Intelligent Task Management using Preconditions and Goals

Steven H. Schwartz

COINS Technical Report 86-01
January 1986

Department of Computer and Information Science
University of Massachusetts, Amherst
Amherst, MA 01003

This research was supported, in part, by the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, New York 13441-5700, and the Air Force Office of Scientific Research, Bolling AFB, DC 20332 under contract no. F30602-85-C-0008. This contract supports the Northeast Artificial Intelligence Consortium (NAIC). This research was also supported by a grant from the External Research Program of Digital Equipment Corporation.

Abstract

The current implementation of the POISE plan recognition and generation system can be extended in several different directions: a more comprehensive task model, a more intelligent scheduler, a structured VAX/VMS command interface, and a non-volatile task-management database. This paper discusses designs for these extensions and presents TASKMAN, a planning-only subset of POISE which incorporates these designs. TASKMAN is demonstrated in the domain of software development. The paper concludes with discussion concerning the integration of POISE and TASKMAN, cooperative work among TASKMAN users, and task learning.

Contents

List of Figures

List of Tables

1 Introduction

The POISE system, developed at the University of Massachusetts at Amherst, performs plan recognition and generation. Plans are modeled as hierarchies of parameterized tasks. Task constraints restrict parameter values and control both plan execution and task interpretation. [CL84]

POISE can be extended in several directions. Precondition and goal information, part of the task description design, was never implemented; this information would enable the task scheduler to operate more intelligently, as well as simplify the description of the temporal ordering of a task's subtasks. The interface between task definitions and real-world tools cannot be easily modified. Task-management data structures are not maintained across POISE sessions.

We discuss these major extensions and some minor ones. A design is presented for a planning-only subset of POISE which implements these extensions. The TASKMAN system is a partial implementation of this design. We describe TASKMAN's architecture and control structure. TASKMAN is demonstrated in the domain of software development tasks.

The paper ends with a discussion of how portions of TASKMAN, the task database in particular, can be integrated with POISE. We also discuss cooperative work on tasks, learning task sequences, and other planning issues.

2 The POISE System

2.1 Capabilities

The following introduction to POISE is taken from [CL84]:

> The POISE system provides task support on the basis of hierarchies of task (or procedure) descriptions. Procedure descriptions specify the steps typical to the task, the tool invocations which correspond to those steps, and their goals. The ability to combine recognition of user actions and plans through use of descriptions and goals gives POISE great flexibility in the kinds of task support it can provide.
>
> POISE acts as an intelligent interface between the user and the tools available in an office system...Three types of information are used by the interface. The procedure library contains the procedure descriptions. The semantic database contains descriptions of the objects used in the procedures and descriptions of the available tools. The model of a particular user's state includes partial instantiations of procedure descriptions with parameters derived from specific user actions as well as instantiations of semantic database objects.

POISE, running under VAX LISP, a CommonLisp dialect, performs two basic activities. Given a high-level goal, POISE *plans* (and executes) detailed sequences of low-level tasks. It also *recognizes* individual task invocations as components of more abstract tasks and goals, and verifies that task sequences are appropriate and meaningful. [CLLH82]

2.2 The Task Model

In the current implementation of POISE, a task definition has the following components [LC85]:[1]

- **name**: a unique name for the task;

[1] A rationale for this structure appears in [HL82].

- **description**: a text comment;
- **icon**: a symbol used in graphic task-hierarchy displays;
- **is-clause**: a shuffle expression [Gis81] in the EDL event description language [BW83], describing the temporal ordering of the task's subtasks;[2]
- **with-clause**: definitions of task attributes (parameters) in terms of subtask parameters, constants, and semantic database objects;
- **cond-clause**: attribute constraints for the task and its subtasks.

The following components were included in the POISE architecture description, but not yet implemented:

- **precondition-clause**: conditions which must be met before the task can begin;
- **satisfaction-clause** (goal-clause): conditions which must hold when the task finishes;
- **effects-clause**: world state changes caused by performing this task.

A plan is modeled as a tree-hierarchy of tasks. Nodes represent tasks which have been performed, are in progress, or are anticipated. The tree itself is a statement about how these tasks are interrelated. There may be numerous possible interpretations of a temporal task sequence: determining the "right" interpretation is one of POISE's primary functions.

2.3 Room for Growth

The current version of POISE can be extended in several different directions. First, the precondition-, goal- and effects-clauses are not implemented. This information

[2]A task defined in terms of subtasks is called a composite task. The opposite, a primitive task, specifies a VAX/VMS or LISP operation (see Section 2.3).

would allow the task scheduler to function more intelligently. POISE could postpone a task which should not be started, or skip one whose goals are already satisfied. If a task does not meet its goals, POISE would assume that one or more preconditions were not satisfied, and select appropriate remedial tasks.

The shuffle expression used in the is-clause component of the task description is necessarily complex: it must describe the exact subtask execution sequence. This requires fourteen distinct temporal operators. With precondition/goal information, the is-clause can simplified significantly. In fact, it can be omitted entirely: the scheduler is capable of choosing the proper sequence of subtasks to achieve any goal. This paradigm, however, is wasteful in domains where standard task sequences are frequently performed. It is thus desirable to retain the is-clause in simpler form.

There is a potential disadvantage to this scheme. A POISE is-clause simply specifies a temporal ordering of tasks; preconditions and goals require more comprehensive knowledge about the nature of a task. While there is a danger of overlooking a subtle constraint in the latter scheme, it allows a task to be (partially) specified in terms of intuitive and first principles knowledge.

The **primitive** tasks — tasks which specify low-level computer operations, like compiling a program or sending mail — are implemented in POISE's low-level code. It is difficult for a POISE system programmer to add new primitive tasks, and virtually impossible for a user. [CLLH82,LC85] discuss this problem, but do not present specific solutions.

POISE's data structures — the task template library and task instance hierarchies — are maintained in dynamic LISP memory.[3] This has several consequences. The task templates must be loaded every time LISP is started. Task instantiation hierarchies are lost when POISE terminates; a task cannot be suspended from one POISE session to another. Assertions about the state of the world are also lost. This information could be preserved by creating a suspended VAX LISP session, but this is expensive in terms of both time and memory.

Another consequence of this data management paradigm is that POISE users cannot share information. POISE cannot control sharing of objects such as files.

[3]A separate semantic database for managing virtual objects is currently under development by Carol Broverman.

Managers cannot generate reports describing their subordinates' work status. Multiuser cooperation on a task is impossible.

The POISE planner displays the name of each subtask as the subtask is started. However, no instance-specific information, such as parameter values, is displayed. In addition, it is easy to lose track of which top-level task is executing.

3 Improving on POISE

We now develop a design for a task management system which corrects the deficiencies discussed above. This system will more closely achieve the fundamental desideratum:

> ...an intelligent interface that both recognizes and corrects local and global errors while permitting the user to interact with the system at many different levels of abstraction: from existing, low-level resource-oriented commands to high-level commands that represent non-procedural specifications of the desired activity. [HL82]

3.1 A New Task Model

To design a better task management system, we begin with a new task model. The task is modeled as a semantic network. Figure 1 shows the task model using a Bachman data-entity diagram. [Bac69]

A **TASK** has the following attributes:

- a unique **name**;
- a **text description**;
- a **performed** flag, indicating whether this instance of the task finished successfully;
- a **display run description** flag (see Section 3.5).

3.1.1 Parameters

A TASK may have any number of **parameters**. Unlike POISE's generic attributes, this design uses five types of PARAMETER structures: integers, reals, strings, file names, and (named) file contents.

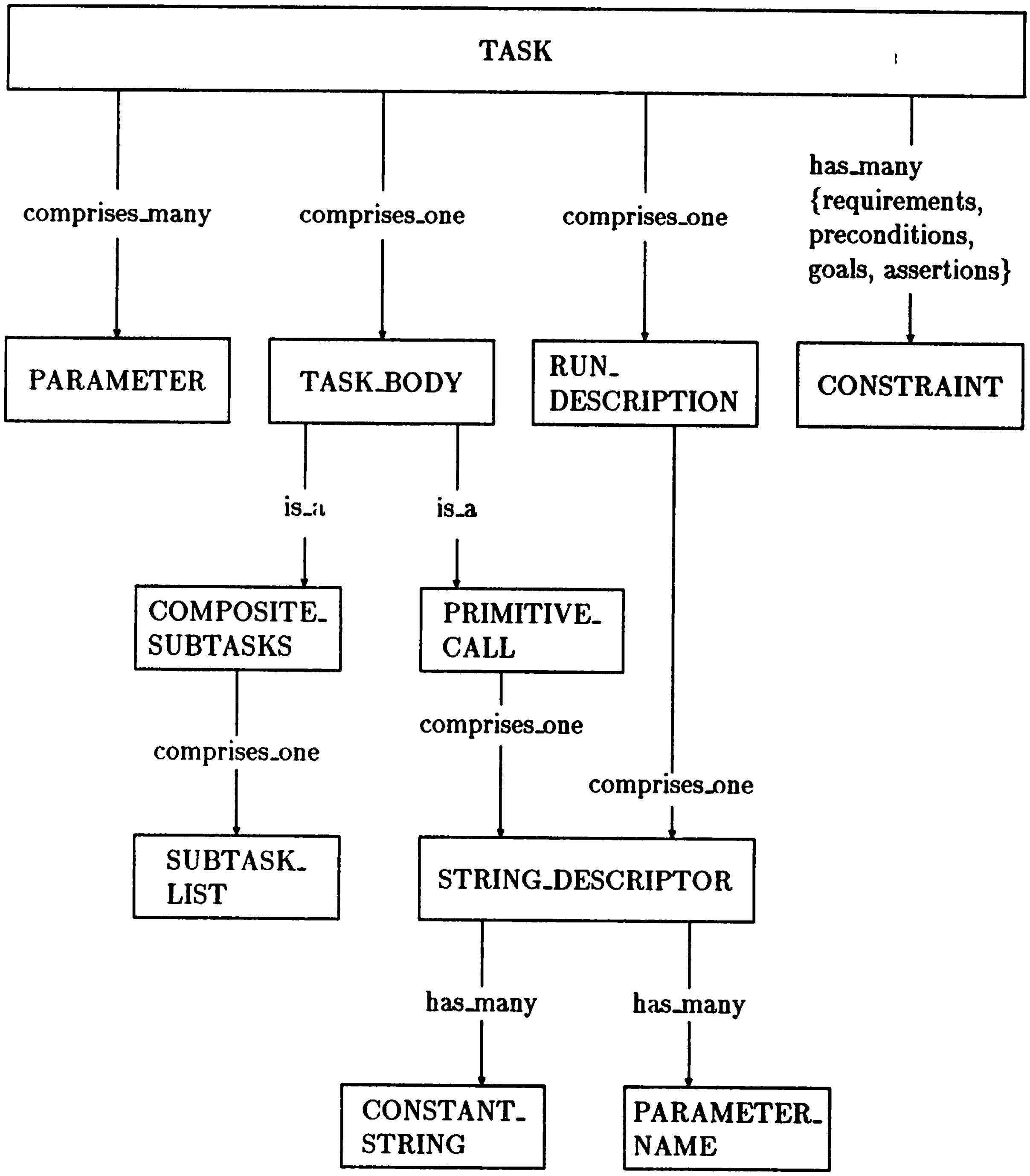

Figure 1: The Task Model.

In this model, a primitive TASK represents a VAX/VMS command. When a VAX/VMS command is executed, a status code is assigned to the DCL[4] symbol *$STATUS*. All primitive tasks have a special PARAMETER named $STATUS; this PARAMETER is assigned the value of its DCL namesake after the task is performed.

3.1.2 The Task Body

The **TASK BODY** specifies what actions a task performs. For primitive TASKs, the **PRIMITIVE CALL** structure specifies the text of the corresponding VAX/VMS command. This text is the concatenation of **CONSTANT STRINGS** and PARAMETER values. The PARAMETERs are referenced by name.

In composite tasks, the TASK BODY specifies a **SUBTASK LIST** (see Figure 2). This structure consists of an operator and a sequence of subtasks: the operator specifies the order in which the subtasks should be performed. The subtasks can be TASKs (referenced by name) or other SUBTASK LISTs. The operators are:

- **CONCAT** — all the subtasks should be performed, in the specified order;
- **SHUFFLE** — all the subtasks should be performed, but in any order;
- **CHOOSE** — only one of the subtasks need be performed.

This grammar is much simpler than POISE's shuffle expressions. The simplification is possible because of the inclusion of precondition/goal information in the task definition.

In both templates and instances of TASKs, subtasks are represented by **UNINSTANTIATED TASK** structures. This structure is not replaced by the full TASK structure until the subtask is actually performed. This paradigm has several benefits. A user can define a composite task before its subtasks are defined. A

[4]Digital Command Language, the VAX/VMS command language.

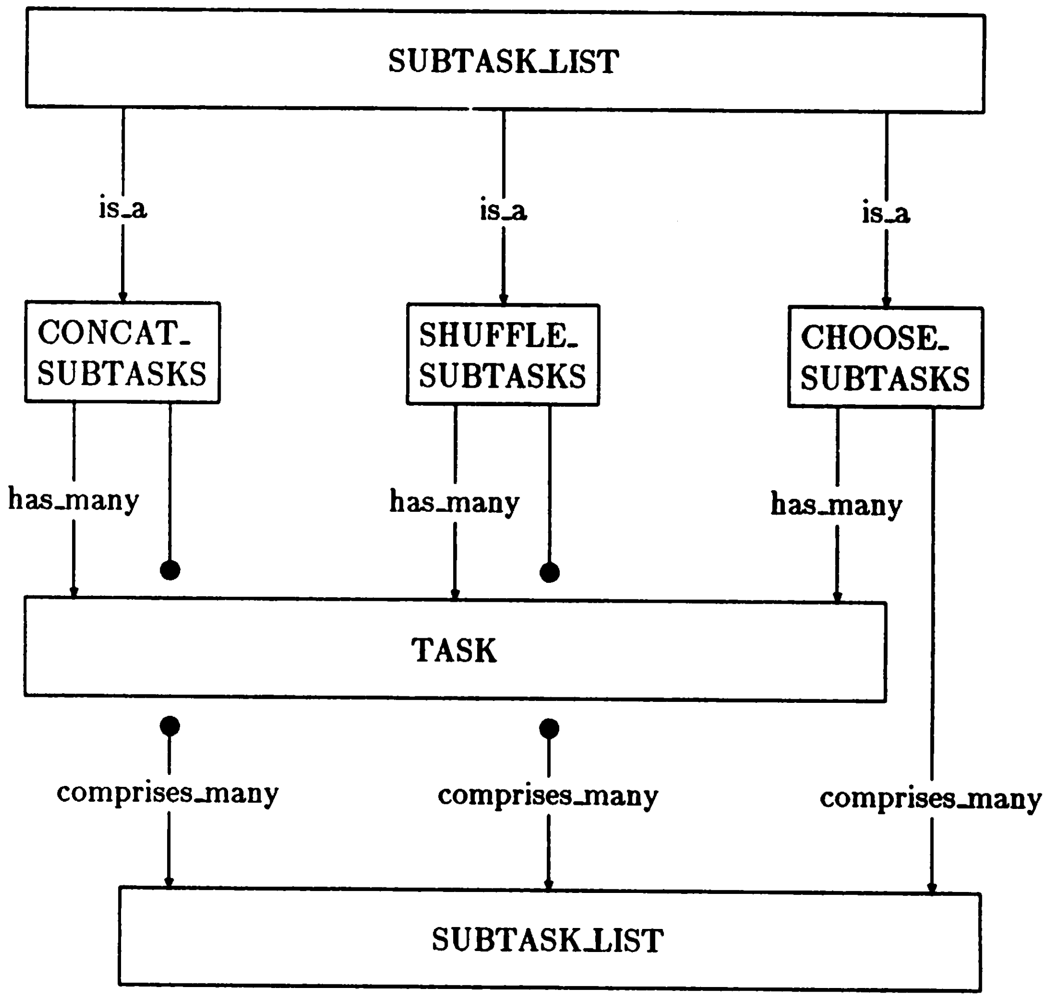

Figure 2: The Subtask List.

task's library definition can be changed without concern for other TASK hierarchies. Some subtasks of a composite task may never be performed; resources are saved by avoiding needless database operations.

3.1.3 Constraints

Constraints are boolean expressions which control task execution and restrict parameter values. There are two classes of constraints. **Explicit** constraints describe tangible conditions, i.e., which can be observed by the computer. **Implicit** constraints describe conditions which cannot be measured directly, but are arbitrary statements about the state of the world. An implicit constraint is asserted by posting it to a JOURNAL (see Section 3.2), and negated by deleting the corresponding JOURNAL entry.

A CONSTRAINT has a **TYPE** attribute and a set of **ARGUMENTs**, which can be CONSTANTs, PARAMETERs, or SUBPARAMETERs (PARAMETERs of subtasks). If the TYPE is known to be explicit, the CONSTRAINT can be evaluated by applying an appropriate function to the ARGUMENTs. Otherwise, the CONSTRAINT is considered implicit: it is true if it has been asserted more recently than it has been negated.

A task can use constraints in four different ways:

- **PRECONDITIONS** are CONSTRAINTs which must be true in order to start a task. If any precondition is false, the task cannot start. Note that a precondition may become false *after* the task begins.
- **REQUIREMENTS** are CONSTRAINTs which must be true while the task is in-progress. Only explicit constraints may be specified as requirements. If a contradiction arises between two REQUIREMENTS, the task management system should recover gracefully.
- **GOALS** are CONSTRAINTs which must be true when a task concludes successfully. If any goal is false, the task is not considered successful; again, the system should invoke a recovery procedure. Note that, for primitive tasks,

the $STATUS parameter must be odd, meaning a successful VAX/VMS return code, in addition to the declared goals.

- **ASSERTIONS** are implicit CONSTRAINTs which are declared to be true when a task concludes successfully, i.e., after the goals are found to be true. A *negate* assertion revokes a specified assertion.

3.1.4 The Run Description

A **RUN DESCRIPTION** describes an instance of a task. Unlike the text description, the run description can reference parameters. For example, a *COMPILE_PASCAL* task might have, "Compiles a PASCAL program," as its text description; the corresponding run description would be, "compiling program FLOAT_AVG." The run description thus informs the user what a task is actually doing.

3.2 A Task Management Database

The task model developed in Section 3.1 is the basic structure of a non-volatile database. The general database structure is illustrated in Figure 3.

The database is divided into three *realms*. The **TASK LIBRARY** realm contains TASK templates. The **PUBLIC CATALOG** contains templates for TASKs available to all users. In addition, each user has his own **CATALOG** for personal TASKs. Users can easily share TASK templates if the owner grants permission.

The **PENDING TASKS** realm contains instantiations of in-progress TASKs. Each user has an **AGENDA**, which contains his pending TASKs. Users can share their work by transferring TASKs from one AGENDA to another. When a task finishes or is aborted, the corresponding structure in PENDING TASKS is deleted.

The **USER JOURNALS** realm contains a **JOURNAL** structure for each user. Recall that implicit constraints (see section 3.1.3) cannot be evaluated directly

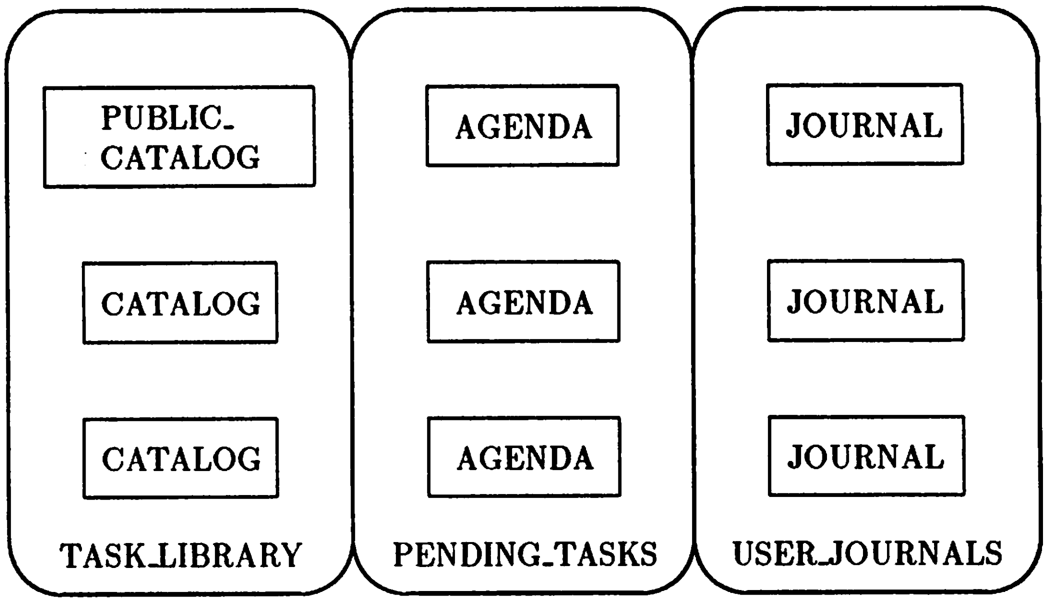

Figure 3: The Task Management Database.

When a task asserts an implicit constraint, a copy of the CONSTRAINT[5] is posted to the user's JOURNAL. If a CONSTRAINT is revoked (by asserting a *negate* CONSTRAINT), it is removed from the JOURNAL. A task evaluates the truth of an implicit constraint by checking for its presence in the JOURNAL.

With this paradigm, a user can easily define new tasks. He edits a *task definition language* program, then loads the definition into his personal catalog. If he has appropriate privileges, he can load the task definition into the PUBLIC CATALOG, or copy definitions from other users' CATALOGs. A personal task definition would, of course, override a PUBLIC CATALOG definition.

3.3 Intelligent Task Scheduling

The use of preconditions, goals, and implicit constraints (assertions) allows this task management system to schedule tasks more intelligently than POISE's planner. The preconditions prevent a task from starting before conditions are appropriate. The goals provide precise criteria for successful task performance: if the goals (and assertions) are *already* true, the scheduler can skip the task altogether.

A task's failure to complete successfully may be due to goals that are not achieved, requirements that are violated, or implicit preconditions that are not *really* true. When a task fails, the scheduler searches for *remedial* tasks whose goals or assertions include one or more of the problematic constraints. A TASK template index, using goal/assertion constraints as pointers, can save much time in this search process. The index, however, is not a critical feature.

The most likely candidates for failure recovery are tasks which are part of the active top-level task; however, any task in the task library is eligible. Primitive tasks are generally preferred to composite tasks. The remedial task may be invoked automatically, or after the system queries the user. If more than one task is identified, the user is asked to select one.

Preconditions and goals are particularly useful with the SHUFFLE and

[5]All parameter and subparameter references are posted as CONSTANTs. If a CONSTRAINT to be posted contains a reference to an unassigned (valueless) parameter, the CONSTRAINT is not posted and the user is notified.

CHOOSE operators (see Section 3.1.2). The constraints provide selection criteria for choosing the (next) subtask to perform.

The task definition in this design is thus a *partial solution.* [DLC85] It does not specify precisely which steps will achieve its goals. It *does* contain sufficient information to construct a solution, and suggests one solution (the task body). This solution, however, is non-binding: the scheduler may select a totally different series of step (or none at all), and still satisfy the specified goals. Davis and Chien suggest that such partial plans "...are responsible for much of the efficiency of human problem solving." [DC77]

The template can also be considered to describe a *scenario*:

> ...a set of "snapshots" of the future that outline possible courses of action, actions of foreign processes, and their consequences. This scenario can then be used to decide courses of action, outline contingency plans or force reexamination of goals set in the problem statement. [WR82]

3.4 A Structured VAX/VMS Interface

A primitive task can specify any VAX/VMS command. Some commands require user input; some produce output. All return a DCL status code when they finish. The task management system may need to assert different levels of control over user $\Longleftrightarrow$ VMS interaction for various commands.

Figure 4 is a model for a structured user/machine interface. The *Agent*, a separate process which executes commands, communicates with the task management system via a set of virtual mailboxes. The user communicates directly with the task management system, but can only speak to the Agent via the mailbox interface. The task management system controls the mailbox interface, restricting user/Agent communications to an appropriate level for each command.

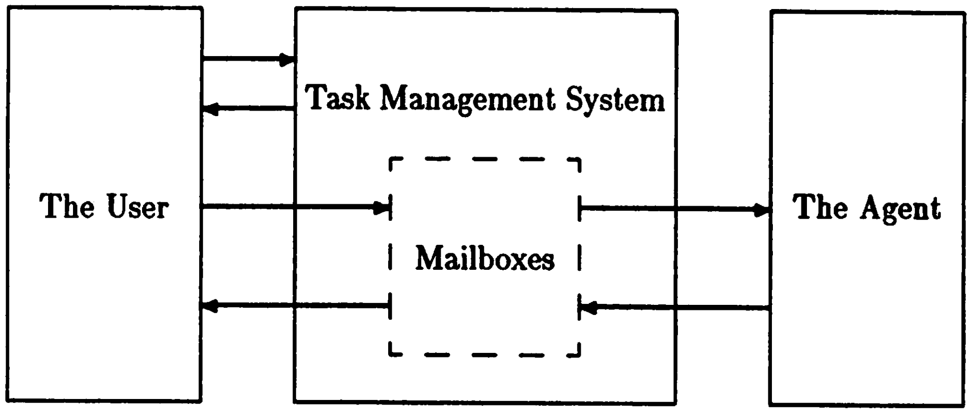

Figure 4: The User/Machine Communications Model.

3.5 The User-interface

The task management system responds to a small set of commands, listed in Appendix A. The user starts a task by naming it. He can interrupt the task by pressing CONTROL-C, which suspends the task and immediately returns the user to the command-processor. The suspended task is assigned a number, which can be used to continue or abort the task. Several tasks may be pending simultaneously, though only will can be active at a time.

The top line of the user's screen is reserved for the run description of the active top-level task. The second line displays the run description of the current subtask. These two lines are continually updated while a task is executing, provided constant information about task execution.

Pressing CONTROL-L clears and redraws the screen. This is useful for erasing VAX/VMS broadcast messages and other unexpected displays. The CONTROL-L key may be used while performing a task without interrupting it.

4 TASKMAN

4.1 A POISE Subset

The system design developed in Section 3 was used to create the TASKMAN Task Management System. Our aim is show how POISE can be extended in particular directions. We have thus concentrated on developing POISE's planning capabilities and omitted the plan recognition facility. Also absent are some of POISE's more aesthetic features: the menu facility; the graphic task-hierarchy display; the natural-language parsing and generation facilities. In their place, we have constructed a simple, yet powerful command system for manipulating multiple pending tasks.

4.2 The Prototype

This section describes a prototype TASKMAN system. This version includes most of the features described in Section 3. The following differences should be noted:

- The SHUFFLE and CHOOSE subtask-list operators (Section 3.1.2) are not implemented.
- TASK CATALOGs (Section 3.2) are not implemented. All TASK templates are stored individually in the TASK LIBRARY and are available to all users. Users cannot define their own tasks.
- There are no facilities for multiuser cooperation on a TASK.
- The user is directly connected to the VAX/VMS interface mailboxes (Section 3.4) while performing primitive (sub)tasks.

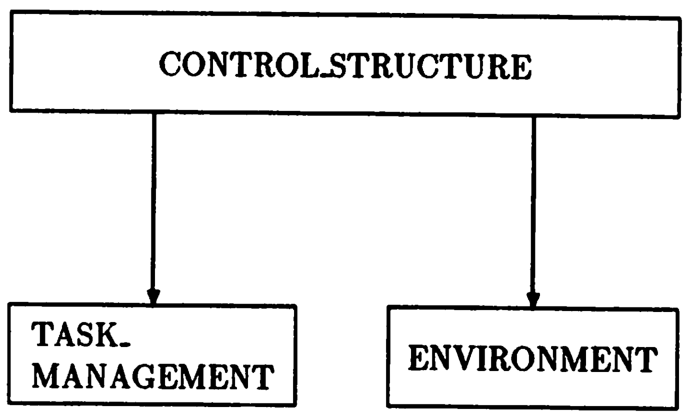

Figure 5: TASKMAN Architecture Summary.

4.3 The Architecture

4.3.1 Overview

One of the primary objectives of this project is to create planning technology which can be integrated with POISE (see Section 5.1). Accordingly, a major objective of the architecture is to isolate this technology — constraint enforcement, agenda management, task scheduling — from other parts of the system.

TASKMAN is implemented in VAX LISP, a dialect of CommonLisp. The architecture can be viewed in terms of three major clusters of VAX LISP packages (Figure 5). The **TASK_MANAGEMENT** cluster implements various data structures and includes the interface to the database management system. The **ENVIRONMENT** cluster controls the user-interface and the VAX/VMS Agent (see Section 3.4). It also provides an online documentation system, a terminal scripting facility, and various low-level I/O routines. The **CONTROL_STRUCTURE** cluster contains the system initialization routines, the command processor, and the task scheduling/execution system. Each cluster will now be described in detail.

Table 1: Database Realm-management Packages.

PACKAGE	*REALM*	*TOP-LEVEL STRUCTURE*
TASK_LIBRARY	TASK_LIBRARY	CATALOG
AGENDA	PENDING_TASKS	AGENDA
JOURNAL	USER_JOURNALS	JOURNAL

4.3.2 The TASK_MANAGEMENT Cluster

The TASK_MANAGEMENT cluster, detailed in Figure 6, implements all data structures related to TASK management. Each of the three packages referenced by the CONTROL_STRUCTURE cluster — *TASK_LIBRARY*, *AGENDA*, and *JOURNAL* — provides access to one of the three database realms and implements the top-level structure in that realm (see Table 1). The CONTROL_STRUCTURE also accesses the *TASK* and *DBMS* packages to facilitate task execution.

The TASK structure exists in both the TASK_LIBRARY (as a template) and the PENDING_TASKS realms (as an instance); thus, the *TASK* package is referenced by both TASK_LIBRARY and AGENDA. Similarly, a *CONSTRAINT* can belong to either a TASK (as part of the task's definition) or the JOURNAL (as an implicit assertion). Note that TASK references JOURNAL directly in order to efficiently determine whether implicit constraints are posted. *PARAMETER*s and *CONSTANT*s are elements of CONSTRAINTs as well as other task structures. All TASK elements not explicitly mentioned here — PRIMITIVE CALL, SUBTASK LIST, RUN DESCRIPTION — as well as UNINSTANTIATED TASKs, are managed by the TASK package.

The *DBMS* package is the interface to the database management system (described in Section 4.4). The database manager is not directly accessible from VAX LISP; a PASCAL procedure serves as an intermediary. Database commands are text strings; a VMS integer status code is returned.

The *work-area*, used for passing data to and from the database manager, is im-

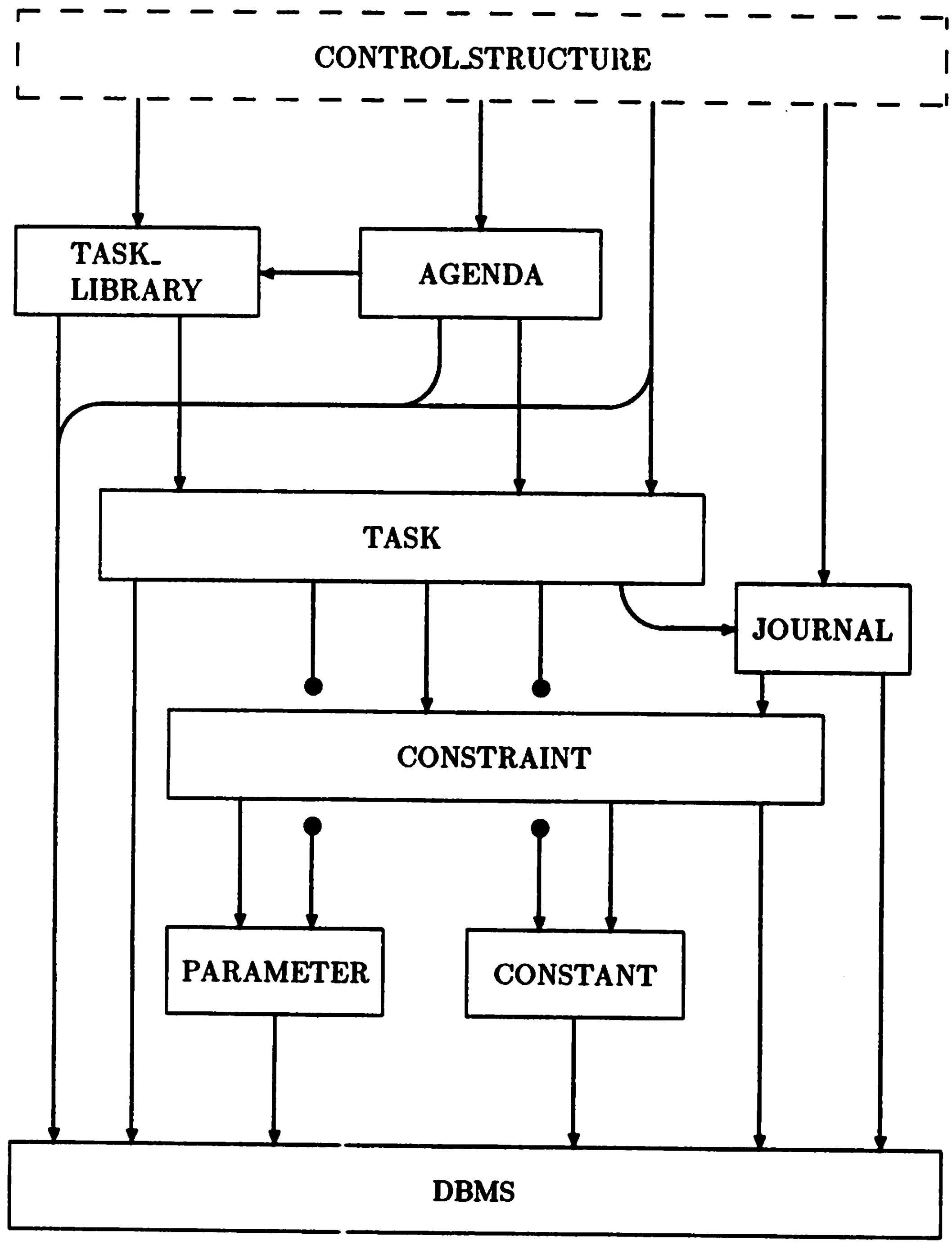

Figure 6: TASK_MANAGEMENT Cluster Architecture.

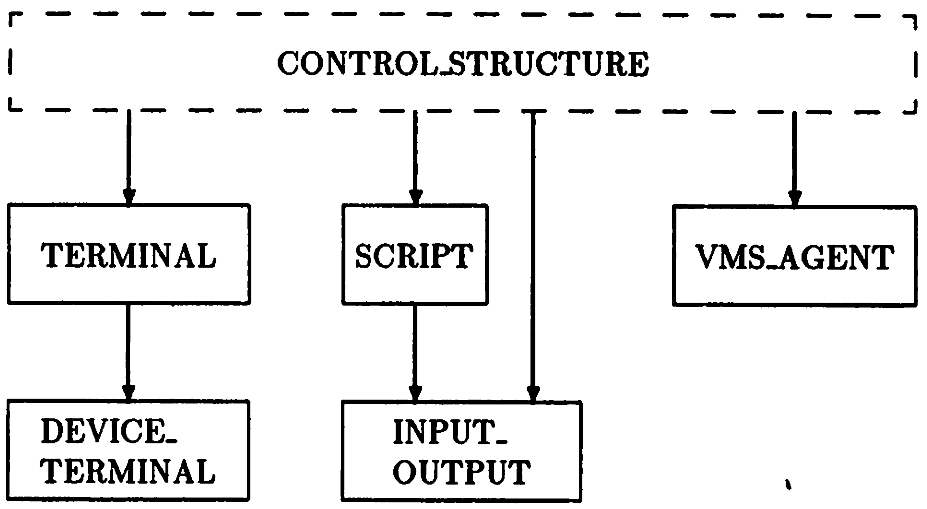

Figure 7: ENVIRONMENT Cluster Architecture.

plemented as an *alien-structure*. This is an aggregate data structure defined as a block of VAX/VMS memory. The DBMS package provides machine-independent work-area access functions for other packages in the TASK_MANAGEMENT cluster.

4.3.3 The ENVIRONMENT Cluster

The ENVIRONMENT cluster (Figure 7) provides a structured interface to VAX/VMS and the user. This cluster has three major components:

The *VMS_AGENT* package controls the VMS Agent and its mailboxes (see Section 3.4). This includes functions for transmitting VAX/VMS commands generated from primitive tasks and interpreting the resulting output.

The *SCRIPT* package provides a facility for recording terminal interaction during the TASKMAN session. *INPUT_OUTPUT* implements file-system-related functions for SCRIPT (as well as the CONTROL_STRUCTURE cluster).

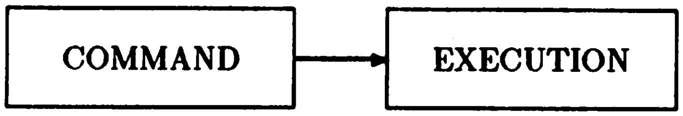

Figure 8: CONTROL_STRUCTURE Cluster Architecture.

The *TERMINAL* package manages the terminal screen, including the RUN DESCRIPTION display. It also provides a screen-oriented file browser. *DEVICE_TERMINAL* translates generic screen functions (e.g., "clear-to-end-of-line") into device-dependent command sequences.

4.3.4 The CONTROL_STRUCTURE Cluster

The CONTROL_STRUCTURE cluster is TASKMAN's operating system. It is divided between two packages.

The *COMMAND* package contains TASKMAN's top-level control code. This includes the initialization and shutdown sequences, the command processor, and the commands listed in Table 2. This package also maintains the variables which are manipulated with the SET command.

The *EXECUTION* package contains the task scheduling facility and the code which performs tasks.

4.4 The Database Management System

The database manager is VAX-11 DBMS, a CODASYL-compliant system. The database schema is maintained in VMS's "Common Data Dictionary"; data structures are distributed among four VMS files.

The database schema is completely independent of the task domain; under normal conditions, it never needs modification. The schema fully defines the realms and the data structures described in Section 3. Note that there is no intrinsic difference between a TASK template and a TASK instance; they are distinguished by realm.

DBMS provides checkpointing of database transactions and allows controlled sharing with protection of data integrity. Optional subschema definitions permit database enforcement of TASKMAN user classes.

There are several interfaces to the DBMS system. TASKMAN calls a subroutine which dispatches on text commands and returns a VAX/VMS integer status code. The most useful status codes are exported as global constants from the DBMS package. Any package which accesses the DBMS package can thus perform arbitrary database operations. It is each VAX LISP function's responsibility to only manipulate appropriate database objects.

4.5 The Flow of Control

4.5.1 System Initialization

TASKMAN is started by calling the (*taskman*) function in the COMMAND package. The screen is immediately cleared, and displays "start-up" RUN DESCRIPTIONs. The TASKMAN control-keys are bound to their functions, the VMS Agent is started, and the user's AGENDA and JOURNAL are located in the database. Any pending tasks are then listed on the screen.

Control now passes to the command processor. Commands typed by the user invoke corresponding functions in the COMMAND package. These may, of course, call appropriate routines in other packages.

4.5.2 Task Execution

When the user invokes a new task, a copy of the template is instantiated in the PENDING TASKS realm and connected to the user's AGENDA. Control transfers to the EXECUTION package, which initializes the task's parameters. The preconditions of this new task (and its parent task, if any) are examined as parameters are modified: discrepancy activates the failure-recovery code (described in Section 4.5.3).

Next, the goals and assertions are checked. If these are all true, the task is *averted*:[6] it is marked "successfully performed," and the scheduler selects the next step to perform.

Now the preconditions are checked: if any precondition is false, the task cannot begin. A top-level task is suspended; for any other task, the failure-recovery facility is activated.

The task body is now performed:

- *primitive tasks*: the corresponding VAX/VMS command is sent to the VMS Agent. The user's terminal handles any required input or resulting output. When the command finishes, its status code (returned by the Agent) is stored in the $STATUS PARAMETER.
- *composite tasks*: the scheduler selects a subtask according to the subtask list operator. For the CONCAT operator, each subtask is performed in sequence. SHUFFLE subtasks are performed in arbitrary order until all succeed; if no unfinished subtask can be performed, the failure-recovery facility is activated. For the CHOOSE operator, the scheduler only performs one subtask; failure recovery is used only when *no* subtask can be performed successfully.

  Precondition/goal information is used to schedule subtasks intelligently, as discussed above and in Section 3.3. In particular, this information is used to choose among subtask alternatives. SHUFFLE subtasks whose preconditions are not satisfied are delayed until other subtasks are performed. CHOOSE subtasks with unsatisfied preconditions are not even considered.

The goals are again checked when the task body finishes. If any goal is false, failure recovery is started; otherwise, all assertions are posted to the user's journal and the task is marked "successful." The scheduler now selects another subtask.

[6]If *no goals* are specified in the task definition, the task will not be averted, since it is not known what the task accomplishes.

When a top-level task finishes successfully, its entire PENDING TASKS structure is deleted, and the user returns to the command processor.

The user may interrupt an active task (or TASKMAN command) by pressing CONTROL-C. Control immediately returns to the command processor. The active task is now *pending*. It can be continued or stopped with an appropriate command. If not explicitly stopped, it will remain pending indefinitely.

4.5.3 Failure Recovery

> Execution may be thought of as a process of pulling the nonlinear plan through a hole that can only accept one step at a time. As each step is pulled through, the remaining plan is deformed into a new structure. [Sac77]

We mentioned earlier (Section 3.3) that the task definition represents a partial solution: the actual subtask sequence invoked by the scheduler may differ significantly from the original task body. We have just described (Section 4.5.2) how the scheduler uses constraint information to refine the subtask sequence specified in the task template. Now we will examine how TASKMAN responds when this subtask sequence fails entirely.

Section 4.5.2 mentions several situations in which failure to satisfy constraints precludes following the usual subtask sequence. The failure-recovery mechanism resolves the difficulty by selecting one or more tasks whose goals or assertions satisfy the problematic constraints. These tasks are inserted into the task instance hierarchy[7] under the CONCAT operator and executed. This should permit the scheduler to reexecute the task which failed.

TASKMAN uses several heuristics to identify and assign priorities to remedial tasks. The underlying criteria are relevance to the current task and execution cost. The heuristics:

- The current task hierarchy is searched first.

[7]The task template is not affected; but see Section 5.6.

- Within the current hierarchy, the failure-recovery mechanism will first look for a task in the current subtask list, then in the parent subtask list, and so on until the top-level task is reached. Other subtask lists are then searched in depth-first order.
- Within a CONCAT expression, earlier-scheduled tasks are preferable to later ones. Within other subtask lists, there is no inherent preference.
- If a task specifies one of the problematic constraints as a precondition, the task is immediately removed from consideration.
- If no remedial tasks are identified within the current task hierarchy, the entire task library is searched.
- Once a set of tasks has been identified, primitive tasks are ordered before composite tasks.

If only one candidate is identified, the user is asked to confirm its execution. Otherwise, the three most preferable tasks are presented to the user in a menu. He may select a menu option, "scroll" the menu to see the other candidates, or suspend execution of the top-level task.

4.5.4 The Unified Control Structure

It is now clear that TASKMAN's control structure does not firmly separate the plan construction and plan execution steps. The scheduler expands each "node" of the task hierarchy (from an UNINSTANTIATED TASK to a full TASK structure) when that node is needed. The failure-recovery mechanism, a logical part of scheduling, is invoked as needed during execution to alter the basic plan structure. This concurs with Sacerdoti:

> The basic strategy of hierarchical planning is to create a cheap, if incorrect, plan by throwing much information away. Then the cheap plan is expanded into a more detailed plan. The process of expansion continues, building ever more detailed plans until a sufficiently accurate one has been built. [Sac77]

The criteria used by the scheduler and failure-recovery mechanism effectively constitute *planning critics* [Sac77]. These critics refine the partial solution (Section 3.3) during task execution, developing a structure appropriate for this instance of the task.

4.6 The Test Domain: Software Development

Bonar and Soloway [BS83] discuss programming as essentially a planning process. The entire software development process is a particularly interesting planning domain because of its non-algorithmic nature. A programmer edits his source code "some number" of times before the code compiles successfully. He then performs several test-and-edit cycles. Some tests will require prior preparation of external procedures or data files. The programmer may, in fact, be developing several programs concurrently, jumping from one task to another.

Obviously, software development cannot proceed in totally arbitrary fashion. There are inherent constraints, such as the linker requiring object code. There can also be artificial constraints: a manager requires that a user requirements document be submitted before design begins; programs which manage financial transactions must compile without any errors, even informational messages.

The critical feature in a software development environment is *flexibility*. Creating software is not a linear task. A good development environment guides the programmer through his work. It provides for deviation from the normal "tool path" when appropriate, even mandating such deviation when necessary. Importantly, the environment does not penalize the programmer for these deviations, by making him remember "where he was" and retype commands.

To do all this, the environment needs to know the nature of the programming task. This includes the nature of the individual tools and how they interact. It also includes the ability to record the status of ongoing development tasks.

The software development domain is well suited to TASKMAN, since software tasks can be represented as a series of computer commands. These commands can include mail transmission, database operations, and text processing, in addition to software manipulation. This domain was therefore chosen for testing TASKMAN.

We have written task descriptions in which TASKMAN models a simple software development environment. This environment supports the basic steps of the entire software development process. Emphasis was placed on using constraints to pass information among tasks and control scheduling. The tasks are described in Appendix B.

4.7 Observations

The POISE design provides for distinguishing among multiple instances of a task within a subtask hierarchy, e.g., several *EDIT*'s as subtasks of *MODIFY_LIBRARY*. There is no provision for doing this in TASKMAN. This does not reduce the set of definable tasks, since arbitrary portions of a subtask hierarchy can be redefined as independent tasks.

It *is* possible that the failure-recovery mechanism will select a remedial task which is already within the top-level task's hierarchy, leading to ambiguous interpretations of "subtask *X*." Excluding extant subtasks from the set of remedial tasks could lead to irrecoverable situations. This issue deserves further study.

5 Further Directions

5.1 Integration with POISE

TASKMAN has concentrated solely on improving the planning aspect of POISE. A substantial portion of POISE is concerned with task recognition, a multifaceted topic. POISE typically maintains several possible interpretations for a series of task instances. Each interpretation is modeled as task hierarchy; the root task represents the user's highest-level goal. As each task is performed, the POISE interpreter tries to connect it to an interpretation where the task is expected. If the task cannot be integrated into any existing interpretation, a new interpretation is constructed. During this recognition process, task parameters are propagated up and down the interpretation hierarchies. Parameter constraints serve as additional criteria for selecting the "correct" interpretation.

This process will generally lead to an abundance of possible plan interpretations. POISE's *focuser* selects the most likely interpretations, based on several planning heuristics. As recognition proceeds, interpretations will shift into and out of the *focus set.* [ML83]

The recognition facility would unquestionably benefit from adding precondition and goal information, as well as journals containing implicit constraints. This information could substantially reduce the size of the focus set as well as the set of possible plan interpretations,

Combining POISE and TASKMAN is [pardon the expression] a substantial task. It includes, among other things, interleaving two plan schedulers, POISE's interpreter and focuser, TASKMAN's failure-recovery facility, and two parameter/constraint management systems. It is more advisable to integrate desired portions of TASKMAN into POISE in stages.

5.2 The Task Database

The task management database is a significant aspect of TASKMAN. It is perhaps the easiest feature to add to POISE. Each POISE form which manipulates a TASK structure (or substructure) is translated to a call to an appropriate routine in the TASK_MANAGEMENT cluster. The database schema would be modified to include POISE's task-description syntax. Any code to be added would be concerned solely with data management, not task execution; the overall database integration is thus substantial, but not technically complex.

5.3 A Task Definition Facility

Section 3.2 discussed how TASKMAN users can define their own tasks. This is, in fact, how TASK templates are constructed: after TASKMAN is interrupted by a VAX LISP breakpoint, task definitions, written as LISP function calls, are loaded from a file. A formal task definition facility should feature a more structured task definition language and a *load* command. Security features should prevent users from casually modifying the PUBLIC CATALOG.

5.4 Cooperative Work

The use of a common task-management database allows users to cooperate on tasks. In the simplest version of cooperative work, each task *belongs* to one user, who can dispose of it as he pleases: he can work on the task, abort it, or transfer it to someone else. Task ownership is represented by the AGENDA to which the TASK is connected: ownership is transferred by simply reconnecting the TASK to another AGENDA. The TASK-passing mechanism must also deal with journal entries related to the task, including file names which refer to different files in different users' environments. This corresponds to Chang's concept of a weak participant system: "...users interact with a system in the performance of shared tasks without essential communication between them." [Cha85]

TASKMAN is also capable of more complex forms of cooperation. The control structure could use a negotiations management tool [MC85] to directly control user participation in task execution. TASKMAN commands, or perhaps the tasks themselves, would mediate the necessary interuser communications. Distributed planning and execution become feasible objectives.

5.5 Goal-oriented Planning

TASKMAN provides *procedure-oriented planning*: the user selects a task to perform. [LC85] also discusses *goal-oriented planning*: the user specifies the goal(s) he wants to achieve, and the system invokes one or more tasks to achieve those goals.

Goal-oriented planning can be added to TASKMAN very easily. In one scenario, the user issues an *achieve* command. TASKMAN queries for a **goal set**, a named list of implicit constraints. The scheduler instantiates a new TASK with the specified name and GOALS. The failure-recovery facility then selects appropriate subtasks to achieve these goals.

5.6 Learning Better Subtask Sequences

> Nilsson [Nil73] characterizes two broad classes of events that an execution monitor must handle: failures and surprises. Failures occur when the execution of an action fails to update the real world in the way that the model of the action updates the model of the world. Surprises occur when some fact, unrelated to the current action, becomes known. Surprises may indicate that some future steps will be unnecessary, or that they will no longer be appropriate, or that they will fail. [Sac77]

The failure-recovery mechanism modifies individual TASK instances to deal with unexpected situations. This is sufficient for a surprise event. However, for a failure, as defined by Nilsson, we may want TASKMAN to permanently revise the task template. This can be a straightforward modification of the task body (e.g., removing a subtask which is always skipped) or a refinement of the constraints (insertion/deletion). It can also be a more exotic change: renaming a user-written task which is shadowing its namesake in the PUBLIC CATALOG.

A *learning facility* should be modeled as an execution critic, monitoring the scheduler. When any exception occurs, whether or not the failure-recovery mechanism is invoked, the critic decides whether a template change is warranted. In practice, one exceptional event does not warrant a permanent change. An effective critic would maintain a history of significant exceptions, using event frequency, exception severity, and recovery expense to determine what action to take.

Two users may display different execution patterns for the same task. A learning facility can customize a shared task to each user, depositing new TASK templates in each user's personal CATALOG.

5.7 In-depth Plan Criticism

The control structure criticizes the task hierarchy using the characteristics of the top-level task. This implicitly assumes that these characteristics — subtask sequences, preconditions, goals — are consonant with the characteristics of the subtasks themselves. If this is not the case, the control structure will modify the execution sequence accordingly.

Sacerdoti [Sac77] discusses constructive critics which examine an *entire* plan hierarchy. These critics look for inconsistencies and contradictions among the subtasks. A task is scheduled on the basis of its neighbors' characteristics, as well as its own. This in-depth criticism might be helpful for TASKMAN. There is a trade-off of time spent on plan criticism *vs.* time consumed executing unnecessary, possibly destructive tasks.

6 Acknowledgements

Many people had a hand in this project. The initial design of TASKMAN was done with Bruce Croft. Victor Lesser helped with the design and implementation of the task management structures, and provided information on constraints. Jack Wileden and Alex Wolf furnished material on software development environments. Dan Corkill and Tom Gruber were indispenable in helping the author master the intricacies of VAX LISP. Larry Lefkowitz answered numerous questions on POISE task structures; Roger Thompson provided aid with database management. Both Lefkowitz and Thompson helped with the preparation of this report. The author again thanks Bruce Croft and Jack Wileden, for lending their time and support to the TASKMAN project.

A The TASKMAN Command Set

The TASKMAN system is intended for intelligent task execution; users are not expected to use TASKMAN's "operating system" for much work. Accordingly, TASKMAN provides a small set of commands, mostly oriented towards task management.

The command processor recognizes TASK names as commands to start a new instance of the corresponding task. If parameter values are required, TASKMAN will query the user. The user can suspend the currently-active task by pressing CONTROL-C. Any number of tasks may be pending (suspended) at a time; each pending task is assigned a number. A pending task can be restarted from the point of interruption[8] by specifying its number as a command. A task automatically deletes itself after successful completion.

The TASKMAN command set is described in Table 2. Note that using a question mark in place of an argument generally displays a list of possible arguments.

[8]There is no differentiation among a task which never began, a task which was interrupted during processing, and a task which completed unsuccessfully; all are marked as "not performed." However, *subtasks* of a composite task *do* retain their "performed" status when a parent task is interrupted.

When a composite task is restarted, the scheduler may start a different subtask than the one which was executing at the time the task was interrupted.

Table 2: TASKMAN Commands.

Command /Argument	*Description*
COMMENT MESSAGE	Appends MESSAGE to the user's *comment* file.
HELP (also "?")	Displays general information about TASKMAN.
PRINT OBJECT	Prints OBJECT on the default printer queue.
SET VAR	Assigns a value to VAR, a TASKMAN variable. Variables include: the names of the *comment* and *script* files; the default printer queue; the *control* directory (default for *comment* and *script* files); and the *working* directory (default for the user's work files).
SHOW ITEM	Displays the specified item. Possibilities include: a list of TASKMAN commands; the values of variables which can be SET; the contents of the *comment* and *script* files; a list of TASKs in the TASK LIBRARY; a description of any particular TASK; and a list of pending TASKs.
START SCRIPT STOP SCRIPT	The START and STOP commands begin and terminate scripting of (portions of) the TASKMAN session.

B Software Development Tasks

TASK NAME	*TYPE*	*DESCRIPTION*
COMPILE_PROGRAM	primitive	Compiles a source-language program into object code.
LINK_PROGRAM	primitive	Links object code into an executable image.
RUN_PROGRAM	primitive	Executes an image.
MAKE_PROGRAM	composite	Creates a program.
EDIT_PROGRAM	composite	Edits a program's source code.
EDIT_DOCUMENT	composite	Edits a document's text.
EDIT_FILE	primitive	Edits an arbitrary file.
PRINT_FILE	primitive	Prints a file.
PRINT_IN_OFFICE	composite	Prints a file on the high-quality office printer.
LIST_FILES	primitive	Displays the contents of the user's work directory.

C Sample Task Descriptions

Task name:	LINK_PROGRAM
Type:	Primitive
Text description:	Links object code into an executable image.
Parameters:	OBJECT_FILE (file name; ask user for value)
	THE_PROGRAM (file name)
Run description:	"Linking " THE_PROGRAM "."
Display subtask run description:	
	Yes.
Primitive call elements:	"LINK" OBJECT_FILE

Preconditions:
 (PROGRAM_COMPILED THE_PROGRAM)
Requirements:
 (HAS_FILE_NAME THE_PROGRAM OBJECT_FILE)
Assertions:
 (PROGRAM_LINKED THE_PROGRAM)

```
Task name:         MAKE_PROGRAM
Type:              Composite
Text description:  Creates a program.
Parameters:        SOURCE_FILE (file name; ask user for value)
                   THE_PROGRAM (file name)
Run description:   "Making program " THE_PROGRAM "."
Display subtask run description:
                   Yes.
Composite subtasks:
                   —CONCAT—
                   EDIT_PROGRAM
                   COMPILE_PROGRAM
                   LINK_PROGRAM
                   RUN_PROGRAM
                   EDIT_DOCUMENT
Requirements:
 (HAS_FILE_NAME THE_PROGRAM SOURCE_FILE)
 (EQUALS SOURCE_FILE EDIT_PROGRAM.SOURCE_FILE)
 (EQUALS SOURCE_FILE COMPILE_PROGRAM.SOURCE_FILE)
 (EQUALS THE_PROGRAM LINK_PROGRAM.OBJECT_FILE)
 (EQUALS THE_PROGRAM RUN_PROGRAM.EXECUTABLE_FILE)
 (HAS_FILE_NAME EDIT_DOCUMENT.TEXT_FILE THE_PROGRAM)
 (HAS_FILE_TYPE EDIT_DOCUMENT.TEXT_FILE "XXX.TXT")
Goals:
 (PROGRAM_LINKED THE_PROGRAM)
```

References

[Bac69] C. Bachman. Data structure diagrams. *ACM Data Base*, 1(2), 1969.

[BS83] Jeffrey Bonar and Elliot Soloway. *The Bridge from Non-programmer to Programmer.* Technical Report 83-18, Department of Computer and Information Science, University of Massachusetts at Amherst, 1983.

[BW83] Peter C. Bates and Jack C. Wileden. *High-level Debugging of Distributed Systems: The Behavioral Abstraction Approach.* Technical Report 83-29, Department of Computer and Information Science, University of Massachusetts at Amherst, 1983. Appeared in *Journal of Systems and Software 3*, 1983, pp. 255–264.

[Cha85] Ernest Chang. *Participant Systems.* Technical Report, Advanced Technologies, Alberta Research Council, Calgary, Alberta, December 1985.

[CL84] W. Bruce Croft and Lawrence S. Lefkowitz. Task support in an office system. *ACM Transactions on Office Information Systems*, 2(3):197–212, July 1984.

[CLLH82] W. Bruce Croft, Lawrence S. Lefkowitz, Victor R. Lesser, and Karen E. Huff. *POISE: An Intelligent Interface for Profession-based Systems.* Technical Report 82-19, Department of Computer and Information Science, University of Massachusetts at Amherst, 1982.

[DC77] P. R. Davis and R. T. Chien. Using and re-using partial plans. In *5th International Joint Conference on Artificial Intelligence*, Cambridge, MA, 1977. vol. 1.

[DLC85] Edmund H. Durfee, Victor R. Lesser, and Daniel D. Corkill. *Coherent Cooperation Among Communicating Problem Solvers.* Technical Report 85-15, Department of Computer and Information Science, University of Massachusetts at Amherst, 1985.

[Gis81] J. Gischer. Shuffle languages, petri nets, and context-sensitive grammars. *Communications of the ACM*, 24(9):597–605, September 1981.

[HL82] Karen E. Huff and Victor R. Lesser. *Knowledge-based Command Understanding.* Technical Report 82-6, Department of Computer and Information Science, University of Massachusetts at Amherst, 1982.

[LC85] Lawrence S. Lefkowitz and Norman F. Carver. POISE — The Architecture. 1985. Unpublished working paper.

[MC85] D. Marca and P. Cashman. Towards specifying procedural aspects of cooperative work. In *3rd International Workshop on Software Specification Proceedings*, IEEE Publishers, August 1985.

[ML83] Daniel McCue and Victor Lesser. *Focusing and Constraint Management in Intelligent Interface Design.* Technical Report 83-36, Department of Computer and Information Science, University of Massachusetts at Amherst, 1983.

[Nil73] N. J. Nilsson. *A Hierarchical Robot Planning and Execution System.* Technical Note 76, Artificial Intelligence Center, Stanford Research Institute, Menlo Park, CA, April 1973.

[Sac77] Earl D. Sacerdoti. *A Structure for Plans and Behavior.* Elsevier, New York, 1977.

[WR82] Rajendra S. Wall and Edwina L. Rissland. Scenarios as an aid to planning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 176–180, Pittsburgh, 1982.