which processes this input and produces, in the **short-term memory** (STM) of an actor,

4) the **snapshot**, in which the objects currently perceived by the actor are stored, with their parameters, to be scanned by

5) the **goal generator** component of the reasoning mechanism (the **cognitive** module) which produces

6) the list of **candidate goals**, that contains all the goal instances that the actor has at a certain time, including the ones added after the new input was processed. In making its decisions, the **goal generator** uses the data stored in

7) the actor's **long-term memory** (LTM), which contains knowledge about

   a) the **beliefs** the actor has about

   -- **objects** in the objective world, including self-beliefs

   -- actor's **goals**

   -- domain-specific and metalevel **processes** (stored as **plans**)

   b) the acquired **values** the actor has about these beliefs: what is more important, when and why, etc.

   c) the **event** memory that embodies past experience.

8) The **scheduler** component of the reasoning mechanism chooses (schedules) a goal instance in the list of candidates and selects the appropriate plan for its achievement. The **executor** component of the reasoning mechanism then attempts to execute the plan. Lower-level primitive plans are, in fact, actions that are performed by

9) the **output** module; these actions can introduce changes into the world, into the list of candidate goals and the long-term memory.

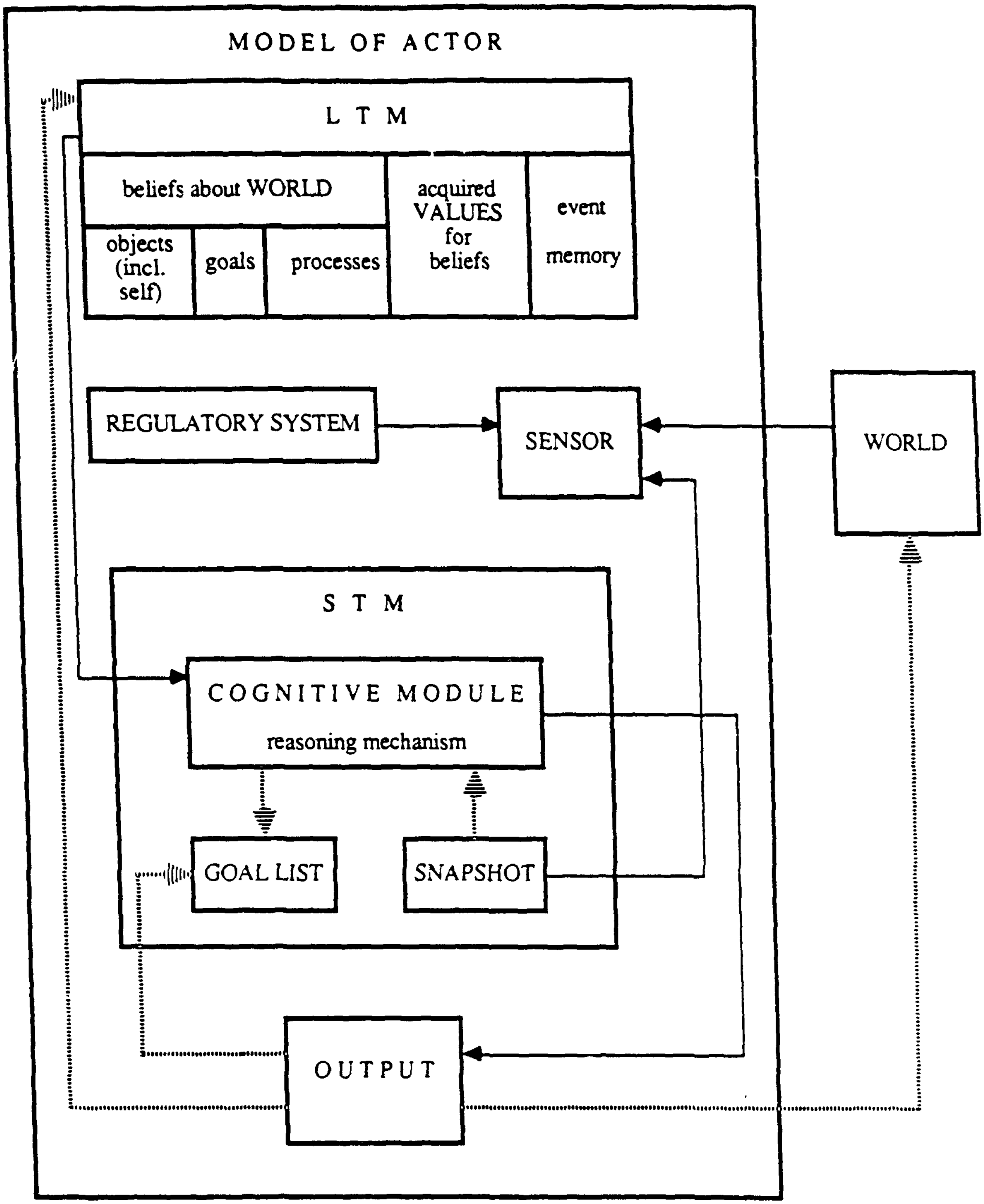

Figure 1. The conceptual architecture of POPLAR.

3.1.4. The Implementation.

POPLAR 1.3 is an implementation of the above conceptual schema in a concrete application domain. It has been implemented in PEARL (Deering et al., 1981) which runs in Franz Lisp under Unix 4.2.

The world in which POPLAR 1.3 is immersed is reminiscent of that of the well-known 'Adventure' or 'Dungeon' games. We represent a cave in which POPLAR's actor can find and react to enemies, treasures, tools, weapons, food and other objects. It is important to understand, however, that POPLAR 1.3 is **not** a game-playing system. We are in the process of applying the system in a different domain (the office world).

At present POPLAR's actor is supplied with three basic goals:

1) 'Don't get killed', dubbed Preserve-Self-1 or PS1

2) 'Don't die of hunger, thirst or fatigue', Preserve-Self-2 or PS2

3) 'Collect as much treasure as possible', Get-Treasure or GTR.

In POPLAR 1.3 the system is making the decisions about what to do next, while it is the responsibility of the user to provide it with input and means for verification of success of actions. The user, therefore, provides the testing ground for the system's empirical experience in the world.

With this caveat in mind, let us see how POPLAR 1.3 is organized to allow its actor to 'act' in this environment.

3.1.5. The System Architecture of POPLAR 1.3.

POPLAR 1.3's system architecture (Figure 2) represents the conceptual architecture of Figure 1 with implementation restrictions superimposed.

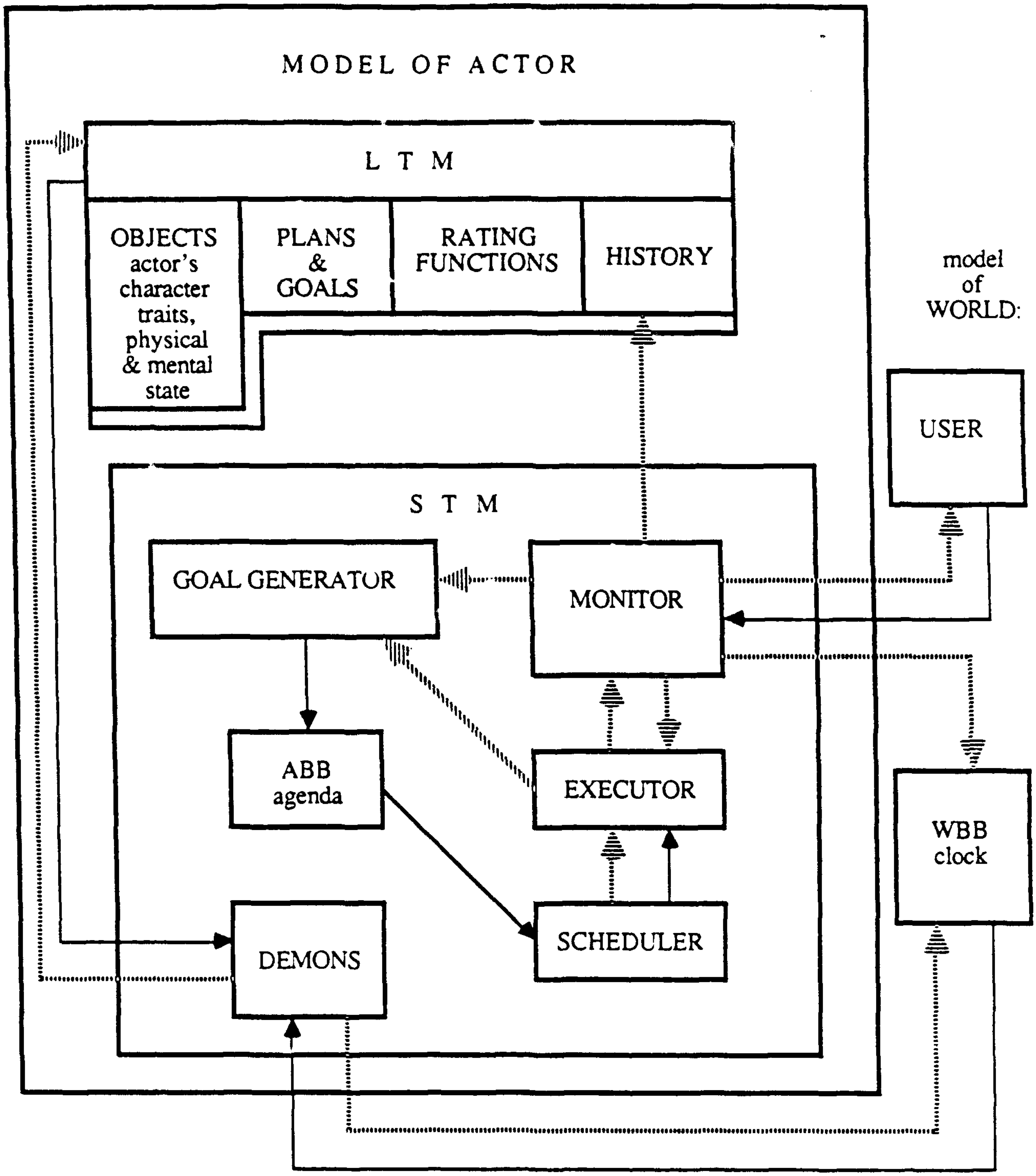

Figure 2. The system architecture of POPLAR 1.3

In POPLAR 1 3 the role of the objective world including the provision of its rules, 'the laws of nature', is assumed by the human user/experimenter. The user also interactively introduces and removes objects in the cave and modifies their parameters. (In future versions we intend to implement ongoing changes in the objective world generated by the operation of if-added demons on a World Blackboard (cf. 3.1.5.2.1).

The user also either permits or forbids certain primitive operations to simulate the actor's pragmatic experience. For example, the user might forbid the actor to pick up an object that is 'too heavy' but previously believed by the actor to be manipulable. This natural state of affairs underscores the difference between the objective world and the world of POPLAR's actor and his beliefs. It is also a means of modeling mistakes (a necessary first step in trying to learn how to recover from them).

The sensor and the output block are simulated in POPLAR 1.3's monitor (though mental actions are performed by demons (see below).

When the user decides to add an object to the current world, it does it by listing it on the **world blackboard** (WBB), the data structure interfacing the objective world and the world of POPLAR's actor. WBB also contains a clock which guides all temporally spread processing.

The STM of POPLAR's actor has the reasoning mechanism (the monitor and the executor with their associated bookkeeping functions, demons) permanently connected with it†. STM contains one-instance metaplans: the goal generator and the scheduler. STM also includes the **actor blackboard** (ABB), which contains slots relating to the current state of POPLAR actor's activities, including notably the **agenda** of activated goal instances.

POPLAR actor's LTM contains his objects, plans, rating functions and history. Cf. a detailed discussion in 3.1.5.1.

POPLAR actor's knowledge about his own regulatory system and that of others is linked in the implementation with the representation of these objects in LTM. In addition to knowledge about objects, LTM contains knowledge about plans, history of processing and proper scheduling and selection.

Let us discuss the components of POPLAR 1.3 in greater detail.

3.1.5.1. LTM.

3.1.5.1.1. Objects.

Several typical object frames and the semantics of their slots are described in Appendix 1. The choice of character traits is at present empirical. However, in parallel to implementing POPLAR 1.3, we have been conducting extensive psychological experiments seeking to establish the set of 'primitive' personality characteristics and their

† The monitor, the executor and the bookkeeping functions stand out among the components of STM in that they are not 'conscious' functions; the actor performs them 'instinctively', while of other elements of STM the actor is consciously aware

mapping into more complex notions that are used by intelligent actors in personality-based decision-making. A separate set of experiments will determine the primitives for specifying mental states of the actor.

3.1.5.1.2. Plans.

POPLAR's knowledge about the dynamics both in the objective world an in the actor world is represented as a set of declarative structures called plans.

Plans in POPLAR 1.3 are classified into several groups (cf. Figure 3).

	TOP-LEVEL	INTERMEDIATE	PRIMITIVE
DOMAIN PLANS	PS1 PS2 GTR	FIGHT, EAT, GET, etc.	move, take, find, etc.
METAPLANS	GG as		as, gg-input, etc.

Figure 3. Classification of plans in POPLAR.

First, there are domain plans that describe actions in the world and metaplans that describe the processes that manipulate other plans. These include such plans as the goal-generator (gg), the plan-selector, the agenda-scheduler (as), etc. Second, there are top-level plans whose instances appear in POPLAR's agenda as representatives of the three main goals; and primitive plans that are no further decomposable into sequences of actions and provide the proper framework (of preconditions, effects, etc.) for their main action.

The plans that are neither top-level nor primitive are called intermediate. Intermediate plans are never scheduled other than in the process of expanding a top-level plan. There are no intermediate metaplans. Also, all of the metaplans are primitive (decomposable), and two of them, at the same time, top-level.

To illustrate the above discussion, consider, for instance the top-level plan of dealing with enemies, such as, in the POPLAR 1.3 objective world, snakes, crocodiles or trolls. The actor can have a number of (intermediate plan) possibilities: to fight, to flee, to hide, to wait and see what happens, etc. All of the above are decomposed into strings of lower level plans (such as get, take, find, etc.), and the process of decomposition continues until all the final decompositions contain only primitive plans (such as, for instance, move or take).

Plans in POPLAR 1.3 are represented in a modified version of the language EDL (Bates et al., 1381; cf. also Croft & Lefkowitz, 1984). The frame for a plan contains the following slots (clauses):

ID	the name of the plan
TOP-LEVEL-FLAG	is this plan top-level?
IS	contains the temporal and causal expansion of the plan
COND	used to pass parameters ('propagate constraints') to lower-level plans upward propagation will be added for the plan recognition task
WITH	specifies the parameters with which the current plan will be processed
CONTROL	contains predicates to choose whether to execute optional steps in the plan this slot has the form of an a-list (<(Control# <s-expr>)>*
PRECONDITIONS	predicates that allow the processing of the current plan to start; differ in principle from CONTROL predicates by being independent of the current context of plan processing
STATUS	one of 'on-agenda', 'executed', 'succeeded', or 'failed'; used for communications with the reasoning mechanism
ACTION-FOR-PRIMITIVE	if plan is domain primitive permission is requested for its completion and the main action is performed (the rest being 'effects')
TIME	number of time cycles the plan takes (only for primitives) -- either integer or s-expression that evaluates to integer
RATING-FUNCTION	scheduling knowledge, see below
EFFECTS	auxiliary (including bookkeeping) modifications accompanying the success of the plan

Figure 4 contains a grammar of the plans implemented in POPLAR 1.3, and Appendix 2 contains annotated examples of POPLAR 1.3 plans.

```
I ::= PS1 | PS2 | GTR | GG ('Goal-Generator') | as ('Agenda-Scheduler')
PS1 ::= FIGHT | HIDE | WS ('Wait-and-See')
PS2 ::= EAT | DRINK | SLEEP
GTR ::= {FIGHT | find} GET
GG ::= gg-input | gg-objects-perceived | gg-physical-states-perceived
FIGHT ::= find {find GET} move attack
HIDE ::= find move
WS ::= do-nothing
EAT ::= find {find GET} ingest
DRINK ::= find {find GET} ingest
SLEEP ::= find do-nothing
GET ::= move take
```

Vertical bars separate disjoined elements; in practice, the 'or'-ed plans are chosen on the basis of their ratings through the application of a special metaplan we call the Plan-Selector, not shown in the grammar;

curly brackets enclose optional plans; the decision whether to execute the optional plan(s) is made on the basis of control functions that are stored in the parent plan and govern the parsing of its IS slot;

plans shown in lower case are primitive.

Figure 4. A grammar of plans in POPLAR 1.3.

3.1.5.1.3. The Rating Functions.

The knowledge that POPLAR's actor has about the relative importance of a top-level goal instance and the relative merits of one plan of action aimed at achieving a goal over another is embodied in the **rating functions**. In the current implementation rating functions are associated with every plan that can serve as parameters in the plan-selector and the agenda-scheduler.

The rating functions calculate a numerical value for a plan, a rating, in all situations where a choice among plans that can be pursued is possible. They draw upon:

a) knowledge of the objects involved in an objective world situation;

b) the character traits, mental and physical states of the actor;

c) the actor's beliefs about the character and current physical/mental state of any other cognitive entity participating in the situation;

d) the actor's event memory, the history of past processing.

Thus, if two actors, Actor1 and Actor2 find themselves in an identical threatening situation (e.g. a snake), but one of them is more courageous (a character trait) and/or is in general not very fearful of snakes (a situational characteristic), the actors may respond to the situation by choosing different plans (e.g. Flee for Actor1 and Fight for Actor2) or even altogether different goals (while Actor1 is likely to choose 'Preserve-Self' against the snake -- because high levels of attention to threats can be expected from actors with low courage values; Actor2 may choose, say, an instance of 'Get-Treasure', because the snake is not serious enough a threat).

The construction of rating functions is an empirical process of gradual refinement. Even without changing the knowledge used by the rating functions one can always manipulate parameters of a function to calibrate its results.

One of the objectives of the psychological experimentation conducted in parallel with this project (cf. Section 4) is to better understand the nature and parameters of t^he rating functions.

Examples of rating functions are presented in Appendix 3.

3.1.5.1.4. History

This part of the actor's LTM contains his memory of past processing. In principle, history can have a very rich structure and be used in a wide variety of ways. Special demon-type functions can be defined, for example, to introduce modifications into the actor's beliefs about objects and processes in the real world based on certain patterns in the event memory†. This is one more location in POPLAR 1.3's architecture where a measure of learning can and is planned eventually to be introduced.

At present the history contains only two types of data: a) the record of all the recursive calls to the executor in the form of paths that the processing took in the grammar of plans and b) a list of the objects (physical or mental) found by all instances of the Find plan; this knowledge is used to retrieve the status and the results of various plan instances. A typical instance of history is presented in Appendix 4.

3.1.5.2. Actor/World Interfaces.

As mentioned above, in the current implementation of POPLAR 1.3 there are two blackboards that facilitate links between the world and the actor.

3.1.5.2.1. The World Blackboard.

WBB is used for introducing new sensory input and managing temporal relations in the system. POPLAR 1.3 has time-triggered demons that automatically update the values of the actor's physical and mental state based on the amount of time he engages in a certain activity.

† An example Suppose that in an internalized plan for fighting crocodiles 'stick' is listed as the best weapon Then during one invocation of the plan Fight (Actor Crocodile Weapon) no stick could be found, so that Actor had to use a gun It appeared that both the results were better and the fatigue increase was smaller After this plan execution was written into the history, a comparison is made (by the above demons) and the old belief about the stick being the best weapon is changed

In their simplest manifestations, the time-related modifications deal with increasing the actor's hunger, thirst and fatigue values at predetermined independent rates. When the value of any of the above parameters becomes greater than a predefined threshold, a message to this effect automatically registers in ABB's 'states-perceived' slot, as a result of which at the next pass of the monitor an instance of Preserve-Self-2 goal will be activated, and the corresponding top-level plan will appear on the agenda.

Temporal knowledge is also used to implement a simple model of attention. A detailed discussion of this mechanism will be deferred till Section 3.1.6.3.

3.1.5.2.2. The Actor's Blackboard.

ABB contains information about

a) the list ('objects-perceived') of object instances that the actor has perceived in the current environment;

b) the list ('states-perceived') of all physical states currently perceived that warrant the attention of the goal generator (e.g. the level of hunger above a threshold);

c) the agenda of all top-level plans (the representatives of the main goals) vying for the attention of the cognitive processor of the actor at any given time;

d) the stack ('current-path') of plans currently being executed (from a top-level plan to a primitive).

In future implementations, specifically when plan recognition will be added to the *repertoire of* POPLAR *and the number of actors inhabiting its world will be allowed to* be greater than one, the number of ABBs in the system may grow to as many as the square of the number of actors. This is because every actor stores his beliefs about other actors' activities in instances of ABB attached to his representation of these other actors. Therefore, each actor theoretically can be aware of all the other actors and contain an ABB for each, including himself.

A typical example of ABB and WBB contents is presented in Appendix 5.

3.1.6. The Algorithms.

3.1.6.1. The Monitor.

The top-level control function of POPLAR 1.3, the monitor, is an infinite loop (our actors do not die -- only if killed by enemies!) which performs the following tasks:

a) it maintains contact with the user (to obtain new input);

b) it starts the executor loop that consists of i) processing new input; ii) scheduling an action; and iii) executing this action

c) it displays selected situations in the world with the help of a (rather simple) graphic interface.

3.1.6.2. The Executor.

The main bulk of POPLAR 1.3 processing is performed by the executor. To understand how POPLAR 1.3 works it is sufficient to trace a cycle of its activities.

The *executor is called many times during one monitor cycle.* First, it processes the goal generating plans using the information obtained by the monitor from the objective world as well as that from the actor's regulatory system. As a result of this stage, the agenda of competing top level plan instances is updated. Second, it executes the agenda scheduler plan select the best candidate plan. Finally, it executes the chosen top-level plan (this involves a number of recursive calls to the executor). When eventually the execution ends, the result of current processing (success or failure) is reported, and a new cycle of the monitor begins.

Omitting a few overly technical details, we can describe the activities of the *executor generally as follows:*

a) obtain a plan to process; if it is not a plan **instance** (the agenda holds only plan instances, e.g. 'CTR19'; whereas **is** clauses of plans are formulated in terms of plan **types**, e.g. 'Find'), create a new instance of this plan;

b) check the plan's **preconditions** clause; if preconditions do not hold, report failure and its reason and exit; otherwise,

c) expand the plan by considering its **is** clause: call the **is** clause parser;

c′) if the **is** clause is 'primitive', then **action-for-primitive** is performed (most often this is a request to the 'laws of nature', the user, to allow an update in the objective world, e.g. a move by the actor; if the permission is given the processing proceeds as specified in e′) below; if the action is not allowed the processing proceeds as in e″). (Let us repeat that the semantics of this situation is that the actor's beliefs about the objects and/or plans and/or values are somewhere wrong, as a result of which some indication of imminent failure must be given to prevent the 'automatic' success of most planners in situations where the internalized preconditions of a plan hold.)

c″) if the IS clause is not 'primitive' the parser has to make specific control decisions: i) whether to execute an optional subpath in the IS clause; ii) which of any possible number of disjoined subplans to choose for fulfilling the current plan. (The ability to choose one of a number of 'shuffled' subplans (those that can be fulfilled in any temporal order) will be added to POPLAR in near future.) The knowledge about whether to execute an optional subpath is encoded in the **control** slot of the plan whose **is** slot is parsed. The knowledge selecting one of disjoined subplans is contained in the plan-selector metaplan and the **rating function** slot of the current plan. Once it becomes clear what member of the **is** clause should be processed first, the executor

d) calls itself recursively with this plan; this event is recorded on ABB, specifically, in a data structure called **current-path**; the old content of **current-path** is added to **history**.

e′) if an **is** clause is processed to its end (cf. the special case of 'primitive' in c′) above), the **status** slot of the plan is set to 'succeeded' and the **effects** clause is evaluated;

e″) if for some reason the **is** clause cannot be processed to its end, the **status** slot is set to 'failed' and

f) this information is communicated to the parent plan; the current plan is discarded from the **current-path** stack, and the processing of the **is** clause of the parent resumes. When, eventually, the outcome of the top-level (and bottom-of-stack in **current-path**) plan becomes known, then

g′) if it succeeded, then the **effects** clause is evaluated and the corresponding top-level plan instance is removed from the agenda (and added to **history**);

g″) but if it failed, then, assuming that the need that had spawned this goal has not been satisfied, the executor creates a new instance of the same top-level plan and adds it to the agenda instead of the failed one (which goes to **history**).

h) a new cycle of the monitor starts.

3.1.6.3. Modeling Attention.

The previous section described the normal flow of control in a monitor cycle. In real life, however, an actor can hardly have the luxury of being able to finish the processing of a top-level plan without taking in new information about the objective world. In future implementations of POPLAR the temporal relations among plans will be elaborated to include the many possibilities of concurrent processing (cf. Allen, 1983a, for the description of a model of time that can be adapted for use in our model; cf. also McCue & Lesser, 1983) for a temporal logic in the POISE system).

POPLAR 1.3 reacts to this problem as follows. When a top-level domain plan is chosen from the agenda and passed over to the executor, its rating is used for calculating the number of time cycles this plan will be allowed to execute without being interrupted. The more 'important' the plan (i.e., the higher its rating) the longer it is allowed to execute uninterrupted. This current programming device is a rough simulation of the actor's concentration or attention to the task. Intuitively, the more immersed one is into a task, the less one would be inclined to be distracted by new sensory inputs. It is obvious that character traits and physical/mental states affect the ability to concentrate.

When an interrupt occurs, the entire **current-path** is suspended; the instance of the top-level plan is deleted from the agenda and another instance is created and added to it (the new instance reflects the knowledge of the stage at which the processing was suspended; **history** is used for this purpose). Then the monitor starts a new cycle.

3.1.6.4. An Example.

Suppose we want to test the performance of POPLAR 1.3 in the following situation of the world. We want to put the actor in a cave with a rock, a snake and an apple and to set its hunger well above the detecting threshold.

POPLAR 1.3 acts as follows:

a) asks the user whether he wants to remove certain objects from the world; we do not, so we answer in the negative;

b) asks the user whether he wants to change any of the properties of the objects already present in the world; this is the time to input the (high) value of actor's hunger;

c) asks whether the user wants to add new objects to the world; we do; since our perception module is simulated, we submit prefabricated instances of objects to POPLAR 1.3; we write: (rock1 snake1 apple1).

d) adds the above object instances to ABB.objects-perceived. Since snakes spawn the need for protection (by virtue of their being descendants of 'creature'), the goal Preserve-Self-1 is activated (by the gg-input plan) and an instance of its corresponding top-level plan, PS10, is added to ABB.agenda (which already contain the unique instance of the Agenda-Scheduler plan that resides there permanently); appropriate messages are issued by POPLAR 1.3;

e) detects, through gg-states-perceived, the actor's hunger; 'hunger' is added to ABB.states-perceived and an instance, PS20, of the top-level plan of the Preserve-Self-2 goal is added to ABB.agenda; appropriate messages are issued;

f) since no objects had been present in the world before, and, therefore, no changes to their properties could be introduced, gg-objects-perceived will not be needed in this case, a message to which effect will be issued;

g) at this point ABB.agenda is (agenda-scheduler PS10 PS20); the monitor calls the executor with the scheduler plan, as a result of which the two domain plans receive ratings. Suppose now that PS20's rating is higher (because the actor is very hungry and at the same time not too afraid of snakes); this being the goal choice,

h) the scheduler is called with PS20(Actor hunger); checks its preconditions (empty!) and expands its **is** clause; the plan-selector, using the rating functions in the plans Eat, Drink and Sleep, decides to select Eat; an instance of Eat, Eat0(Actor) is created and pushed onto **current-path**

i) Eat0's preconditions are checked (empty!), and its own **is** clause is expanded; this means creating a new instance of Find, Find0(Actor food Actor.inventory), -- that is, first the actor wants to check whether he is carrying some food;

j) the controll predicate chooses whether to execute the optional Find and Get plans; the predicate essentially returns 'true' if the previous Find failed; the optional subpath corresponds intuitively to the situation when the actor looks around him trying to find some food; suppose now that Find0 fails; in this case,

k) Find(Actor food ABB.objects-perceived) is executed; Find's IS clause is 'primitive'; its **action-for-primitive** is to record the object found; Find1 finds apple1;

l) next, GET0(Actor, apple1) is created and pushed onto **current-path**; this instance's **is** clause consists of Move followed by Take; (in reality, Get has three parameters, the third being the indication of the time that the actor can spend on retrieving the object -- this is very handy as a precondition if, for example, an adversary can reach the desired object first!)

m) Move0(Actor Apple1) is created and pushed onto **current-path**; Move is a primitive plan, so its **action-for-primitive** asks the user for permission for the actor to move to the point where apple1 is. We grant the permission; Move0 evaluates its **effects**, updating the positions of the actor and all the objects in his inventory and sets its **status** to 'succeeded';

n) **current-path** is appended to **history**; Move0 is popped, and the next plan in the IS clause of Get0 is pushed onto **current-path**: Take0(Actor apple1);

o) Take0 is primitive; its processing is similar to the processing of Move0; it succeeds, one of its effects being that apple1 is added to the actor's inventory, and after manipulations with **current-path** similar to those in l), Ingest0(Actor apple1) is sent to the executor;

p) Ingest is primitive; suppose we allow the actor to ingest the apple; then, after the appropriate (and by now familiar) bookkeeping operations, we find ourselves at the point where Eat0 is proclaimed as succeeded; at this point we evaluate its **effects** and pop it from **current-path** (which at the time contains only PS20, known to have succeeded);

q) **effects** of PS20 are evaluated (the hunger level of the actor is decreased, and a message to this effect is issued), and with this PS20 is popped from **current-path**, which remains empty; this signifies the completion of a cycle of the monitor.

3.2. POPLAR 2.0.

3.2.1. Introduction.

This section describes the changes introduced into the Colgate personality-oriented planner in the new version, POPLAR 2.0. A number of technical improvements were made to support new functionality. POPLAR 2.0 runs on a Symbolics 3600 Lisp Machine. This is a companion text to a previous report on POPLAR: S.Nirenburg , I.Nirenburg and J.Reynolds, POPLAR: A Testbed for Cognitive Modelling, Research Report COSC7, Division of Natural Science, Colgate University, June 1985. This document is structured as follows. First we highlight the additions to the functionality of the system. Next we describe the changes in the knowledge representation introduced in POPLAR 2.0. This is followed by a description of the modified algorithms. Finally, we include a discussion of implementation-related decisions and an example run of POPLAR 2.0. Example plan representations in POPLAR 2.0 can be found in the appendix.

3.2.2. Comparing the functionality of POPLAR 1.3 and POPLAR 2.0.

3.2.2.1. Review of Functionality in POPLAR 1.3.

Before talking about the differences between POPLAR 1.3 and POPLAR 2.0 let us recall the planning algorithm of POPLAR 1.3.

The top level function **Monitor**:

```
begin
 repeat forever
   call the function MAINTAIN-WORLD;
    {which obtains from the user information about changes in the
     world and records it}
   for every object in ABB.objects-perceived† do
      if the object is connected to a certain behavioral need (goal)
      and ABB.agenda* does not contain an instance of a top level
      plan to achieve this goal then create an instance of a top
      level plan to achieve this goal (satisfy this need) and
      add it to the ABB.agenda
   for every physical state in ABB.states-perceived** do
      if ABB.agenda does not contain top level plans for achieving
      corresponding goals (preserve self from hunger, thirst and/or
      fatigue, called the MAINTAIN goals) then create an instance of
      corresponding top level plan and add it to ABB.agenda;
   if ABB.agenda is empty then EXIT (end-repeat);

   for every plan in ABB.agenda do
      produce a rate for this plan by evaluating its rating function;
   choose the plan with the top rating;
   call Executor with the top-rated plan
  end.
```

The function MAINTAIN-WORLD:

```
begin
 repeat
    ask the user whether he/she wants to add objects to the world
    if 'yes' then obtain an object instance name and add it to the
    ABB.objects-perceived
  until the answer is 'no';
  repeat
    ask the user whether he/she wants to remove any object instance
    from the world
```

† ABB objects-perceived is a slot on 'Actor-Blackboard' (see Nirenburg et al., 1985) which holds a list of object instances perceived by Actor at a given time

* ABB agenda (see below) is a slot on 'Actor-Blackboard' which holds a list of top-level plans associated with Actor's goals at the moment

** ABB states-perceived (also see below) is a slot on 'Actor-Blackboard' which holds a list of Actor's physical states (hunger,fatigue, etc) perceived at a given moment

```
    if 'yes' then obtain an object instance name and delete it from the
    ABB.objects-perceive
  until the answer is 'no';
  repeat
    ask the user whether he/she wants to change any object's
    attributes
    if 'yes' then obtain an object name, names and new values for
    attributes and record them
  until the answer is 'no'
end.
```

The central function of POPLAR is **Executor**. It is described in detail in Nirenburg et al., 1985. Briefly, the algorithm is as follows:

```
begin
 obtain a plan to process
 if the plan's preconditions do not hold
   then report failure and exit
   else expand the plan by considering its is slot:
    if the is clause is 'primitive'
       then perform its Action-for-primitive and
         if was completed successfully then evaluate that plan's effects
       else for every subplan in the is slot
         call Executor with that subplan
 if is slot is processed to its end
     and the last executed subplan was completed successfully
   then report SUCCESS of the current plan,
       evaluate its effects, EXIT;
   else report FAILURE, EXIT;
end.
```

One cycle of the **Monitor** in POPLAR 1.3 covers the choice and the execution of a single top-level plan. This process lasts N time cycles where N is equal to the number of primitive plans involved. (Note that the plan **Move** is also primitive so that it lasts only one time cycle irrespective of the distance between Actor's starting and end positions.)

Changes in the world are made only once during one **Monitor** cycle (at the beginning). This means that no changes obtained during processing are recorded before the beginning of the next **Moni**t**or** cycle. In other words the world remains **monotonic** during one **Monitor** cycle.

In POPLAR 1.3 an attempt was made to approach the solution of this problem in the following way: when a top-level plan is chosen its rating is used to calculate the number of time cycles that this plan will be allowed to execute without interruption. The more 'important' the plan, the longer it is allowed to execute uninterrupted. When an interrupt occurs, the entire **current-path** (a data structure where the stack of executed plans is stored) is sent to **history**. Then the **Monitor** starts a new cycle and if the same top-level plan is chosen execution starts 'from scratch' even if some of its sub-plans were completed successfully during the previous **Monitor** cycle.

Both problems described above (the possibility to make the world **nonmonotonic** and ability of saving partial results for the further use) are solved in the new implementation of the POPLAR: POPLAR 2.0.

3.2.3. Improvements in POPLAR 2.0.

A major difference between POPLAR 2.0 and POPLAR 1.3 is the separation of the task of executing an entire top-level plan into two different tasks:

a) execution of non-primitive and 'mental-primitive' plans (this is done by **Executor** in POPLAR 2.0) and

b) execution of 'physical-primitive' plans (this is done by **Effector**) (a detailed description of both algorithms see below in Section 3.1.6.).

The new **Executor** performs a top-level plan execution by expanding its **is** slot - considering its sub-plans (i.e. goes down in the plan hierarchy, or in other words, lowers the level of abstraction). When a 'physical-primitive' plan is encountered control is passed to **Effector** with that plan as a parameter. **Effector** performs a 'physical-primitive' plan execution.

This distinction gives an opportunity in future to perform these two tasks in parallel. The idea behind this decision is that in real life people tend to perform physical and mental actions simultaneously (for example, a person can start scheduling weekend activities while driving to his job on a Thursday).

It is postulated that the execution of a 'physical-primitive' plan lasts one time cycle. Therefore a new **Monitor** cycle now also lasts one time cycle. (Note that the primitive plan **Move** is now represented as one 'step' of the Actor.) So, changes in the world could be perceived by the Actor after every time cycle.

When the top-level plan chosen for execution is already partially executed, **Executor** then starts processing at the point where it left off at the previous step. This point could be found by detecting differences between plan **type** name and plan **instance** name Subplans that were already processed are presented by plan instance names in the **is** slot.

3.2.4. Plans in POPLAR 2.0.

For the new version changes were introduced into the plan grammar of POPLAR 1.3. (See Figure 5 for the new version of the grammar.) Note that the most important changes were introduced into the plans that involve **finding** objects (e.g., **Fight**, **Eat**, **Drink**). In POPLAR 1.3 these plans contained 'optional' subplans and a decision whether to execute them was made with the help of what we called 'control' functions.

In POPLAR 2.0 instead of both optional paths and control functions a new 'mental-primitive' plan **Get-Selector** is used.

The **get-selector** plan creates new instances of the **Get** plan: one instance for each object instance that was 'found' by the previously performed LOCATE plan. (for example, Actor is looking for food. Apple1, apple2 and carrot1 are located in the surroundings of the Actor by the plan Locate. Then three instances of the plan **Get** are created: Get1 (object apple1), Get2 (object apple2) and Get3 (object carrot1)). All these instances of the **Get** plan obtain the status 'candidate' and the real-world-flag value of 'no'†. A list of these instances is stored in the **is** slot of the **get-selector** plan for future use. Then one of them is chosen for further execution. The choice is made on the basis of rating. The information on which the rating process is based is contained in the **rating function** slots of the **Get** plans.

When the **Get-Selector** plan is executed for the second or further time, it chooses one of the **Get** plan instances that it had created previously (and stored in its **is** slot). The algorithm for making the choice is as follows: if there is a plan with the status 'suspended', choose *it* for execution, else if there are plans with the status 'candidate' rate them and pick the one with the maximum rate, otherwise report FAILURE.

I ::= P | M | GTR
PS1 ::= FIGHT | HIDE | wait-and-see
PS2 ::= EAT | DRINK | SLEEP
GTR ::= GET
FIGHT ::= FIND move attack
HIDE ::= *locate* move
EAT ::= FIND ingest
DRINK ::= FIND ingest
SLEEP ::= *locate* do-nothing
GET ::= *move take*
FIND ::= *locate get-selector* GET

Plans shown in lower case are physical-primitive; plans shown in *italics* are mental-primitive.

Vertical bars separate disjoined subplans; in practice, the 'or'-ed plans are chosen on the basis of their ratings through the application of a special 'mental-primitive' metaplan **Plan-Selector**, not shown in the grammar.

Figure 5. A grammar of plans in POPLAR 2.0.

The **Plan-Selector** makes a specific control decision as to which of any possible number of disjoined subplans to choose for fulfilling the current plan. The knowledge for selecting one of disjoined subplans is contained in the **rating-function** slots (methods) of the disjoined subplans.

† The instantiation of plan tokens for possible future use is essentially a way of modeling one-step 'look-ahead' The Actor as if thinks about all possible plans at this point and choose the best of them to perform

3.2.5. The Algorithms.

In this section we present the algorithms of the functions that have been changed in POPLAR 2.0.

Executor (P)

a) obtains as parameter a plan, P; P could be either a plan **instance** or a plan **type**[†]. If P is a plan **type** then

  i) create a new instance of this plan

  ii) substitute the plan name P with the name of the new created instance in the **is** slot of the parent plan

  else (i.e. P is a plan **instance**)
  case status
  'failed': exit, reporting failure,
  'succeeded'· exit, reporting success[‡].

b) check the plan's **preconditions** clause; if preconditions do not hold, (are false) report failure and its reason and exit; otherwise,

c) expand the plan by substituting the contents of its **is** clause for the plan itself;
  if the **is** clause is 'physical-primitive' then exit;
  if the **is** clause is 'mental-primitive' then perform its **Action-for-primitive**. If the current plan is either **Plan-Selector** or **Get-Selector** (metaplans that chose the next plan for execution) and **Action-for-primitive** was performed successfully (a plan is selected) call **Executor** recursively with the selected plan. If **Action-for-primitive** failed then report failure and exit.
  if the IS clause is not 'primitive' then call **Executor** recursively with this plan.

  end **Executor**.

Another new function is **Effector**, whose task it is to monitor actual (simulated) execution of a primitive plan.

The algorithm of **Effector** is as follows:

a) if FAILURE was reported by **Executor** then for each plan in the **current-path** set **status** to 'suspended': exit:
  else obtain a primitive (physical) plan instance P (from **Executor**). Perform P's **Action-for-primitive**.

b) for each plan in the **current-path** (including P) do: check **satisfaction-condition**; if it holds then perform that plan's **effects** and set **status** to 'succeeded', else set **status** to 'suspended'.

[†] note that the agenda holds only plan instances, e g 'GTR19', whereas **is** clauses of plans can contain both plan **types**, e g 'Find', and plan **instances** - in case when those plans were already processed once by **Executor**

[‡] This means that plans that were already completed are not executed again

3.2.6. Implementation.

POPLAR 2.0 is implemented in Zetalisp and runs on a Symbolics 3600 Lisp Machine. The new implementation uses the knowledge representation system native to the 3600 Lisp Machine, the Flavors system. Although the overall structure of the knowledge base remains unchanged, a number of internal technical modifications were made.

3.2.6.1. Plan structure.

To maintain the ability to continue execution of a plan from the point it was interrupted, a number of new slots (both instance variables and methods) were added to the basic plan frame (flavor), as specified in the extended EDL of Nirenburg et al., 1985. The syntax of several slots was modified as follows:

WITH

now contains a list of parameters in the form: ((agent <object-instance-name>) (object <object-instance-name>) (instrument <object-instance-name>) (place <position>)(time-for-execution <time>)

SATISFACTION-CONDITION

one or more of world states that become true after this plan is executed successfully. This slot is used for understanding the status of the plan, that is, whether the plan is already completed or not.

REAL-WORLD-FLAG

holds YES when a plan instance is created for immediate execution, and NO if that instance creation is 'look-ahead'. This slot is used by the plan **Get-Selector** (see above).

STATUS

one of 'on-agenda', 'executed', 'suspended', 'succeeded', 'failed' or 'candidate'.

Appendix 6 contains examples of POPLAR 2.0 plans.

3.2.6.2. The Actor's Blackboard.

Two slots were added to the Actor-Blackboard to help trace Actor's behavior:

CURRENT-GOAL

holds the Actor's current goal

CURRENT-PLAN

holds the name of the plan presently executed by Actor.

3.2.6.3. The User Interface.

In POPLAR 2.0 the function MAINTAIN-WORLD which directs the acquisition of information from the user is menu-driven. It contains a menu for choosing an operation (insertion, modification, etc.), a menu for changing an attribute(s) of an object in the world, etc.

The world is represented by the board which contains 200 (10 x 20) mouse sensitive squares. Objects could be moved from place to place or removed from the board using the mouse.

The screen is divided into three separate windows:

a) the Actor's world (see above);

b) a LISP Listener window where all messages generated by the program appear

c) the TRACE window which monitors the current values of the most important parameters and data structures including

    i) the CLOCK of the system

    ii) Actor's blackboard which contains AGENDA of goals, a current plan, a current goal as well as

    iii) Actor's physical state parameters.

All the values are updated immediately after changes were introduced.

3.2.7. An Example.

Suppose we want to test POPLAR's performance in the following world situation. The room will contain, in addition to the Actor itself, a rock, a snake, a sword, a dog and a stick. The positions of all the objects will be as shown in Figure 6. Recollect that the world in which POPLAR operates at the moment is that of an actor in a room with sources of danger, food and treasure. The actor is 'programmed' to plan survival and maintenance of self plus getting as much of the treasure as possible in its possession.

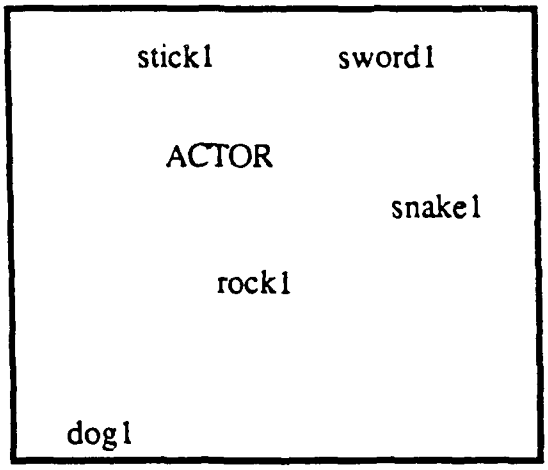

Figure 6. An Instance of the POPLAR World.

POPLAR 2.0 acts as follows:

a) obtains a new input from the user:

i) first it produces a menu containing the list of possible operations of the following types:
creating new object instances,
adding existing object instances to the world or
changing attributes of the object instances in the world.
The user clicks on 'creating new object instances', then

ii) the next menu containing the list of all object types that are predefined in the system is popped.

The user chooses, say, 'snake'. Then

iii) the menu containing the list of the parameteis for 'snake', with default values, is popped, so that the user can make changes there.

The procedure iterates until specifically told to exit, so that the user inserts all desired objects into the world.

POPLAR 2.0

b) processes the new input: since snakes and dogs spawn the need for protection, two instances of the goal Protect (P1 and P2) are created and added to ABB.agenda.

c) checks Actor's physical state (hunger, etc.). Suppose that none of the hunger, thirst and fatigue is above the detecting threshold, therefore, ABB.states-perceived is empty.

d) the function Agenda-Scheduler is called. The goal P1 ((agent Actor)(object snake1)) gets the highest rate because the Actor (as it is adjusted for this particular run of POPLAR) is more concerned about snakes than about dogs. Note also that the snake is closer to the Actor than the dog.

e) the **Executor** is called with P1 which is now posted in both ABB.current goal and ABB.current-plan. P doesn't have any preconditions, so its **is** clause is expanded: first a new instance of **Plan-Selector** is created. Since **Plan-Selector** is a 'mental-primitive' plan, its **Action-for-Primitive** is performed:

f) new instances of the plans **Fight**, **Hide** and **Wait-and-See** are created and rated (using their rating functions). Fight1 is selected. ABB.current-path now contains (P1 Fight1).

g) **Executor** is called with Fight1. After checking the preconditions of Fight1 (there are none) the IS clause is expanded to (Find Move Attack). Find1 with ((agent Actor)(object weapon)) is created and pushed onto ABB.current-path. Then Find1 is in its turn expanded to (Locate, Get-selector Get).

h) Locate1 with ((agent Actor)(object weapon)) is created and since it is 'mental-primitive', its **Action-for-primitive** is performed. The result is the list of objects 'found' (stick1 sword1). (Actor 'knows' that both sticks and swords could be used as weapons against snakes). The plan Locate1 is completed successfully.

i) Get-selectorl is created and its **Action-for-primitive** is performed. As the result of this two instances of **Get** (Getl ((agent Actor)(object stickl)) and Get2 ((agent Actor)(object swordl)) are created and rated. The **Get** with the highest rate (in this case Get2 because sword is more effective weapon then stick) is selected for execution.

j) **Executor** is called with Get2 which is expanded to (Move Take).

k) **Executor** is called with **Move**; Movel is created. Since its **is** slot is 'physical-primitive' control is passed to **Effector**.

l) **Effector** performs Movel's **Action-for-primitive** which is 'make one step towards the position of swordl'. This step is made, i.e. the position of Actor is changed. Then the SATISFACTION-CONDITIONl of all plans in the ABB.current-path are checked (the current path at this point contains (Movel Get2 Fightl Pl). None is satisfied, so the values of their 'status' slots are set to 'suspended'.

m) a new cycle of the **Monitor** begins. The user has again an opportunity of changing the world. Suppose, the user does not want to change anything. So, the agenda of goals remains the same and the top rated goal is the same Pl.

n) the path of 'suspended' plans is found (Movel Get2 Findl Fightl Pl). Since Movel is not yet completed, control is passed to the **Effector** with Movel.

o) the **Effector** performs the next 'step' of the Actor's movement towards swordl. Then SATISFACTION-CONDITIONs of all plans in the ABB.current-path are checked, none is satisfied, so the values of their 'status' slots are set to 'suspended'.

p) a new cycle of the **Monitor** begins.

4. Background and Related Work.

In designing and implementing POPLAR 2.0 a number of conceptual and technical decisions and choices had to be made. The following is an incomplete, though representative list.

1) how does one approach, and justify, construction of a multi-faceted system when little is known about the peculiarities of its components? Where is the starting point?

2) how might the problem of personality influences upon cognition be addressed?

3) within cognitive component, how are goals and plans related? How are they each related to such concepts as needs, drives, performance, etc.?

4) What is the structure of the planning module in cognitive systems? How is the scheduling of the cognitive system's activities performed?

5) What is the relationship between the use of internalized (canned) and newly created plans?

6) What is the relation between plan production and plan understanding?

The realization of the above and some additional problems was instrumental in the design stage. While not all of the decisions have been already made at this stage, our desire was to avoid design choices that would preclude or hamper a future improvement or extension.

None of the theoretical or design decisions were made without the influence of other, previous, related work. In this section we briefly review the bases for the various decisions as well as mention other work on the problems we faced.

Fundamental to the development of POPLAR was the approach to the task, faced by most cognitive modelers, of building a structure consisting of a number of distinct constituents, the details of many of which were (and at present remain) unknown. How does one construct a global model when many of its components are uncertain, and each one is itself a mystery? Here we adopted the attitudes advocated by Haugeland (1981), who suggests that it is appropriate to study an entire information processing system (IPS), consisting of several modules each of which (plus the IPS itself) is a black box, without first completing the study of the components; thus, we studied the cognitive actor even though we had not (and, obviously, could not) first provided an account for perception and performance.

Norman (1981) was very instrumental in specifying the tasks to be tackled in cognitive modeling. We also owe much to Anderson's (e.g. 1983) work on the architecture of cognitive entities. Sloman & Croucher (1981) discuss the introduction of motives, moods, attitudes and emotions in natural and artificial intelligent systems. Although no formalism is suggested for encoding this type of information, the general thrust of the approach is valuable for those who consider the introduction of certain personality characteristics into a class of AI systems. Wallace (1981) addresses similar problems in the context of learning.

Uhr & Kochen (1969) is an early work that addressed similar issues. Many of the important points for POPLAR have been anticipated in that work. Unfortunately, Uhr & Kochen's approach cannot be even called knowledge-based. It was an attempt to perform an important piece of research with inadequate means.

Wood (1983) discusses planning in a dynamically changing world with multiple actors. Her system, AUTODRIVE, uses the world of the automobile driver as the domain. Although the design of the system depends too strongly on the implementation world, the idea of interaction between the actor and the world (in fact, the mere separation of the objective world and that of the actor -- through a program called SIMULATOR) is very fruitful.

Schank & Abelson (1977) and Schank & Lehnert (1979) informally discuss and catalog human (including interpersonal) goals. Carbonell (e.g. 1979) discusses the use of the concept of personal goals in the context of understanding stories. Wilensky (1983) also discusses everyday goals and metagoals, as well as various cooperative and competitive relations among them.

The relation between goals and plans is an interesting question that had to be addressed in our work. Our solution was to use this term only for top-level goals recognized by the goal-generators but made manifest in the system through the instantiation of a top-level plan. We did not use the concept of goals at lower levels in planning (i.e. we did not use the term 'subgoaling', cf. Lesk, 1984).

It is argued (cf. e.g. Barber (1983) or Berlin (1984)) that subgoaling is preferable to the use of 'canned' plans because if the latter are used then there is no possibility of ever achieving a goal in a non-standard way. But in the subgoaling approach, within the current state of the art, no unexpected results can be obtained either. To introduce these, one has to build a learning system, one capable of creating and not only recreating. But at present the planning of the subgoaling type remains no less 'canned' than the the 'forward' planning.

It seems that these two approaches to planning relate essentially in the same manner in which backward chaining relates to forward chaining in inference making. Our opinion is that the choice between the two is not strategically important and should reflect the peculiarities of the domain and other 'weak' considerations, so typical for AI.

Another important issue related to goals and plans is whether to build systems that in scheduling an action take into consideration the knowledge of how many different plans and/or goals will be furthered by it. The main empirical body of Wilensky's book (1983) is devoted to such issues. Cf. also Hammond (1983) for a philosophically related approach. Hayes-Roth & Hayes-Roth (1979) also want their planner to have this capability. Our position on this topic (cf. also Carver et al., 1984) is that in the type of planners we are building the goal cooperation or conflict does not play a role. We argue that to treat this topic as central in modeling planning in intelligent actors is similar to consider such non-everyday tasks as playing chess and solving differential equations central topics for AI. The latter methodological fallacy has been amply criticized.

General works on planning that immediately influenced this project include Stefik's work (e.g. 1981) on metaplanning and planner architecture. Hayes-Roth & Hayes-Roth (1979) describe a very rich planning domain and offer a good discussion of what the editors of **The Handbook of AI** (Cohen & Feigenbaum, 1982, p. 519; cf. also pp. 22 - 27) call opportunistic planning. It does not seem, however, that a non-trivial, involved implementation of the itinerary planner they suggest is possible.

Hayes-Roth (1984) is a definitive proposal concerning the architecture for planners. It addresses the control problem in AI systems as a whole. It also contains a comparison with other current proposals concerning control. In its architectural part this proposal (in fact, not only this proposal!) draws heavily on the earlier work in the HEARSAY-II speech understanding system that introduced and popularized the blackboard architecture (cf. Erman et al., 1980).

The crucial idea of metalevel reasoning is discussed, with different emphases, in Stefik (1981), Hayes-Roth (1984), Wilensky (1983) and Genesereth (1983).

The basic architecture of POPLAR has a number of common points with that of Wilensky's planner (cf. Wilensky, 1983, pp.22-23). The two models, however, display major differences, notably in the attention paid in POPLAR to the problem of scheduling or in importance attributed to the idea of the independent representation of the objective world. Insufficient attention to scheduling and to describing the planning process at the system level were prominent among the criticisms in some reviews of Wilensky's book (cf. Russell, 1984; Berlin, 1984).

Many interesting ideas about scheduling can be found in Sathi et al. (1984) and Fox (1983).

Work on understanding plans in the POISE (Croft et al., 1983) and Argot (e.g. Litman & Allen, 1984) projects has helped in formulating some parts of our approach.

5. Status and Future Development.

5.1. General.

POPLAR is a working system that generates and executes a relatively small number of plans in a rich, though simulated, environment. Its scheduling capabilities actually seem to transcend the immediate necessities of the domain. The system is designed in such a way that both domain planning and metaplanning are performed by one executor (that is, POPLAR can reason about its own actions). A number of features have been included that make POPLAR a model of a human planner in a real world.

At the same time, the possibilities of development and improvement that this basic system offers are probably even more exciting than experimentation with the current version of POPLAR. There are many points at which the system can be improved. Some of them are discussed below.

First (and simplest) of all, the POPLAR actor's knowledge about the objects, goals and processes both in the objective world and its own 'mind' can and will be augmented. In parallel, the control knowledge (rating and control functions) will be constantly adjusted and tuned, both through the introduction of additional character trait, mental state and situation parameters and through devising more appropriate ways of amalgamating them in the decision functions. Extensive experimentation with POPLAR will help to verify such decisions.

In parallel and in conjunction with the POPLAR project, these authors have been involved in designing a general model of human cognitive activity. Initial results of that research are reported elsewhere (Nirenburg & Reynolds, 1983; Reynolds & Nirenburg, in preparation). An aspect of that project extremely helpful to POPLAR is research aimed at deriving a set of 'primitive' character traits, motivations and mental states, such that weighted combinations of them will correspond to the 'higher-level' parameters (e.g. 'aggressiveness') that we would like to use in POPLAR's decision functions.

One can see that the above are actually two separate problems: 1) to extract primitives; 2) to express complex entities in terms of the primitives. It was decided to adapt the primitives suggested by Cattell (cf. e.g. Cattell & Child, 1975). Extensive psychological experimentation with humans is pursued in order to find answers to the second problem. The benefits of having a system that boasts psychologically valid (and not 'folk psychology'-based) control parameters are enormous and self-evident. And, therefore, this is one of the most immediate improvements we plan to make.

We also plan to add plan understanding to plan production. The world is inhabited by more than one cognitive actor (consider, for instance, the trolls in the current POPLAR). In order to behave correctly an actor must be able to discern plans of others. We believe that POPLAR's machinery will be able to handle plan recognition without the necessity to introduce major changes. An actor will maintain as many blackboards as there are cognitive actors around. It will assume that all other actors operate in the same manner. It will have beliefs about their character traits, etc. and will 'project' plans for them much in the same manner as it plans.

A logical extension to adding plan recognition is to introduce verbal behavior into POPLAR. There exist a number of interesting approaches to discourse analysis and plan understanding in dialogues (e.g. Allen, 1983b; Litman & Allen, 1984; Carberry, 1983; Reichman, 1984; etc.). A study in modifying POPLAR to involve verbal behavior and discourse analysis can be found in Nirenburg & Pustejovsky (1985).

The inclusion of multiple actors into the objective world can lead to the development of an experimental testbed for modeling conflict resolution, cooperation and many more important 'real-life' situations. The possibilities here are definitely substantial and quite unexplored.

The mechanism for modeling attention will undergo serious modifications, as will the treatment of time and the interaction between the actor(s) and the objective world.

And, finally, a most important avenue of improvement is the introduction of learning capabilities to the system. There are many modules in POPLAR where planning can be introduced; and there are many different types of learning to be studied. Some examples of this may be modifying the scheduling behavior depending on results of previous processing or after seeing somebody achieve a goal in a way not previously used; modifying beliefs about objects; being able to 'create' new plans, by analogy or otherwise; and many many more. This topic is one of the more complex ones, but any progress in this direction may have a very beneficial effect on the field of planning in AI.

5.2. Specific Plans: POPLAR 3.0.

The development plans for the coming year include the development and implementation of

1) strategies for combining plan **recognition** and plan **production** in one system;

2) a **mixed strategy** for planning: the use of canned plans for standard situations and 'first principles' knowledge when non-standard situations arise;

3) the development of a model of (a subset of) the world of I & W.

Bibliography.

Allen, J., 1983a. Maintaining knowledge about temporal intervals. *Communications of ACM,* vol. 26, 832 - 843.

Allen, J., 1983b. Recognizing intentions from natural language utterances. In: M. Brady & R.C. Berwick, eds., **Computational Models of Discourse.** Cambridge, MA: MIT Press.

Anderson, J.R. 1983. **The Architecture of Cognition.** Cambridge MA: Harvard University Press.

Barber, G., 1983. Supporting organizational problem solving with a work station. *ASM Transactions on Office Automation Systems.* Vol.1, No.1, January. 45 - 67.

Bates, P., J. Wileden and V. Lesser, 1981. A language to support debugging in distributed systems. University of Massachusetts COINS Technical Report 81-17.

Berlin, Daniel, 1984. Review of Wilensky (1983). *Artificial Intelligence,* vol. 23, 242-244.

Carberry, S., 1983. Tracking user goals in an information-seeking environment. Proceedings of AAAI-83. Washington, DC.

Carbonell, J., 1979. Computer models of human personality traits. Proceedings of IJCAI-79. 121 - 123.

Carver, Norman F., Victor R. Lesser and Daniel L. McCue, 1984. Focusing in plan recognition. Proceedings of AAAI-84. Austin, TX.

Cattell, R.B. and D. Child, 1975. **Motivation and Dynamic Structure.** NY: Wiley.

Cohen, P.R. and E.A. Feigenbaum (eds.), 1982. **The Handbook of Artificial Intelligence.** Volume III. Los Altos CA: Kaufmann.

Croft, W.B. and L. Lefkowitz, 1984. Task support in an office system. *ACM Transactions on Office Automation Systems.* Vol.2, No.3 197 - 212.

Croft, W.B., L. Lefkowitz, V. Lesser and K. Huff, 1983. POISE: An intelligent interface for profession-based systems. Conference on Artificial Intelligence. Oakland, Michigan, 1983.

Deering, M., J. Faletti and R. Wilensky, 1981. PEARL - a package for efficient access to representations in Lisp. Proceedings of 7th IJCAI, Vancouver, BC. 930 - 932.

Erman, L.D., F. Hayes-Roth, V.R. Lesser and D.R. Reddy, 1980. The HEARSAY-II speech understanding system: integrating knowledge to resolve uncertainty. *Computing Surveys*, vol. 12, 213-253.

Fox, M., 1983. Constraint-directed search: a case study of job-shop scheduling. CMU Robotics Institute Technical Report 83-22.

Genesereth, M.R., 1983. An overview of meta-level architecture. Proceedings of AAAI-83. Washington, DC.

Hammond, K.J., 1983. Planning and goal interaction: the use of the past solutions in present situations. Proceedings of AAAI-83, Washington, DC.

Haugeland, J., 1981. The nature and plausibility of cognitivism. In: J. Haugeland (ed.), **The Mind Design.** Cambridge MA: MIT Press.

Hayes-Roth, B., 1984. A blackboard model of control. Stanford University Heuristic Programming Project Report HPP 83-38 (revised August 1984).

Hayes-Roth, B and F. Hayes-Roth, 1979. A cognitive model of planning. Cognitive Science, vol. 3, No.4.

Lesk, M., 1984. Universal Subgoaling. CMU PhD Thesis.

Litman, D. and J.Allen, 1984. A plan recognition model for clarification subdialogues. Proceedings of COLING-84. Stanford, 302 - 310.

McCue, Daniel and Victor Lesser, 1983. Focusing and Constraint Management in Intelligent Interface Design. University of Massachusetts COINS Technical Report 83-36.

Nirenburg, S. and J. Pustejovsky, 1985. Plan Recognition and Production for Verbal (Discourse) and Non-Verbal Behavior. COINS Technical Report, University of Massachusetts.

Nirenburg, S. and J.H. Reynolds, 1983. An architecture for a computer model of human cognitive systems. Colgate University COSC Technical Report 4-83.

Norman, D., 1981. Twelve issues for cognitive science. *Cognitive Science*, vol. 4, No.1.

Reichman-Adar, R. Extended person-machine interface. *Artificial Intelligence*, vol.23, 157 - 218.

Reynolds, J.H. and S.Nirenburg, in preparation. The relationship between motivational traits, goal generation and plan selection.

Russel, Daniel.M., 1984. Review of Wilensky (1983). *Artificial Intelligence,* vol. 23, 239-242.

Sathi, A., M. Fox, M. Greenberg and T. Morton, 1984. Callisto: an intelligent project management system. CMU CS working paper.

Schank, R.C. and R. Abelson, 1977. **Scripts, Plans, Goals and Understanding**. Hillsdale, NJ: Erlbaum.

Schank, R.C. and Lehnert, W., 1979. The conceptual content of conversations. Proceedings of 6th IJCAI. 769 - 771.

Sloman, A. and M. Croucher, 1981. Why robots will have emotions. Proceedings of 7th IJCAI. Vancouver, BC, 197-202.

Stefik, M, 1981. Planning with constraints (MOLGEN: Part 1 and Part 2). *Artificial Intelligence* 16, 111-170.

Uhr, L. and M. Kochen, MIKROKOSMS and robots. Proceedings of IJCAI-69, Washington, D.C., 541 - 556.

Wallace, J.G., 1981. Motives and emotions in a general learning system. Proceedings of 7th IJCAI. Vancouver, BC, 84-86.

Wilensky, Robert, 1983. **Planning and Understanding**. Reading, MA: Addison-Wesley.

Wood, S., 1983. Dynamic world simulation for planning with multiple agents. Proceedings of IJCAI-83, Karlsruhe, Germany.

Appendix 1. Representation of OBJECTS in POPLAR 1.3

We present objects in two ways: first, in the way the object is stored in LTM, and second, from within a POPLAR run (as an annotated script). The difference is due to inheritance of parents' properties by children in the hierarchy.

A.

```
(dbcr exp creature person ;this is a PEARL header for a frame
     (id person)
     (type creature)     ;CREATURE is the parent of PERSON
     (h-process-roles  lisp ((Take Who)
                              (Put Who)
                             (Find Who)))
                           ;the above are the roles in which an instance
                           ;of this type can appear in specified
                           ;processes by virtue of its having properties
                           ;of a "human": humans can act as agents in
                           ;TAKE, PUT and FIND
     (mental-state struct) ;humans have mental states -- cf. the
                             ;default values in the script listing below
     (character-traits struct char-traits) ;ditto
     (weapon-against ((sword 100 3)(knife 50 1)(rock 10 20)))
               ;POPLAR knows (believes) that weapons against people
               ;include swords, knives and rocks; the numbers (a b)
               ;indicate the efficiency of the weapon and the maximum
               ;range
     (power 50) ;maximum
     (speed 50) ;maximum
     (fearsomeness 25) ;what is the level of fear that such objects
                      ;typically elicit in POPLAR (default: 25)
     (mass 55)
     (inventory lisp) ;the objects this person is perceived by POPLAR
                      ;to be carrying
 )
```

B.

```
POPLAR> person
  (person (id person)
        (type creature)
;------------------ ---------------------------
        (o-process-roles ((Find What)))
;this property is inherited by virtue of PERSON's being a
;descendant of OBJECTS: any object can occupy the "what" slot in Find,
;because finding mental objects is recollecting their representations in
;memory
;---------------------------------------------
        (shape nil)
        (color nil)
        (mass 55)
        (position nil)
        (p-process-roles ((Take What) (Put What)))
        (goal-parameters ((PS1 adv)))
;the above properties are inherited by virtue of person being a descendant
;of PHYSICAL-OBJECTS; the goal-parameters slot specifies an instance of
;what goal is created when an object of this type is perceived. In this
;case the intuition behind the entry is that the appearance of a person
;spawns the creation of a goal instance of Preserve-Self-1, that is,
;persons are perceived by POPLAR as potential enemies
;---------------------------------------------
        (edibility nil)
;this property is inherited by virtue of PERSON's being a descendant of
;+alive; nil is the default value with the semantics of "unknown"
;---------------------------------------------
        (c-process-roles
         ((Eat Who)
          (Ingest Who)
          (Drink Who)
          (Move Who)
          (Attack (Who Whom))))
;the above properties are inherited by virtue of PERSON's being a
;descendant of CREATURE; creatures are considered by POPLAR to be able
;to be agents of eating, drinking and moving, and agents and objects of
;attacking
;---------------------------------------------
        (weapon-against ((sword 100 3) (knife 50 1) (rock 10 20)))
        (power 50)
        (fearsomeness 25)
        (speed 50)
        (orientation nil) ;this shows whether this particular person
                   ;LOOKS at POPLAR at the moment of processing
;-----------------------------------------------------
;the following are physical states (conceptually, they are a part of
;the regulatory system)
        (hunger 0)
        (thirst 0)
        (fatigue 0)
        (injury 0)
        (h-process-roles ((Take Who) (Put Who) (Find Who)))
        (mental-state (nilstruct))
;character traits are a component of the regulatory system
        (character-traits
         (char-traits (greed 20)
                     (pedantism 10)
                     (hunger-tolerance 5)
                     (thirst-tolerance 20)
                     (fatigue-tolerance 20)
                     (courage 25)
                     (aggression 40)
                     (impulsiveness 30)
                     (articulateness 40)
                     (extravertedness 50)
                     (loquaciousness 40)
                     (curiosity 55)))
         (inventory nil))
```

Appendix 2. Examples of PLAN representation in POPLAR 1.3.

```
(dbcr exp PLANS PS1
  (ID PS1)
  (Type PLANS)
  (Top-level-flag yes)
  (IS ((Plan-Selector  Fight Wait-and-See))) ;Flee Hide
  (With   (Actor Adversary))
  (COND  ((Plan-Selector '(Fight Wait-and-See)   ;Hide Flee
                    current-plan)
        (Fight Actor Adversary)
        (Flee Actor Adversary)
        (Hide Actor Adversary)
        (Wait-and-See Actor Adversary)))
  (Preconditions (and (member 'Adversary (getpath ABB '(OBJECTS-PERCEIVED)))
          ;Adversary is among objects perceived by the Actor
                (or (= 'Actor 'self)
                   (and (structurep Actor)
                       (not (structurenamep 'Actor))
                       (= (getpath (eval Actor) '(type) 'person))))))
          ;Actor is either "self" or any instance of person
  (Rating-function (rating-func-PS1)))
```

```
(dbcr exp PLANS Plan-Selector
  (ID Plan-Selector)
  (Type PLANS)
  (Top-level-flag no)
  (IS (primitive))
  (Action-for-primitive (Schedule-of-plan 'list-of-plans 'calling-plan))
  (With (list-of-plans calling-plan))
  (Time 1) )

(dbcr exp PLANS Fight
  (ID Fight)
  (Type PLANS)
  (Top-level-flag no)
  (IS (Find (Control1 (Find ! (Control2 (Get)))) (Control3 (Move ! Attack))))
  (COND ((Find  Actor (getpath  Adversary '(weapon-against))
          (getpath Actor '(inventory)))
        (Find  Actor (getpath  Adversary '(weapon-against))
            (getpath ABB '(OBJECTS-PERCEIVED)))
        (Get Actor  (car result-find)
           (div (distance Adversary (car result-find))
              (getpath Adversary '(speed))))
        (Move Actor (prog (weapon-range)
          ;position to Move to
           (cond ((<= (distance Actor Adversary)
                  (setq weapon-range
                    (caddr (assoc (getpath (eval (car result-find))
                                   '(type))
                              (getpath (eval Adversary)
                                   '(weapon-against))))))
              ;if distance between Actor and Adversary is less
              ;(or equal) than the range of the Actor's weapon
              ;then Actor doesn't need to move towards Adversary
                (return (getpath (eval Actor) '(position))))
               (t (return (calculate-position Actor
                                  Adversary weapon-range)))
           )))

        (Attack Actor Adversary (car result-find))))

  (Control ((Control1 (Fight-Control1 Actor Adversary))
        (Control2 (Fight-Control2 Adversary))
        (Control3 (Fight-Control3))))
  (With  (Actor Adversary))
  (Rating-function (rating-func-fight)) )
```

```
(defun Fight-Control1 (Actor Adversary)
    (cond ((not (= (car ABB.CURRENT-PATH.Status)
               'succeeded))
          t)
   ;EITHER the last executed plan (which is Find) failed
         ((= (cadar Adversary.weapon-against)
            (cadr (assoc (car result-find).type
                     Adversary.weapon-against)))
          nil)
   ;OR actor's current weapon is NOT the most efficient weapon
   ; against this adversary
         ((lessp
              (div (times (distance Actor Adversary)
                      (diff (cadar Adversary.weapon-against)
                          (cadr (assoc (car result-find).type
                                   Adversary.weapon-against))))
                  Actor.character-traits.impulsiveness)
           fight-control1-threshold))
   ;OR even if the actor does not have the best weapon, he may decide not to
   ;look for a better one -- if the distance between him and the adversary
   ;is too small, if the actor is very impulsive or if the weapon is not
   ;much worse than the best one
 ]

 (defun Fight-Control2 (Adversary)
 (cond ((null (cadr result-find)) t)
    ;no weapon was found in actor's possession
     ((greaterp (cadr (assoc (car result-find).type
                      Adversary.weapon-against))
             (cadr (assoc (cadr result-find).type
                      Adversary.weapon-against))))
    ;the weapon that was found "around" is BETTER than
    ;the weapon in actor's possession ]
```

Appendix 3. Examples of POPLAR 1.3 rating functions.

A. The rating function for the Preserve-Self-1 goal (and top-level plan)

```
(defun rating-func-PS1 (actor adversary)

  (fix (div
         (times
              (calculate-fear actor adversary)
              actor.aggression)
         actor.courage)))
```

```
(defun calculate-fear (actor adversary)

  (fix (div
        (times adversary.orientation
             (add adversary.mass adversary.speed)
             adversary.power
             adversary.aggr
             adversary.fearsomeness)
        (times (fix (add1 (log (distance actor adversary))))
             actor.courage
             actor.power
             (add actor.mass actor.speed)))))
```

B. The rating function for the Fight intermediate plan.

```
(defun rating-func-fight (actor adversary)

 (fix (div
        (times adversary.weapon-against.efficiency
             actor.courage
             actor.power
             (add1 adversary.injury)
             (expt actor.aggression 2))
        (times
             (calculate-fear actor adversary)
             adversary.power
             (add1 actor.injury)
             adversary.fearsomeness
             (add1 actor.fatigue)))))
```

Appendix 4. HISTORY in POPLAR 1.3.

;this is the way HISTORY looks at the end of the example run of 5.4.

```
POPLAR> HISTORY
  ((Ingest0 Eat0 PS20)
   (Take0 Get0 Eat0 PS20)
   (Move0 Get0 Eat0 PS20)
   (Get0 Eat0 PS20)
   (Find1 Eat0 PS20)
   (Find0 Eat0 PS20)
   (Eat0 PS20)
   (Plan-Selector0 PS20))
```

Appendix 5. BLACKBOARDS in POPLAR 1.3.

Typical contents of the worldand the actor blackboards.

```
POPLAR> WBB
       (World-Blackboard (ID WBB)
                      (NEW-INPUTS troll1 apple2 crocodile2)
                      (TIME (Base-Time (ID Time)(act-time 17))))

POPLAR> ABB
       (Actor-Blackboard (ID ABB)
                      (OBJECTS-PERCEIVED (troil2 sword1 gold-nugget2))
                      (STATES-PERCEIVED (hunger fatigue))
                      (AGENDA PS14 PS22 GTR4 Agenda-Scheduler)
                      (CURRENT-PATH (find7 fight3 PS14)))
```

Appendix 6. Examples of PLAN representation in POPLAR 2.0.

```
(defflavor Plans (id
                  is
                  top-level-flag
                  with
                  status
                  satisfaction-cond
                  param-binding
                  real-world-flag)
  ()
 :settable-instance-variables
 :gettable-instance-variables
 :initable-instance-variables
)

(defflavor P ((id P)
              (is ’((Plan-Selector Fight Flee Hide Wait-and-See)))
              (top-level-flag ’yes)
              (with ’((agent actor)(object adversary)(instrument weapon)))
              (status ’init))
   (Plans)
 :settable-instance-variables
 :gettable-instance-variables
 :initable-instance-variables
)

(defmethod (P :init) (options)
  (setq param-binding ‘((Plan-Selector (Fight Wait-and-See Hide) ,id)
                        (Fight ,(assoc ’agent with)
                               ,(assoc ’object with)
                        (weapon nil))
                        (Wait-and-See ,(assoc ’agent with))
                        (Hide  ,(assoc ’agent with)
                               ,(assoc ’object with)))
        satisfaction-cond ‘(not (member (quote ,(cadr (assoc ’object with)))
                                      (send ABB :objects-perceived))))
       ;goal P is achieved when Adversary is not among
       ;the objects perceived by Actor
)

(defmethod (P :rating-function) ()
 (let* ((actor (eval (assoc ’agent with)))
        (adversary (eval (assoc ’object with)))
        (dist (distance actor adversary))
```

```
      (adv-mass (send  adversary :mass))
      (actor-mass (send  actor :mass))
      (orientation (cond ((equal (send  adversary :orientation) 'yes)
                           2)
                         (t 1)))
      (adv-power (send  adversary :power))
      (actor-power (send actor :power))
      (actor-courage (send  actor :courage))
      (adv-speed (send  adversary :speed))
      (actor-speed (send  actor :speed))
      (adv-aggr (send  adversary :aggression))
      (actor-aggr (send  actor :aggression)))

  (fix (div (times orientation
                 (add adv-mass adv-speed)
                 adv-power
                 adv-aggr
                 actor-aggr
                 (send  adversary :fearsomeness))
           (times (fix (add1 (log dist)))
                 actor-courage
                 actor-courage
                 actor-power
                 (add actor-mass actor-speed)))
 )
)


(defflavor Find ((id Find)
              (is  '(Locate (Plan-Selector Get)))
              (top-level-flag 'no)
              (with '((agent actor)(object obj)))
              (status 'init))
   (Plans)
 :settable-instance-variables
 :gettable-instance-variables
 :initable-instance-variables
)


(defmethod (Find :init) (options)
  (setq param-binding `((Locate ,(assoc 'agent with)
                           ,(assoc 'object with))
                    (Plan-Selector (Get) ,id)
                    (Get  ,(assoc 'agent with)
                     (object obj)))
      satisfaction-cond '(member (cadr (assoc 'object with)))
                         (send (eval (cadr (assoc 'agent with)))
                             :inventory))
     ;goal Find is achieved when Actor has Object
     ;in his possession
)

(defmethod (Find :preconditions) ()
  (cadr (assoc 'object with))
  ;the object is specified (Actor 'knows' what is to be found)
)


(defflavor Take ((id Take)
             (is  '(physical-primitive))
             (top-level-flag 'no)
             (with '((agent actor)(object obj)))
             (status 'init))
    (Plans)
 :settable-instance-variables
 :gettable-instance-variables
 :initable-instance-variables
 )


 (defmethod (Take :init) (options)
   (setq satisfaction-cond '(member (cadr (assoc 'object with)))
                           (send (eval (cadr (assoc 'agent with)))
                               :inventory))
       ;goal Take is achieved when Actor has Object
       ;in his possession
 )

 (defmethod (Take :preconditions) ()
   (and (member (cadr (assoc 'object with))
             (send ABB :objects-perceived))
      ;Object is among the objects perceived by Actor
        (equal (send (eval (cadr (assoc 'agent with))) :position)
             (send (eval (cadr (assoc 'object with))) :position)))
      ;Object and Actor are on the same position
  )

  (defmethod (Take :action-for-primitive) ()
 ;the part of the code that maintains the graphic
 ;part of the program is omitted

(let ((actor  (cadr (assoc 'agent with)))
      (object (cadr (assoc 'object with))))
 (send  (eval actor) :set-inventory
        (cons object
              (send (eval actor) :inventory)))
   ;add Object to Actor's inventory
 (format t "~& ~a ~a ~a ~a ~&"
         object
        "is now in"
        actor
        "'s possession.")
 (send ABB :set-objects-perceived
       (delete object (send ABB :objects-perceived)))
 )
)
```

PROVIDING INTELLIGENT ASSISTANCE IN DISTRIBUTED OFFICE ENVIRONMENTS[1]

Sergei Nirenburg
Colgate University

Victor Lesser
University of Massachusetts

Abstract. We argue that a task-centered, an agent-centered and a cognition-oriented perspective are all needed for providing intelligent assistance in distributed office environments. We present the architecture for a system called OFFICE that combines these three perspectives. We illustrate this architecture through an example.

1. Introduction.

In this paper we describe OFFICE, a system that provides intelligent assistance in the office environment. A schematic diagram of the type of system we are proposing is shown in Figure 1.

In this diagram the office worker operating together with his/her workstation constitute one node in the office problem solving network. The initiative in such a problem-solving environment is mixed: it can be originated by the office worker performing a low-level task or specifying a high-level goal to be accomplished or the office system OFFICE requesting the worker to perform a task. Thus, we see OFFICE as an intelligent assistant to the office worker.

We argue that a task-centered, an agent-centered and a cognition-oriented perspective are all needed for providing intelligent assistance in distributed office environments. We need knowledge from each of these perspectives in order to support not only effective local interaction between OFFICE and the office worker, but also to coordinate cooperative problem solving among the nodes in the system. Coordinating problem solving is an especially difficult task, given the semi-autonomous nature of processing at each node; the bandwidth of the communication channel (which makes it not feasible for nodes to have a complete global view of problem solving in the network); the diversity of the types of knowledge necessary for coordinating and scheduling office activities; and the necessity to provide guidance to the office worker about how to prioritize his own tasks so that they are coherent with the goals of the whole system.

[1] This work was supported by the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, NY 13441-5700, and the Air Force Office of Scientific Research, Bolling Air Force Base, D C 20332 under contract number F30602-85-C-0008

We s:e the coordination problem as breaking down into a number of subproblems, which include managing resources; equalizing workload distribution; managing goal conflicts; maintaining a proper level of redundancy in task execution and especially in information flow; analyzing dependencies in the sets of goals, plans and events, etc. Automation of any of the above tasks clearly involves manipulation of many types of knowledge, both domain and control.

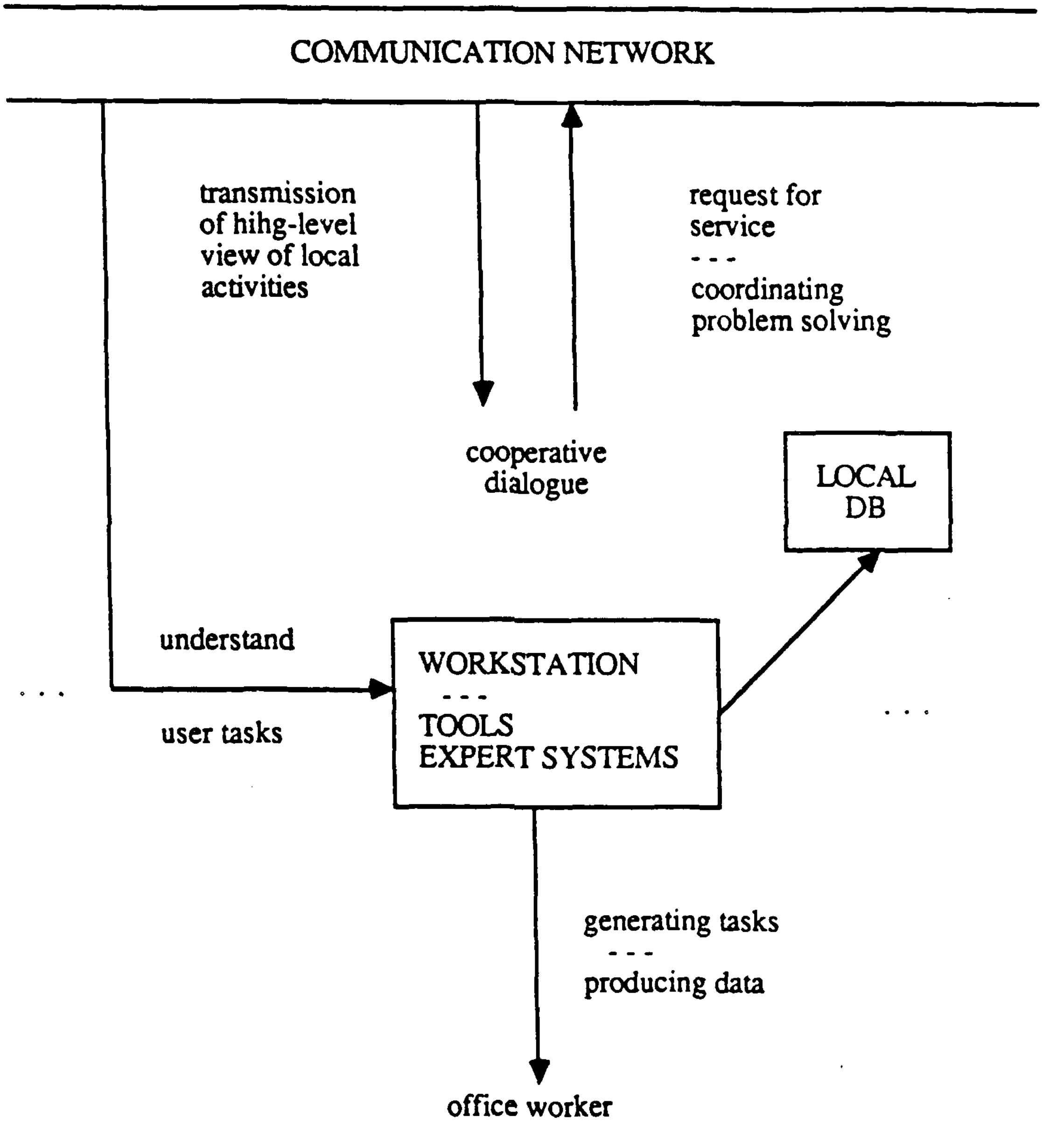

Figure 1. A node in a network of cooperative office workstations.

To illustrate the problem of local scheduling that takes into account global coherence, consider an office consisting of an executive, E, and his/her secretary, S. Suppose, E is dictating letters to S, and the telephone rings. S answers, and the call appears to be about a very important shipment, and S is asked to provide some information about it. The scheduling choice here is between continuing with the letters (task Tl) and performing the request that came over the phone (task T2). We want our system to consider a number of factors here, including the relative importance of the tasks (say, a number of people may be idle in the company because of the lack of raw materials that are to be shipped), the time limitations (suppose, the information is needed before the end of the business day, and it's already 4 p.m.; also, the estimated time of finding the requested information), personal characteristics of S and E, etc. If the secretary were scheduling purely locally, he/she may prefer to schedule T2, but knowing that E will be detained by her doing so, S may prefer Tl based on global coherence considerations. S's knowledge about personal characteristics of E can also be a factor: if E is very conscious of his/her status and importance, then the decision of scheduling Tl is even more strengthened; if not, and if S has the characteristic of being assertive, T2 may be preferred, after an explanation to E.

In what follows we, first, trace the project's genesis from three research projects in connected fields and discuss its functionality. Second, we describe how an office can be modelled in a distributed computer system such as OFFICE and describe its architecture and the basic processing cycle. Finally, we give an example of OFFICE operation where we concentrate on its reasoning capabilities.

The Task-Oriented Perspective.

Our initial effort in developing an expert system in the office domain is the task support system POISE (Croft et al., 1983). POISE has been designed to support office workers in their problem solving activities through the use of plan recognition and planning In the plan recognition mode the system obtains messages about certain atomic events (such as too! invocations) and tries to determine into which of typical tasks known to the system this event fits In this manner POISE is able to monitor the activities in an office, predict future activity and detect errors. If, as a result of the monitoring, the system understands the user's task, it can in principle take over its completion. This task completion mode is integrated with the planning mode of operation. In the planning mode POISE is supplied with a typical tasks and its parameters and tries to execute as much of it as possible, based on its knowledge of the task structure and the status of domain objects in a semantic database.

POISE's knowledge takes the form of an hierarchy of typical tasks. Each task is represented by a precondition statement that defines the necessary conditions for its execution; a goal statement that specifies the intended effect of the task; the sequence of subtasks needed to be performed in order to accomplish the task and the constraints among the parameters of the subtasks and those of the task. See Figure 2 for an example.

```
PROC    Purchase-items (Purchasing Amount It°ms Vendor)
DESC    Procedure for purchasing items with non-state funds
IS      Receive-purchase-request
        ' (Process-purchase-order | Process-purchase-requisition)
        ' Complete-purchase

COND
        Process-purchase-order.Amount          = Receive-purchase-request.Amount
        OR
        Process-purchase-requisition.Amount    = Receive-purchase-request.Amount
        Process-purchase-order Items           = Receive-purchase-request Items
        OR
        Process-purchase-requisition Items     = Receive-purchase-request.Items
        Process-purchase-order Vendor          = Receive-purchase-request.Vendor
        OR
        Process-purchase-requisition.Vendor    = Receive-purchase-request Vendor
        Process-purchase-order.Amount          = Complete-purchase.Amount
        OR
        Process-purchase-requisition Amount    = Complete-purchase.Amount
        Process-purchase-order Items           = Complete-purchase Items
        OR
        Process-purchase-requisition Items     = Complete-purchase.Items
        Process-purchase-order.Vendor          = Complete-purchase.Vendor
        OR
        Process-purchase-requisition.Vendor    = Complete-purchase.Vendor

WITH    Purchaser    = Receive-purchase-request.Purchaser
        Amount       = Receive-purchase-request.Amount
        Items        = Receive-purchase-request Items
        Vendor       = Receive-purchase-request.Vendor
```

Figure 2. A plan in POISE

POISE plans are structured so that they in principle allow concurrent execution of subtasks of a task. Straightforward transformation of POISE into a distributed system cannot, however, be performed. Since POISE does not have a developed agent-oriented perspective, there is no way in it to express a fact such as 'requests made by the manager of the office have priority over those made by other workers' or the fact that even though certain workers are better at doing certain types of jobs, if they are not available to do a job of this type, then other workers have to be assigned this responsibility. There is also no way of talking about seemingly independent tasks being actually parts of a cooperative problem solving situation. This includes the considerations of arbitration of competing claims for limited resources.

POISE does not distinguish or reason about the agents' roles and the objects in plans. Thus, for instance, it does not have the possibility to understand that an unusual event happened if it gets the message that the president of a company typed a letter (and not a secretary). Therefore it cannot infer that the secretary may have a day off or that a goal must be instantiated of changing workload distribution among the employees.

Another deficiency of POISE is that the plan recognition and planning architectures are not designed for being distributed and assume a global blackboard and a single locus of control. POISE gives us some ideas about what an intelligent assistant could be but its architecture is not appropriate for use in a distributed environment and it lacks a distributed agent-oriented perspective.

The Distributed Agent-Oriented Perspective.

One of the research areas where we can look for ideas of how to implement the distributed agent-oriented perspective is the field of distributed AI. One of the current approaches there is the study of functionally accurate, cooperative (FA/C) distributed problem solving (Lesser and Corkill, 1983; Corkill, 1982; Durfee et al., 1984, 1985). With this approach, a problem is solved in cooperation by a set of semi-autonomous processing nodes (agents) that may have inconsistent and incomplete local databases. each node independently generates tentative partial solutions, communicates them through a network to other nodes, receives messages (partial solutions, goals, plans and facts) from other nodes, and modifies its processing in accordance with new input. The experience of this work has shown that the control problem is difficult; that the network communication is both difficult and computationally expensive; most importantly, it was found that the key to global coherence is having sophisticated agents who can reason about their own view of processing as well as the views of other agents. They have developed a system in which each node is guided by a high-level strategic plan for cooperation among the nodes in the network. This plan, which is a form of metalevel control, is represented as a network organizational structure that specifies in a general way the information and control relationships among the nodes. Examples of this information include static priorities among local tasks, to whom and what information to communicate and how to prioritize tasks that have been requested by other nodes versus those that were locally generated.

Other work by Smith and Davis (1981) has focused on the knowledge and the protocols necessary for nodes to decide in a distributed way how to allocate subtasks to other nodes. This involves a two-way bidding protocol in which the contractors (taking on the task perspective) and bidders (taking on an agent perspective) communicate to determine the best task allocation.

The work by Lesser et al. focuses on how to do local scheduling given a static task allocation that may redundantly allocate tasks among nodes, while Smith and Davis focus on dynamic task allocation. The office domain requires an integration of both approaches together with augmenting the knowledge used by both approaches for scheduling. The office domain also presents challenges to both approaches because of the tighter and more complex interactions among agents that exists in this domain, compared to the distributed interpretation domain from which both of the above approaches evolved.

The Cognition-Oriented Perspective.

The distributed problem solving approaches described above concentrated on the architecture of the network and the nodes, with the view of organizing the control structure. The types of knowledge necessary for control and communication in OFFICE are studied in the field of cognitive agency research (e.g. Georgeff, 1984, Moore, 1985, but mainly Nirenburg et al., 1985, 1986). The view of the world in this field is that cognitive agents are immersed in a world which is non-monotonic, in the sense that changes in the world can be introduced not only because of the activities of a single agent but also through uncontrolled external events. Agents are capable of a variety of cognitive tasks. They can perceive objects and events in the world. They possess a set of goal types and means of achieving goals of these types: plans. They perform goal and plan generation, selection and execution in complex situations in which many goals and plans coexist and compete for the attention of the agent's conscious processor.

The study of the knowledge that underlies the reasons for particular choices of goals and plans by an agent (in other words, reasons for scheduling and communication decisions) is the central theme of this approach. This knowledge is claimed to involve such factors as personality traits, and physical and mental states of the agent, in addition to the knowledge about the domain situation and the typical tasks and goals. Our approach is to use all the types of knowledge discussed in the cognitive agency approach within the architectural framework inspired by the distributed AI research.

2. An Architecture for a Distributed Office System.

We present here, through an example, an architecture for an intelligent assistance system that integrates the task-, agent- and cognition-oriented perspectives.

2.1. Representing an office.

An office is modelled as a network whose nodes are interpreted as office workers and edges, as communication channels. Every node in the network is a complete problem solver that consists of an office worker and his/her workstation. Following POISE, OFFICE deals with typical activities in a university-based research project (RP), namely: purchasing equipment, hiring and travel. The types of agents in the RP office include Principal Investigator (PI), Research Associate (RA), Graduate Student (GS), Secretary (S), Vendor (V) and Accountant (A). A typical instance of a project may involve 1 PI, 2 RA's, 6 GS's, 1 S, 3 V's (e.g., DEC, Symbolics and TI) and and 2 A's (say, one in Accounts Receivable and one in Personnel).

Figure 3 shows the communication channels for the RP office.

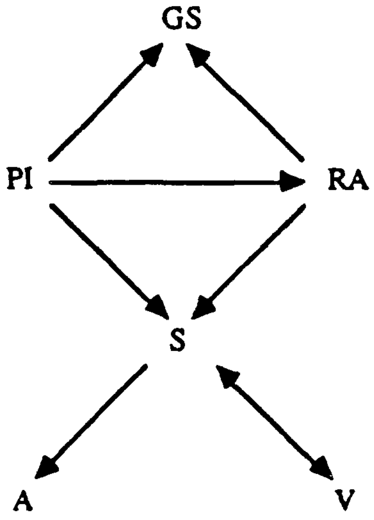

Figure 3. The network of processing nodes in a model of an RP office.
The arrows illustrate authority relationships (see below).

Every node in the office network is aware of its responsibilities to carry out parts of certain plans. They also know who or where from they can and should seek information that is necessary for them to perform their tasks (recall that information about the typical agents for all types of tasks is among the knowledge that every agent possesses).

At any moment t each agent in OFFICE has an agenda of current goals or, more precisely, of current goal instances, as illustrated in (1),

$$\left\{PU^{5}_{\sigma_1}, PU^{6}_{\sigma_1}, HI^{1}_{\sigma_3}, TR^{2}_{\sigma_4}\right\}_t \qquad (1)$$

where PU^i, HI^j and TR^k stand for instances of goal types *Purchase, Hire* and *Travel*, and σ_i designate subsets of network nodes that are working cooperatively on particular goals. Intuitively, at any given moment the office workers are pursuing a number of goals, working in teams. Note that some such goals can be in conflict or can compete for resources. Therefore, the agents must have means of resolving these conflicts.

The architecture of an agent in OFFICE is illustrated in Figure 4. A frame-based representation is used for objects, goals, plans and actions, including messages. Plans are represented in extended EDL (cf. Nirenburg et al., 1985). An agent has knowledge about the goals it is typically responsible for as well as about plans that are typically used to accomplish these goals. (If node A has a goal G on its agenda, then A is *responsible* for achieving G.) It also has the knowledge about the current state of its goal agenda, as well as a subset of the contents of other agents' agendas. Scheduling knowledge used by the agent to select goals and plans for processing is represented as a set of condition-action rules. The agent also is aware of the authority relationships in the office, illustrated in Figure 3, that are part of the agent's scheduling knowledge.

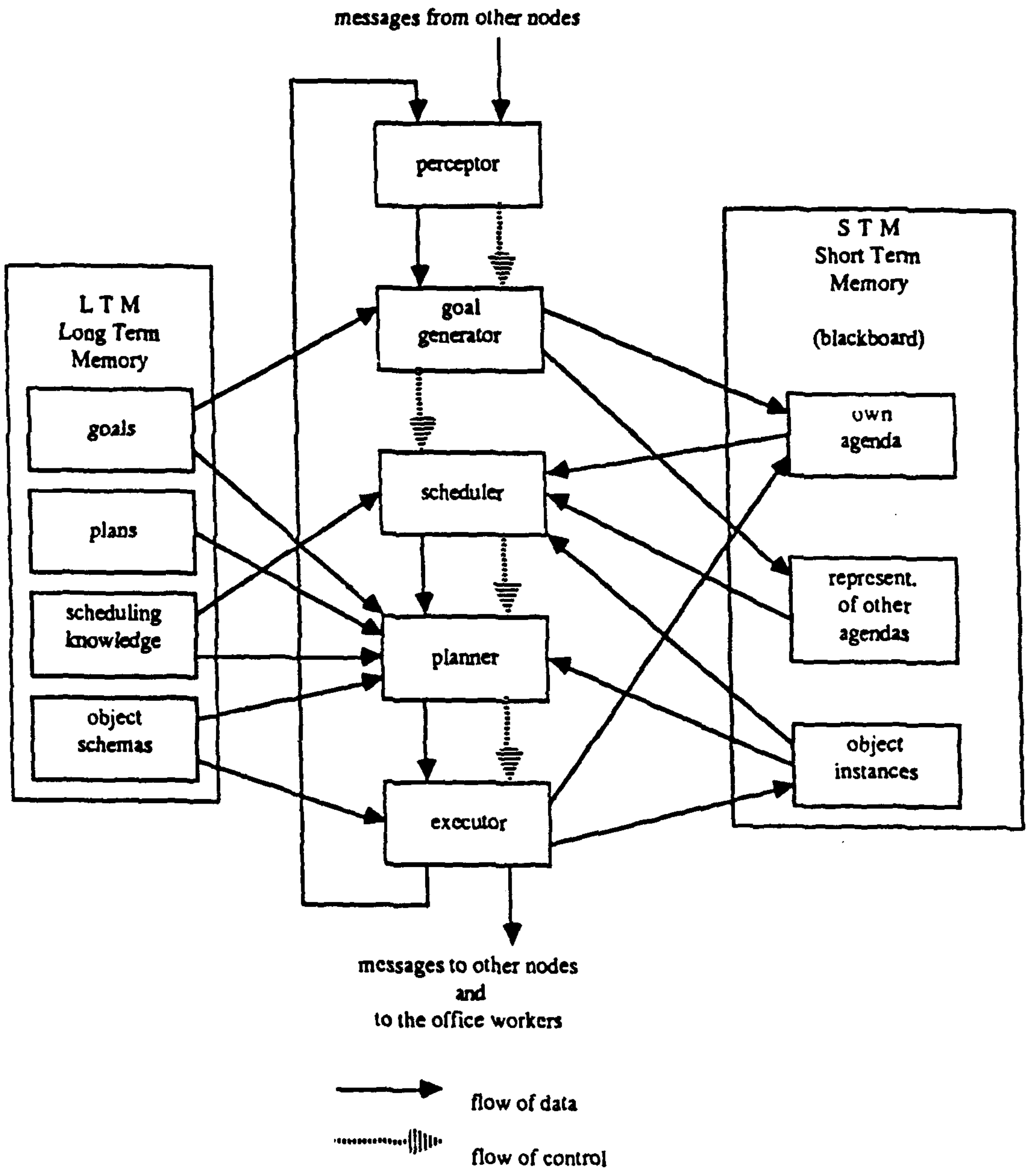

Figure 4. Node Architecture

Representations of two top-level goal types in OFFICE are given in Figure 5. A number of OFFICE plans are illustrated in Figure 6.

```
(goal HIRE-PERSON
  (Typical-Responsible-Agents RP F;)
   (Typical-Plan Hire-plan)
   (time-scale days)
   (importance 2)
  (beneficiary Research-Project)
  (Supergoals  Conformity-between-Workers-and-Work-Amount
              Use-All-Resources-Available)
  (Trigger (or (sum of expenditures is less than available funds)
              (there are less workers than needed to do work)))
)

(goal PURCHASE
  (Typical-Responsible-Agents  PI)
  (Typical-Plan Purchase-plan)
  (time-scale days)
  (importance 1)
  (beneficiary (PI S RA GS))                    ;any member of RP
  (Supergoals  Get-Equipment
              Use-All-Resources-Available)
  (Trigger  (and (there are funds available)
               (the beneficiary's resources are incomplete,
                compared to the typical resources allocated
                to this role-holder)))
)
```

Figure 5. The goals HIRE-PERSON and PURCHASE.

```
(Purchase-plan
   (icon PU)
   (With ((Agent RP.member)
          (Object POBJ) ;is not specified at the moment of
                         ;plan instantiation
          (Amount int)  ; - " -
        (    ..))
   (is ((specify-item-to-buy (agent = RP.member
                          object = item
                          approx-price = int))
       (make-document (agent = RP.member
                      doc-type = purchase-request
                      object = item))
       (communicate (agent = RP.member
                    destination = Secretary
                    object = purshase-request))
       (plan-selector ((process-purchase-order (agent = Secretary
                                          object = item))
                   (process-purchase-requisition (agent = Secretary
                                              object = item)))
          ;both plans are compound
       (complete-purchase (agent = Secretary
                        object = item))
  )
    (preconditions  (Agent has money  Vendor has Object))
    (effects (Agent has less money.
             Vendor has more money.
             Agent has Object))
   )
```

```
(Process-Purchase-Order
   (with (agent = secretary
        object = item
        destination = vendor
        price = int))
   (preconditions (approx-price < $250 ))
   (is (make-document (agent = secretary
                  doc-type = purchase-order
                  object = (item vendor)))
      (communicate (agent = secretary
                object = purchase-order
                destination = vendor)))
)

(Complete-Purchase
  (with (agent= secretary
       object = item
       source = vendor))
  (is (# (communicate (agent = vendor
                 destination = secretary
                 object = item))
       (communicate (agent = vendor
                 destination = secretary
                 object = bill)))
     (check-goods (agent = secretary
               object = item))
     (plan-selector ((pay-for-goods (agent = secretary
                            destination = vendor
                            object = item
                            amount = bill.amount)
                 (cancel-goods  (agent = secretary
                            destination = vendor
                            object = item
                            amount = bill.amount))))
 )

 (make-document              ;a primitive plan
  (with (agent = person
       doc-type = purchase-request | bid-request | purchase-order |
               disbursement-form | item-rejection-form | cv | offer ..
       destination = person | organization
       object = (item, price ...)  ;parameters that are mentioned in
                            ;the document
  )
  (is primitive)
  (effects     )  ,the document exists
  )

  (check-goods
   (with (agent = person
        object = item))
   (is primitive)
  )
```

```
(Communicate
 (with (agent = person
       destination = person
       object = message
       type = assertion | question | order
       instrument = medium))
                  ;medium is a list of phone, mail, csnet, etc.
 (is primitive)
 (action-for-primitive))


(Get-Info
  (with (agent = person
       object = message
       was-invoked-by = person3
       instrument medium))
 (is (plan-selector (ask-track (agent = person1
                               destination = person2
                               was-invoked-by = person3
                               instrument = medium))
                    find-track (agent = person
                                object = message)))
 (effects (communicate (agent = person1
                        destination person3
                        object = message
                        instrument = medium) ))
)
```

Figure 6. A sample of OFFICE plans.

Local and Global Scheduling Knowledge

A special part of the knowledge in OFFICE is the knowledge about scheduling and prioritizing activities by the nodes in the network. A part of the scheduling knowledge is static, that is, is considered true irrespective of the circumstances in which the scheduling takes place. The other portion of the scheduling knowledge is dynamic in that it takes into account the presence of other goals on the node's agenda and the suggests the ways of dealing with goal conflict.

The static part of an agent's scheduling knowledge includes the authority and responsibility structure of the office and the profiles of actual workers in specific roles within the organization. The latter includes both the workers' stated attitudes and preferences with respect to the types of jobs they are performing and their personality profiles, as understood by the current agent, on the basis of which the above attitudes and preferences can be inferred.

The dynamic part of this knowledge includes a snapshot of problem solving activities from the current agent's perspective; a representation of time and other resources; and a set of operational rules that contribute to the task of scheduling. In this paper we will present these rules as a set of scheduling heuristics, bypassing, for the sake of clarity and understandability the actual formalism in which they are expressed. The scheduling heuristics are as follows:

1. Static priorities are stated for all the types of top-level goals. The instances of goals with higher static priorities will be preferred. Thus, for instance, *Purchasing* can be declared more important than *Hiring*.
2. The more time a goal spends on the agenda, the higher the priority it acquires.
3. The less time the accomplishment of a goal will take (as estimated by an agent), the higher the priority it acquires.
4. The smaller the effort needed for the accomplishment of a goal (as estimated by an agent), the higher the priority it acquires. This rule measures effort in terms of both the amount of energy exertion on the part of the agents and the number of intermediate steps (plans) still estimated as needed to accomplish the goal.
5. If a precondition for a plan selected to achieve a goal is false, the goal's priority goes down; however, for specific types of preconditions and nodes a new goal of satisfying this precondition can be established.
6. The higher the authority of the node responsible for a goal, the higher the priority it acquires.
7. Beliefs about the agendas of other network nodes weigh less in the decision process than the contents of own agenda. For example, if the level of authority responsible for a goal G is *inferred* by a node then it will increase the priority of G to a lesser degree than in the case when the authority level was explicitly obtained as input.
8. If the accomplishment of a goal satisfies preconditions for the execution of a plan (or a number of plans) leading to the achievement of other goals (on any of the goal agendas in the network), the priority of the goal is considered higher.

The influence of prioritizing rules based on the above scheduling heuristics is calibrated to produce a general dynamic priority for every goal on a node's agenda.

2.2. How Do the Agents Operate?

A cycle of processing by each agent involves a consecutive invocation of the perceptor, the goal generator, the scheduler, the planner and the executor (cf. Figure 4).

The **perceptor**

obtains as input (either through the network or from the office worker) *messages* about changes in the world that were received since the previous time cycle (changes are various new states, including results of actions performed by agents in the system).

Input messages are classified according to their *speech act* character. Messages can be either assertions or requests. Assertions can be definitions, opinions, facts, promises, threats and advice. Requests can be questions (request-info) or commands (request-action). Commands are orders, suggestions or pleas. This classification is needed to improve the understanding capabilities of the system (as compared, e.g., with POISE). Also, it allows a clear way of setting goals for the nodes in the network.

Next, the perceptor 'understands' these actions in terms of *plans* they are parts of and, correspondingly, in terms of what was the *goal* that the agent of that action pursued. This step embodies the *plan recognition* activity of the system, since, in the general case, it must understand plans of others in order to perform its own plan production.

The **goal generator**

updates the agenda of the node's goals due to new inputs. Thus, the arrival of the following input:

```
(message-14
  (instance-of message)
  (speech-act order)
  (sender PI-1)
  (receiver Secretary-33)
  (proposition (communicate Secretary-33
                            Vendor-101
                            'what is the price of desk-22?'
                             Phone))
```

will lead to the generation of the low-level goal instance 'Get-Info-34' that will be fulfilled when the secretary knows the price of the desk. The plan selection for reaching this goal also is specified in the message: using the telephone. 'Get-Info-34' is added to secretary's agenda of goals to accomplish.

There are thus two kinds of sources of goals for every node. One source is the state of the (office) world (if there are more workers than workstations, the goal of purchasing equipment will be generated and put on the office head's agenda). The other source, as in the above example, is messages (requests and orders) from other nodes.

The **scheduler**

selects a goal to pursue from among a number of candidate goals on the agenda. It applies condition-action rules designed on the basis of the above scheduling heuristics and evaluates the current local state of problem solving from the current agent's perspective. After the scheduler finishes operation, one goal from the node's agenda is selected for processing, and control is passed to the planner.

The **planner**

has the task of providing a plan for the achievement of the goal scheduled by the scheduler. If the agent knows of a *canned plan* that typically leads from the current state to the goal state, the planner simply passes the plan to the executor (see below). If more than one plan can be used to achieve a given goal, the planner selects one of them, based on the scheduling rules. The same heuristics that are used for scheduling goals are also used for plan selection. This is in itself a scheduling heuristic.

The knowledge needed by the planner includes the list of plan types, the list of plans that are believed by the node to be instrumental in achieving the goal selected by the scheduler, and the for competing plans.

The **executor**

is called after the planner selects a plan for achieving the current goal.[1] It performs the following sequence of steps:

a) creates an instance of the chosen plan (if such an instance does not already exist) and lists it under the corresponding goal on the agenda.

b) checks preconditions of the plan; if preconditions do not hold (the plan is not immediately applicable) then sets precondition states to be (sub)goal states; puts them on the goal agenda (note that one of preconditions is 'to have values for all non-optional parameters') else expands the agenda tree by substituting the current plan by the sequence of its component plans.

c) if the first subplan in this sequence has the current node as its agent, it is processed by the executor; if another role in the office is the agent of a subplan, the execution of the current plan is interrupted and a value of its 'status' slot is set to 'suspended' and a corresponding message is issued to the agent of the next subplan.

d) if the plan is 'primitive' the actions specified in it are performed. Then the executor checks whether the plan is completed; if yes, the executor reports this, through the communication channels, to the node responsible for supergoal of the goal which the current plan helped achieve. In this way responsibility relationships are both statically and dynamically introduced into the system.

3. An Example Run of OFFICE.

We will consider 2 top-level goals: Purchase and Hiring. The processing will be traced from the standpoint of one specific network node, that of Secretary (S). At the beginning of the run S already has a *nonempty* agenda of plans and goals. It also has a representation of agendas of other nodes in the network. This representation may contain mistakes, because it is mainly a result of plan understanding activities of the node. The contents of S's agenda and S's belief about the agendas of a sample of other nodes at the beginning of our manual trace are given in Figure 7.

[1] This is a simplification In reality, planning and execution steps can be interleaved

S's own agenda:

```
AGENDA ITEM 1
 Purchase-plan3 (object = terminal)
   communicate (agent = S, destination = PI, object =
                    [communicate (agent = V1, object = terminal,
                                  destination = S)
                     communicate (agent = V1, object = bill,
                                  destination = S)])
   check-goods (agent = PI, object = terminal16)
   plan-selector (agent = S, object =
                    [pay-for-goods (agent = S, destination = V1,
                                  object = bill)
                     cancel-goods (agent = S, destination = V1,
                                  object = (terminal16 bill)])

AGENDA ITEM 2
  process-purchase-order5 (object = book)
    make-document (agent = S, document-type = purchase-order,
                   object =book, destination = V2)
    communicate (agent = S,object = purchase-order, destination = V2)
```

Secretary's beliefs about PI's agenda:

```
AGENDA ITEM 1:
  Purchase-plan3 (object = terminal16)
    complete-purchase (agent = PI, object = terminal16)

AGENDA ITEM 2:
  Hiring-plan2 (RA)
    evaluate (agent = PI, object = candidate3)
    make-document (agent = S, object = offer, destination = candidates)
    communicate (agent = S, object = offer, destination = candidates)
    select (agent = candidate, object = accept/rej)
    make-doc (agent = candidate, object = accept/rej)
    communicate  (agent = candidate, object = accept/rej)
    plan-selector (agent = S, object =
                     [acceptance-track rejection-track])
```

S's representation of RA1's agenda

```
AGENDA ITEM 1
  PU1 (object = book11)
    process-purchase-order (agent = S, object = book11)
    complete-purchase (agent = S, object = book11)
```

An agenda item consists of the name of a goal and the names of those of the plans selected for its accomplishment that are not yet (completely) executed, with the bindings for their parameters. Plan names are printed in **bold**. Plan names with numbers appended represent plan instances. The above agendas say that the secretary has the plans to facilitate the purchase of a terminal and to facilitate purchasing of a book asked for by a research associate (Purchase-plan1); S believes PI has plans to hire a research associate (Hiring-plan2) and to facilitate the purchase of a terminal (Purchase-plan3). S also believes that RA1 has the plan of purchasing a book (Purchase-plan1). PI is responsible for both g on its agenda; S is co-responsible for the Purchase-plan3. In contrast, S is responsible only for a subplan of the top-level plan Purchase-plan1. RA1 is responsible for Purchase-plan1.

Figure 7. Sample Contents of the Agendas of an Agent.

Now let us trace the operation of OFFICE through a number of time slices starting with the above state, observing the decision S makes and the changes to its agenda due to new inputs.

------ time slice 1 ------

Suppose, there is a message posted on the secretary S's blackboard : message19 from research associate RA2, of type *order*, that asks to get a price for a desk from vendor V by phone. This message is received by S and a new goal, GET-INFO11, is generated and put on its agenda. S also updates its representation of RA2's agenda by adding there the (inferred) plan of buying a desk. Note that the inferred Purchasing goal is not on S's agenda; therefore, S is not responsible for it.

Next, the scheduler must choose one of the 3 goals on the agenda (PU3 P-P-O5 and GET-INFO11) for immediate processing.

In our example the Get-Information goal will be chosen. This happens because the Purchasing goal is out of contention since it is in the stage of waiting for ordered goods (terminal) to come (Scheduling Heuristic 5). The choice is, therefore, between the Process-Purchase-Order and the Get-Information. P-P-O has, of course, been on agenda for a longer time (Scheduling Heuristic 2), but GET-INFO can be performed by just placing a phone call, while P-P-O requires typing out a form (Scheduling Heuristic 4). There is no rush on the book order, so the goal that can potentially be achieved sooner (Scheduling Heuristic 3) is selected (Scheduling Heuristics 2 and 3 prevail in this case over Scheduling Heuristic 4).

Next, a plan **get-info** is found for achieving the chosen goal; this plan is instantiated and the executor runs its first subplan: **communicate15** (agent = S, object = message34, proposition = message19.proposition, destination = V2, type = question, instrument = phone). As a result of that subplan, the vendor is informed about the question.

------ time slice 2 -----

New inputs: a) Message20: a terminal and a bill arrived from vendor V1 b) Message 21: the price for the book arrived from V2.

The messages are perceived and understood as the execution of specific plans traced on S's agenda: a) refers to the two *communicate* plans that are objects of the next component of the plan chosen for the Purchase3 goal instance; b) is the response to message19 above.

The above messages do not lead to the generation of any new goals. The schedulcr now has the following choice: PU3, P-P-O5 and GET-INFO11. P-P-O5 has the same status as at the previous cycle. PU3 is now at a point where the PI must be told that preconditions are met for the execution of the *check-goods* plan (because the terminal arrived). Only one action remains to be performed in GET-INFO11, and that is to relay the information obtained from V2 to RA2.

At this point GET-INFO11 is chosen for the following reasons. S knows that PI is currently in a meeting with a candidate for hiring. Even though the importance of the *check-goods* plan is relatively high (Scheduling Heuristic 1), it cannot be performed at this point (the presence of PI is necessary) and is therefore rated low. GET-INFO11 is closer to completion than the other goals. In accordance with Scheduling Heuristic 3, it is selected, and S sends the plan (**communicate** agent= S, Destination = RA2, Object = Message21.proposition) to the executor.

After this plan is executed, the entire tree for GET-INFO11 is deleted from the agenda.

4. Summary and Status.

We hope we have shown that in order to provide assistance in distributed office environments we need to integrate the agent-centered, the task-centered and the cognition-oriented perspectives. It is important to carefully choose the task and delineate the world corresponding to it. It is equally important to provide an architecture that can support sophisticated scheduling activities by nodes in a distributed problem solving network. At the same time one should try to explore the sources of real-world knowledge that is used as the basis for scheduling. In addition to the observable world situation the scheduling algorithm must have access to the knowledge about the internal states of the processors, or, in other words, the 'personality profile' of the agents to whom the system provides assistance.

The node-level knowledge and processors have been implemented in Zetalisp on a Symbolics 3600 Lisp Machine. We are currently developing the network level of the system.

References

Corkill, D.D., 1982. A Framework for Organizational Self-Design in Distributed Problem Solving Networks. Ph.D. Dissertation, University of Massachusetts, Amherst. (Available as COINS Technical Report 82-33.)

Corkill, D.D. and V.R. Lesser, 1983. The use of meta-level control for coordination in a distributed problem solving network. Proceedings of 8th IJCAI, 748 - 756.

Croft, B.W., L.S.Lefkowitz; V.R. Lesser and K.E. Huff, 1983. POISE: an intelligent interface for profession-based systems. Proceedings of the Conference on Artificial Intelligence, Oakland University, Rochester, MI.

Durfee, E.H., D.D. Corkill and V.R. Lesser, 1984. Distributing a distributed problem solving network simulator. COINS Internal Memo, University of Massachusetts.

Durfee, E.H., D.D. Corkill and V.R. Lesser, 1985. Increasing coherence in a distributed problem solving network. Proceedings of Ninth IJCAI, Los Angeles, August 1985, 1025 - 1030.

Georgeff, M., 1984. A theory of action for multiagent planning. Proceedings of AAAI-84, 121 - 125.

Lesser, V.R., D.D.Corkill, 1981. Functionally accurate, cooperative distributive systems. *IEEE Transactions on Man, Systems and Cybernetics*, SMC-11, 81-96.

Lesser, V.R., D.D.Corkill, 1983. The distributive vehicle monitoring testbed: a tool for investigating distributed problem solving networks. *AI Magazine*, 4, 15-33.

Nirenburg, I., S. Nirenburg and J. Reynolds, 1985. POPLAR: Toward a Testbed for Cognitive Modelling. Technical Report COSC7, Colgate University.

Nirenburg, S., I.Nirenburg and J. Reynolds, 1986. Studying the Cognitive Agent. Technical Report COSC9, Colgate University.

Smith, R.G. and R. Davis, 1981. Frameworks for cooperation in distributed problem solving. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-11, 61-70.

POPLAR: A Testbed for Cognitive Modeling.

Irene Nirenburg, Sergei Nirenburg and James H. Reynolds

Colgate University, Hamilton, NY 13346

ABSTRACT

This paper presents an overview of a cognitive modeling system centered around a personality-oriented planner, and then describes in detail the types of knowledge it uses to make control decisions. POPLAR is a model of an intelligent actor capable of planning sequences of control and domain actions in a simulated world that exists independently of the planner. The world is a simplification of the 'Dungeon' computer game environment. The actor makes control decisions on the basis of situational knowledge as well as its personality characteristics (character traits, physical and mental states) and its beliefs about personality of other cognitive entities in the world. POPLAR is a step toward an AI system whose behavior is psychologically justified and can provide the basis for an experimental testbed in cognitive modeling.

1. Setting the Stage.

The POPLAR planner is a component in a model of an intelligent actor. It is an approximation of the human actor in that:

i) like humans, it possesses multiple goals with associated plans;

ii) like in humans, its control decisions depend upon multiple sources of information, e.g. input from the 'objective' world, its permanent character traits, its temporary physical and mental states, and past experience;

iii) like humans, it is immersed into an 'objective' world, changes in which can be introduced not only by the actor, but also by events beyond the actor's control, making it necessary to deal with non-monotonicity.

We believe that the essence of an intelligent actor's cognitive activity is best described in terms of the following loop:

1) perceive input stimuli (sensory, proprioceptive or mental);

2) generate goals connected with these stimuli;

3) schedule the most important goal instance for the given period of time: the one to which the actor's cognitive resources are allocated;

4) choose (occasionally, create) and

5) execute plans to achieve this goal, including performance of physical, verbal or mental actions that are components of these plans. Executions of the loop provide continuous change and stimulation at several levels. Physical actions introduce changes in the objective world. Verbal actions can provide sensory input for other intelligent actors in the world. Mental actions introduce changes in the world of the actor himself (his event memory and beliefs). So, the actions by the actor and other actors in the objective world change this world, and therefore, provide new inputs for the system.

POPLAR offers a solution to above loop components 2), 3), the non-creative part of 4), and the mertal action part of 5). The visual perception portion of 1) and the physical actions of 5) are **simulated** through interaction with the human user of POPLAR.

In the current implementation there is no natural language capability (i.e. the verbal behavior of 1) and 5) are not addressed). Nor do we tackle in any complete and principled manner the extremely complex problem ɔf learning (one facet of which is the creative part of 4).

The central cognitive and archi.ectural points that distinguish the current version of POPLAR are, in addition to i) - iii) above, as follows:

A. The choice of the type(s) of knowledge for scheduling (cf. 3 above) and selecting (cf. 4 above) activities. We proceed from the assumption that in a non-trivial world these operations should be based on a psychologically justified model of human cognitive behavior. This property makes POPLAR personality-oriented, i.e. provision is made in the present model for introducing personality factors that influence goal generation and plan selection.

B. Decisions concerning the organization of metaknowledge that monitors and directs the cognitive processes of goal generation and plan selection. POPLAR represents such metaknowledge in the same framework as the domain plans (top-level, intermediate and primitive). This allows them to be processed by the same reasoning mechanism.

A discussion of POPLAR's relation to other work in the field is in Section 6.

2. The Conceptual Architecture of POPLAR.

The conceptual architecture of POPLAR, as presented in Figure 1, consists of the following modules:

1) the objective **world**, information from which and from

2) the **regulatory system** of the actor, where the non-cognitive knowledge about the actor's character and physical and mental states is stored (cf. Norman, 1981), is obtained by

3) the **sensor**, which processes this input and produces, in the **short-term memory** (STM) of an actor,

4) the **snapshot**, in which the objects currently perceived by the actor are stored, with their parameters, to be scanned by

5) the **goal generator** component of the reasoning mechanism (the **cognitive** module) which produces

6) the list of **candidate goals**, that contains all the goal instances that the actor has at a certain time, including the ones added after the new input was processed. In making its decisions, the **goal generator** uses the data stored in

7) the actor's **long-term memory** (LTM), which contains knowledge about

   a) the **beliefs** the actor has about

   - **objects** in the objective world, including self-beliefs
   - actor's **goals**
   - domain-specific and metalevel **processes** (stored as **plans**)

   b) the acquired **values** the actor has about these beliefs: what is more important, when and why, etc.

   c) the **event** memory that embodies past experience.

8) The **scheduler** component of the reasoning mechanism chooses (schedules) a goal instance in the list of candidates and selects the appropriate plan for its achievement. The **executor** component of the reasoning mechanism then attempts to execute the plan. Lower-level primitive plans are, in fact, actions that are performed by

9) the **output** module; these actions can introduce changes into the world, into the list of candidate goals and the long-term memory.

3. The Implementation.

POPLAR is an implementation of the above conceptual schema in a concrete application domain. It has been implemented in PEARL (Deering et al., 1981) which runs in Franz Lisp under Unix 4.2.

The world in which POPLAR is immersed is reminiscent of that of the well-known 'Adventure' or 'Dungeon' games. We represent a cave in which POPLAR's actor can find and react to enemies, treasures, tools, weapons, food and other objects. It is important to understand, however, that POPLAR is **not** a game-playing system. We are in the process of applying the system in a different domain (the office world).

At present POPLAR's actor is supplied with three basic goals:

1) 'Don't get killed', dubbed Preserve-Self-1 or PS1

2) 'Don't die of hunger, thirst or fatigue', Preserve-Self-2 or PS2

3) 'Collect as much treasure as possible', Get-Treasure or GTR.

In POPLAR the system is making the decisions about what to do next, while it is the responsibility of the user to provide it with input and means for verification of success of actions. The user, therefore, provides the testing ground for the system's empirical experience in the world.

With this caveat in mind, let us see how POPLAR is organized to allow its actor to 'act' in this environment.

4. The System Architecture of POPLAR.

POPLAR's system architecture (Figure 2) represents the conceptual architecture of Figure 1 with implementation restrictions superimposed.

In the current version of POPLAR the role of the objective world including the provision of its rules, 'the laws of nature', is assumed by the human user/experimenter. The user also interactively introduces and removes objects in the cave and modifies their parameters. (In future versions we intend to implement ongoing changes in the objective world generated by the operation of if-added demons on a World Blackboard (cf. 4.2.).)

The user also either permits or forbids certain primitive operations to simulate the actor's pragmatic experience. For example, the user might forbid the actor to pick up an object that is 'too heavy' but previously believed by the actor to be manipulable. This natural state of affairs underscores the difference between the objective world and the world of POPLAR's actor and his beliefs. It is also a means of modeling mistakes (a necessary first step in trying to learn how to recover from them)

The sensor and the output block are simulated in POPLAR's monitor (though mental actions are performed by demons (see below).

When the user decides to add an object to the current world, it does it by listing it on the **world blackboard** (WBB), the data structure interfacing the objective world and the world of POPLAR's actor. WBB also contains a clock which guides all temporally spread processing.

The STM of POPLAR's actor has the reasoning mechanism (the monitor and the executor with their associated bookkeeping functions, demons) permanently connected with it†.
STM contains one-instance metaplans: the goal generator and the scheduler. STM also includes the **actor blackboard** (ABB), which contains slots relating to the current state of POPLAR actor's activities, including notably the **agenda** of activated goal instances.

POPLAR actor's LTM contains his objects, plans, rating functions and history. Cf. a detailed discussion in 4.1.

POPLAR actor's knowledge about his own regulatory system and that of others is linked in the implementation with the representation of these objects in LTM. In addition to knowledge about objects, LTM contains knowledge about plans, history of processing and proper scheduling and selection.

Let us discuss the components of POPLAR in greater detail.

4.1. LTM.

4.1.1. Objects.

Several typical object frames and the semantics of their slots are describ*e*d in Appendix 1. The choice of character traits is at present empirical. However, in parallel to implementing POPLAR, we have been conducting extensive psychological experiments seeking to establish the set of 'primitive' personality characteristics and their mapping into more complex notions that are used by intelligent actors in personality-based decision-making. A separate set of experiments will determine the primitives for specifying mental states of the actor.

4.1.2. Plans.

POPLAR's knowledge about the dynamics both in the objective world and in the actor world is represented as a set of declarative structures called plans.

Plans in POPLAR are classified into several groups (cf. Figure 3).

First, there are domain plans that describe actions in the world and metaplans that describe the processes that manipulate other plans. These include such plans as the goal-generator (gg), the plan-selector, the agenda-scheduler (as), etc. Second, there are top-level plans whose instances appear in POPLAR's agenda as representatives of the three main goals; and primitive plans that are no further decomposable into sequences of actions and provide the proper framework (of preconditions, effects, etc.) for their main action.

The plans that are neither top-level nor primitive are called intermediate. Intermediate plans are never scheduled other than in the process of expanding a top-level plan. There are no intermediate metaplans. Also, all of the metaplans are primitive (decomposable), and two of them, at the same time, top-level.

† The monitor, the executor and the bookkeeping functions stand out among the components of STM in that they are not 'conscious' functions; the actor performs them 'instinctively', while of other elements of STM the actor is consciously aware.

To illustrate the above discussion, consider, for instance the top-level plan of dealing with enemies, such as, in the POPLAR objective world, snakes, crocodiles or trolls. The actor can have a number of (intermediate plan) possibilities: to fight, to flee, to hide, to wait and see what happens, etc. All of the above are decomposed into strings of lower level plans (such as get, take, find, etc.), and the process of decomposition continues until all the final decopositions contain only primitive plans (such as, for instance, move or take).

Plans in POPLAR are represented in a modified version of the language EDL (Bates et al., 1981; cf. also Croft & Lefkowitz, 1984). The frame for a plan contains the following slots (clauses):

ID	the name of the plan
TOP-LEVEL-FLAG	is this plan top-level?
IS	contains the temporal and causal expansion of the plan
COND	used to pass parameters ('propagate constraints') to lower-level plans upward propagation will be added for the plan recognition task
WITH	specifies the parameters with which the current plan will be processed
CONTROL	contains predicates to choose whether to execute optional steps in the plan this slot has the form of an a-list: (<(Control# <s-expr>)>*
PRECONDITIONS	predicates that allow the processing of the current plan to start; differ in principle from CONTROL predicates by being independent of the current context of plan processing
STATUS	one of 'on-agenda', 'executed', 'succeeded', or 'failed'; used for communications with the reasoning mechanism
ACTION-FOR-PRIMITIVE	if plan is domain primitive permission is requested for its completion and the main action is performed (the rest being 'effects')
TIME	number of time cycles the plan takes (only for primitives) – either integer or s-expression that evaluates to integer
RATING-FUNCTION	scheduling knowledge, see below
EFFECTS	auxiliary (including bookkeeping) modifications accompanying the success of the plan

Figure 4 contains a grammar of the plans implemented in this version of POPLAR, and Appendix 2 contains annotated examples of POPLAR plans.

4.1.3. The Rating Functions.

The knowledge that POPLAR's actor has about the relative importance of a top-level goal instance and the relative merits of one plan of action aimed at achieving a goal over another is embodied in the **rating functions**. In the current implementation rating functions are associated with every plan that can serve as parameters in the plan-selector and the agenda-scheduler.

The rating functions calculate a numerical value for a plan, a rating, in all situations where a choice among plans that can be pursued is possible. They draw upon:

a) knowledge of the objects involved in an objective world situation;

b) the character traits, mental and physical states of the actor;

c) the actor's beliefs about the character and current physical/mental state of any other cognitive entity participating in the situation;

d) the actor's event memory, the history of past processing.

Thus, if two actors, Actorl and Actor2 find themselves in an identical threatening situation (e.g. a snake), but one of them is more courageous (a character trait) and/or is in general not very fearful of snakes (a situational characteristic), the actors may respond to the situation by choosing different plans (e.g. Flee for Actorl and Fight for Actor2) or even altogether different goals (while Actorl is likely to choose 'Preserve-Self' against the snake – because high levels of attention to threats can be expected from actors with low courage values; Actor2 may choose, say, an instance of 'Get-Treasure', because the snake is not serious enough a threat).

The construction of rating functions is an empirical process of gradual refinement. Even without changing the knowledge used by the rating functions one can always manipulate parameters of a function to calibrate its results.

One of the objectives of the psychological experimentation conducted in parallel with this project (cf. Section 7) is to better understand the nature and parameters of the rating functions.

Examples of rating functions are presented in Appendix 3.

4.1.4. History

This part of the actor's LTM contains his memory of past processing. In principle, history can have a very rich structure and be used in a wide variety of ways. Special demon-type functions can be defined, for example, to introduce modifications into the actor's beliefs about objects and processes in the real world based on certain patterns in the event memory†. This is one more location in POPLAR's architecture where a measure of learning can and is planned eventually to be introduced.

At present the history contains only two types of data: a) the record of all the recursive calls to the executor in the form of paths that the processing took in the grammar of plans and b) a list of the objects (physical or mental) found by all instances of the Find plan; this knowledge is used to retrieve the status and the results of various plan instances. A typical instance of history is presented in Appendix 4.

† An example. Suppose that in an internalized plan for fighting crocodiles 'stick' is listed as the best weapon. Then during one invocation of the plan Fight (Actor Crocodile Weapon) no stick could be found, so that Actor had to use a gun. It appeared that both the results were better and the fatigue increase was smaller. After this plan execution was written into the history, a comparison is made (by the abore demons) and the old belief about the stick being the best weapon is changed.

4.2. Actor/World Interfaces: the blackboard.

As mentioned above, in the current implementation of POPLAR there are two blackboards that facilitate links between the world and the actor.

4.2.1. The World Blackboard.

WBB is used for introducing new sensory input and managing temporal relations in the system. POPLAR has time-triggered demons that automatically update the values of the actor's physical and mental state based on the amount of time he engages in a certain activity.

In their simplest manifestations, the time-related modifications deal with increasing the actor's hunger, thirst and fatigue values at predetermined independent rates. When the value of any of the above parameters becomes greater than a predefined threshold, a message to this effect automatically registers in ABB's 'states-perceived' slot, as a result of which at the next pass of the monitor an instance of Preserve-Self-2 goal will be activated, and the corresponding top-level plan will appear on the agenda.

Temporal knowledge is also used to implement a simple model of attention. A detailed discussion of this mechanism will be deferred till Section 5.

4.2.2. The Actor's Blackboard.

ABB contains information about

a) the list ('objects-perceived') of object instances that the actor has perceived in the current environment;

b) the list ('states-perceived') of all physical states currently perceived that warrant the attention of the goal generator (e.g. the level of hunger above a threshold);

c) the agenda of all top-level plans (the representatives of the main goals) vying for the attention of the cognitive processor of the actor at any given time;

d) the stack ('current-path') of plans currently being executed (from a top-level plan to a primitive).

In future implementations, specifically when plan recognition will be added to the repretoire of POPLAR and the number of actors inhabiting its world will be allowed to be greater than one, the number of ABBs in the system may grow to as many as the square of the number of actors. This is because every actor stores his beliefs about other actors' activities in instances of ABB attached to his representation of these other actors. Therefore, each actor theoretically can be aware of all the other actors and contain an ABB for each, including himself.

A typical example of ABB and WBB contents is presented in Appendix 5.

5. The Algorithms.

5.1. The Monitor.

The top-level control function of POPLAR, the monitor, is an infinite loop (our actors do not die – only if killed by enemies!) which performs the following tasks:

a) it maintains contact with the user (to obtain new input);

b) it starts the executor loop that consists of i) processing new input; ii) scheduling an action; and iii) executing this action

c) it displays selected situations in the world with the help of a (rather simple) graphic interface.

5.2. The Executor.

The main bulk of POPLAR processing is performed by the executor. To understand how POPLAR works it is sufficient to trace a cycle of its activities.

The executor is called many times during one monitor cycle. First, it processes the goal generating plans using the information obtained by the monitor from the objective world as well as that from the actor's regulatory system. As a result of this stage, the agenda of competing top level plan instances is updated. Second, it executes the agenda scheduler plan select the best candidate plan. Finally, it executes the chosen top-level plan (this involves a number of recursive calls to the executor). When eventually the execution ends, the result of current processing (success or failure) is reported, and a new cycle of the monitor begins.

Omitting a few overly technical details, we can describe the activities of the executor generally as follows:

a) obtain a plan to process; if it is not a plan **instance** (the agenda holds only plan instances, e.g. 'GTR19'; whereas **is** clauses of plans are formulated in terms of plan **types**, e.g. 'Find'), create a new instance of this plan;

b) check the plan's **preconditions** clause; if preconditions do not hold, report failure and its reason and exit; otherwise,

c) expand the plan by considering its **is** clause: call the **is** clause parser;

   c') if the **is** clause is 'primitive', then **action-for-primitive** is performed (most often this is a request to the 'laws of nature', the user, to allow an update in the objective world, e.g. a move by the actor; if the permission is given the processing proceeds as specified in e') below; if the action is not allowed the processing proceeds as in e''). (Let us repeat that the semantics of this situation is that the actor's beliefs about the objects and/or plans and/or values are somewhere wrong, as a result of which some indication of imminent failure must be given to prevent the 'automatic' success of most planners in situations where the internalized preconditions of a plan hold.)

   c'') if the IS clause is not 'primitive' the parser has to make specific control decisions: i) whether to execute an optional subpath in the IS clause; ii) which of any possible number of disjoined subplans to choose for fulfilling the current plan. (The ability to choose one of a number of 'shuffled' subplans (those that can be fulfilled in any temporal order) will be added to POPLAR in near future.) The knowledge about whether to execute an optional subpath is encoded in the **control** slot of the plan whose **is** slot is parsed. The knowledge selecting one of disjoined subplans is contained in the plan-selector metaplan and the **rating function** slot of the current plan. Once it becomes clear what member of the **is** clause should be processed first, the executor

d) calls itself recursively with this plan; this event is recorded on ABB (cf. 4.2.2), specifically, in a data structure called **current-path**; the old content of **current-path** is added to **history** (cf. 4.1.4.).

e') if an **is** clause is processed to its end (cf. the special case of 'primitive' in c') above), the **status** slot of the plan is set to 'succeeded' and the **effects** clause is evaluated;

e'') if for some reason the **is** clause cannot be processed to its end, the **status** slot is set to 'failed' and

f) this information is communicated to the parent plan; the current plan is discarded from the **current-path** stack, and the processing of the **is** clause of the parent resumes. When, eventually, the outcome of the top-level (and bottom-of-stack in **current-path**) plan becomes known, then

g') if it succeeded, then the **effects** clause is evaluated and the corresponding top-level plan instance is removed from the agenda (and added to **history**);

g'') but if it failed, then, assuming that the need that had spawned this goal has not been satisfied, the executor creates a new instance of the same top-level plan and adds it to the agenda instead of the failed one (which goes to **history**).

h) a new cycle of the monitor starts.

5.3. Modeling Attention.

The previous section described the normal flow of control in a monitor cycle. In real life, however, an actor can hardly have the luxury of being able to finish the processing of a top-level plan without taking in new information about the objective world. In future implementations of POPLAR the temporal relations among plans will be elaborated to include the many possibilities of concurrent processing (cf. Allen, 1983a, for the description of a model of time that can be adapted for use in our model; cf. also McCue & Lesser, 1983) for a temporal logic in the POISE system).

At present, however, POPLAR reacts to this problem as follows. When a top-level domain plan is chosen from the agenda and passed over to the executor, its rating is used for calculating the number of time cycles this plan will be allowed to execute without being interrupted. The more 'important' the plan (i.e., the higher its rating) the longer it is allowed to execute uninterrupted. This current programming device is a rough simulation of the actor's concentration or attention to the task. Intuitively, the more immersed one is into a task, the less one would be inclined to be distracted by new sensory inputs. It is obvious that character traits and physical/mental states affect the ability to concentrate.

When an interrupt occurs, the entire **current-path** is suspended; the instance of the top-level plan is deleted from the agenda and another instance is created and added to it (the new instance reflects the knowledge of the stage at which the processing was suspended; **history** is used for this purpose). Then the monitor starts a new cycle.

5.4. An Example.

Suppose we want to test POPLAR's performance in the following situation of the world. We want to put the actor in a cave with a rock, a snake and an apple and to set its hunger well above the detecting threshold.

POPLAR acts as follows:

a) asks the user whether he wants to remove certain objects from the world; we do not, so we answer in the negative;

b) asks the user whether he wants to change any of the properties of the objects already present in the world; this is the time to input the (high) value of actor's hunger;

c) asks whether the user wants to add new objects to the world; we do, since our perception module is simulated, we submit prefabricated instances of objects to POPLAR; we write: (rock1 snake1 apple1).

d) adds the above object instances to ABB.objects-perceived. Since snakes spawn the need for protection (by virtue of their being descendants of 'creature'), the goal Preserve-Self-1 is activated (by the gg-input plan) and an instance of its corresponding top-level plan, PS10, is added to ABB.agenda (which already contain the unique instance of the Agenda-Scheduler plan that resides there permanently); appropriate messages are issued by POPLAR;

e) detects, through gg-states-perceived, the actor's hunger; 'hunger' is added to ABB.states-perceived and an instance, PS20, of the top-level plan of the Preserve-Self-2 goal is added to ABB.agenda; appropriate messages are issued;

f) since no objects had been present in the world before, and, therefore, no changes to their properties could be introduced, gg-objects-perceived will not be needed in this case, a message to which effect will be issued;

g) at this point ABB.agenda is (agenda-scheduler PS10 PS20); the monitor calls the executor with the scheduler plan, as a result of which the two domain plans receive ratings. Suppose now that PS20's rating is higher (because the actor is very hungry and at the same time not too afraid of snakes); this being the goal choice,

h) the scheduler is called with PS20(Actor hunger); checks its preconditions (empty!) and expands its **is** clause; the plan-selector, using the rating functions in the plans Eat, Drink and Sleep, decides to select Eat; an instance of Eat, Eat0(Actor) is created and pushed onto **current-path**

i) Eat0's preconditions are checked (empty!), and its own **is** clause is expanded; this means creating a new instance of Find, Find0(Actor food Actor.inventory), – that is, first the actor wants to check whether he is carrying some food;

j) the controll predicate chooses whether to execute the optional Find and Get plans; the predicate essentially returns 'true' if the previous Find failed; the optional subpath corresponds intuitively to the situation when the actor looks around him trying to find some food; suppose now that Find0 fails; in this case,

k) Find(Actor food ABB.objects-perceived) is executed; Find's IS clause is 'primitive'; its **action-for-primitive** is to record the object found; Find1 finds apple1;

l) next, GET0(Actor, apple1) is created and pushed onto **current-path**; this instance's **is** clause consists of Move followed by Take; (in reality, Get has three parameters, the third being the indication of the time that the actor can spend on retrieving the object – this is very handy as a precondition if, for example, an adversary can reach the desired object first!)

m) Move0(Actor Apple1) is created and pushed onto **current-path**; Move is a primitive plan, so its **action-for-primitive** asks the user for permission for the actor to move to the point where apple1 is. We grant the permission; Move0 evaluates its **effects**, updating the positions of the actor and all the objects in his inventory and sets its **status** to 'succeeded';

n) **current-path** is appended to **history**; Move0 is popped, and the next plan in the IS clause of Get0 is pushed onto **current-path**: Take0(Actor apple1);

o) Take0 is primitive; its processing is similar to the processing of Move0; it succeeds, one of its effects being that apple1 is added to the actor's inventory, and after manipulations with **current-path** similar to those in l), Ingest0(Actor apple1) is sent to the executor;

p) Ingest is primitive; suppose we allow the actor to ingest the apple; then, after the appropriate (and by now familiar) bookkeeping operations, we find ourselves at the point where

EatO is proclaimed as succeeded; at this point we evaluate its **effects** and pop it from **current-path** (which at the time contains only PS20, known to have succeeded);

q) **effects** of PS20 are evaluated (the hunger level of the actor is decreased, and a message to this effect is issued), and with this PS20 is popped from **current-path**, which remains empty; this signifies the completion of a cycle of the monitor.

6. Related Work.

In designing and implementing POPLAR a number of conceptual and technical decisions and choices had to be made. The following is an incomplete, though representative list.

1) hοw does one approach, and justify, construction of a multi-faceted system when little is known about the peculiarities of its components? Where is the starting point?

2) how might the problem of personality influences upon cognition be addressed?

3) within cognitive component, how are goals and plans related? How are they each related to such concepts as needs, drives, performance, etc.?

4) What is the structure of the planning module in cognitive systems? How is the scheduling of the cognitive system's activities performed?

5) What is the relationship between the use of internalized (canned) and newly created plans?

6) What is the relation between plan production and plan understanding?

The realization of the above and some additional problems was instrumental in the design stage. While not all of the decisions have been already made at this stage, our desire was to avoid design choices that would preclude or hamper a future improvement or extension.

None of the theoretical or design decisions were made without the influence of other, previous, related work. In this section we briefly review the bases for the various decisions as well as mention other work on the problems we faced.

Fundamental to the development of POPLAR was the approach to the task, faced by most cognitive modelers, of building a structure consisting of a number of distinct constituents, the details of many of which wer (and at present remain) unknown. How does one construct a global model when many of its components are uncertain, and each one is itself a mystery? Here we adopted teh attitudes advocated by Haugeland (1981), who suggests that it is appropriate to study an entire information processing system (IPS), consisting of several modules each of which (plus the IPS itself) is a black box, without first completing the study of the components; thus, we studied the cognitive actor even though we had not (and, obviously, could not) first provided an account for perception and performance.

Norman (1981) was very instrumental in specifying the tasks to be tackled in cognitive modeling. We also owe much to Anderson's (e.g. 1983) work on the architecture of cognitive entities. Sloman & Croucher (1981) discuss the introduction of motives, moods, attitudes and emotions in natural and artificial intelligent systems. Although no formalism is suggested for encoding this type of information, the general thrust of the approach is valuable for those who consider the introduction of certain personality characteristics into a class of AI systems. Wallace (1981) addresses similar problems in the context of learning.

Uhr & Kochen (1969) is an early work that addressed similar issues. Many of the importnant points for POPLAR have been anticipated in that work. Unfortunately, Uhr & Kochen's approach cannot be even called knowledge-based. It was an attempt to perform an important piece of research with inadequate means.

Wood (1983) discusses planning in a dynamically changing world with multiple actors. Her system, AUTODRIVE, uses the world of the automobile driver as the domain. Although the design of the system depends too strongly on the implementation world, the idea of interaction between the actor and the world (in fact, the mere separation of the objective world and that of the actor -- through a program called SIMULATOR) is very fruitful.

Schank & Abelson (1977) and Schank & Lehnert (1979) informally discuss and catalog human (including interpersonal) goals. Carbonell (e.g. 1979) discusses the use of the concept of personal goals in the context of understanding stories. Wilensky (1983) also discusses everyday goals and metagoals, as well as various cooperative and competitive relations among them.

The relation between goals and plans is an interesting question that had to be addressed in our work. Our solution was to use this term only for top-level goals recognized by the goal-generators but made manifest in the system through the instantiation of a top-level plan. We did not use the concept of goals at lower levels in planning (i.e. we did not use the term 'subgoaling', cf. Lesk, 1984).

It is argued (cf. e.g. Barber (1983) or Berlin (1984)) that subgoaling is preferable to the use of 'canned' plans because if the latter are used then there is no possibility of ever achieving a goal in a non-standard way. But in the subgoaling approach, within the current state of the art, no unexpected results can be obtained either. To introduce these, one has to build a learning system, one capable of creating and not only recreating. But at present the planning of the subgoaling type remains no less 'canned' than the the 'forward' planning.

It seems that these two approaches to planning relate essentially in the same manner in which backward chaining relates to forward chaining in inference making. Our opinion is that the choice between the two is not strategically important and should reflect the peculiarities of the domain and other 'weak' considerations, so typical for AI.

Another important issue related to goals and plans is whether to build systems that in scheduling an action take into consideration the knowledge of how many different plans and/or goals will be furthered by it. The main empirical body of Wilensky's book (1983) is devoted to such issues. Cf. also Hammond (1983) for a philosophically related approach. Hayes-Roth & Hayes-Roth (1979) also want their planner to have this capability. Our position on this topic (cf. also Carver et al., 1984) is that in the type of planners we are building the goal cooperation or conflict does not play a role. We argue that to treat this topic as central in modeling planning in intelligent actors is similar to consider such non-everyday tasks as playing chess and solving differential equations central topics for AI. The latter methodological fallacy has been amply criticized.

General works on planning that immediately influenced this project include Stefik's work (e.g. 1981) on metaplanning and planner architecture. Hayes-Roth & Hayes-Roth (1979) describe a very rich planning domain and offer a good discussion of what the editors of **The Handbook of AI** (Cohen & Feigenbaum, 1982, p. 519; cf. also pp. 22 - 27) call opportunistic planning. It does not seem, however, that a non-trivial, involved implementation of the itinerary planner they suggest is possible.

Hayes-Roth (1984) is a definitive proposal concerning the architecture for planners. It addresses the control problem in AI systems as a whole. It also contains a comparison with other current proposals concerning control. In its architectural part this proposal (in fact, not only this proposal!) draws heavily on the earlier work in the HEARSAY-II speech understanding system that introduced and popularized the blackboard architecture (cf. Erman et al., 1980).

The crucial idea of metalevel reasoning is discussed, with different emphases, in Stefik (1981), Hayes-Roth (1984), Wilensky (1983) and Genesereth (1983).

The basic architecture of POPLAR has a number of common points with that of Wilensky's planner (cf. Wilensky, 1983, pp.22-23). The two models, however, display major differences, notably in the attention paid in POPLAR to the problem of scheduling or in importance attributed to the idea of the independent representation of the objective world. Insufficient attention to scheduling and to describing the planning process at the system level were prominent among the criticisms in some reviews of Wilensky's book (cf. Russell, 1984; Berlin, 1984).

Many interesting ideas about scheduling can be found in Sathi et al. (1984) and Fox (1983).

Work on understanding plans in the POISE (Croft et al., 1983) and Argot (e.g. Litman & Allen, 1984) projects has helped in formulating some parts of our approach.

7. Conclusions and Future Development.

POPLAR is a working system that generates and executes a relatively small number of plans in a rich, though simulated, environment. Its scheduling capabilities actually seem to transcend the immediate necessities of the domain. The system is designed in such a way that both domain planning and metaplanning are performed by one executor (that is, POPLAR can reason about its own actions). A number of features have been included that make POPLAR a model of a human planner in a real world.

At the same time, the possibilities of development and improvement that this basic system offers are probably even more exciting than experimentation with the current version of POPLAR. There are many points at which the system can be improved. Some of them are discussed below.

First (and simplest) of all, the POPLAR actor's knowledge about the objects, goals and processes both in the objective world and its own 'mind' can and will be augmented. In parallel, the control knowledge (rating and control functions) will be constantly adjusted and tuned, both through the introduction of additional character trait, mental state and situation parameters and through devising more appropriate ways of amalgamating them in the decision functions. Extensive experimentation with POPLAR will help to verify such decisions.

In parallel and in conjunction with the POPLAR project, these authors have been involved in designing a general model of human cognitive activity. Initial results of that research are reported elsewhere (Nirenburg & Reynolds, 1983; Reynolds & Nirenburg, in preparation). An aspect of that project extremely helpful to POPLAR is research aimed at deriving a set of 'primitive' character traits, motivations and mental states, such that weighted combinations of them will correspond to the 'higher-level' parameters (e.g. 'aggressiveness') that we would like to use in POPLAR's decision functions.

One can see that the above are actually two separate problems: 1) to extract primitives; 2) to express complex entities in terms of the primitives. It was decided to adapt the primitives suggested by Cattell (cf. e.g. Cattell & Child, 1975). Extensive psychological experimentation with humans is pursued in order to find answers to the second problem. The benefits of having a system that boasts psychologically valid (and not 'folk psychology'-based) control parameters are enormous and self-evident. And, therefore, this is one of the most immediate improvements we plan to make.

We also plan to add plan understanding to plan production. The world is inhabited by more than one cognitive actor (consider, for instance, the trolls in the current POPLAR). In order to behave correctly an actor must be able to discern plans of others. We believe that POPLAR's machinery will be ab'· to handle plan recognition without the necessity to introduce major changes. An actor wil' maintain as many blackboards as there are cognitive actors around. It will assume that all other actors operate in the same manner. It will have beliefs about their character traits, etc. and will 'project' plans for them much in the same manner as it plans.

A logical extension to adding plan recognition is to introduce verbal behavior into POPLAR. There exist a number of interesting approaches to discourse analysis and plan understanding in dial¬gs (e.g. Allen, 1983b; Litman & Allen, 1984; Carberry, 1983; Reichman, 1984; etc.). A study in modifying POPLAR to involve verbal behavior and discourse analysis can be found in Nirenburg & Pustejovsky (1985).

The inclusion of multiple actors into the objective world can lead to the development of an experimental testbed for modeling conflict resolution, cooperation and many more important 'real-life' situations. The possibilities here are definitely substantial and quite unexplored.

The mechanism for modeling attention will undergo serious modifications, as will the treatment of time and the interaction between the actor(s) and the objective world.

And, finally, a most important avenue of improvement is the introduction of learning capabilities to the system. There are many modules in POPLAR where planning can be introduced; and there are many different types of learning to be studied. Some examples of this may be modifying the scheduling behavior depending on results of previous processing or after seeing somebody achieve a goal in a way not previously used; modifying beliefs about objects; being able to 'create' new plans, by analogy or otherwise; and many many more. This topic is one of the more complex ones, but any progress in this direction may have a very beneficial effect on the field of planning in AI.

Bibliography.

Allen, J., 1983a. Maintaining knowledge about temporal intervals. *Communications of ACM*, vol. 26, 832 - 843.

Allen, J., 1983b. Recognizing intentions from natural language utterances. In: M. Brady & R.C. Berwick, eds., **Computational Models of Discourse.** Cambridge, MA: MIT Press.

Anderson, J.R., 1983. **The Architecture of Cognition.** Cambridge MA: Harvard University Press.

Barber, G., 1983. Supporting organizational problem solving with a work station. *ASM Transactions on Office Automation Systems.* Vol.1, No.1, January. 45 - 67.

Bates, P., J. Wileden and V. Lesser, 1981. A language to support debugging in distributed systems. University of Massachusetts COINS Technical Report 81-17.

Berlin, Daniel, 1984. Review of Wilensky (1983). *Artificial Intelligence*, vol. 23, 242-244.

Carberry, S., 1983. Tracking user goals in an information-seeking environment. Proceedings of AAAI-83. Washington, DC.

Carbonell, J., 1979. Computer models of human personality traits. Proceedings of IJCAI-79. 121 - 123.

Carver, Norman F., Victor R. Lesser and Daniel L. McCue, 1984. Focusing in plan recognition. Proceedings of AAAI-84. Austin, TX.

Cattell, R.B. and D. Child, 1975. **Motivation and Dynamic Structure.** NY: Wiley.

Cohen, P.R. and E.A. Feigenbaum (eds.), 1982. **The Handbook of Artificial Intelligence.** Volume III Los Altos CA: Kaufmann.

Croft, W.B. and L. Lefkowitz, 1984. Task support in an office system. *ACM Transactions on Office Automation Systems.* Vol.2, No.3 197 - 212.

Croft, W.B., L. Lefkowitz, V. Lesser and K. Huff, 1983. POISE: An intelligent interface for profession-based systems. Conference on Artificial Intelligence. Oakland, Michigan, 1983.

Deering, M., J. Faletti and R. Wilensky, 1981. PEARL - a package for efficient access to representations in Lisp. Proceedings of 7th IJCAI, Vancouver, BC. 930 - 932.

Erman, L.D., F. Hayes-Roth, V.R. Lesser and D.R. Reddy, 1980. The HEARSAY-II speech understanding system: integrating knowledge to resolve uncertainty. *Computing Surveys*, vol. 12, 213-253.

Fox, M., 1983. Constraint-directed search: a case study of job-shop scheduling. CMU Robotics Institute Technical Report 83-22.

Genesereth, M.R., 1983. An overview of meta-level architecture. Proceedings of AAAI-83 Washington, DC.

Hammond, K.J., 1983. Planning and goal interaction: the use of the past solutions in present situations. Proceedings of AAAI-83, Washington, DC.

Haugeland, J., 1981. The nature and plausibility of cognitivism. In: J. Haugeland (ed.), **The Mind Design.** Cambridge MA: MIT Press.

Hayes-Roth, B., 1984. A blackboard model of control. Stanford University Heuristic Programming Project Report HPP 83-38 (revised August 1984).

Hayes-Roth, B and F. Hayes-Roth, 1979. A cognitive model of planning. Cognitive Science, vol. 3, No.4.

Lesk, M., 1984 Universal Subgoaling. CMU PhD Thesis.

Litman, D. and J.Allen, 1984. A plan recognition model for clarification subdialogues. Proceedings of COLING-84. Stanford, 302 - 310.

McCue, Daniel and Victor Lesser, 1983. Focusing and Constraint Management in Intelligent Interface Design. University of Massachusetts COINS Technical Report 83-36.

Nirenburg, S. and J. Pustejovsky, 1985. Plan Recognition and Production for Verbal (Discourse) and Non-Verbal Behavior. COINS Technical Report, University of Massachusetts.

Nirenburg, S. and J.H. Reynolds, 1983. An architecture for a computer model of human cognitive systems. Colgate University COSC Technical Report 4-83.

Norman, D., 1981. Twelve issues for cognitive science. *Cognitive Science*, vol. 4, No.1.

Reichman-Adar, R. Extended person-machine interface. *Artificial Intelligence*, vol.23, 157 - 218.

Reynolds, J.H. and S.Nirenburg, in preparation. The relationship between motivational traits, goal generation and plan selection.

Russel, Daniel.M., 1984. Review of Wilensky (1983). *Artificial Intelligence*, vol. 23, 239-242.

Sathi, A., M. Fox, M. Greenberg and T. Morton, 1984. Callisto: an intelligent project management system. CMU CS working paper.

Schank, R.C. and R. Abelson, 1977. **Scripts, Plans, Goals and Understanding**. Hillsdale, NJ: Erlbaum.

Schank, R.C. and Lehnert, W., 1979. The conceptual content of conversations. Proceedings of 6th IJCAI. 769 - 771.

Sloman, A. and M. Croucher, 1981. Why robots will have emotions. Proceedings of 7th IJCAI. Vancouver, BC, 197-202.

Stefik, M, 1981. Planning with constraints (MOLGEN: Part 1 and Part 2). *Artificial Intelligence* 16, 111-170.

Uhr, L. and M. Kochen, MIKROKOSMS and robots. Proceedings of IJCAI-69, Washington, D.C., 541 - 556.

Wallace, J.G., 1981. Motives and emotions in a general learning system. Proceedings of 7th IJCAI. Vancouver, BC, 84-86.

Wilensky, Robert, 1983. **Planning and Understanding**. Reading, MA: Addison-Wesley.

Wood, S., 1983. Dynamic world simulation for planning with multiple agents. Proceedings of IJCAI-83, Karlsruhe, Germany.

Appendix 1. Objects in POPLAR.

Representation of OBJECTS in POPLAR.

We present objects in two ways: first, in the way the object is stored in LTM, and second, from within a POPLAR run (as an annotated script). The difference is due to the inheritance of parents' properties by children in the hierarchy.

A.

```
(dbcr exp creature person       ; this is a PEARL header for a frame
        (id person)
        (type creature) ; CREATURE is the parent of PERSON
        (h-process-roles lisp ( (Take Who)
                                (Put Who)
                                (Find Who)))
                                ; the above are the roles in which an instance
                                ; of this type can appear in specified
                                ; processes by virture of its having properties
                                ; of a "human": humans can act as agents in
                                ; TAKE, PUT, and FIND
        (mental-state struct)   ; humans have mental states -- cf. the
                                ; default values in the script listing below
        (character-traits struct char-straits)      ; ditto
        (weapon-against ((sword 100 3) (knife 50 1) (rock 10 20)))
                ; POPLAR knows (believes) that weapons against people
                ; include swords, knives and rocks; the numbers (a b)
                ; indicate the efficiency of the weapon and the maximum
                ; range
        (power 50)      ; maximum
        (speed 50)      ; maximum
        (fearsomeness 25)       ; what is the level of fear that such objects
                                ; typically elicit in POPLAR (default: 25)
        (mass 55)
        (inventory lisp)        ; the objects this person is perceived by POPLAR
                                ; to be carrying
```

B.

```
POPLAR> person
        (person (id person)
                (type creature)
; -------------------------------------------------------------------------------
                (o-process-roles ((Find What)))
; this property is inherited by virtue of PERSON's being a
; descendant of OBJECTS: any object can occupy the "what" slot in Find,
; because finding mental objects is recollecting their representations in
; memory
; -------------------------------------------------------------------------------
                (shape nil)
                (color nil)
                (mass 55)
                (position nil)
                (p-process-roles ((Take What) (Put What)))
                (goal-parameters ((PS1 adv)))
; the above properties are inherited by virtue of a person being a descendant
; of PHYSICAL-OBJECTS; the goal-parameters slot specifies an instance of
; what goal is created when an object of this type is perceived. In this
; case the intuition behind the entry is that the appearance of a person
; spawns the creation of a goal instance of Preserve-Self-1, that is,
; persons are perceived by POPLAR as potential enemies.
; -------------------------------------------------------------------------------
                (edibility nil)
; this property is inherited by virtue of PERSON's being a descendant of
; +alive; nil is the default value with the semantics of "unknown"
; -------------------------------------------------------------------------------
                (c-process-roles
                        ((Eat Who)
                        (Ingest Who)
                        (Drink Who)
                        (Move Who)
                        (Attack (Who Whom))))
; the above properties are inherited by virtue of PERSON's being a
; descendant of CREATURE; creatures are considered by POPLAR to be able
; to be agents of eating, drinking, and moving, and agents and objects of
; attacking
; -------------------------------------------------------------------------------
                (weapon-against ((sword 100 3) (knife 50 1) (rock 10 20)))
                (power 50)
                (fearsomeness 25)
                (speed 50)
                (orientation nil)         ; this shows whether this particular person
                                          ; LOOKS at POPLAR at the moment of processing
; ------------------------------------------------------------------------------
; the following are physical states (conceptually, they are part of
; the regulatory system)
                (hunger 0)
                (thirst 0)
                (fatigue 0)
                (injury 0)
                (h-process-roles ((Take Who) (Put Who) (Find Who)))
                (mental-state (nilstruct))
; character traits are a component of the regulatory system
                (character-traits
                        (char-traits    (greed 20)
                                        (pedantism 10)
                                        (hunger-tolerance 5)
                                        (thirst-tolerance 20)
                                        (fatigue-tolerance 20)
                                        (courage 25)
                                        (agression 40)
                                        (impulsiveness 30)
                                        (articulateness 40)
                                        (extravertedness 50)
                                        (locquaciousness 40)
                                        (curiosity 55)))
                        (inventory nil))
```

Appendix 2. Examples of PLAN representation in POPLAR.

```
(dbcr exp PLANS PS1
    (ID PS1)
    (Type PLANS)
    (Top-level-flag yes)
    (IS ((Plan-Selector Fight Wait-and-See))) ; Flee Hide
    (With (Actor Adversary))
    (COND ( (Plan-Selector '(Fight Wait-and-See) ; Hide Flee
                                                    current plan)
                (Fight Actor Adversary)
                (Flee Actor Adversary)
                (Hide Actor Adversary)
                (Wait-and-See Actor Adversary)))
    (Preconditions (and (member 'Adversary (getpath ABB '(OBJECTS-PERCEIVED)))
                        ; Adversary is among the objects perceived by Actor
                            (or (= 'Actor 'self)
                                (and  (strcutrep Actor)
                                        (not (structurenamep 'Actor))
                                        (= (getpath (eval Actor) '(type) '(person))))))
                        ; Actor is either "self" or any instance of person
    (Rating-function (rating-fuc-PS1)))
(dbcr exp PLANS Plan-Selector
    (ID Plan-Selector)
    (Type PLANS)
    (Top-level-flag no)
    (IS (primitive))
    (Action-for-primitive (Schedule-of-plan 'list-of-plans 'calling-plan))
    (With (list-of-plans calling-plan))
    (Time 1)
)

(dbcr exp PLANS Fight
    (ID Fight)
    (Type PLANS)
    (Top-level-flag no)
    (IS (Find (Control1 (Find ! (Control2 (Get)))) (Control3 (Move ! Attack))))
    (COND ((Find Actor (getpath Adversary '(weapon-against))
                     (getpath Actor '(inventory)))
                 (Find Actor (getpath Adversary '(weapon-against))
                    (getpath ABB 'OBJECTS-PERCEIVED)))
                 (Get Actor (car result-find)
                     (div (distance Adversary (car result-find))
                         (getpath Adversary '(speed))))
                 (Move Actor (prog (weapon-range)
                         ; position to move to
                     (cond ((<= (distance Actor Adversary)
                                 (setq weapon-range
                                     (caddr (assoc (getpath (eval (car result-find))
                                                             '(type))
                                                 (getpath (eval Adversary)
                                                             '(weapon-against))))))
                         ; if distance between Actor and Adversary is less
                         ; (or equal) than the range of the Actor's weapon
                         ; then Actor doesn't need to move toward Adversary
                             (return (getpath (eval Actor) '(position))))
                         (t (return (calculate-position Actor
                                                         Adversary weapon-range)))
                      )))

                  (Attack Actor Adversary (car result-find))))

      (Control ( (Control1 (Fight-Control1 Actor Adversary))
                  (Control2 (Fight-Control2 Adversary))
                  (Control3 (Fight-Control3))))
      (With (Actor Adversary))
      (Rating-function (rating-func-fight))
  )
```

```
(defun Fight-Control1 (Actor Adversary)
        (cond ((not (= (car ABB.CURRENT-PATH.Status)
                       'succeeded))
                t)
    ; EITHER the last executed plan (which is Find) failed
                 ((= (cadar Adversary.weapon-against)
                     (cadr (assoc (car result-find).type
                                     Adversary.weapon-against)))
                 nil)
    ; OR actor's current weapon is NOT the most efficient weapon
    ; against this adversary
                 ((lessp
                     (div (times (distance Actor Adversary)
                                 (diff  (cadar Adversary.weapon-against)
                                          (cadr (assoc (car result-find).type
                                                          Adversary.weapon-against))))
                         Actor.character-traits.impulsiveness)
                     fight-control1-threshold))
    ; OR even if the actor does not have the best weapon, he may decide not to
    ; look for a better one -- if the distance between him and the adversary
    ; is too small, if the actor is very impulsive, or if the weapon is not
    ; much worse than the best one
]

(defun Fight-Control2 (Adversary)
    (cond  ((null (cadr result-find)) t)
         ; no weapon was found in the actor's possession
             ((greaterp  (cadr (assoc (car result-find).type
                                          Adversary.weapon-against))
                          (cadr (assoc (cadr result-find).type
                                          Adversary.weapon-against))))
         ; the weapon found "around" is BETTER than
         ; the weapon in the actor's possession
 ]
```

Appendix 3. Examples of POPLAR rating functions.

A. The rating function for the Preserve-Self-1 goal (and top-level plan)

```
(defun rating-func-PS1 (actor adversary)

    (fix (div
              (times
                     (calculate-fear actor adversary)
                     actor.aggression)
              actor.courage)))

(defun calculate-fear (actor adversary)

    (fix (div
              (times adversary.orientation
                     (add adversary.mass adversary.speed)
                     adversary.power
                     adversary.aggr
                     adversary.fearsomeness)
              (times (fix (add1 (log (distance actor adversary))))
                     actor.courage
                     actor.power
                     (add actor.mass actor.speed)))))
```

B. The rating function for the Fight intermediate plan.

```
(defun rating-func-fight (actor adversary)

    (fix (div
              (times adversary.weapon-against.efficiency
                     actor.courage
                     actor.power
                     (add1 adversary.injury)
                     (expt actor.aggression 2))
              (times (calculate-fear actor adversary)
                     adversary.power
                     (add1 actor.injury)
                     adversary.fearsomeness
                     (add1 actor.fatigue)))))
```

Appendix 4. The History mechanism in POPLAR.

; this is the way HISTORY looks at the end of the example run of 5.4.

```
POPLAR> HISTORY
   ((Ingest0 Eat0 PS20)
     (Take0 Get0 Eat0 PS20)
     (Move0 Get0 Eat0 PS20)
     (Get0 Eat0 PS20)
     (Find1 Eat0 PS20)
     (Eat0 PS20)
     (Plan-Selector0 PS20))
```

Appendix 5. Blackboards in POPLAR.

Typical contents of the world and the actor blackboards.

```
POPLAR> WBB
          (World-Blackboard (ID WBB)
                              (NEW-INPUTS troll1 apple2 crocodile2)
                              (TIME (Base-Time (ID Time) (act-time 17))))

POPLAR> ABB
          (Actor-Blackboard (ID ABB)
                              (OBJECTS-PERCEIVED (troll2 sword1 gold-nugget2))
                              (STATES-PERCEIVED (hunger fatigue))
                              (AGENDA PS14 PS22 GTR4 Agenda-Scheduler)
                              CURRENT-PATH (find7 fight3 PS14)))
```

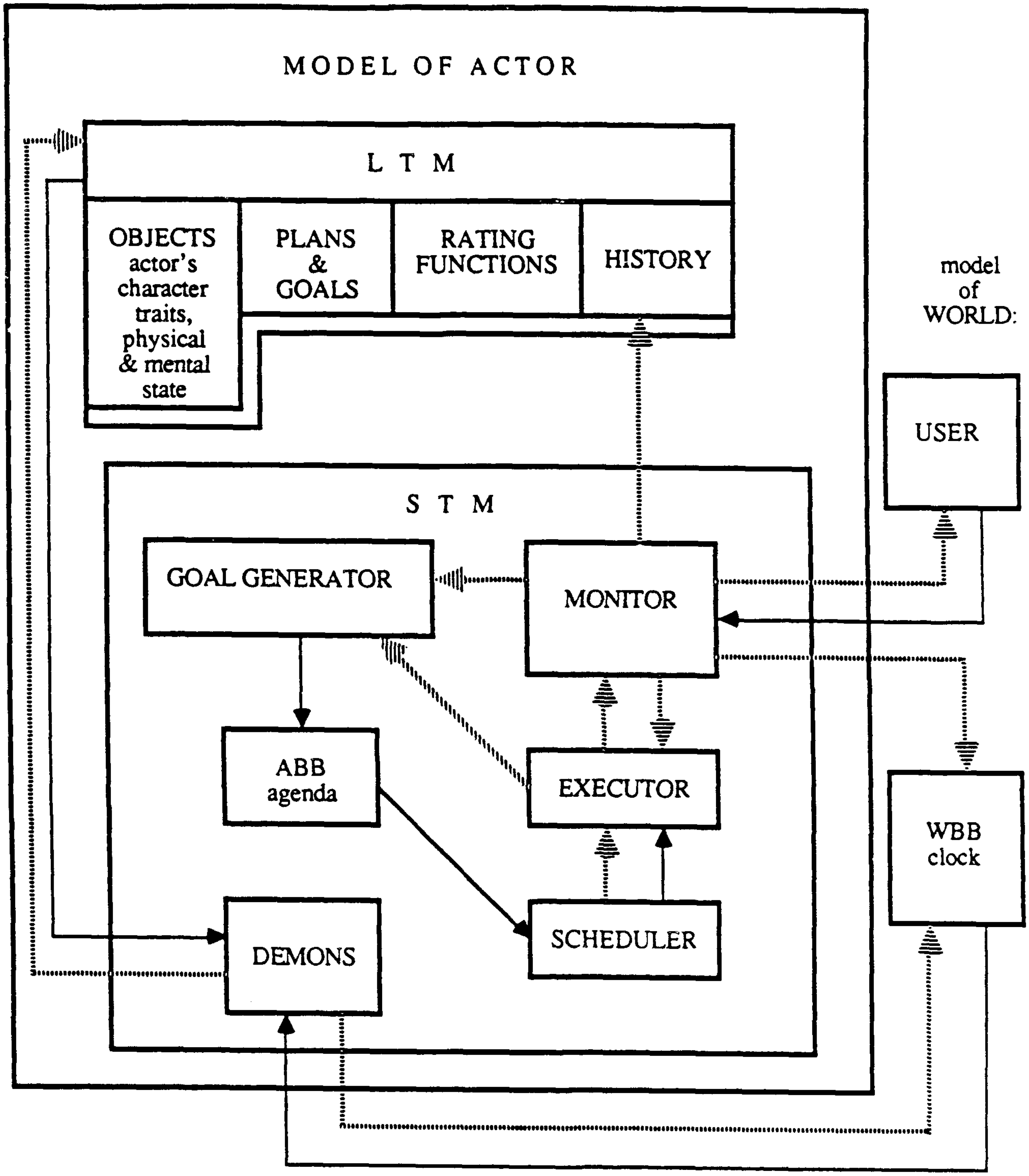

Figure 2. The system architecture of POPLAR 1.3

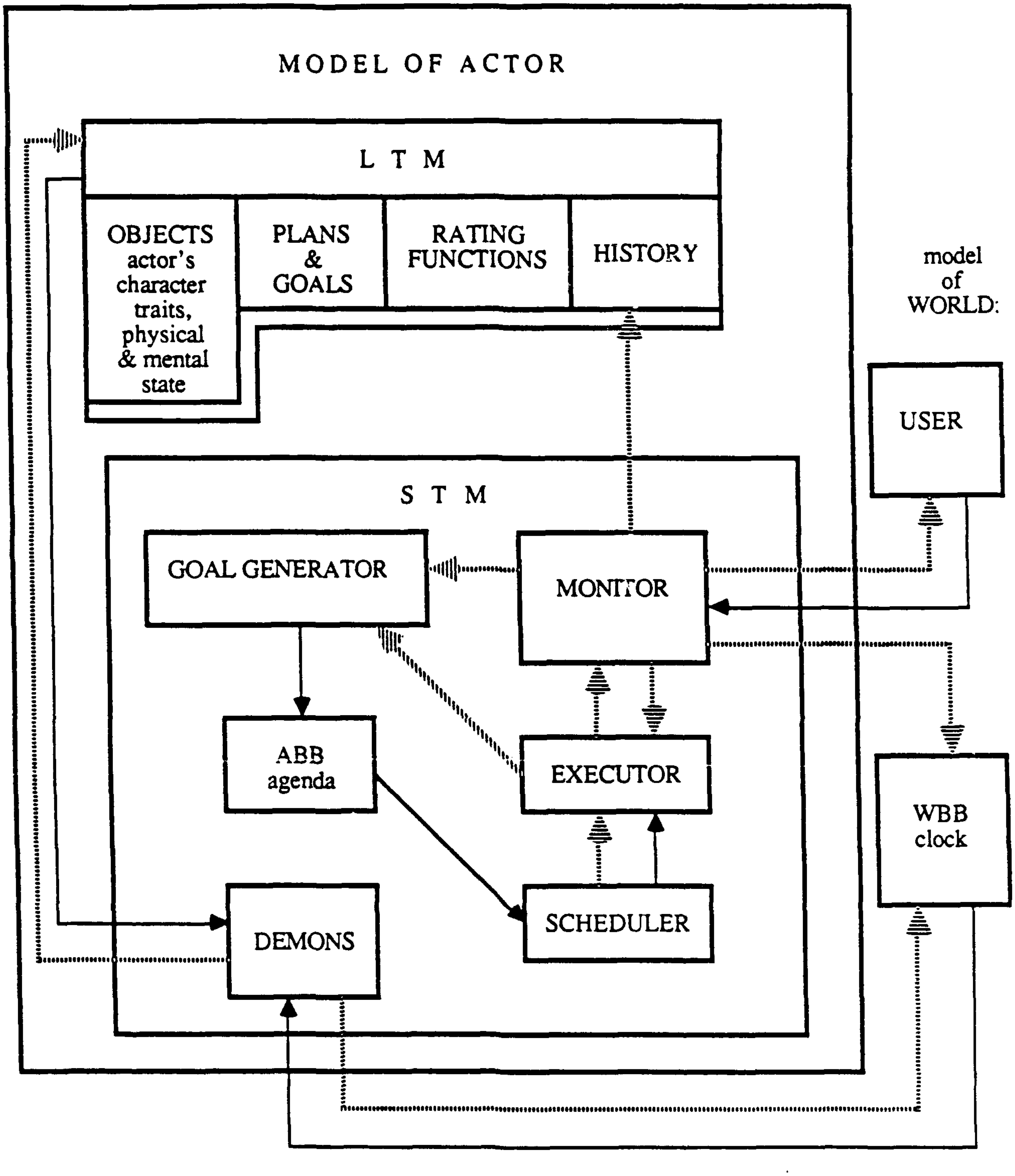

Figure 2. The system architecture of POPLAR 1.3

	TOP-LEVEL	INTERMEDIATE	PRIMITIVE
DOMAIN PLANS	PS1 PS2 GTR	FIGHT, EAT, GET, etc.	move, take, find, etc.
METAPLANS	GG as		as, gg-input, etc.

Figure 3. Classification of plans in POPLAR.

I ::= PS1 | PS2 | GTR | GG ('Goal-Generator') | as ('Agenda-Scheduler')
PS1 ::= FIGHT | HIDE | WS ('Wait-and-See')
PS2 ::= EAT | DRINK | SLEEP
GTR ::= {FIGHT | find} GET
GG ::= gg-input | gg-objects-perceived | gg-physical-states-perceived
FIGHT ::= find {find GET} move attack
HIDE :: find move
WS ::= do-nothing
EAT ::= find {find GET} ingest
DRINK ::= find {find GET} ingest
SLEEP ::= find do-nothing
GET ::= move take

Vertical bars separate disjointed elements; in practice, the 'or-ed' plans are chosen on the basis of their ratings through the application of a special metaplan we call the Plan-Selector, not shown in the grammar;

curly brackets enclose optional plans; the decision whether to execute the optional plan(s) is made on the basis of control functions that are stored in the parent plan and govern the processing of its IS slot;

plans shown in lower case are primitive.

FIGURE 4. A grammar of plans in POPLAR.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. The areas of technical competence include communications, command and control, battle management information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic reliability/maintainability and compatibility.

END