KNOWLEDGE-BASED COMMAND UNDERSTANDING:

An Example for the Software Development Environment

Karen E. Huff and Victor R. Lesser

82-6

Computer and Information Sciences
University of Massachusetts, Amherst

June 30, 1982

CONTENTS

LIST OF FIGURES

1. INTRODUCTION

One or more times during a normal computer terminal session, a user will probably say in frustration after misspelling a command name, leaving off an essential parameter, or giving the wrong file name: "Why didn't the system know what I meant -- it is obvious what I intended?" or "This command was supposed to repeat the last action I performed, with only minor differences. Why can't the system work it out for me?".

Another scenario, especially common when what needs to be done is particularly intricate or when there is time pressure, is for two people to carry out work cooperatively at a single terminal. Together, they suggest actions, explore the ramifications of the ordering of actions, and attempt to plan ahead for the most effective way of getting their joint work done. Often, one person will act as kibitzer, whose specific purpose is to look for any flaws in the developing plan of attack. This style of working is based on the assumption that two minds working on the same problem will reduce the chances of making costly errors.

These phenomena are indicative of the gap which exists between the user and the user interface, which in the typical computer-based environment is the command language of the operating system. This language and the underlying facilities of the operating system are oriented towards the efficient management of resources: manipulation of file storage and execution of programs. The interface has no general knowledge of the contents of files and the way in which the user will typically operate on them, or knowledge of the constraints on the execution of specific programs. In particular, there is no knowledge base upon which to draw in order to detect (global) errors or to correct errors (local or global) which occur. If a history of past commands is maintained at all, it is maintained in uninterpreted form as a linear list of unanalyzed strings.

Because of this lack of knowledge, current interface designs are limited in the scope of the assistance they can provide to users. They offer only help in avoiding or minimizing the impact of errors in individual commands, where the error can be syntactically detected and possibly corrected through the use of highly local methods, such as spelling correction, command completion through defaults, or use of a dictionary of names of objects currently being manipulated. Those approaches to interface design which attempt to assist the user through highly structured menu-based command generation or high-level command invocation through natural language or non-procedural specifications are not appropriate to all user interactions. These alternative approaches either restrict the style/mode of users' interactions with the system (thus dictating how they must approach their problem-solving), or limit the users' abilities to efficiently manage resources based on the needs of the problem at hand.

We see the need for 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. At the core of such an intelligent interface (schematically depicted in Figure 1)

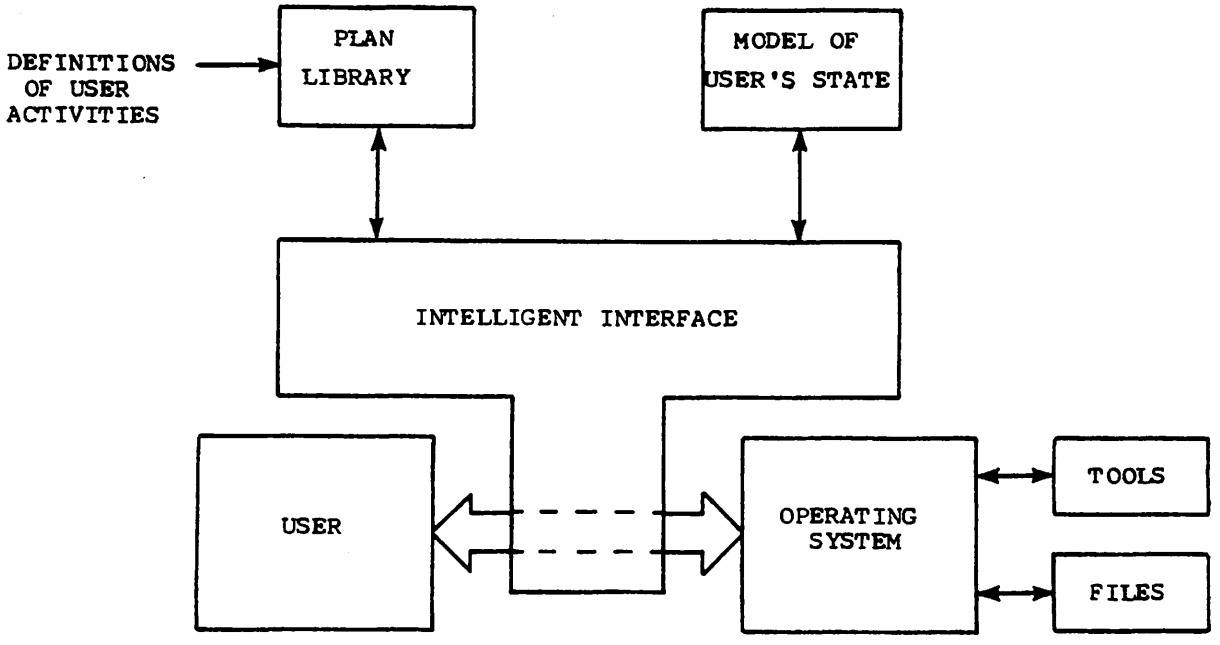

Figure 1. THE INTELLIGENT INTERFACE

is a program which monitors user commands in order to interpret sequences of low-level commands as partial instantiations of some high level plan (command) implicit in the user activities and inversely which performs sophisticated planning in order to generate from a high-level command a sequence of low-level commands that realize the desired activity. This interface will thus provide a spectrum of support from analysis of commands on the one hand to synthesis of commands on the other, using a single unified implementation framework. Through these capabilities, the interface can mediate between the system view of activities and the user's own view of those same activities.

If such an approach to an intelligent interface is to be realizable, then a basic premise must be satisfied: namely, that professional activities in the computer-based environment (e.g., programming environment, CAD/CAM design environment, office information system, etc.) are not unstructured and unpredictable, as has commonly been assumed. Rather, we believe that there is significant structure in, and a large body of knowledge applied to, the task of carrying out work in such environments. This structure has been obscured by the fact that, as humans carry out these tasks, they do so in a highly concurrent fashion: working first on this task, then on another, then back to the first. Often, the order in which certain tasks are accomplished is immaterial, but there do exist fixed synchronization points in these processes when several on-going subsidiary tasks must have been completed in order for further progress to be made. In addition, the initiation of certain activities usually gives valuable clues as to what additional activities must be undertaken in the future, although the exact ordering will be unknown. This interleaving of tasks is an inherent feature of the way that people work.

However, we also feel that there are limits to this structuring which must be confronted. Although these are fairly well-defined domains, we are still, after all, dealing with the actions of people. There is likely to be inherent ambiguity in the definitions of what the user's activities are and how they are structured. That is, it may not be possible to state a priori sufficient conditions to distinguish two actions which are in fact different. Such disambiguation may require heuristic arguments based upon current context. There will likely be sharing of activities, where a single action has effects which meet more than one goal of the user; here, the action plays more than one role, both in the user's view and in the syntactic/semantic structure through which commands are understood. These issues will pose significant challenges to our command understanding system, as we shall see later in this paper, and the need to deal with these issues is the primary motivation for employing artificial intelligence techniques

The remainder of this paper will discuss how the knowledge of user activity patterns can be formulated in a pragmatic fashion through the use of a set of hierarchically related plan elements and a database representing semantic relationships among objects the user is manipulating. Based upon this representation, the activities of the software development professional have been specified and example terminal sessions based upon these plans have been created showing the intelligent interface in action. The plans are preliminary in nature and are known to be incomplete, but serve to show the kind of information which is required to provide the intelligent interface functions we have envisioned. The example terminal sessions presented serve to show the kinds of support offered by the interface and to sketch the implementation approach which is envisioned.

An important aspect of these example sessions is showing how the knowledge gained from the forward and backward propagation of constraints within a partially instantiated set of hierarchically related plan elements, together with the retrieval of information from the semantic database, can be used to:

* make summaries of past accomplishments
* answer questions about current state
* manage agendas of future activities
* detect global errors
* correct local or global errors
* provide predictive information which permits the early interpretation of the pattern of interleaving of user activities
* interpret ambiguous statements of desired activity
* perform high-level command completion.

2. PLANS AS THE UNITS OF KNOWLEDGE

In order for the intelligent interface to carry out its interpretation and planning activities, it must in some way have explicit domain knowledge about the types of tasks the user will perform and the tools available to the user to perform those tasks: specifically, the input and output characteristics of the tools and the problem solving context in which the tools are applied. We feel this domain knowledge should be of a pragmatic, rather than abstract nature. That is, tool/command invocations (user actions) should be defined in terms of what they can be applied to and when, in relation to other invocations, it is appropriate/required to apply them. We wish to emphasize the active, applicative aspects of this knowledge, rather than the passive, descriptive aspects. By this pragmatic approach, we hope to achieve a system which is well-suited to interpretation and planning of sequences of actions by knowledgeable users, rather than one which is tailored more specifically to the needs of novice users.

Two related research efforts, the CONSUL project at USC/ISI [1, 2, 3] and the Graceful Interaction project at CMU [4] differ from the intelligent interface described here in that they both take a one-command-at-a-time approach to assisting the user. Thus, they are not currently taking advantage of the valuable information available from considering the global context in which the commands are issued; such information is of particular value in disambiguating incomplete commands and in identifying fundamental errors in the plan of attack being used to accomplish some activity. In the case of CONSUL, the project emphasis is on natural language communication and in the case of Graceful Interaction, the project emphasis is on integrated, multi-media communication including natural language, graphics, voice, mouses, and the like.

2.1 Plan Structure

We propose to organize the domain knowledge into units or clusters called plans. These plans are related hierarchically, so that detail can be contained at appropriately low levels and abstracted away at higher levels. Thus, actions can be viewed from these varying perspectives. A hierarchical view of the deep structure of commands has not previously been attempted. Operating systems have a one-level or flat syntactic/semantic model of commands: each command gives the name of an executable entity along with names of files to serve as input and output repositories; in such a model, all commands are syntactically and semantically equal. There is an analogy with the distinction between syntax-directed editors and more traditional editors. The traditional editor has simple view of text as a sequence of characters forming a sequence of lines. In contrast, the syntax-directed editor has a more sophisticated understanding of the structure of text based upon syntactic units such as

statement, keyword, block and the like; here, not all characters or words are considered equal in syntactic role.

In response to the requirement to support concurrency (i.e., the interleaving of on-going activities), a formal description of plan interrelationships is based upon regular expressions, augmented with two concurrency operators: the shuffle and the closure of shuffle which is called interleave. Just as context-free constructs, with their provision for matched bracketing and nesting, model the essential features of programming languages, so concurrent constructs are required to model the features of programming environment work. Shuffle allows arbitrary ordering among a fixed number of expressions while interleave provides arbitrary ordering among an arbitrarily large number of expressions. These two operators have been used in several formal systems addressing concurrency, which are discussed further below in section 2.4. The use of interleave implies that the grammar for plans is not regular, but rather context-sensitive.

While augmented regular expressions are used to describe the basic or syntactic form of plans, additional semantic constraints must also be accommodated. In an analogous fashion, programming languages are described by a context-free grammar and then supplemented with numerous semantic conditions to arrive at the complete, intended definition. The syntax allowed by the grammar gives an approximation of the desired language, while the semantics add the fine detail. In addition to supplying further detail on the well-formedness of plan occurrences, these conditions define semantic connections that must exist between related sub-parts of a plan. Within a given plan, the semantic constraints make use of the characteristics of the constituent, lower level plans.

In order to support abstraction, each plan specifies a set of attributes, or characteristics, whose definitions are given in terms of the attributes of its constituent plans. These attributes may then be used by any higher level plan in which this plan participates. The role of these attribute definitions is to abstract from all the details of a plan exactly those characteristics that are of interest outside the context of the plan. This follows the style of attributed grammars, with the plan attributes behaving as synthetic attributes.

Finally, each plan contains a specification of the effect that its occurrence has on a model of the state of the user's world. The use of this model, which is recorded in semantic database form, is a significant feature of our approach. Whereas the past occurrences of plans indicate the state of the user in terms of actions performed, the purpose of the semantic database is to describe the cumulative effects of those actions in terms of the objects that have been created and the characteristics that those objects have. In addition to concrete objects, the database also

describes the concepts, of which the concrete objects represent instantiations. This database of facts is used to provide additional semantic information which can be referenced by the plans or used as the basis for reasoning from first principles. As an encapsulation of state, either command history alone or database alone would be sufficient; used together, they are expected to provide significant predictive power in the interface, as will be seen later.

The combination of the plan and its effects on the model of the user's world provide our pragmatic definition of the semantics of the user tools and command language, by giving meaning in terms of an underlying model of the "state" of the user's actions and the "state" of the objects being manipulated. A simple command to invoke the editor to make changes to a file is given semantic meaning as the establishment of a historical revision relationship between the old and the new file contents. Depending on whether the file contained source or a testcase, to take two examples, the state of the user's activities is advanced so that certain other actions are now known to be pending (compiling the source or running the testcase). Plans also make semantic distinctions between commands which are syntactically similar. Thus the invocation of the editor simply to browse through a file's contents is distinguished from an invocation where actual changes are made, in that the file system state is updated differently for the two cases. Or, editing of source is distinguished from editing of testcases in that the user state is updated differently. These kinds of distinctions are not made in current interfaces, with the result that reasoning as to the user's past and future actions cannot be performed.

The basic plan framework builds upon other research currently being performed at U/MASS Amherst, namely the work on EDL by Bates, Wileden, and Lesser [5]. EDL is an event description language designed to support the debugging of distributed computer software. There are several significant differences between EDL and the plan structure described above, arising from differences of goals and from differences inherent in the application. In the interests of efficient interpretation, EDL has not incorporated the interleave operator. Further, EDL does not use the notion of a model of the state of the user's world over and above that given by the history of past events recognized. Requirements to handle early interpretation, uncertainty in interpretation, error correction, and prediction exist in the software development environment application, but are not of such immediate importance in distributed debugging. It is certain that the research discussed here will draw heavily upon AI techniques for precisely these reasons, whereas EDL need not.

2.2 A Specification Framework for Plans

In this section, we introduce by way of an example, the framework that we have developed for specifying plans. As our example, we have chosen a hypothetical software development environment. Such an environment lends itself well to the application of an intelligent interface because many of the activities are largely mechanical, highly error-prone, and have straightforward bookkeeping aspects. Much programming work consists not of creating programs, but entering the programs into the computer, changing them, compiling them, manipulating files containing them, writing reports about them, and communicating with other programmers about them. Thus, it is a good test case both for showing the usefulness of an intelligent interface, and (just as importantly) for justifying the premise that activities in this domain are structured and employ identifiable knowledge, allowing for the interpretation of partial sequences of interleaved actions as higher level units of activity.

The plans for this hypothetical software engineering environment form a hierarchy which is given in Figure 2; the links in the hierarchy show which lower level plans are used in the definitions of higher level plans. The plans in their entirety appear in Appendix A. One primitive plan (issue_command) representing the execution of a load module, system command, or user-written command procedure with certain inputs, outputs, parameters, and completion code is provided. Certain forms of this primitive plan are used to construct the next level of plans; for example, when the entity being executed is the Ada compiler, then it qualifies as an instance of the plan compile. These plans in turn form part of other plans, such as the plan for update source unit, which is formed from compile, along with edit and check results.

These example plans assume (or define) a particular toolset and a particular style of tool usage appropriate to that toolset. The plans could just as easily define a different style with the same toolset, or adapt a new toolset to this style. Other approaches to supporting programmers in a software development environment have tended to focus on the issue of what tools ought to be in the toolset and to develop one particular style which is well-suited to those tools. An example is the Gandalf project [6, 7, 8], which uses a "constructive" approach to software development via syntax directed editing, compilation-based technology to support incremental development, and system construction via pre-defined parts lists. Research into tools and research into intelligent ways of interfacing those tools are in fact complementary, not contradictory. One could use the plan framework described here to define the Gandalf toolset behavior and style, for example.

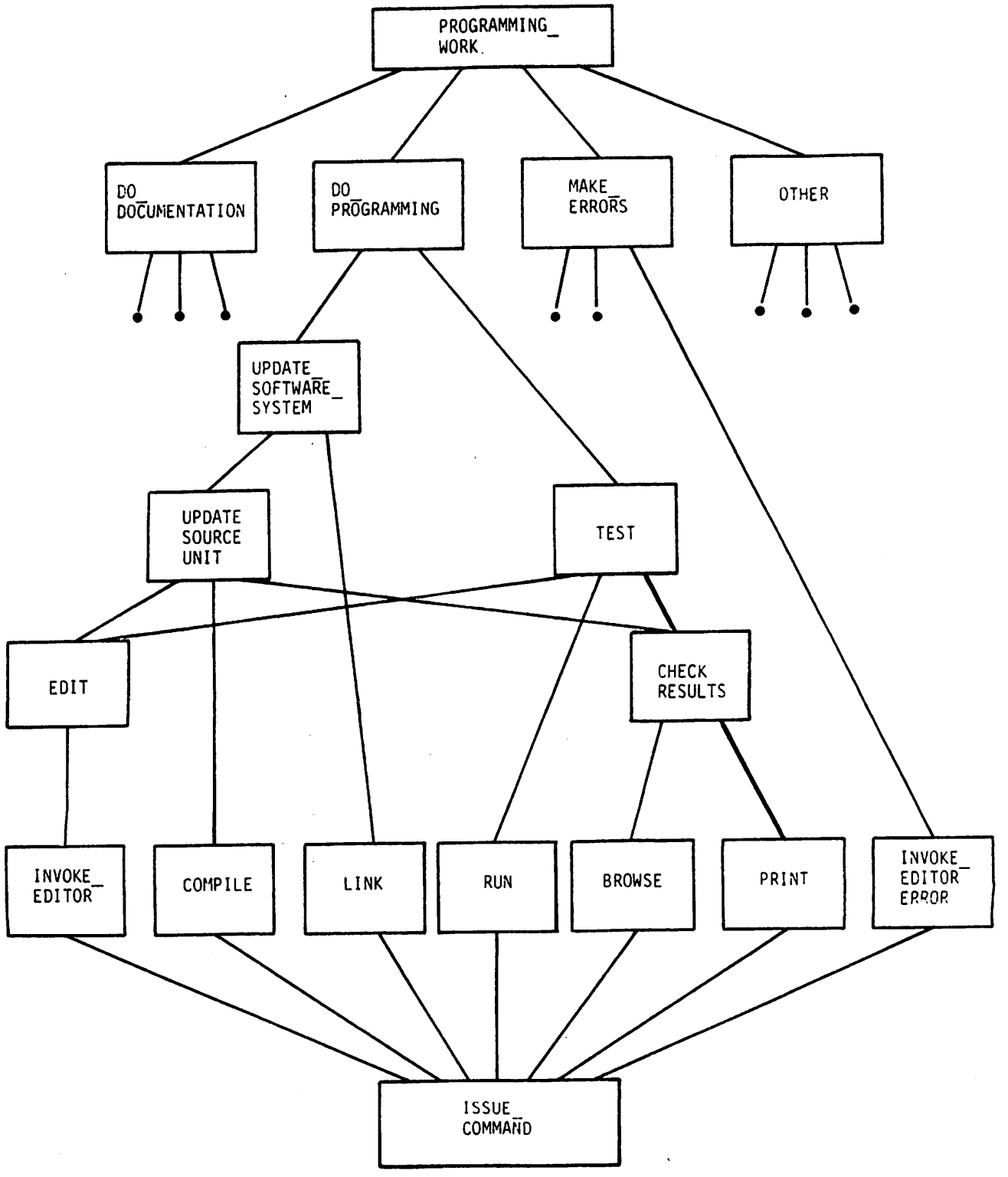

Figure 2. Plan Hierarchy

2.3 A Typical Plan

A detailed view of the plan representation can be obtained from a description of an example plan. We have chosen for this example the plan for "invoke_editor". The definition of "invoke_editor" (as it appears in Appendix A) is repeated here as Figure 3.

All plans except primitive (builtin) plans are defined using shuffle expression notation (regular operators augmented with concurrency operators) over other plans, thus creating a plan hierarchy wherein higher level plans are composed of lower level plans. Associated with a plan is a set of conditions (the COND clause) which must be met and a set of conditions (the LIKELY clause) which cause warnings if not met. Attributes associated with each plan are given in the WITH clause along with information about how to determine their values. The attribute values of lower level plans may be referenced in a higher level plan which is defined via the lower level plans. Finally, the EFFECTS clause states what facts in a model of the real-world become true when an instance of this plan has occurred; this clause acts as a kind of state-machine specification of the semantics of the plan. The model provides a database of information which is referenced in the COND, WITH, and EFFECTS clauses of the plans.

2.3.1 IS Clause Shuffle Expression The definition of invoke_editor begins with an IS clause, here stating that the plan invoke_editor is an occurrence of the plan issue_command (subject to the further constraints of the COND clause.) This is an example of the simplest kind of plan expression possible, merely the name of another plan; in general, plan expressions may take advantage of the full set of shuffle expression operators which have been chosen to handle plan concurrency (defined more fully later in Figure 4). In the plan hierarchy, this establishes invoke_editor as higher in level than issue_command, as was shown in Figure 2. The plan expression has the secondary effect of making visible, within the definition of invoke_editor, all of the attributes of issue_command, such as program_name or execution_info; these attributes are defined in the WITH clause of issue_command.

2.3.2 COND Clause Constraints Next in the specification of invoke_editor is the COND clause, giving additional constraints on interpretation. The first constraint in that COND clause specifies that only those issue_commands in which the program being executed is a text editor are possible occurrences of invoke_editor. The next constraint requires either that the file being edited already exists and is of editable type, or that the file to be edited is in fact being created by this edit operation. This is an example of use of the database information, where all the interesting facts about the existence and attributes of objects are modeled. Whereas the first constraint was oriented toward distinguishing the various kinds

```
PLAN invoke_editor IS

     issue_command

COND

     program_name INSET {"vi", "rand"};
     || two editors, vi and rand, are assumed to be available

     (NOT exists (in_file[1])) OR (kind(in_file[1]) INSET
          {source, testcase, doc, memo, mail,
          unknown text_type});
          || if an existing file is being edited, then it must be
          || of an editable kind

     modification_flag = change;
     || actual changes must have been made to the input file
     || (see also the plan for browse)

WITH

     successful := (return_code = ok);

     file_created := (NOT exists (in_file[1]));

     file_updated := NOT file_created;

     new_file := NEW_VALUE'get_contents (out_file [1]);

     old_file := NEW_VALUE'get_contents (in_file [1]);

EFFECTS

     IF successful THEN
          EFFECTS_OF'create_file
               (out_file[1], unknown_text_type);
               || enter the file and its contents in the
               || database, along with their attributes.
               || kind will be updated when a defining use
               || of the file is recognized at a later time
               || via another plan.
          IF file_updated THEN
               NEW_VALUE'successor
               (in_file[1], out_file[1]) = TRUE;
               || describe new_file in database as the successor
               || historically to old_file
          ENDIF;
     ENDIF;
END PLAN;
```

Figure 3. Definition of Plan Invoke_Editor

of operations available to the programmer, the second is oriented towards recognizing errors on the programmer's part. The third constraint, requiring that actual changes are made to the file in the course of the editor session, is present in order to make a semantic distinction between changing a file and merely looking at it (the latter being covered by another plan).

In order to recognize an instance of invoke_editor, we must satisfy both the syntactic conditions of the plan expression as well as the detailed constraints of the COND clause. If these constraints fail to be satisfied for a particular instance of issue_command, matching against other plans in which issue_command can occur may succeed. Matching can be thought of as proceeding in parallel, and need not return a single answer; we are prepared to handle non-determinism in plan interpretation, arising from cases where plan definitions lead to overlapping sets of plan instances.

This example plan does not have a LIKELY clause; if one were to appear, it would list constraints which are expected in general to be met, but which need not be met in the unusual case. A LIKELY clause otherwise has all the characteristics of a COND clause.

2.3.3 WITH Clause Attributes Assuming that a particular instance of issue_command does match all the constraints, then the WITH clause specifies how to calculate the attributes of an instance of invoke_editor, using the attributes of the instance of issue_command and any database references which those attributes make possible. In the scope of the WITH clause, the database references may be either to the database before the EFFECTS are recorded or to the database after the EFFECTS are recorded. In this example, the switch file_created, used to indicate whether or not a file was in fact created, must be calculated from the existence or non-existence of the in_file before the EFFECTS; after the EFFECTS, the file will perforce have been entered into the database. Also in this example we capture time-independent references to the input and output of this edit operation by explicitly by-passing the file names and using the contents database function; in this case, we make our database references after the EFFECTS have been achieved, as indicated by the use of NEW_VALUE.

2.3.4 EFFECTS Clause Updates Finally, the EFFECTS clause specifies in what ways the database (i.e., the world model) must be updated as a result of this plan having occurred. Here the database is only updated if the invoke_editor was successful (as defined in the WITH clause). When it is updated, two things happen. First, all the effects of the state_changing procedure create_file occur. (The definition of the effects of create_file are given in the formal specification of the database in Appendix B; these effects include the creation of both a file entity and a contents entity in the database, as well as the establishment of

the stored_in relationship between them.) Second, the value of the successor relationship between the in_file and the out_file is made TRUE, thus capturing the information that one file is an historical revision of the other. As is usual for state machine specifications, the EFFECTS are assumed to come about as a unit. The state of the database either before or after the EFFECTS can be referenced, but there is no definition of the state during the change.

2.3.5 Other Syntactic Issues A few other comments are in order on the syntax of the language used to define plans. The plan specifications have been written using upper case for constructs which are "built-in" to the plan language, such as IF, THEN, RANGE, NEW_VALUE, and the like. All other constructs are in lower case. Comments are initiated by "||" and terminated by end-of-line. In plans where more than one instance of a lower-level plan can occur, these instances are indexed in sequential order; thus, compile[i] precedes compile[i+1] in temporal order. However, there is no guaranteed relationship between compile[2] and edit[1]. In fact, the COND constraints often express some semantic (or dataflow) relationship which must exist between occurrences of different lower level plans; for example, in update_source_unit, we see that the edits and compiles are connected by the relationship that the output of the edit is the input to the compile. In any plan, the attributes of the lower level plans can be referenced by attribute name only, if such a reference is unambiguous; otherwise, they are preceded by the plan name using dot selection to resolve the ambiguity.

2.3.6 Assumptions Underlying Example Plans For purposes of this paper, certain simplifying assumptions have been made in order to keep the examples within manageable limits; it is believed that these simplifications do not reduce the difficulty inherent in the problem domain. A very small tool set has been addressed, but this minimal set could be expanded using the same general approach. A rather bland style of tool usage has been incorporated, and a very simple-minded approach to the definition of some activities (such as testing) has been taken. On another level, no semantic distinctions have been drawn between a programmer making an internal release for purposes of exploring a possible fix to a known bug versus making a version about to be released to users. These two activities might differ in the amount of testing to be performed or in triggering certain back-up actions.

An issue which is far richer than we have indicated in the example plans here is the one of information passed from the tools to the interface and from the interface back to the tools. The question of parameters passed to tools has been largely ignored, and no global context information is shown being passed from the interface to the tool. For example, it is assumed that a successful compilation always results in an object module being created, even though compilers typically provide options for

performing syntactic and semantic analysis without subsequent code generation. Very simple information is shown returning from a tool invocation to the interface; the compiler returns a few facts about the compilation unit which it just processed. In a real-world example, one would want to develop this aspect of the tool/interface relationship in much more detail; we have, in fact, created a context in which this kind of information can be utilized effectively. And, of course there is no reason why the actions within a tool cannot be handled with the same plan formalism, thus providing one consistent mechanism whereby all user commands are processed and understood.

In their present form, the plans are very project independent. It is anticipated that usable plans will include project, task, or programmer specific knowledge and constraints. For example, one might take advantage of knowledge about how a project organizes and names its files hierarchically to infer whether a newly created file contains source, documentation, testcases, etc. Similarly, one could also take advantage of interrelationships such as "changes to module A are likely to be necessary if changes to module B are made". Or, the plans might reflect the habits of a particular programmer, say in showing that he always used a particular editor when editing particular kinds of files. Thus, we expect to see some decoration of these skeletal plans along these lines.

On the other hand, the plans given here are not language independent. They do assume that Ada* is the programming language being used. In particular, a somewhat simplified version of the Ada model of separate compilation [9] has been incorporated. Some model had to be selected; because the Ada notion of a library (around which the separate compilation facilities are defined) lends itself well to this application, it was chosen. As a simplification, it is assumed here that an Ada library can contain only one "main" program.

2.4 Shuffle Expressions and Concurrency

The notation used in composing plans to form new plans is based upon regular expressions, augmented with two concurrency operators. Regular expressions provide operators for union, concatenation, sequential closure, and repetition. Several formalisms for providing concurrency have been defined as extensions to the regular expression framework. This includes the flow expressions of Shaw [10] and the event expressions of Riddle [11]. Recently, the subset common to both flow and event expressions has been studied formally by Gischer [12]. That subset, called shuffle expressions, is the basis for our plan expressions. Shuffle expressions provide two additional operators: shuffle, for expression of arbitrary ordering between

* Ada is a registered trademark of the Ada Joint Program Office -- U.S. Government.

two operands, and interleave, for expression of arbitrary ordering among an arbitrary number of operands. Interleave is the closure of shuffle, so that interleave is to shuffle as sequential closure is to concatenation.

Formal definitions of all shuffle expression operators are given in Figure 4. A shuffle grammar covering all the plans of Appendix A is given in Figure 5; this merely extracts from each plan definition its IS clause.

With the addition of only the shuffle operator to the standard regular operators, the languages expressed are still regular languages. However, with the addition of the interleave operator, the languages expressed are no longer regular -- in fact, Gischer has shown that they are context-sensitive [12]. The exact status of languages defined using the interleave operator is still an open question. In [12], it is shown that the set of such languages is not equivalent to the set of labelled Petri net languages (another formalism which has been used to express concurrency). An example of a language which is a labelled Petri net language but not a shuffle language is given there.

An approach to formal definition of office procedures based upon Petri nets was developed by Zisman in the SCOOP system [13]. He used Petri nets to organize production rules associated with asynchronous, concurrent processes. The nets were then used to recognize actions and to complete certain actions automatically. His standard example is the work of a journal editor receiving papers, sending them out for refereeing, receiving referee comments in return, and making publishing decisions. In this example, there are no instances of ambiguity or sharing of actions, characteristics which are pervasive in the domains we are considering, and which require considerable additional sophistication in order to be handled.

We have already mentioned the role of concurrency of tasks from multiple plans in the structure of activities carried out by people. It is essential to be able to express arbitrarily complex instances of such structuring in the plan language. Thus, in our plans both the shuffle and the interleave operators are provided. In EDL, the interleave operator is not allowed; consequently, EDL can use simpler interpretation algorithms (finite automata), at the cost of restricting the extent of the concurrency which can be handled.

2.5 Semantic Database

A significant feature of our plans is the ability to maintain and reference a database describing all aspects of the application "world" of interest, in this case the programmer's world. This is another way in which our plan language diverges from EDL, which uses no such application-specific database. In this

Let A be an alphabet. Let ^ denote the empty string.

If s belongs to A, then s is a shuffle expression denoting the language L(s) = { s }. And, if s1 and s2 are shuffle expressions, then:

Union: s1 | s2
is a shuffle expression denoting the language:

L(s1 | s2) = L(s1) U L(s2)

Concatenation: s1 s2
is a shuffle expression denoting the language:

L(s1 s2) = { xy | x elem L(s1) and y elem L(s2) }

Sequential Closure: s1*
is a shuffle expression denoting the language:

L(s1*) = { ^ } U L(s1) U L(s1s1) U L(s1s1s1) U ...

Repetition: s1+
is a shuffle expression denoting the language:

L(s1+) = L(s1(s1*))

Shuffle: s1 $ s2
is a shuffle expression denoting the language:

L(s1 $ s2) = { x1 y1 x2 y2 ... xn yn |
n >= 1 and
x1 x2 ... xn elem L(s1) and
y1 y2 ... yn elem L(s2) and
xi elem A* for all i and
yi elem A* for all i }

Interleave: s1@
is a shuffle expression denoting the language:

L(s1@) = { ^ } U L(s1) U L(s1$s1) U L(s1$s1$s1) ...

Figure 4. Formal Definition of Shuffle Expressions

```
issue_command -> BUILTIN

invoke editor  -> issue_command  [on the editor]

invoke_editor_error -> issue command  [on the editor]

edit -> invoke_editor +

browse -> issue_command  [on the editor]

print -> issue_command  [on the print queuer]

check_results -> ( browse | print )+

compile -> issue_command  [on the Ada compiler]

update_source_unit -> ( (edit compile check results ) |
                        (edit compile ) |
                        (compile ) |
                        (compile check_results) )+

link -> issue_command  [on the linker]

update_software_system -> ( update_source_unit @ link ) +

run -> issue command  [on some program created by the
                       programmer]

test -> ( (edit run | run ) check results ) @

do programming -> update_software_system $ test

make_errors -> invoke_editor_error +

programming_work -> (do_programming | do_documentation |
                     make_errors ) @
```

Figure 5. The Shuffle Grammar for the Example Plans

database, we collect all facts relating to the objects the programmer is creating and to which the programmer can apply operations. The plans both reference this body of knowledge and specify how it is to be kept up to date; further, this knowledge is of considerable value to the interface for carrying out its interpretation actions, as we shall see later. While this database captures the relevant information describing what objects actions can be applied to, the relevant information about past and pending actions is given by the plan instantiations. The current state of the programmer, then, is the sum of two parts: the state of the database and the state given by the collection of all partially or wholly instantiated plans (represented as a hierarchy.)

2.5.1 The Database as a State Model The EFFECTS clause in a plan delineates the changes to the semantic database which are brought about by the occurrence of that plan. These changes take the form of the existence of new objects, new facts about previously existing objects, and new relationships between objects. Thus, the database is a collection of entities and relationships [14], essentially modeling the state of the file system in use by the programmer. This database captures the kind of knowledge about files and their contents (the objects of interest) that a programmer typically maintains and manages informally. It is exactly our purpose to identify and use that knowledge which programmers carry in their minds about their work.

A similar sort of database is used in the automated consultation system known as the "designer/verifier's assistant" [15]. The purpose of the assistant is to advise verifiers of the impact of change to the programs or formal specifications comprising a software system and to analyze the need for redoing parts of the original system's proofs of correctness. The function of the assistant is to mediate between the designer and the verification system tools, to make sure that the tools are being used efficiently. The database describes all the interesting interrelationships among programs, specifications, and proofs, to a level of abstraction suitable for reasoning about changes.

Integration through a database is part of a general trend in programming environment design (see for example [16, 17]) towards the sharing of information (knowledge) among tools. Previously, all data produced by a tool was either generated temporarily and thrown away (such as the abstract syntax tree from the compiler) or formatted explicitly for the use of some other specific tool (such as the object module formatted by the compiler precisely for the linker). Now, attention is being paid to raising all potentially interesting tool-generated data to the level of a defined, rather than hidden, interface. The intelligent interface described here builds upon that philosophy by taking advantage of various tool-generated information, filtered to an appropriate level of detail. That is, the interface does not

need to know all the variable names associated with a program module, but does need to know the dependency relationships between that program module and other program modules which exist. Taking this knowledge from the tools rather than from the programmer is essential if the knowledge is to be valid and maintained in valid form through time.

The approach to the representation of knowledge about the components is a further development of work by one of the present authors [18]. The database is divided into three "spaces": one containing entities representing files (in the sense of places to store contents), one containing entities representing file contents, and one containing entities representing abstract concepts. The links between the concept space and the contents space take the form of "instance_of" relationships. The links between the contents space and the file space take the form of "stored_in" relationships. File "names" and concept "names" provide the means for programmers to refer to content entities, which are the basic units to which operations can be applied. The database schema is presented in Figure 6.

The distinction between file space and contents space is made in order to separate location information, namely where something is stored, from other descriptive information, namely what something is. This is convenient for accurately describing contents as well as for insuring that only one content entity is "stored_in" a particular file at a particular time. The distinction is critical to being able to use the database to model the state of the file system over time. The truth is that a file name is a time-dependent reference to an object. When a time-independent reference is desired, it must not be through the file name.

If, at time t1, a certain listing is stored in file x, there is no guarantee that at time t2, the exact same listing is still retrievable from x. For example, between t1 and t2, the programmer might have edited the source and recompiled to make a new listing which was stored back in x. Perhaps the first listing showed compilation errors which were corrected by the editing actions. Then, even though the new listing is in many ways like the old, it is still a different listing and must be described accurately (which is to say, differently) in the semantic database. Further, the database must reflect the fact that the old listing (showing the compilation errors) cannot be retrieved from file x any longer. If a programmer copies file x into file y, the database will have to show that the same content entity is "stored_in" both x and y, or alternatively that the content entity can be retrieved from x or from y.

2.5.2 Formal Specification of the Database A formal specification of this database is given in Appendix B; the facilities are divided into two levels (MODULES), one encapsulating all facilities needed by the plans, and one providing the primitives necessary to build these "user-level"

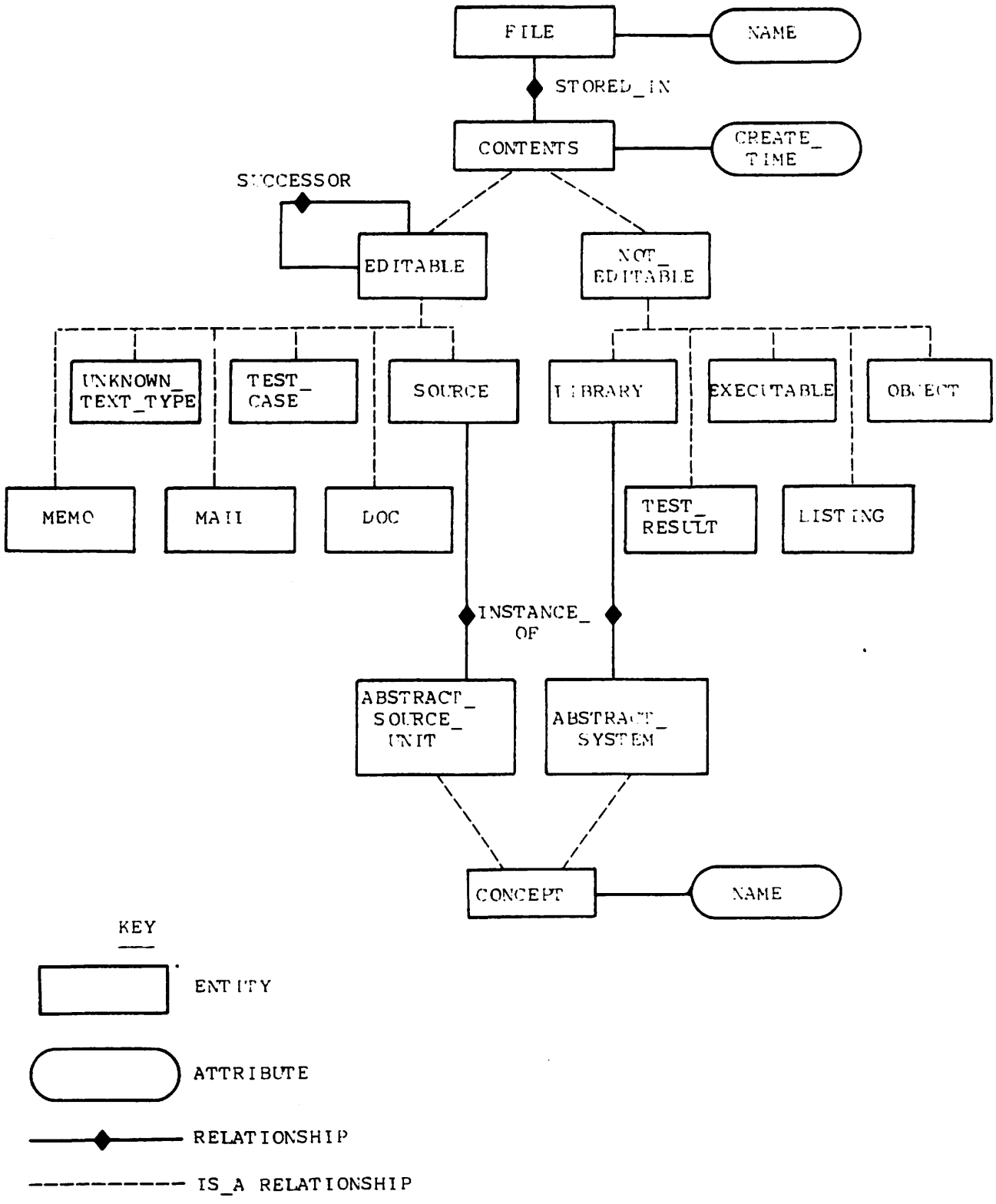

Figure 6. Semantic Database Schema, Part One of Two

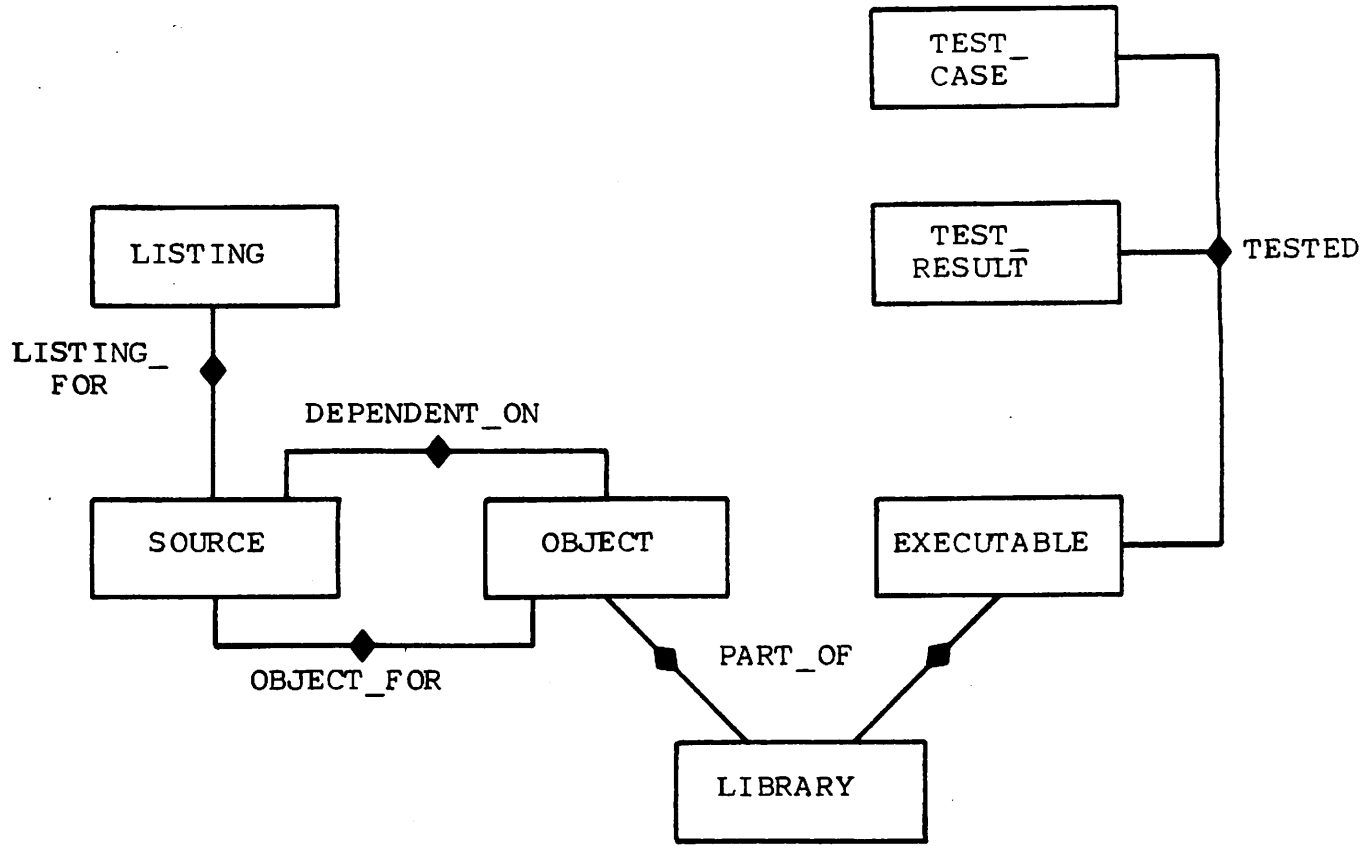

Figure 6: Semantic Database Schema, Part Two of Two

facilities. The MODULE primitive database provides the minimal database primitives. This includes the definitions of the three types of entities (files, contents, and concepts) along with their attributes and all relationships in which they can participate. The function NEW is provided to allocate a new instance of an entity. The database schema of Figure 6 is equivalent to the primitive_database, except that derived relationships have been omitted from the Figure.

The MODULE semantic_database builds upon these primitives to define the full model needed by the plans. It makes visible the database look-up facilities: namely, the ability to transform a "string name" into a database pointer. It makes visible the database retrieval facilities (as in the primitive_database) necessary to retrieve attribute or relationship information and to determine entity existence. Further, it provides some deductive capabilities which make the database more convenient to use. For example, at the primitive database level, "kind" is an attribute of contents but not of files. If we enquire as to the "kind" of a file, what we surely mean is the "kind" of the contents currently stored in that file; this interpretation is supplied by the semantic_database module. With these deductive capabilities, the apparent range of stored information is expanded; the plans need not be concerned with the distinction between stored and calculated knowledge. By keeping explicitly stored information to a minimum, the amount of work to update the database is simplified.

The database specification takes the form of a state machine. This follows the specification style first introduced by Parnas [19], and later exemplified in such systems as SPECIAL [20]. The state of the database is given by a collection of variables and state-revealing functions; for each such variable or function, an initial value is specified using the keyword INITIALLY. The state is changed by invoking procedures which employ the NEW_VALUE operator to change values and/or the NEW function to allocate new (state-revealing) variables. In the specification, each relationship (such as "stored_in") is a state revealing function and each existing entity is a state revealing variable; an example of a state changing function is "create_concept", which allocates new concept entities and fixes the values of their attributes. The plans themselves also change the state, either by use of the NEW_VALUE operator explicitly or by referring to the EFFECTS_OF some state changing procedure.

3. THE OPERATION OF THE INTELLIGENT INTERFACE

Given the foregoing set of plans, it is possible to hypothesize some typical terminal sessions of user commands and to show how the interface performs its roles of interpretation, knowledge collection, planning, and intelligent assistance. The purpose is to show specific examples of the kinds of support which the interface offers, and to sketch some of the internal mechanisms (especially data structures) which are required for the interface to perform its functions. In particular, this will lend some credence to the claims being made for the functional capabilities of the intelligent interface.

The basic assumptions upon which these examples are based include:

* a style of terminal dialog based loosely on UNIX* is used, so that:
    * "%" is the prompt character,
    * input files are listed first following the command name,
    * output files follow after a ">",
    * in order for the reader to distinguish them, system responses are in upper-case and user typing is in lower-case,
    * no system response to a particular command implies that the command completed normally ("no news is good news").
* a running commentary appears to the right of the terminal command sequence to motivate the programmer's actions.
* nonsense file names (such as foo_1) have been used throughout so that the human reader is not mislead into reading more into the example than the intelligent interface can. (If the interface were to be equipped with information about the conventions being used to name files, then it could use these cues in much the same way that people do. We leave this subject for future exploration.)
* editing always takes place on a copy of the file being edited. At any given time, only the current version and the previous version are automatically saved by the system. That is, when the editor is used to change a file, the old version is replaced by the current version and the new version becomes the current version. The file name used for

* UNIX is a Trademark of Bell Laboratories.

the previous version is the same name with a tilde ("~") appended to the end. (We could just as easily have defined the conventions so that all versions were saved in a limitless stack, in the way that VAX VMS behaves.)

* the programmer is assumed to be starting from scratch with the first terminal session; that is, there are no existing files known to the interface.

* actions (such as adding lines, changing lines, etc.) within an invocation of an editor are ignored. (Apropos of earlier comments on this subject, the rationale is to keep the example within manageable limits.)

* certain cooperation from the tools within the environment is assumed. For example, we assume that the Ada compiler returns information identifying the name and type of the source unit just compiled. Also, we assume that the editors can be initiated in a mode where changes are not allowed (that is, only examination of contents is permitted.)

3.1 First Terminal Session

We begin with a very simple terminal session, and show what information is built up in the knowledge base by the end of this session and also what data structures representing the interpretation of commands are constructed.

FIRST TERMINAL SESSION

%LOGIN: huff %PASSWORD:	Standard login sequence for a UNIX style system
25 SEPTEMBER 1981 14:03	System responds with date and time of login
%vi foo_1 ... now in editor	An edit step to create a file which will later be seen to be a source file
%vi foo_2 ... now in editor	An edit step to create a different file, type later seen to be of type source
%vi foo_1 ... now in editor	The editing of foo_2 has reminded the programmer of omissions in foo_1.

```
%ada foo_1 foo_3 >foo_4 foo_5
COMPILATION ERRORS
IN PACKAGE_SPEC A
```

An attempt to compile which fails due to compilation errors. This is the defining usage of all of foo_1, foo_3, foo_4 and foo_5, respectively the source unit, the Ada library, the compilation listing, and the object module.

```
%browse
     ... now in editor
```

Browse through the listing to identify the errors. In a typical environment, the command would have to have been:

```
vi foo_4
```

Here we can allow other commands such as "browse", or "vi listing". The command "check results" is also possible, but is ambiguous, because we don't know whether or not hard copy is desired.

```
%vi foo_1
     ... now in editor
```

Back to the editor to fix the errors.

```
%ada foo_1
PACKAGE_SPEC A COMPILED
```

A successful compile this time. Notice that some parameters have been omitted, and that values have been supplied by the interface, from the previous "ada" command occurrence.

```
%ada foo_2 foo_3 >foo_6 foo_7
COMPILATION ERRORS
IN SUBPROGRAM B
WITH (PACKAGE_SPEC A)
```

An attempt to compile foo_2, which fails, let us imagine, on some trivial source error. This is a defining usage for foo_2, foo_6 and foo_7.

```
%browse
     ... now in editor
```

Check the listing to see what the errors actually were. Again, the programmer did not have to name foo_6 or the editor explicitly.

```
%vi foo_8
     ... now in editor
```

Create another file, intended to be a source file.

```
%logout
TOTAL OF 203 CPU SECONDS
25 SEPTEMBER 1981 16:24
```

That completes this terminal session.

```
SUMMARY:
INITIATED:
   UPDATE SOFTWARE SYSTEM
        FOR: foo_3
        STATUS: PRE LINK-STEP
CREATED:
     foo_8
```

The intelligent interface responds with a summary of the session.

3.1.1 The Semantic Database At this point, the interface has collected quite a lot of information about the files that the programmer is working with. This information takes the form of an instantiation of the database schema of Figure 6. The state of the database at the end of this terminal session is given, in semantic network form, in Figure 7. The entities are allocated in three "spaces": file space, contents space, and concept space. Most of the information is available in the form of relationships, or, using semantic net terms, arcs; the database has been formulated so that rather little information is stored in attributes of the entities. The use of relationships in preference to attributes allows database integrity to be enforced automatically from constraints such as "relationship R is 1-to-n."

The strategy taken towards accumulating information about the components which the programmer is manipulating is a conservative one based upon a desire to avoid requiring the programmer to specify facts explicitly which the system could infer. Thus the interface does not ask a programmer to give information identifying the type and "name" of a file's contents when the file is first created; this information is instead accumulated when a defining usage of the file occurs, as for example when the file is compiled. This approach carries with it a penalty in that binding of information occurs later than would otherwise be possible. It remains to be seen, from empirical studies, whether this will undercut the ability to detect errors in a timely fashion. In circumstances where the programmer is doing the bulk of work with existing files, this would not be a problem; in circumstances where the programmer is embarking on a new task, it could well lead to some performance degradation initially.

In response to situations where the uncertainty levels are high, the interface could take the initiative to query the user for just that information which could be applied to reduce the uncertainty. This is a standard ploy in expert systems; for example, in MYCIN, the doctor is queried for specialized patient information only when it is applicable to the problem-solving issues at hand. A limited use of user-querying will increase the effectiveness of the command interpretation without inconveniencing the user and thus jeopardizing his productivity.

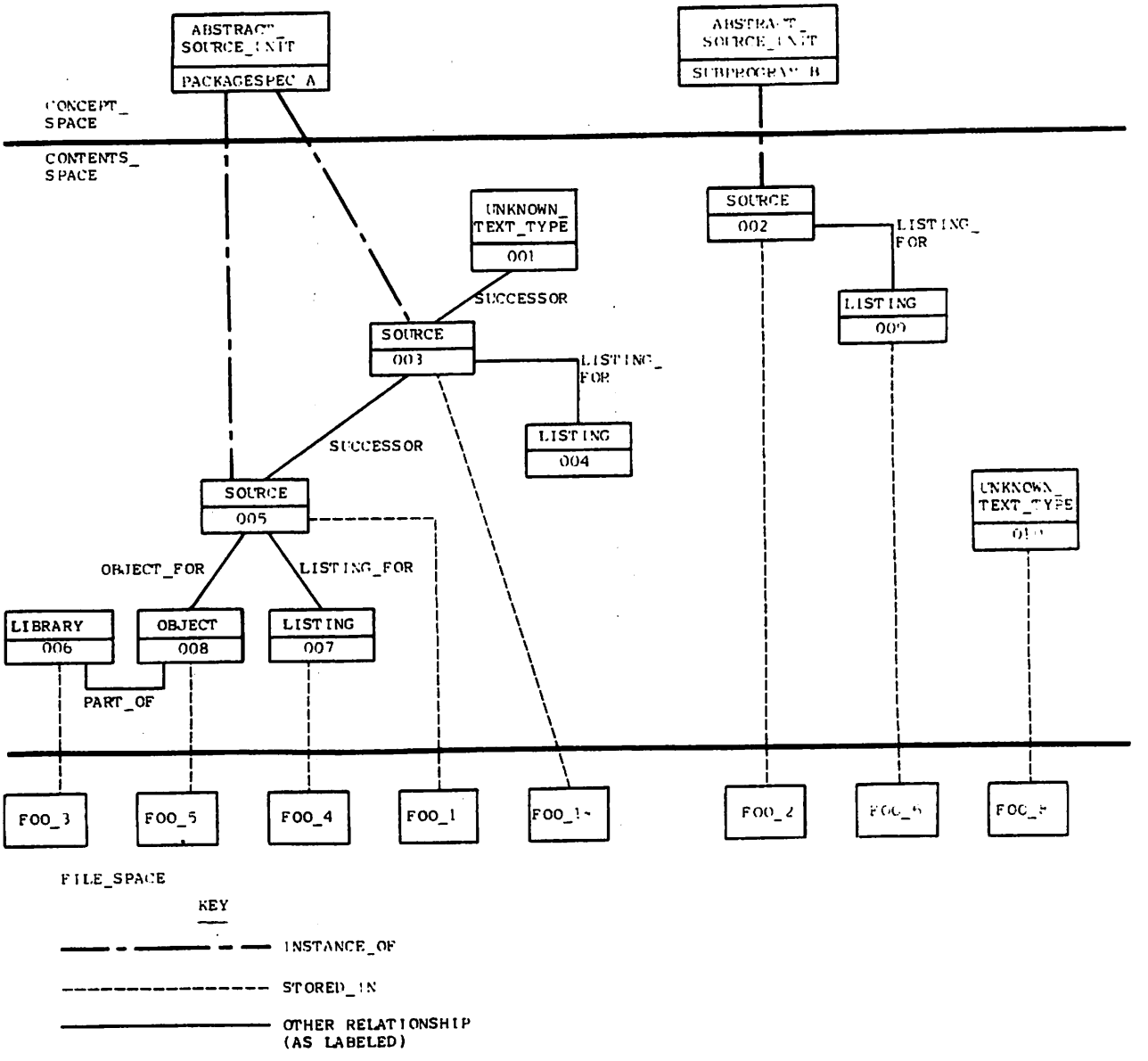

Figure 7. The Semantic Database at End of Terminal Session One

3.1.2 Command Interpretation The "parsing" or interpretation of the command stream also results in information being collected and recorded. Given our grammar-based approach to the definition of user activities, this information rather naturally takes the form of a syntax tree showing the grammatical underpinnings of the command stream. The interpretation process involves construction of such a tree as commands are issued, with the goal that as much underlying structure is elaborated as is possible at each stage of processing. Interpretation is made more complex by the need for early decision making (i.e., early detection of errors), the existence of inherent ambiguity (i.e., non-determinism in plan definitions), and the possibility of sharing (i.e., one action serving two or more roles in the syntax tree).

The syntax tree can be thought of as an attributed tree, where each node (except a leaf node) in the tree contains all the values defined by the WITH clause of the associated plan. The leaf nodes can be thought of as containing the command string as issued by the programmer. Of course, given the addition of the interleave operator which makes plans inherently not context-free, the syntax tree is a little peculiar; in particular, some of the links will cross if the sequential ordering through time of the leaves is preserved. Further, it must be emphasized that the tree of interest represents the application of both syntactic plan information (from the plan IS clauses) and semantic plan information (from the COND clauses), in spite of the fact that we call it a syntax tree in keeping with standard terminology.

We can draw this syntax tree (see Figure 8), by observing the following conventions:

* number each link according to the (relative) time at which all competing hypotheses for the link are resolved to a single possibility,

* number each separate instantiation of a plan,

* ignore the fact that each command is first parsed into an "issue command" (merely in order to simplify the figure),

* ignore the existence of plans at the level of do_programming and above (again, to limit the figure to the most interesting detail).

To reiterate, the figure shows the state of interpretation at the end of the first terminal session. The figure thus does not show those competing hypotheses (e.g., possible interpretations) which were generated at some point in the parse but which have been rejected by the time of the end of the terminal session; only competing hypotheses which are still alive at session end are shown. The justification and comments for each interpretation action are discussed fully in the next section, which may be skipped by the reader not interested in this detailed view.

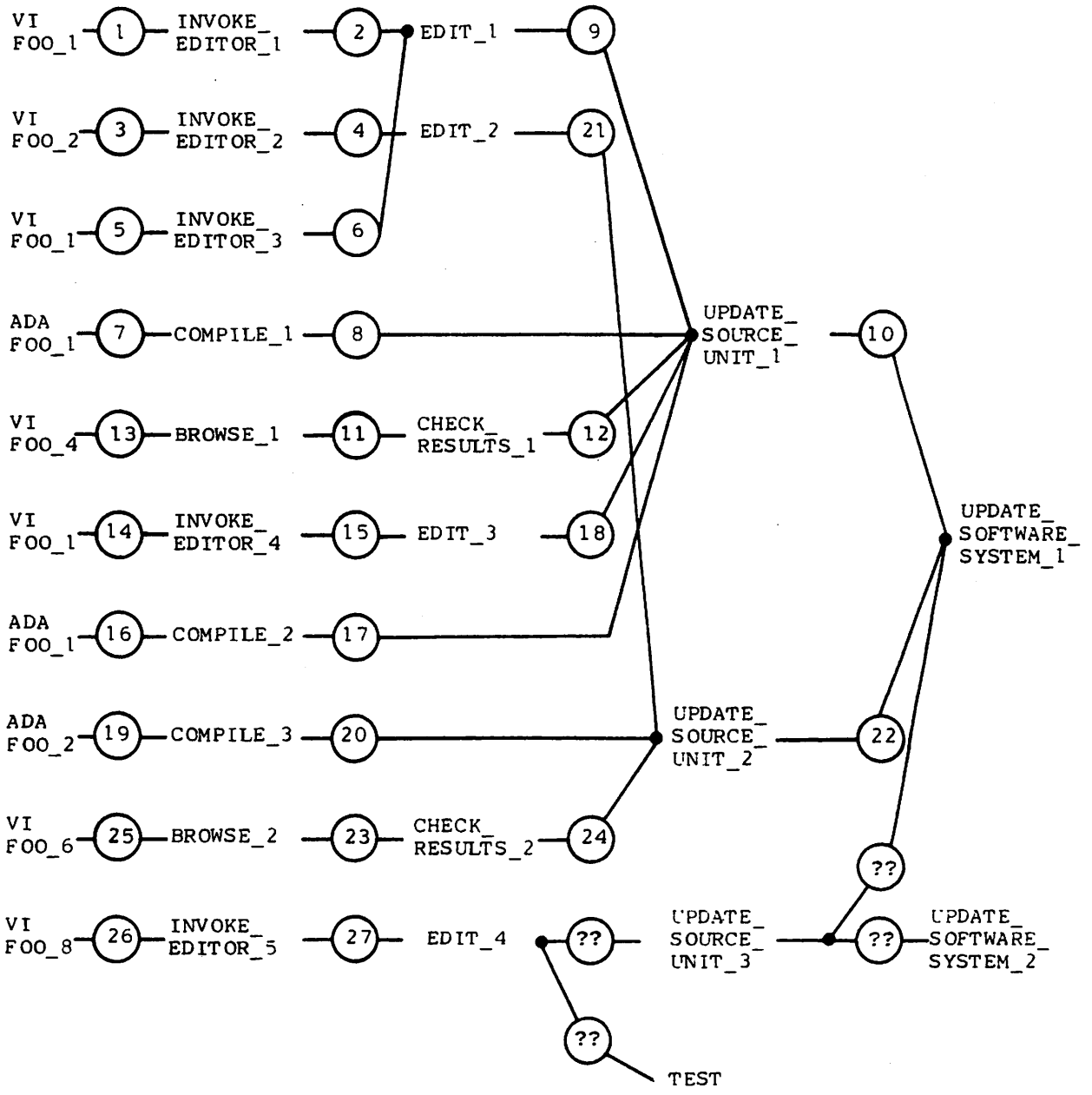

Figure 8. Interpretation of Commands From Terminal Session One

3.1.2.1 Rationale for Interpretation Actions Each paragraph of rationale given here refers to an arc in Figure 8, the syntax tree giving the interpretation of the commands of the first terminal session.

1 - When the command is issued, it can be determined that it does not fit the constraints of either browse or invoke_editor error, both of which require an already existing file. However, it cannot be definitively determined that the command is an invoke editor until after the editor session has terminated and it is known that a file was actually created. At this later time, since a file was successfully created, we can make the command an instance of invoke_editor.

2 - The only plan in which invoke editor participates is edit, and there are no existing instances of edit; so, we create one and connect invoke_editor_1 to it. We can go no further at this time because we do not know if this edit will be part of an update_source unit or a test; these two competing hypotheses will be recorded and held in parallel until it is known which is correct. (Note that since they have one mutually exclusive constraint, relating to the type of the file operated upon, invoke editor_1 cannot be shared between them.)

3 - See reasoning for 1.

4 - The only plan in which invoke_editor participates is edit, and we have one existing instance of edit. However, invoke_editor_2 cannot be part of edit 1 since the dataflow constraint concerning a chain of files edited is not satisfied. Therefore, invoke editor_2 must be part of a new instance of edit, namely edit 2. Again, lacking type information on the contents of foo_2, we cannot say if edit_2 will be part of update source_unit or of test, so we carry these two competing hypotheses.

5 - Since foo_1 now exists and we know it to be of editable type from the effects of invoke editor_1, this cannot be an instance of invoke editor error. Until the edit session finishes, we do not know if it is a browse or an invoke editor. When the session finishes, we see that foo 1 was modified, so this action is made into invoke_editor_3.

6 - Now we see that the output of invoke_editor 1 is the input to invoke_editor_3 and therefore, that invoke editor 3 could be part of edit 1; it cannot be part of edit_2 because the chaining constraint fails to be met. We cannot rule out the possibility that the output of invoke_editor_1 will be used, say in some future compilation or test activity. If that should happen and the output of invoke_editor 3 is likewise used, then invoke_editor_1 would be shared, i.e., it would act as part of the editing chain associated with each use. This sharing does not rule out the interpretation that invoke_editor_1 and

invoke_editor_3 are both part of edit 1. In the absence of any sharing possibilities, we would automatically connect invoke_editor_3 to edit_1. For the sake of this example, let us treat sharing when it is seen to arise, rather than treating it each time it might possibly arise. Thus, we may later activate the hypothesis that invoke editor_1 is part of edit_i for some i, but we do know that we need not retract the interpretation action we have decided to take here.

7 - The only case of issue-command in which the program_name is "ada" is the plan compile, so this must be an instance of compile. The fact that foo_1 is of unknown_text_type is not inconsistent with this interpretation.

8 - The only plan in which compile participates is update_source_unit, so that compile_1 must be part of some instance of update_source_unit.

9 - Given an instance of update_source_unit from link 8, there may or may not be an edit instance which lead to the existence of the source unit being compiled. (Edit is optional in the definition of update source unit). We look around, and see that there is such an edit instance, in edit_1. It may be that later the output of edit 1 is also compiled into some other Ada library, in which case edit 1 will be shared between two instances of update source unit. For now, we can connect edit_1 into update source_unit 1, leaving only the potential for sharing to future determination.

10 - The only plan in which update source unit participates is update software_system, and having no existing instance of this higher level plan, we create one.

11 - The only plan in which browse is used is check_results.

12 - There are two ways in which check_results is used, and in neither case is it a legal first operation in those plans. There are no hypotheses relating to any tests having been performed (via the run plan). Therefore, this check results must fit with update_source_unit_1, which is the only existing or hypothesized instance to which it could be connected. Further, we know that check results is mandatory in the case where the compile has failed, as happened here.

Note that although the outcome of this reasoning is that the browse is associated with the immediately previous operation in the command sequence, that is not the reason why it was connected that way. A more complex notion of command context is being used here. The implications and benefits of this context reasoning, as compared with simpler but less powerful mechanisms, are discussed more fully in section 4.2.2 below.

13 - Now that we know (links 11 and 12) what context the browse fits in, we know what is to be browsed (the compilation listing of compile_1) so we can generate the "vi foo_4" command. Note that this link points leftward, not rightward as the others have done.

14 - See 5.

15 - Note that invoke_editor_4 cannot be part of edit 1, even though all local constraints are satisfied, because edit_1 has been interpreted as part of update_source unit and within that plan, edits and compiles cannot intermix -- they must be strictly sequential; there has already been a compile to "consume" the product of the edit 1. New editing must be part of a new edit operation, thus becoming edit_3.

16 - See 7.

17 - The only plan in which compile participates is update_source unit. Therefore, either this compilation is part of the continuation of update source_unit_1 (all of whose constraints it satisfies) or it is part of a new instance of update_source_unit, in which case we know that a new (unnamed and as yet non-existent) Ada library must be involved. We know further that a new compile is LIKELY in update_source_unit 1 in order to overcome the errors found on the last compile. Arguing heuristically that a continuation of an existing plan is more likely to hold than the start of a new plan, and further that only association with update source_unit_1 supplies all omitted parameter values, we commit to the interpretation that this compile belongs to update_source unit 1.

18 - See 9.

19 - See 7.

20 - Using COND clause constraints, we see that compile 3 cannot be part of update source_unit 1 even though the same ada library is involved because the source unit is not the same. Therefore, it must be a new instance of update source_unit.

21 - See 9.

22 - The only plan in which update_source_unit participates is update_software system. There is an existing instance of this, and the COND clause is satisfied if this new action is part of the existing instance. Therefore, we connect them to one another.

23 - See 11.

24 - Check_results 2 could be part of either update_source_unit_1 or update source_unit 2. Arguing heuristically that it is more

likely to go with the activity closer in time sequence (update_source_unit 2, the subject of the previous command to compile) and that in any case, update source unit_1 has one instance of the mandatory check_results which is the minimal requirement, we connect it with update source_unit 2. This is another instance of a heuristic argument in this interpretation sequence.

25 - See 13.

26 - See 1.

27 - See 4.

3.1.2.2 Interpretation Algorithms For the most part, the interpretation proceeds in a bottom-up manner; that is, the commands issued by the programmer are the leaves of the tree and the superstructure is built up from the leaves. However, when the programmer issues a command which is a plan name, then the interpretation proceeds middle-up to arrive at a context (higher level plan) sufficient to provide bindings of parameters (attributes); from this context, the algorithm can proceed middle-down to arrive at commands (and all parameters) which the operating system will recognize. Two instances of this occurred in the first terminal session, when the programmer twice gave the plan name "browse", which were resolved into "vi foo_4" and "vi foo 6" from the middle down, only after the "browses" had been associated with their respective update source_unit instantiations in middle up order. When an instance of browse is interpreted as belonging to an instance of update source_unit, then the item browsed must be the compilation listing of that update source_unit; that is how the parameter binding is accomplished.

At the end of this terminal session, there are three competing hypotheses for an interpretation of the very last command, "vi foo 8". These are shown in Figure 8, as the links marked with "??" showing that they have not been uniquely resolved as yet. The first possibility is that foo_8 is a source module destined to be eventually compiled into the existing Ada library. The second possibility is that foo 8 is a source module destined to be eventually compiled into some new Ada library. The third possibility is that foo 8 is a test case, in which event we would expect to see it used in an instantiation of the test plan. More possibilities would of course exist if there were additional plans dealing with documentation and project-communication (e.g., mail).

A distinguishing aspect of the interpretation strategy involves carrying forward several competing interpretation hypotheses, and checking for errors against those possibilities still alive. The fact is that, unlike classical parsing in a compiler context, the interpretation decisions cannot always wait until all necessary

information is available; such a timid strategy would lead to a situation in which opportunities for error detection would be lost. It is hardly acceptable to announce, at the end of a terminal session, that an error was made back at the third command issued. Thus, interpretation must provide for ranking competing alternatives and using heuristics to select between alternatives when such a selection can not be made deterministically. The trade-offs involving the inefficiencies of un-doing and re-doing interpretation decisions and the possibility of missing errors if only the hypotheses of greatest likelihood are followed will undoubtedly have to be set after more empirical analysis of actual case studies. Of course, the more aggressive strategies also lead to opportunities for incorrectly inferring what the programmer intends.

The need to make heuristic interpretation decisions on-the-fly with incomplete information distinguishes the approach described here from several other systems which perform automated consultation. In the case of the MACSYMA advisor [21], the advisor must be explicitly invoked by the user when a problem arises and, further, the advisor has the luxury of access to the entire erroneous plan when it begins to analyze the difficulties. Thus, the plan debugging paradigm used by the advisor takes place in the context of complete information as to what the user did.

The Programmer's Apprentice project [22, 23, 24] uses plans to represent the deep structure of programs in a way similar to our use of plans to represent the deep structure of programming work; these plans form the basis of their approach to both program synthesis and program analysis. They have recently tackled the specific issue of deriving the underlying plan structure from an existing program [25]. Again, given the entire program, they have the freedom to delay interpretation decisions until all information is available and even to make multiple passes over the program in the process of constructing the underlying plan. In both their application and ours, the not insignificant issues of ambiguity and sharing must be confronted.

Another plan recognition system, in a much more open ended domain than ours, is the BELIEVER system [26]. In BELIEVER, human actions (such as making an ice cream cone in the kitchen) are interpreted in light of the interrelationships between outcomes of actions and goals which those outcomes might be intended to achieve. An additional dimension which the system must deal with, over and above the state of the world in which the players find themselves, is the set of intents, beliefs and expectations of the players (their psychological models). BELIEVER operates strictly with goal and effect reasoning and does not employ a set of predefined plans such as we have proposed. Thus, it could be said to reason strictly from first principles. In our case, where there are a reasonable number of activities specifiable in advance, we have avoided "real-time" plan synthesis through the use of the given plan definitions.

3.2 Additional Terminal Sessions

The focus of the first terminal session was primarily on the collection of information, rather than the active use of that information. The intelligent interface performed three visible functions for the programmer: first, in providing translation of commands expressed in more abstract terms into commands acceptable to the operating system (browse -> vi foo 4 and browse -> vi foo 6); second, in providing parameter completion on one command (ada foo_1 -> ada foo 1 foo 3 > foo 4 foo 5); and third, in providing a summary of the activities of the terminal session. At this point, the interface is in quite a good position to be helpful for further terminal sessions.

Each terminal session beyond the first can begin with a summary of pending activities, which will be similar in flavor to the summary which appears at the end of the previous terminal session. It will however be different in scope, in that pending activities resulting from multiple terminal sessions will be included. Thus, the second terminal session might open as follows:

```
%LOGIN: huff
%PASSWORD:

26 September 1981  8:58

PENDING ACTIVITIES:
     UPDATE SOFTWARE SYSTEM
          FOR: foo_3
          STATUS: PRE LINK-STEP
          NOTE: COMPILE OF B FAILED

     USE OF foo 8

% .......
```

The interface is now in a position to provide error detection for a variety of situations which could occur. Rather than providing an actual terminal session in which a wide variety of errors were made (which would stretch the limits of credibility), we list some cases which would be detected:

* an attempt to issue a link on foo_3 without ever having made a change to foo 2 or foo_1, that is without having done something to correct the problems underlying the compilation failure on B.

  This action fails to satisfy the LIKELY clause in update_source_unit for B, because the plan instantiation currently ends with an unsuccessful compile. Actually, the programmer is doubly protected from this error because there

is also the reminder in the terminal session opening summary which helps to bring this potential problem to his attention.

* re-issuing a compile command on foo_2 without having made changes to foo_1.

  This is not so much an error, as it is an exact repetition of a command already issued. This is also excluded by the conditions given for compile in the LIKELY clause.

* an attempt to change, by editing, the listing file from some compilation (instead of making the changes to the source file itself).

  This error is recognized because files of type listing cannot be edited, only "browsed". Note that the interface is in a position not only to flag this as an error but also to suggest the file which the programmer most likely intended to edit.

3.2.1 Second Terminal Session Having given examples of what our intrepid programmer might inadvertently attempt to do, let us assume that all such traps are avoided and that the next terminal session proceeds as follows:

SECOND TERMINAL SESSION

.....	Session opens as sketched above.
%vi B	Proceed to correct errors in the most recent revision of B. The interface supplies the binding to the desired revision.
%compile B COMPILED SUBPROGRAM B WITH (PACKAGE_SPEC A)	Compile new version of B. Binding to desired revision and filling in of additional file names performed by the interface.

```
%compile C
IS C INSTANTIATED IN foo_8?
yes
COMPILED SUBPROGRAM C
WITH (PACKAGE_SPEC A)
SEPARATE (SUBPROGRAM B)
```

Here the command given is a plan name and the object to be operated upon is a concept name. The interface has no information about this concept, but can find only one file which it might be. The user is queried to see if this is correct, and the response is yes. Then all proceeds as normal.

```
%link foo_3 >foo_11
MAIN SUBPROGRAM IS B.
```

Having achieved 3 successful compiles, the link step is performed.

```
%vi foo_12
```

Make up first test case.

```
%vi foo_13
```

Make up second test case.

```
%foo_11 foo_12 >foo 14
```

Run first test case.

```
%run ABC foo_13 >foo_15
IS ABC INSTANTIATED IN foo_11?
yes
```

Try second test case. ABC becomes an "alias" for the abstract system consisting of A, B, and C (also known by the name of its main program, B).

```
%check_results foo_15
HARDCOPY?
yes
```

Get listing of second test case outcome.

```
%logout
TOTAL OF 506 CPU SECONDS
26 September 1981 10:45

COMPLETED:
     UPDATE-SOFTWARE-SYSTEM
          FOR:  ABC IN LIBRARY foo_3

INITIATED:
     TEST
          FOR: ABC AS foo_11
          STATUS: TWO CASES TRIED,
                  ONE CHECKED.
```

Session termination.

This terminal session shows a mixture of command and plan invocations, with a considerable amount of parameter binding

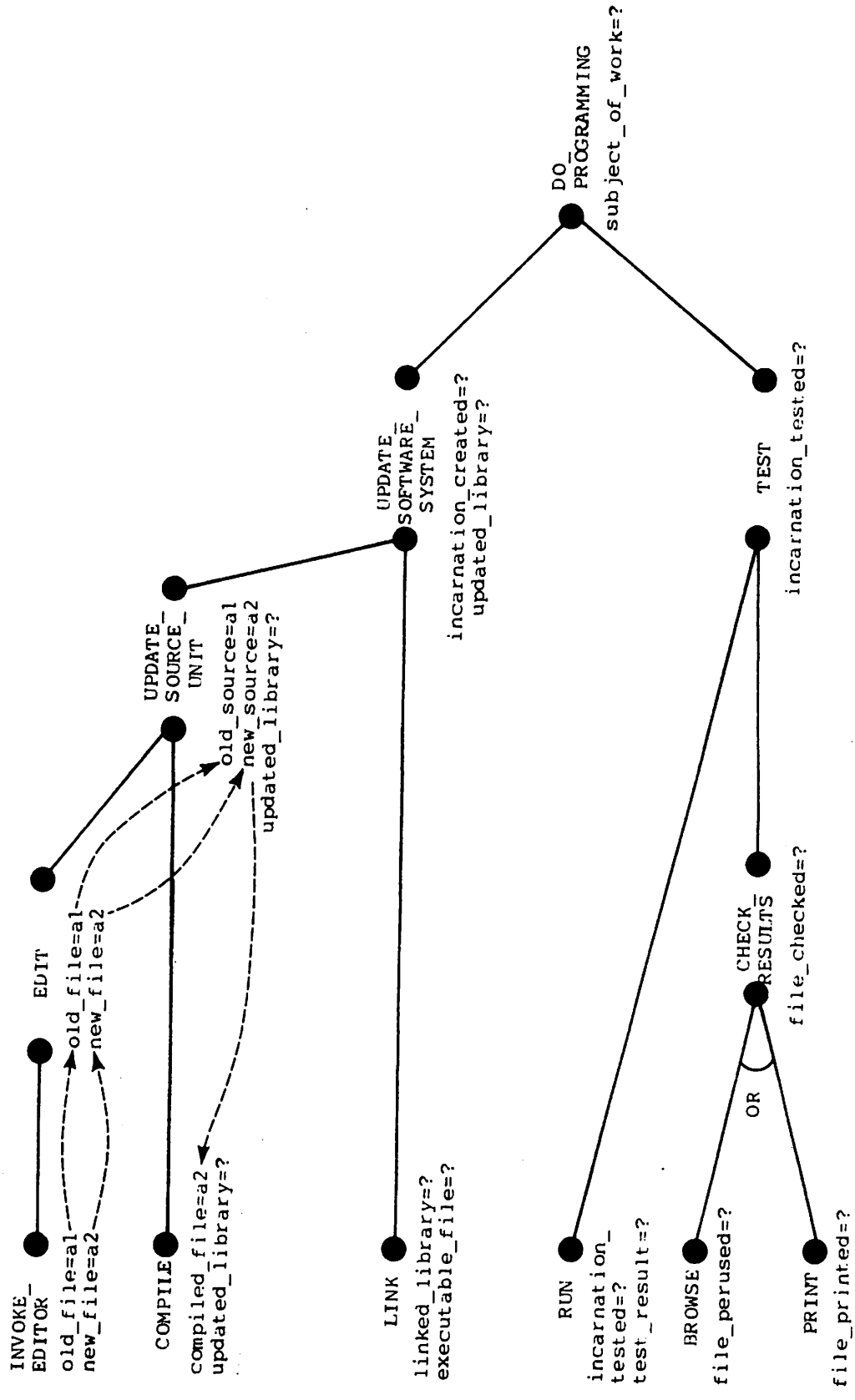

Figure 9. An Example of Prediction and Forward Propagation

being performed by the interface. It is now the case that the interface has seen a complete instantiation of the update_software system plan for a system made up of three components (A, B, and C) incorporated into an Ada library stored in foo_3. Further, the interface has seen how the programmer intends to go about testing of this system, and has observed two test cases actually run. The interface is now in a good position to be of significant assistance for future work on the same system. Before considering a final terminal session example, it is appropriate to explore the issue of prediction.

3.2.2 Prediction and Forward Propagation The information collected in the database, taken together with the plan definitions themselves, provides a sufficient mechanism for making predictions of future actions. These predictions take the form of partial instantiations of entire plan hierarchies. Such an instantiation can be constructed when the programmer issues the first command in a new set of activities, and can be augmented and updated as further commands are issued. There will of course be some uncertainties in the hierarchy, where parameters are unknown, where interleaving is unknown, and where numbers of occurrences of some actions are also unknown. For some unknowns, there will be guesses which can be made using the database information. In general, without the information in the database, there would be so many uncertainties that the instantiations would be useless. The predicted hierarchies can be used for agenda management and additional types of error detection, as well as in the implementation of command interpretation algorithms.

To take an example, consider the prediction that can be made when the programmer issues "vi foo 1", say at the start of the next terminal session. Because we know that foo 1 is a source module, we can follow this line of reasoning: invoking the editor on foo_1 can only be part of editing source, which can only be part of updating a source unit (which implies certain compile and possibly check result actions), which can only be part of updating a software system (which implies certain link actions), which can only be part of doing programming (which implies certain test actions, which themselves imply certain possible edits, runs, and checking of results.) The plan hierarchy which can be instantiated from this single command is given in Figure 9, and discussed in the next few paragraphs.

We can identify the most interesting parameters in this hierarchy as the Ada library, the sources to be changed, and the test cases to be run. (Other parameters can be deduced from these "significant" parameters.) These parameters are exactly the attributes of the WITH clauses of the instantiated plans. As parameters become bound, we can propagate them through the hierarchy in order to make bindings or predictions of bindings of the remaining free parameters. This propagation (occurring both upward and downward in the hierarchy) follows the attribute value

relationships established by the COND clauses and the WITH clauses of the plan definitions.

Knowing just the first source unit being changed, we can make predictions as to the Ada library: it is likely to be the Ada library to which the pre-edited source already belongs, some other Ada library which is an instantiation of the same system, or possibly a new Ada library. Knowing the source unit also suggests what test cases are candidates for being run. After the first compile, we will know for certain the binding of the Ada library parameter. This binding can be propagated all the way up and down to the link command, based upon a chain of COND and WITH clause constraints relating attributes successively through update source_unit, update_software_system, and link. As time goes by, the hierarchy will become increasingly definitive through this propagation.

The figure depicts the minimal set of predicted future actions based upon the single occurrence of invoke_editor. For each predicted action, the associated parameters (i.e., the WITH attributes of the plan) are shown; they can be thought of as the variables associated with this instantiation. The propagation of the only known parameters (the oldfile and newfile subject to editing) is shown with dotted lines. If the next action in this task is another invoke_editor on the newfile a2 to produce a3, then the parameter bindings in edit, update_source unit, and compile will change accordingly. If another source unit is edited, then another instance of update source unit will be added into the hierarchy. When the compile action is seen, the library parameter will be bound and propagated.

We can also make predictions relative to future states of the database. As the parameters are bound and propagated, we can predict the existence of certain database objects, with certain attributes. For example, we know there will be a new object module for each source changed, and we know there will be a new executable eventually associated with the Ada library after the link action. In addition, there are constraints placed upon the database as a result of the plan prediction. We know that the Ada library being operated upon by a plan hierarchy must continue to exist for the duration of the plan; otherwise, it will be impossible to complete the necessary plan actions.

The current collection of partially instantiated plan hierarchies is exactly the current agenda of the programmer. Filtered to some appropriate level of detail, these hierarchies are exactly what is needed to enumerate the pending activities, as was shown for the previous terminal session. There are two ways that pending activities could be described: looking forward, as the set of actions yet to be performed, or looking backward, as the status achieved through the actions already performed. Predicted hierarchies are necessary for the first kind of description to be made; interpretation of past commands is all that is necessary

for the second kind of description.

The use of prediction hierarchies in error detection is potentially quite interesting, because it allows a new class of errors to be identified. These errors are actions which prove to be inconsistent with known future actions, not just known past actions. It is even possible that the predicted actions from some newly-issued command are inconsistent with the predicted actions of some previous command; this would be a second-order example of the same class of error. Typically, this kind of error will arise when an action makes it impossible to complete a partially instantiated plan, usually because some database object necessary for completion is replaced, deleted, or overridden.

Two examples are as follows. If a programmer has recently achieved a successful link on a modified Ada library, and without having tested it, decides to "clean up his file space" and inadvertently issues a command to delete the library or the executable module, then he should be challenged by the interface on this inconsistency. Alternatively, if, under the same circumstances, he proceeds to modify that library by editing and compiling some source into it, then the interface should advise the programmer about the implications of his action: namely, that compiling is expected to be followed by linking which will destroy the existing (untested) load module.

The final use of prediction is in the implementation of the command interpretation algorithms. Here, there is a need to use all available information in order to make interpretation decisions as early as possible, as has been mentioned previously. If one possible interpretation of a new incoming action matches with a predicted action, then we have significant support for preferring that interpretation over others which are not tied in with preceding actions. In the interests of optimizing the interpretation process, one might screen an incoming action against the existing predictions to make a first cut at interpretation and possibly by-pass the more exhaustive (and more expensive) searching. Thus, the prediction hierarchies provide supporting arguments and possibly a cheap interpretation mechanism which are of value in command interpretation.

3.2.3 Third Terminal Session Let us consider one final example of a terminal session, in which the programmer continues work on the system ABC by adding some new functionality. Thus, we will be able to see the interface employ all the knowledge it has gained from the previous activities of the programmer.

THIRD TERMINAL SESSION

```
%LOGIN: huff
%PASSWORD:

27 SEPTEMBER 1981 12:43

PENDING ACTIVITIES:
     TEST:
          FOR: ABC AS foo 11
          STATUS: RESULTS IN foo_14
                  FROM foo_12
                  NOT YET CHECKED.

%browse foo 14

%update_software system ABC
UNITS ARE A, B, AND C
WHICH UNITS DO YOU WANT TO UPDATE?
B
  ....now in editor for B

%DO YOU WANT TO COMPILE NOW OR WAIT?
wait
%DO YOU WANT TO CHANGE ANY OTHER UNIT?
A
  ....now in editor on A

%DO YOU WANT TO COMPILE OR WAIT?
compile all

PACKAGE_SPEC A COMPILED



SUBPROGRAM B COMPILED
WITH (PACKAGE_SPEC A)

SUBPROGRAM C COMPILED
WITH (PACKAGE_SPEC A)
SEPARATE (SUBPROGRAM B)

LINK PERFORMED

UPDATE_SOFTWARE_SYSTEM COMPLETED
```

The expected login sequence.

Finish checking last test case results.

This very high level plan invocation causes the interface to go into a planning mode.

Some intelligence is required by the interface here to select the optimum order in which to compile the modules. With the Ada notion of separate compilation, modules must be compiled after the modules upon which they depend.

```
%test                                          || Again, a planning
THERE ARE TWO TEST CASES AVAILABLE             ||      mode is set-up.
EDIT OR RUN?
run all

foo_12 RESULTS READY

foo_13 RESULTS READY

HARDCOPY DESIRED?
yes
TWO TEST RESULTS PRINTED

%logout                                        || Session ends.
TOTAL OF 867 CPU SECONDS                       ||      Standard summary
27 September 1981  4:35                        ||      provided.

INITIATED AND COMPLETED:
     UPDATE SOFTWARE SYSTEM
          FOR: B IN LIBRARY foo_3
     TEST
          FOR: B AS foo_11
NO ACTIONS PENDING.
```

This last terminal session consists of a very high level of dialog between the programmer and the interface, for which the interface draws heavily on the information it has saved from the previous command activity in both the semantic database and the interpretation parse tree. The actions of the interface have shifted from being interpretation oriented, as in the first two terminal sessions, to being planning oriented; the initiative has shifted from the user to the interface. This terminal session has all the features of an interactive planning session in which the indeterminacies in the user's statements of desired actions are resolved cooperatively by the interface and the user. The plans have served as partial definitions of plans and the previous terminal sessions have served as examples of how those plans are intended to be instantiated for the software system ABC.

4. USE OF KNOWLEDGE FOR SOPHISTICATED INFERENCING

In this section, we re-visit the topic of command understanding in order to consider in more depth some of the issues and potential solutions. Two important issues are detection of errors or ambiguities in commands together with associated corrective actions and the resolution of uncertainty in command interpretation to accommodate early decision making. An important class of these interface functions involves the use of inferencing heuristics based on the global knowledge available in the semantic database and in the partial instantiations of plans. We close the section by considering one additional function of the interface which is made possible by the overall approach.

4.1 Resolving Ambiguity in User Specification of Commands

The issues of errors in commands and ambiguities in commands are closely related. An error arises when the command cannot be "parsed" into the hierarchy of existing plan instantiations or when a LIKELY clause constraint fails to be met; the latter might be termed a "weak" error. Ambiguities arise when the command is incomplete through omission of parameter values.

4.1.1 Explicit Error Correction Knowledge One kind of error is made when the programmer gives a command which matches all the constraints of an error plan. These are errors which have been anticipated and explicitly pre-defined in the plans. For example, it is an error to attempt to edit an object module; this was handled in the sample plans by providing a special plan to catch any kind of editing on the wrong type of file. In order to deal with these types of errors, certain error-correction capabilities can be explicitly built into these error plans. For example, in the invoke_editor_error plan, we might add the following local reasoning:

> if file_kind is object_module, then infer that the file to be edited is the source_module from which that object module was built.

We could accommodate this by having ERROR_PLANS (as well as vanilla flavored PLANS) which have SUGGESTED_CORRECTIONS associated with them. Then the interface would behave as follows: if an error plan is identified, and if one of the corrective actions applies, then suggest the correction to the user, else report error to user.

4.1.2 Error Correction Through Inferencing In cases where the command and its arguments do not match an explicit error plan nor do they parse as they stand, certain inferences must be drawn to arrive at what the user actually meant. Inferencing is also necessary when the command is simply incomplete as the user stated it. Corrective possibilities include modifying the values

of certain parameters given or modifying the name of the command itself or both. We consider each such possibility below.

Consider the case of an input file name value which is missing or unidentifiable. It may be neither a known file nor a known concept. Suppose the user says "compile C" where C is unknown (this was shown in the second terminal session). There are two possibilities: either C was not meant at all or C is a new concept not previously recorded. (N.B. Let us assume for the moment that the command name is correct. Then, C cannot be a new file because compile requires an existing file.) In exploring these possibilities, the following approaches could be taken:

- what instantiated plans could a compile be associated with?
- what source modules exist which could be compiled?
- if other arguments to the compile command were given (such as the name of the library to be used) then the previous two answers could be restricted to those items consistent with the additional arguments.

The interface can easily supply values when output file names are missing; it need only pick some unique names not likely to be used by the user. Note that the user can always access the contents of these files by describing what is wanted ("listing for most recent compilation of program A", and so forth).

The only tricky part of inferring output file names comes when the user would have rewritten an existing file if he had supplied the name himself. This often happens because a programmer will think of a particular file as representing "the most recent version of such-and-such". Some possibilities are: if the last time this operation was performed an error occurred, then re-use the file (i.e., rewrite it). Or, if there is one such file associated with a revised object, then re-use; consider the case of a load module and an Ada library: if the programmer revises the library (as opposed to making a new library), then he would have rewritten over the old load module if he had given the complete command himself.

Suppose we assume that the file names are "right" but that the command name is in error. We might want to do this if the command failed to match any existing plan instantiation and was not itself a legal way to begin any new plan instantiation. This is the other way to disambiguate "compile C" where C is unknown. Here we could proceed in a way analogous to that for dealing with input file name resolution. Hopefully we would be led to concluding that what was meant was "vi C", that is a creation of C to be followed by its compilation.

4.2 Resolving Uncertainty in Command Interpretation

We have stressed the need for the interface to make interpretation decisions as early as possible in order to provide good error detection. In this section, we consider some ways in which the interface can arrive at reasonably accurate interpretation decisions in the face of partial information. There is an interplay between completing a single command and giving it an interpretation; parameter completion for example may depend upon an interpretation having been made, and a definitive interpretation may require the complete identification of parameters. Some rules are given which help to break this circularity.

4.2.1 Heuristics for Ordering Alternatives When the interface has identified several possible plan instantiations to which a particular command might belong, it is necessary to order the possibilities according to their likelihood, as has been mentioned previously. The most desirable ordering heuristics will be those which are domain independent and can be used with any particular set of plans. Taking the command "vi foo_8" at the end of the first terminal session as an example, we might argue as follows:

- prefer an existing plan instantiation over a new instantiation: thus, foo_8 is source for update_software_system_1 (an existing instantiation) in preference to source for update_software_system_2 (a new instantiation)

- prefer a new instantiation related to some existing instantiation over a new instantiation which has no relationship to current activities: thus, foo_8 is a test case for test_1 where test_1 will exercise the load module of update_software_system_1, in preference to foo_8 source for update_software_system_2.

Another example of a useful ordering rule, with slightly more complex reasoning, is:

- if two possibilities are at "the same level of abstraction in the plan hierarchy", then if there is a higher level plan containing them both and that plan imposes an ordering, prefer the plan which must precede the other in the time sequence.

Each of these heuristics is based upon assumptions about the way that people work: that there is coherency and connectivity in the actions being performed, and that actions are not random.

Although examples from our software development plans were used, there is nothing inherently plan or even domain dependent in these heuristics. Thus, they can be built into the interpretation system for use with any set of plans; they need not be specified anew for each application or environment.

A related plan recognition system using similar heuristics to make decisions in the face of partial information addresses the application of natural language dialog comprehension. Allen [27] has developed a system which uses plan recognition to analyze intention in dialogues so as to be able to generate question responses which give more information than explicitly requested or which are reasonable replies to indirect speech acts or to fragmentary speech acts. The system attempts to deduce the plan of the speaker and then formulates a response which assists in the satisfaction of the goal of the plan. Plan deduction is primarily a bottom-up process, although top-down reasoning is used when the resulting set of alternatives is not unreasonably large. The system makes use of domain-independent heuristics to rate the competing plan hypotheses, which must be pruned to the single most likely candidate in order to generate the response.

4.2.2 Context Knowledge We can define the current context as the path through the instantiated plan hierarchy to which the last user command belongs; of course, there may be several competing possibilities, in which case the previously stated rules for ordering these possibilities could be used. This path will consist of a chain of plan instantiations, from lowest to highest level of abstraction, representing the "understanding" of the command. The highest level plan instantiation (for the programming environment we have been discussing it would be do_programming) will never change. The lowest level plan instantiation (for our example it would be issue_command) will always change from one command to the next. If we consider some appropriate intermediate level, such as update_software_system, then we would expect context changes relatively infrequently, based upon our assumption that activities are not entirely random. Some empirical studies are necessary to verify that indeed such context changes are infrequent and that the number of contexts associated with a given terminal session is small.

This notion of a current context has considerable value as compared with the simple notion of context which has been used in other user interfaces. The concept which has been used before relies on a straightforward notion of "last". If arguments are omitted, for example, they are taken from the "last" occurrence of the same command; here "last" is defined by viewing the commands as a linear list which is searched in most-recent to least-recent order. We believe that this notion can be augmented by partitioning the commands, as the plans do, into related subsets and factoring this into the notion of context as suggested above. Although a more sophisticated notion of context obviously incurs a greater processing cost, we believe this cost

is balanced by the value gained. Two examples of the gain are as follows.

The current context can be used in performing inferencing to complete details in a command in an intelligent way. For example, one approach to selecting an instantiation of a concept is to pick the most recently created of all such instantiations. However, this is unnecessarily simple-minded. Suppose the programmer is working with two different Ada libraries, one representing the released version of the system in use by users and one representing the experimental development version. The sources associated with the experimental version are thus the most recently created instantiations of the programs. If the programmer has just been testing the released version to track down the behavior of some bug, a command to edit a particular program most likely refers to the instantiation which is part of the released version. The simple-minded resolution rule fails in this case, whereas the context related rule succeeds.

Another use of the current context is for ordering of alternative interpretations of commands. It leads to the following heuristic rule:

- prefer an interpretation which is connected to the current context over one which is not so connected.

This is a stronger form of the second heuristic ordering rule given above, which is useful if there are several interpretations which connect with previous activities; the new rule provides for selection of the interpretation which builds from the current context.

It is the case that in many of these inferencing situations, it will be helpful to make a distinction between guesses which are nearly certain and guesses with greater degrees of uncertainty. In the latter cases, the interface should probably query the user as to the accuracy of the interpretation it has decided to use. On the other hand, if the interface has to check with the programmer every time it has drawn some knowledgeable conclusion, the programmer will become rather annoyed. This is an area for experimental tuning.

4.2.3 Attention Focusing Knowledge There are plan instantiations in the programming environment which are of particular importance to the programmer and to the interface. A compile which is successful is not as interesting as one which fails; the failure indicates that there is a piece of work which remains to be done. In updating a software system, if new software modules are introduced or if new dependencies are introduced, that too is of more interest than the case where such things did not happen. There are several different ways in which such interesting events can be exploited by the interface.

To begin with, such interesting plans are candidates for special notes in the session summaries. They might also be of use in inferring when the programmer is entering a situation which is more than normally error-prone. Another use of noting such interesting events is that interpretation hypotheses which exploit these events are more applicable than hypotheses which do not. In the case of interesting events which represent failures, there might be some domain independent reasoning which applies. For example, in the event of a failure, the interface might post a prediction that the action would be retried, as well as some predictions about the actions necessary before the re-try occurs.

4.2.4 Using Knowledge for Other Actions There is a spectrum of initiative possibilities from all programmer initiated actions, through mixed programmer and interface initiative, to all interface initiated actions. We can include in the plan expressions certain plans which will not necessarily ever be invoked by the programmer, but in general will be executed by the interface in a manner largely transparent to the programmer.

One example is as follows. We might like to set up a programming environment in which the efficient management of the file space was handled automatically. Suppose it was desired to use a text management tool to store all the historical revisions to source modules as successive deltas, any one of which could be retrieved at any time. In that case, we don't want to have to make a submission to this tool every time a file is edited, because many such revisions are not of interest. However, we might say that a significant revision had been achieved if it was compiled, and in that plan we would want to connect the submission to our tool with the compilation action. Thus, one of the actions in the compile plan would be to execute the record new_version plan, if this was the first compile of this revision. This leads to the idea that programming objects can be retrieved from files directly or indirectly (via some invocation of a tool). The usual command languages of operating systems cannot handle this in a way transparent to the programmer.

4.3 Passing Database Knowledge Through to the User

We end this section with the simple observation that the information in the semantic database is of value not only to the intelligent interface, but also to the programmer. With this store of information, references to the components the programmer is working with need no longer be exclusively through the file names. Rather, the programmer can use descriptors such as "listing for most recent revision of A" to refer to foo.4. Further, the programmer can make direct queries on the knowledge base, such as "find all libraries into which A has been compiled". Such facilities appear to be extremely convenient, and provide a much closer match between the way that programmers would verbally describe what they are trying to do and the way that they must express that to the system. This can be achieved

using a suitable database query language, without going to the extreme of implementing an entire natural language dialog capability.

When the programmer makes a direct query on the database to retrieve the description of some file, the interface can choose to return a standardized description of the file. Alternatively, it can use context information from the plan interpretation data structures to return a description tailored to the related actions the programmer is currently engaged in. Thus, when asked to describe foo 1, the interface could respond with an unbiased report:

```
MOST RECENT REVISION OF PACKAGE_SPEC A
```

or it could respond with a report biased towards the fact that foo 1 is related to the recent "interesting" plan that the compile of foo_2 failed:

```
REVISION OF PACKAGE_SPEC A USED IN THE
COMPILATION OF SUBPROGRAM B WHICH HAD
COMPILE ERRORS.
```

This could prove an extremely useful capability since it can be well-tuned to the on-going activities using the command interpretation data structures.

5. STATUS

We have explored in some detail the nature and functionality of an intelligent interface and demonstrated by way of our examples its applicability and usefulness in the programming environment. We are currently involved in studying actual terminal session transcripts from programmers [28] to assist us in the process of identifying the domain knowledge which is necessary and useful for achieving our intelligent interface. This will complete the validation of our assumptions about the structure of programming activities and the expertise that is involved. Although only a relatively few plans have been developed for this present paper, we feel that there is considerable unexplored depth to the problem, both in terms of further plans covering the full range of programming activities and in terms of further knowledge which is available from tools and can be exploited by the interface.

A variety of issues remain undeveloped so far. We have not dealt with the important practical issue of the acquisition of plan definitions and support systems which might make this rather complex process straightforward. We wish to explore further the issue of the types and uses of domain independent knowledge. Another issue is the integration of our interface with natural language generation facilities or graphical communication facilities.

We are just beginning to turn our attention to the important issue of an overall system architecture for the interface, in which the applicable knowledge can be effectively utilized. It is our goal to make a clear distinction between domain specific knowledge (expressed in the plans) and domain independent knowledge which drives the interpretation and planning, by operating on the domain specific knowledge. Further, we wish to construct an architecture which will lend itself to further experimentation with the design of intelligent interfaces. The architecture must accommodate the following types of components:

* interpretation algorithms
* database maintenance, retrieval, and reasoning facilities
* heuristics for resolving uncertainty
* prediction algorithms to instantiate representations of actions expected in the future
* forward propagation algorithms to propagate known parameters through a plan hierarchy to bind future actions.

Our approach to the development of this architecture will draw heavily upon AI techniques in order to achieve non-deterministic interpretation, keep multiple hypotheses alive for analysis, use probabilistic information for making choices among competing

hypotheses, proceed effectively with only partial information, simulate the implication of selected choices, and use an opportunistic control strategy to decide where to focus attention. One general purpose architecture which we are currently investigating is the Hearsay-III architecture [29].

6. SUMMARY

In this paper we have defined an intelligent interface which offers significant leverage on the problem of human productivity in a computer-based work environment. This interface operates at a conceptual level which is closer to that of the computer user that that of the operating system. This conceptual level captures the extra knowledge that users informally maintain in their minds and draw upon as they manage their work. In current systems, this knowledge is not a shared domain of discourse between the user and the system.

We have shown how the interface can be equipped with explicit domain expertise in the form of plans for the various programming activities. These plans are hierarchically related, so that the activities can be viewed from multiple perspectives of varying detail. We believe that we have provided sufficient justification for the existence of such a plan structure, organized on the principle of interleaving of multiple tasks, in the programming environment. In conjunction with these plans, a semantic database has been described which captures all the relevant information necessary to model the state of the user's world.

Instantiations of the plan and database schemata are the focus of the interpretation and planning activity of the intelligent interface. We have shown how interpretation occurs as instances of the plans are perceived in the user input command stream. By operating in a non-invasive manner, performing inferencing automatically in preference to requiring explicit direction from the programmer, the interface preserves the flexibility which users have in current systems while simultaneously providing new levels of support.

Using this structure, we have shown how the intelligent interface can:

* detect actual and potential errors, notably errors of a more fundamental nature that locally-determined illegal command syntax

* perform error recovery, over and above error detection, to correct errors automatically or query the user for information which will lead to an error correction

* create and manage agendas of work yet to be performed, by prediction of future actions based upon past actions and information in the semantic database

* summarize the accomplishments of terminal sessions, partitioning the activities into related groups as defined by the plans, and using a higher level of abstraction than merely repeating the commands verbatim

* automatically complete certain plans (planning and plan execution), perhaps with the assistance from the user (interactive planning)

* shift from modes in which the user has the initiative to modes in which the interface assumes control for performing work

7. ACKNOWLEDGEMENTS

This research was initiated in collaboration with our colleagues Bruce Croft and Larry Lefkowitz, who contributed to the development of the framework described here and in [30] for interpretation and planning. They are currently pursuing the application of this framework to office automation [31]. Computer support for the preparation of this report was provided by Intermetrics, Inc.

8. REFERENCES

[1] Lingard, R.W., "A Software Methodology for Building Interactive Tools", Proceedings of the Fifth International Conference on Software Engineering, ACM and IEEE (March, 1981) pgs. 394-399.

[2] Wilczynski, D., "Knowledge Acquisition in the Consul System", IJCAI-7, (August, 1981) pgs. 135 - 140.

[3] Mark, W., "Representation and Inference in the Consul System", IJCAI-7, (August, 1981) pgs. 375-381.

[4] Ball, E. and Hayes, P., "Representation of Task-Specific Knowledge in a Gracefully Interacting User Interface," Proceedings AAAI National Conference, (August, 1980) pgs. 116-120.

[5] Bates, P., Wileden, J., and Lesser, V., "EDL: A Language to Support Debugging in Distributed Systems," COINS Technical Report, University of Massachusetts at Amherst, (Spring, 1981).

[6] Haberman, A.N., "An Overview of the Gandalf Project", 1978-79 CSD Research Review, Carnegie-Mellon University, Pittsburgh, PA. (1979).

[7] Tichy, W., "Software Development Control Based On Module Interconnection", Proceedings of the Fourth International Conference on Software Engineering, ACM and IEEE (1979) pgs. 29-41.

[8] Feiler, P. and Medina-Mora, R., "An Incremental Programming Environment", Proceedings of the Fifth International Conference on Software Engineering, ACM and IEEE (1981) pgs. 44-53.

[9] "Reference Manual for the Ada Programming Language: Proposed Standard Document", U.S. Department of Defense, (July, 1980).

[10] Shaw, A.C., "Software Descriptions with Flow Expressions", IEEE Transactions on Software Engineering, Vol. SE-4, No. 3 (May, 1978) pgs. 243-254.

[11] Riddle, W., "An Approach to Software System Behavioral Specification", Journal of Computer Languages, Vol. 4, No. 1 (1979) pgs. 22-47.

[12] Gischer, J., "Shuffle Languages, Petri Nets, and Context-Sensitive Grammars", Communications of the ACM, Vol. 24, No. 9 (September, 1981) pgs. 597-605.

[13] Zisman, M.D., Representation, Specification and Automation of Office Procedures, Ph.D. Dissertation, Wharton School, University of Pennsylvania (1977).

[14] Chen, P.P., "The Entity Relationship Model: Toward a Unified View of Data", ACM Transactions on Database Systems, Vol. 1, No. 1 (March, 1976) pgs. 9 - 36.

[15] Moriconi, M., "A Designer/Verifier's Assistant", IEEE Transactions on Software Engineering, Vol. SE-4, No. 4 (July, 1979) pgs. 387-401.

[16] Buxton, J.N., Requirements for Ada Programming Support Environments (Stoneman), Department of Defense (February, 1980).

[17] Osterweil, L., "Software Environment Research: Directions for the Next Five Years," IEEE Computer Magazine, Vol. 14, No. 4 (April, 1981) pgs. 35-43.

[18] Huff, K.E., "A Database Model for Effective Configuration Management in the Programming Environment", Proceedings of the Fifth International Conference on Software Engineering, ACM and IEEE (March, 1981) pgs. 54-62.

[19] Parnas, D.L., "A Technique for Software Module Specification with Examples", Communications of the ACM, Vol. 15, No. 5 (May, 1972) pgs. 330 - 336.

[20] Levitt, K., Robinson, L., and Silverberg, B., The HDM Handbook, Volumes 1-3, SRI International, Menlo Park, California (June, 1979).

[21] Genesereth, M.R., "The Role of Plans in Automated Consultation", IJCAI-6, Vol. 1 (1979) pgs. 311-319.

[22] Rich, C. and Shrobe, H., "Initial Report on a LISP Programmer's Apprentice", IEEE Transactions on Software Engineering, Vol. SE-4, No. 6 (November, 1978) pgs. 456-466.

[23] Rich, C., "A Formal Representation for Plans in the Programmer's Apprentice", IJCAI-7, (August, 1981) pgs. 1044-1052.

[24] Waters, Richard, "The Programmers Apprentice: Knowledge Based Program Editing", IEEE Transactions on Software Engineering, Vol. SE-8, No. 1 (January, 1982) pgs. 1-12.

[25] Brotsky, D., "Program Understanding Through Cliche Recognition", Master's Thesis, Massachusetts Institute of Technology, in preparation.

[26] Sridharan, N., and Schmidt, C., "Knowledge-Directed Inference in BELIEVER", in Pattern-Directed Inference Systems, Waterman and Hayes-Roth, eds, Academic Press (1978) pgs. 361-379.

[27] Allen, J. A Plan-Based Approach to Speech Act Recognition, Ph.D. Thesis, Department of Computer Science, Univeristy of Toronto (1979).

[28] Huff, K., "Analysis of Programmer Terminal Sessions", COINS Technical Report, University of Massachusetts at Amherst, forthcoming.

[29] Balzer, R., Erman, L., London, P., and Williams, C., "HEARSAY-III: A Domain Independent Framework for Expert Systems," Proceedings AAAI National Conference, (August, 1980) pgs. 108-110.

[30] Croft, W.B., Huff, K.E., Lefkowitz, L. and Lesser, V.R., "Design of Intelligent Interfaces," COINS Technical Report, University of Massachusetts at Amherst, forthcoming.

[31] Croft, W.B., and Lefkowitz, L., "An Office Procedure Formalism Used for an Intelligent Interface," COINS Technical Report 82-4, University of Massachusetts at Amherst, (Spring, 1982).

1. APPENDIX A: Plan Definitions

1.1 Plan Issue Command

```
PRIMITIVE PLAN issue command IS

     BUILTIN

WITH

     program name IS STRING;
     || name of program being executed

     in file IS ARRAY OF db_ref;
     || a sequence of input files, from 'FIRST to 'LAST.
     || a file is represented by a database reference,
     || which takes the form of the string name given in
     || the command.

     out_file IS ARRAY OF db_ref;
     || a sequence of ouput files, from 'FIRST to 'LAST,
     || in the same form as for in_file.

     parameters IS STRING;
     || the parameters passed to the program being executed.
     || the use of parameters is not explored in these examples.
     || the form of this value in a full-scale example would be a
     || variant record, varying with the value of program_name.

     return_code IS (ok, user_errors, abnorm_termination);
     || code revealing state at completion of program execution.
     ||  the case of abnormal termination is
     ||  ignored in the following plans.

     execution_info IS RECORD
          WHEN program_name INSET {"vi","rand"}:
               modification_flag IS (change, no_change);
          WHEN program_name = "ada":
               unit_name IS STRING,
               unit_type IS (package_spec, package_body,
                             subprogram_spec, subprogram_body,
                             task_spec, task body),
                unit_level IS (lib_unit, sub unit);
          WHEN program name = "link":
               main_pgm_name IS STRING;
     ENDRECORD;
     || information passed back from program, if any.

END PLAN;
```

1.2 Plan Invoke editor

```
PLAN invoke_editor IS

     issue_command

COND

     program_name INSET {"vi", "rand"};
     || two editors, vi and rand, are assumed to be available

     (NOT exists (in_file[1])) OR (kind(in_file[1]) INSET
          {source, testcase, doc, memo, mail,
          unknown_text_type});
          || if an existing file is being edited, then it must be
          || of an editable kind

     modification_flag = change;
     || actual changes must have been made to the input file
     || (see also the plan for browse)

WITH

     successful := (return_code = ok);

     file_created := (NOT exists (in_file[1]));

     file_updated := NOT file_created;

     new_file := NEW_VALUE'get_contents (out_file [1]);

     old_file := NEW_VALUE'get_contents (in_file [1]);

EFFECTS

     IF successful THEN
          EFFECTS_OF'create_file
               (out_file[1], unknown_text_type);
               || enter the file and its contents in the
               || database, along with their attributes.
               || kind will be updated when a defining use
               || of the file is recognized at a later time
               || via another plan.
          IF file_updated THEN
               NEW_VALUE'successor
               (in_file[1], out_file[1]) = TRUE;
               || describe new_file in database as the successor
               || historically to old_file
          ENDIF;
     ENDIF;
END PLAN;
```

1.3 Plan Edit command error

```
ERROR_PLAN invoke_editor error IS

     issue_command

COND

     program_name INSET {"vi", "rand"};

     exists (in_file[1]);

     kind(in_file[1]) NOT INSET {source, doc, mail, memo,
          testcase, testresult, listing, unknown_text_type};
          || the conjunction of these three conditions
          || implies that the programmer has asked to
          || edit a file which is not of editable kind,
          || such as a load module or object module.

END PLAN;
```

1.4 Plan Edit

```
PLAN edit IS

     invoke_editor+
     || one or more instances of plan invoke_editor

COND

     FORALL i FROM invoke_editor'FIRST + 1
          TO invoke_editor'LAST:
          invoke_editor[i].old_file =
               invoke_editor[i-1].new_file;
               || to be part of the same plan instantiation, all
               || edits must form a chain such that the input to the
               || next edit is the output from the last edit.

     FORALL i FROM invoke_editor'FIRST
          TO invoke_editor'LAST:
          invoke_editor[i].successful = TRUE;
          || we choose to ignore unsuccessful edits

WITH

     old_file := old_file [1];

     new_file := new_file [last];

END PLAN;
```

1.5 Plan Browse

```
PLAN browse IS

     issue_command

COND

     program_name INSET {"vi", "rand"};
     || the program invoked must be an editor

     exists (in_file[1]);
     || an existing file is the input file

     kind(in_file[1]) INSET {source, doc, mail, memo,
          testcase, listing, unknown_text_type,
          testresults};
          || file must be of editable kind

     modification_flag = no_change;
     || the input file is not modified in any way

WITH

     file_perused := get_contents (in file[1]);

END PLAN;
```

1.6 Plan Print

```
PLAN print IS

     issue_command

COND

     program_name = "lp";
     || the program being executed is lp,
     || the line printer queuer.

     exists( in file[1] );
     || printing can only be performed on existing files.

     kind(in file[1]) INSET {source, testcase, doc, memo, mail,
          listing, testresult, unknown text_type};
          || the file must be a printable kind

WITH

     file_printed := get_contents (in_file[1]);

END PLAN;
```

1.7 Plan Check results

```
PLAN check_results IS

     (browse | print)+

     || checking results is any number of
     || browses and/or prints in any order

COND

     THEREEXISTS x: db ref SUCHTHAT
          kind(x) INSET {listing, testresults}
          AND
          FORALL i FROM browse'FIRST
               TO browse'LAST:
               assoc_name (x) = assoc_name (browse[i].file_perused)
          AND
          FORALL i FROM print'FIRST
               TO print'LAST:
               assoc_name (x) =
                    assoc_name (print[i].file_printed);
               || checking takes place on listings (e.g. after
               || compilation) or on test results (e.g. after
               || testing)
               || further, all activity in a single instantiation
               || of check_results is to a single such file.

WITH

     hard_copy := (THEREEXISTS i IN RANGE <PLAN'FIRST .. PLAN'LAST>
                         SUCHTHAT PLAN[i] = print);
                         || hardcopy was produced if there is at
                         || least one instance of print in this
                         || instantiation.

     file_checked := IF hard_copy = TRUE THEN
                              print[1].file_printed
                         ELSE
                              browse[1].file_perused;

END PLAN;
```

1.8 Plan Compile

```
PLAN compile IS

     issue_command

COND

     program_name := "ada";
     || consider an environment in which programming is done
     || in Ada.

     exists (in_file[1]);
     || compiling can only be performed on existing files.

     kind(in_file[1]) INSET {source, unknown text_type};
     || first input file must be a file containing source
     || or this is a defining usage of the file kind as source.

     (NOT exists (in_file[2])) OR (kind(in_file[2]) = library);
     || if the second input file does not exist, then an Ada
     || library will be created and initialized from this
     || compilation. otherwise, the second input file must
     || be an Ada library.

LIKELY

     IF part_of (in_file[1], in_file[2]) THEN
          THEREEXISTS f: db_ref SUCHTHAT
               dependent (f, in_file[1])
               AND
               NOT part_of (f, in_file[2]);
               || if we compile the exact same source back into
               || some ada library, then some part of that ada
               || library which affects the current source must
               || have been changed; if this condition is not
               || met, then the compilation amounts to a no-op.

     system(in_file[1]) = system(in_file[2]);
     || we expect that the source module belongs to the same
     || system that the library represents.  this condition
     || would be violated in retrofitting a module from
     || one system to another, as happens (all too) infrequently.

WITH

     successful := (return_code = ok);

     compiled_file := NEW_VALUE'get_contents (in_file[1]);

     updated_library := NEW_VALUE'get_contents (in_file[2]);
```

```
source_listing := NEW_VALUE'get_contents (out_file[1]);

object_module := NEW_VALUE'get_contents (out_file[2]);

abstract_unit := NEW VALUE'make db_ref
                 ( unit_name CONCAT unit_type,
                 abstract_source_unit);

EFFECTS

     IF successful THEN

          IF NOT exists (in_file[2]) THEN
               EFFECTS_OF'create_file
               (in file[2], library);
                    || enter descriptor into database for newly
                    || created Ada library.
          ENDIF;

          EFFECTS_OF'create_file
          (out_file[2], object);
               || enter descriptor into database for the newly
               || created object module.

          NEW VALUE'part_of
               (out_file[2], in_file[2]) = TRUE;
               || show the relationship between the object module
               || and the library to which it belongs.

          FORALL i FROM 3 TO in_file'LAST:
               NEW_VALUE'dependent
               ( in_file[i], out_file[2] ) = TRUE;
               || this new object module is dependent on
               || any sources the compiler had to "touch"
               || in order to perform the compilation.


          IF NOT exists (make_db_ref ( unit_type CAT unit_name,
               concept space)) THEN
                    EFFECTS_OF'create_concept
                    (unit type CAT unit_name,
                    abstract source unit);
                    || if this is the first occurrence of this
                    || abstract source unit, then create a concept
                    || to represent it.
          ENDIF;

          NEW VALUE'instance of
               ( make db ref ( unit_type CAT unit_name,
               concept_space), in_file[1]) = TRUE;
               || record the relationship between the compiled file
               || and abstract concept it represents.
```

```
    ENDIF;

    EFFECTS_OF'create_file
        (out_file[1], listing);
        || enter description for source_listing in
        || database.

    NEW_VALUE'listing for
        (out_file[1], in_file[1]) = TRUE;
        || show relationship between listing and source.

    IF kind(in_file[1]) = unknown text_type THEN
        NEW_VALUE'kind (in_file[1]) = source;
        || if file input as the source file was of unknown text
        || type, infer that it is of type source by its use in
        || a compilation context.

    ENDIF;

END PLAN;
```

1.9 Plan Update source unit

```
PLAN update_source_unit IS

    ((edit compile check_results) |
     (edit compile) |
     (compile check_results) |
     (compile))+

     || Repeated instances of compile with optional preceeding
     || edit and optional succeeding check results.

COND

     FORALL i FROM edit'FIRST
          TO edit'LAST:
          kind(edit[i].new file) INSET {source, unknown_text_type};
          || only edits of source files or potential source files
          || are considered part of this plan.

     FORALL i FROM edit'FIRST
          TO edit'LAST:
          THEREEXISTS j IN RANGE <compile'FIRST .. compile'LAST>
          SUCHTHAT

               AFTER (compile[j],edit[i]) AND
               edit.new file[i] = compile.compiled_file[j];
               || if the editing of a particular file is part of
               || this plan, then there should be a later
               || compile of that same file.

     FORALL i FROM check_results'FIRST
          TO check results'LAST:
          THEREEXISTS j IN_RANGE <compile'FIRST .. compile'LAST>
          SUCHTHAT
               AFTER (check_results[i], compile[j]) AND
               check_results[i].file_checked =
                    compile[j].source_listing;
                    || checking takes place on source listings
                    || which are created in this plan
                    || instantiation.

     FORALL i FROM compile'FIRST +1
          TO compile'LAST:
          same_source_unit( compile[i].compiled_file,
                            compile[1].compiled_file )

          AND
          compile[i].updated_library = compile[1].updated_library;
          || all compile activity is on the same source unit and
          || with respect to the same ada library.
```

```
     FORALL i FROM compile'FIRST
         TO compile'LAST:

         IF compile[i].successful = false THEN
              THEREEXISTS j IN_RANGE <check results'FIRST ..
                   check results'LAST> SUCHTHAT
                        AFTER (check results[j], compile[i]) AND
                        check results[j] =
                        compile[i].source listing;
                        || if the compile fails, then we require
                        || that the programmer look at the source
                        || listing to identify the errors,
                        || before continuing to edit or
                        || compile.

LIKELY

     compile[LAST].successful = TRUE;
     || the only circumstances in which this would fail to
     || occur would be if the programmer decided to back off
     || and not use the newly created source module at all.

WITH

     old_source := edit[1].old_file;

     new source := compile[i].compiled file SUCHTHAT
                   i = MAX({j | j>=compile'FIRST
                         AND j<= compile'LAST
                         AND compile[j].successful=TRUE});
                         || the source used in the last
                         || successful compile in this plan
                         || instantiation.

     updated_library := compile[1].updated_library;

     aborted := NOT compile[LAST].successful;

     changes made := THEREEXISTS j IN RANGE <compile'FIRST ..
                         compile'LAST> SUCHTHAT
                         compile[j].successful = TRUE;
                         || at least one compile was successful
                         || or no updating occurred to the library.

END PLAN;
```

1.10 Plan Link

```
PLAN link IS

     issue_command

COND

     program_name = "link";
     || the program being executed is the linker

     exists (in_file[1] );
     || linking is done on existing (library) files.

     kind(in_file[1]) = library;
     || linking is done on Ada libraries.

WITH

     successful := ( return_code=ok );
     || we need to know if there were any unsatisfied references,
     || or other such failures.

     auto_recompile := { NEW VALUE'get contents (in file[2])
                         .. NEW_VALUE'get_contents (in_file[LAST]) };
     || certain modules may have been automatically
     || re-compiled because modules which they
     || depend upon have changed.

     linked_library := NEW VALUE'get contents (in file[1]);

     executable_file := NEW_VALUE'get_contents (out_file[1]);

     abstract_sys := NEW_VALUE'make_db ref
                         (main_program_name, abstract_system);

EFFECTS

     IF successful THEN
          EFFECTS_OF'create_file
          (out file[1], executable);
               || create description of executable file

          NEW_VALUE'part_of
               (in_file[1], out file[1]) = TRUE;
               || create relationship between library and
               || executable files.

          IF NOT exists ( make_db_ref (main_pgm_name,
               concept_space)) THEN
                    EFFECTS_OF'create_concept
```

```
                    ( main_pgm name, abstract_system);
                    || record the main program name as the name of
                    || the software system.

          NEW_VALUE'instance of
               (in_file[1], make_db_ref(main_pgm_name,
               concept_space) = TRUE;
               || record this ada library as an instantiation of
               || this abstract system.

     ENDIF;

END PLAN;
```

1.11 Plan Update software system

```
PLAN update_software_system IS

     ( update_source_unit@ link)+
     || any number of update source_units interleaved, then
     || followed by a link, the whole repeated one or
     || more times.

COND

     FORALL i FROM update_source_unit'FIRST
          TO update_source_unit'LAST:

               update_source_unit[i].updated_library =
                    link[1].linked_library;
                    || to be part of the same plan instance, all
                    || activity must be to the same Ada library.

     FORALL i FROM link'FIRST + 1
         TO link'LAST:
              link[i].linked_library = link[1].linked_library;
              || all links are to the same library.

     FORALL i FROM link'FIRST
          TO link'LAST -1:
          link[i].successful = false
     AND
          link[LAST].successful = TRUE;
          || this constitutes a termination condition for this
          || plan.


WITH

     incarnation_created := link[LAST].executable_file

     updated_library := link[1].linked_library;

     modules_changed := { m: db_ref | THEREEXISTS i IN RANGE
          < update_source_unit'FIRST .. update_source_unit'LAST >
          SUCHTHAT m = update_source_unit[i].new_source
          AND update_source_unit[i].changes_made = TRUE };
          || the set of modules changed is the set of all
          || modules successfully updated in the library
          || during this plan.

END PLAN;
```

1.12 Plan Run

```
PLAN run IS

     issue command

COND

     THEREEXISTS x: db_ref SUCHTHAT
          kind(x) = executable
          AND assoc_name (x) = program_name;
          || the program being executed is some load
          || module previously created by the programmer.

     exists (in_file[1] );
     || the test input must be an existing file.

     kind(in_file[1]) INSET {testcase, unknown_text_type};
     || the input must be of kind testcase.

WITH

     test_input := NEW_VALUE'get_contents (in_file[1]);

     test_result := NEW_VALUE'get_contents (out_file[1]);
     || we make the simplifying assumption that
     || testing always involves one input and one
     || output file.

     incarnation_tested := make_db_ref
                           (program_name, file);

EFFECTS

     IF kind(in_file[1]) = unknown_text_type THEN
          NEW_VALUE'kind (in_file[1]) = testcase;
          || this is a defining occurrence of the input file.
     ENDIF;

     EFFECTS_OF'create_file
          (out file[1], testresult);

     NEW VALUE'tested
          (make db_ref (program_name, file_space),
          in_file[1], out file[1]) = TRUE;
          || record the testing plan via the tested
          || relationship.

END PLAN;
```

1.13 Plan Test

```
PLAN test IS

     ((edit run | run) check_results)@

     || testing may optionally begin with creation of
     || or change to a test case, followed by running of the
     || desired program, followed by some kind of examination
     || of the test results.
     || the whole may be repeated, in an interleaved fashion

COND

     FORALL i FROM edit'FIRST TO edit'LAST:
          kind(edit[i].new_file) INSET {testcase, unknown test_type};
          || only edits of test cases or potential test cases are
          || considered part of this plan.

     FORALL i FROM edit'FIRST TO edit'LAST:
          THEREEXISTS j IN RANGE <run'FIRST .. run'LAST> SUCHTHAT
          AFTER (run[j], edit[i]) AND
          edit[i].new file = run[j].test_input;
          || the edit belongs to this plan if what is edited
          || is the data input to the some later run in this plan.

     FORALL i FROM check_results'FIRST
          TO check_results'LAST:
          THEREEXISTS j IN RANGE <run'FIRST .. run'LAST>
               SUCHTHAT
               AFTER (check_results[i], run[j]) AND
               check_results[i].file_checked = run[j].test_results;
               || if the checking belongs to this plan, then
               || the file checked is the outcome of an earlier
               || test run in this plan.

     FORALL i FROM run'FIRST +1
          TO run'LAST:
               run[i].incarnation_tested =
                    run[1].incarnation_tested;
                    || to be part of the same plan instantiation,
                    || all runs must be on the same executable
                    || module.
WITH

     incarnation_tested :=  run[1].incarnation_tested;

     new_test_cases := {t: db_ref | THEREEXISTS i IN RANGE
                          < edit'FIRST .. edit'LAST > SUCHTHAT
                           t = edit[i].new file};

END PLAN;
```

1.14 Plan Do programming

```
PLAN do_programming IS

     ( update_software_system $ test )

COND

     update_software_system.incarnation_created =
          test.incarnation_tested;
          || we require that the system updated be the
          || same as the system tested for the two
          || activities to be part of the same plan
          || instantiation.

WITH

     subject_of_work := update_software system.incarnation_created;

END PLAN;
```

1.15 Plan Make errors

```
PLAN make_errors IS

     ( invoke_editor_error )+

COND

     || all instances of invoke_editor_error belong to this
     || plan, which is essentially a non-terminating
     || catch-all for all error activity.

WITH

     || there are no attributes of interest to the higher
     || level plan.

END PLAN;
```

1.16 Plan Programming work

```
PLAN programming_work IS

     (do_programming | do documentation | make_errors)@

     || interleaving of any number of instances of
     || do_programming and/or do_documentation and/or
     || make_errors.
     || if this example were even more ambitious than it is,
     || then there would be even other alternatives participating
     || in the interleaving.

COND

     || all instances of these plans are unconditionally part of
     || this plan, which captures all on-going activity.

WITH

     || this is the highest level plan explored here.
     || in a multi-person environment, this plan might
     || have WITH attributes describing which programmer
     || was performing the plans.

END PLAN;
```

1. APPENDIX B: Semantic Data Base Schema

1.1 Kernel Schema

```
MODULE primitive_database

     EXPORTS (file access, contents_access, concept access,
              contents_kind, concept_kind,
              prim instance of, prim stored in, prim successor,
              prim listing for, prim_part_of, prim_dependent on,
              prim_object_of, prim_tested, prim same source_unit,
              prim same system);
```

This specification module defines the behavior of the primitive level database, in terms of entities and relationships between those entities. Certain of these definitions are made visible to other modules, via the EXPORTS construct; in particular, these definitions will be imported by the module "semantic database".

An entity declaration takes the form:

```
ENTITY e
    ATTRIBUTE a1 IS t1,
    ....
    ATTRIBUTE an IS tn;
 END;
```

This should be considered derived syntax for:

```
TYPE e IS RECORD
     a1 IS t1 (INITIALLY UNDEFINED),
     ...
     an IS tn (INITIALLY UNDEFINED),
ENDRECORD;

TYPE e access IS ACCESS (e);
```

Further, a built-in function NEW can be used to allocate new instances of type e, returning a value of type e_access which points to the new instance.

```
ENTITY file

     ATTRIBUTE name IS STRING
     || plus other attributes as necessary

END;

TYPE contents_kind IS (source, listing, testcase, testresult,
                       object, executable, library, doc, mail, memo,
                       unknown text type);

ENTITY contents

     ATTRIBUTE kind IS contents kind,
     ATTRIBUTE create time IS TIME STAMP,
     || plus other attributes as necessary

END;

TYPE concept_kind IS (abstract_source unit, abstract system);

ENTITY concept

     ATTRIBUTE name IS STRING,
     ATTRIBUTE kind IS concept kind;
     || plus other attributes as necessary

ENDRECORD;
```

The form of a relationship declaration is:

```
RELATIONSHIP r1 ( arg1: e1, ... argn: en);
    ASSERTIONS
        asrt1; ... asrtm;
END;
```

This is derived syntax for:

```
FUNCTION r1 ( arg1: e1 access, ...
              argn: en access) RETURNS BOOLEAN;

    IF arg1 = NULL OR arg2 = NULL OR ...
       argn =  NULL THEN
          RETURN (FALSE);
    ELSEIF NOT (asrt1' AND asrt2' AND ... asrtm')
          THEN RETURN (FALSE);  ENDIF;
    ELSE  RETURN (INITIALLY FALSE);
    ENDIF;

 END;
```

|| where asrt1', ... asrtn' are derived from asrt1, ... asrtn by systematic replacement of dot selection with pointer dereferences.

A statement of the form: RELATION IS (1:1) is derived syntax for the following global assertion:

```
IF r1 (x1, x2) AND r1 (x1,x3) THEN
     x2 = x3
AND IF r1 (x1, x2) AND r1 (x3, x2) THEN
     x1 = x3;
```

Similarly, the meaning of RELATION IS (1:N) is:

```
IF r1 (x1, x2) AND r1 (x3, x2) THEN
     x1 = x3;
```

RELATION IS (N:M) leads to no global assertions, as there are no restrictions on the instances of the relationship.

```
RELATIONSHIP prim instance of (c1:contents, c2:concept)
     || The basic relationship between contents and concepts.

     ASSERTIONS

          (c1.kind = source AND c2.kind = abstract source unit)
          || sources are instances of abstract sources
          OR
          (c1.kind = library AND c2.kind = abstract_system);
          || libraries are instances of abstract systems.

          RELATION IS (N:1);
          || For each content, there is a unique associated
          || concept.  However, concepts can have multiple
          || instantiations.
END;


RELATIONSHIP prim stored in (c:contents, f:file)
     || The basic relationship between files and contents,
     || which holds between a file and any kind of contents.

     ASSERTIONS

          RELATION IS (1:N);
          || For each file, there is a unique content, but not
          || vice-versa.
END;


RELATIONSHIP prim successor (c1:contents, c2:contents)
     || The successor relationship, established by editing actions,
     || holds between two contents entities.

     ASSERTIONS

          c1.kind INSET {source, doc, testcase, mail, memo,
                         unknown text type};

          c2.kind INSET {source, doc, testcase, mail, memo,
                         unknown text_type};
          || Successor is only defined for editable kinds
          || of entities.

          RELATION IS (N:1);
          || An entity cannot have two predecessors.
END;
```

```
RELATIONSHIP prim listing_for (c1:contents, c2:contents)
     || A relationship established by compilation.

     ASSERTIONS

          c1.kind = source  AND c2.kind = listing;
          || the listing for relationship is defined between
          || sources and compilation listings.

          RELATION IS (1:N);
          || A listing is associated with a single source.
END;


RELATIONSHIP prim part of (c1:contents, c2:contents);
     || A contents entity which is of kind library is an
     || aggregate, made up of one or more object modules
     || and at most one executable module.
     || The part of relationship represents this aggregation.

     ASSERTIONS

          c2.kind = library;
          (c1.kind = object AND RELATION IS (N:M))
          || an object module can belong to several
          || libraries and a library can have several
          || object modules in it.
          OR
          (c1.kind = executable AND RELATION IS (1:N));
          || the relationship of executable module to library
          || is a 1:N relationship.

END;


RELATIONSHIP prim_dependent_on (c1:contents, c2:contents)
     || This relationship is used to capture the inter-
     || dependencies between separately compiled units,
     || from information provided by the compiler.

     ASSERTIONS

          c1.kind = object AND c2.kind = source;
          RELATION IS (N:M);
          || dependencies are recorded between object modules
          || and (secondary) sources which were used in the
          || construction of the object module.
END;
```

```
RELATIONSHIP prim object of (c1:contents, c2:contents);
     || This relationship derives from a compilation action.

     ASSERTIONS

          c1.kind = object AND c2.kind = source;
          RELATION IS (N:1);
          || each object module is related to a single
          || (primary) source module.

END;


RELATIONSHIP prim tested (c1:contents, c2:contents, c3:contents);
     || This relationship records testing activity.

     ASSERTIONS
          c1.kind = executable;
          c2.kind = testcase;
          c3.kind = testresult;
          || Testing actions involve an executable module, a
          || testcase as input and a testresult as output.

END;

IF tested (x1, x2, x3) AND tested (x4, x2, x3) THEN x1 = x4;

IF tested (x1, x2, x3) AND tested (x1, x4, x3) THEN x2 = x4;
     || These two global assertions express the fact that for
     || each testresult, there is a unique testcase/executable
     || pair to which it is related by the tested relation.


|| The next three relations are all derivable from the
|| previous set of relations; for each of the next three
|| relations, therefore, a derivation is given in terms of
|| the previously defined relations.


RELATIONSHIP prim part of (c1: contents, c2: contents);
     || This relationship records more of the aggregate
     || nature of Ada libraries.  Note that the function
     || associated with this relationship is now overloaded
     || three ways: for sources and libraries, for object
     || modules and libraries, and for executable modules
     || and libraries.
     || Note also that the relationship between sources and
     || libraries can be derived using two other relationships
     || and thus a derivation is provided.
```

```
      ASSERTIONS
           c1.kind = source;
           c2.kind = library;
           RELATION IS (N:M);

      DERIVATION
           RETURN (
           THEREEXISTS c3: contents SUCHTHAT
                c3.kind = object AND
                part_of (c3, c2) AND
                object_for (c3, c1) );
END;


RELATIONSHIP prim_same source unit (c1: contents, c2: contents);
      || This relationship holds when two source units are both
      || instances of the same (abstract) concept.  Note that the
      || value of this relationship is derived from other
      || relationships.

      ASSERTIONS
           c1.kind = source AND c2.kind = source;

      DERIVATION
           RETURN (
           THEREEXISTS c3: concept SUCHTHAT
                instance of (c1, c3) AND
                instance of (c2, c3) );
END;


RELATIONSHIP prim same system (c1: contents, c2: contents);
      || This relationship is also derived using other
      || relationships, and it represents the case of two
      || source units which belong to the same (abstract) system.

      ASSERTIONS
           c1.kind = source AND c2.kind = source;

      DERIVATION
           RETURN (
           THEREEXISTS c3, c4: contents SUCHTHAT
                c3.kind = library AND
                c4.kind = library AND
                part_of (c1, c3) AND
                part_of (c2, c4) AND
                THEREEXISTS c5: concept SUCHTHAT
                     c5.kind = abstract_system AND
                     instance_of (c3, c5) AND
                     instance of (c4, c5) );
END;
```

1.2 Virtual Database Schema

```
MODULE semantic database

     IMPORTS primitive database (ALL)

     EXPORTS (db ref, make_db_ref, create_file, create_concept,
              ref space, contents_kind, concept_kind,
              get contents, kind, exists, successor, tested,
              listing for, instance of, part of, same_name,
              same_source_unit, same system );

     || This module specifies all the features of the
     || world model needed by the plans.  It provides
     || the functionality for the plans to both update
     || and query the state of the world, and through its
     || EXPORT clause makes these functions available to
     || the plans.
     || The specification of semantic database is accomplished
     || in terms of the functionality of the primitive_database
     || module, all of whose visible definitions are imported
     || for that purpose in the IMPORTS clause.

TYPE ref space IS (by name, file, concept, contents);

TYPE db ref IS RECORD

     ref_type IS ref space;

     WHEN ref_type = by name:
          name IS STRING,
          space IS ref_space SUBRANGE <file .. concept>;

     WHEN ref type = file:
          file_ptr IS file_access;

     WHEN ref_type = contents:
          contents ptr IS contents access;

     WHEN ref_type = concept:
          concept_ptr IS concept_access;

     || A reference to a database entity may take the form
     || of a name, which is in need of resolution into a
     || database "pointer", or any kind of a database pointer,
     || from which one may reach other database entities by
     || pointer chasing.

ENDRECORD;
```

```
FUNCTION look up ( x:db ref ) RETURNS db ref;
     || This function takes as its argument a db_ref
     || of arbitrary form, and returns as its
     || result, the associated db ref such that the
     || ref_type is not by name.  That is, it performs
     || a database look up, if necessary.

     IF NOT (x.ref_type = by name) THEN RETURN (x);
          || if the db ref is already in the form of
          || a database pointer, then no action is necessary.

     ELSEIF x.space = file THEN
          IF THEREEXISTS z:file_access SUCHTHAT
               z->file.name = x.name THEN
               RETURN ( <file, z> );
          ELSE RETURN ( <file, NULL> );
          ENDIF;
          || if the db ref gives the name of a file, then
          || return the database pointer to that file, if any,
          || else return a null pointer.

     ELSE
          IF THEREEXISTS z:concept access SUCHTHAT
               z->concept.name = x.name THEN
               RETURN ( <concept, z> ):
          ELSE RETURN ( <concept, NULL> );
          ENDIF;
          || if the db ref gives the name of a concept, then
          || return the database pointer to that concept, if any,
          || else return a null pointer.

     ENDIF;

END;


FUNCTION resolve_contents (x: db ref) RETURNS contents access;
     || This function takes a db ref of arbitrary form and
     || returns the associated contents entity.

     IF (x.ref type = by name) THEN
          RETURN ( resolve contents (look-up (x)));
          || if the db ref is in by name form, then
          || we want the contents associated with the
          || database entity whose name is given in the
          || db_ref.

     ELSEIF (x.ref_type = contents) THEN
          RETURN ( x.contents_ptr);
          || if the db ref is a pointer to a contents
          || database entity, then that is the contents
          || entity we want.
```

```
     ELSEIF (x.ref_type = file) THEN
          IF THEREEXISTS y:contents_access SUCHTHAT
               prim_stored_in (y, x.file_ptr) THEN
               RETURN (y);
          ELSE RETURN (NULL);
          || if the db ref is of file form, then
          || we want the contents entity stored in
          || that file, if any.

     ELSE RETURN ( resolve_by_context (x) );
     || Otherwise, we have a db_ref to some concept,
     || and we select its associated contents by
     || use of context information.

     ENDIF;

END;


FUNCTION get_contents (x: db_ref) RETURNS db_ref;
     || This function differs from resolve_contents only
     || in that its return type is db_ref, not contents_access.

     RETURN ( <contents, resolve_contents (x)> );

END;


FUNCTION resolve_concept (x: db_ref) RETURNS concept_access;
     || This function takes a db_ref of arbitrary form and
     || returns the assocaited concept entity.

     IF (x.ref type = by name) THEN
          RETURN ( resolve concept ( look up (x) ));
          || If the db ref is of by name flavor, then
          || we want the concept related to whatever entity
          || is named.

     ELSEIF (x.ref_type = concept) THEN
          RETURN ( x.concept ptr );
          || if the db ref is a concept pointer, then
          || that is the concept to return.

     ELSEIF (x.ref_type = file) THEN
          RETURN ( resolve concept (resolve_contents (x)));
          || If the db ref is  a file, then we want the
          || concept associated with the contents of the file.
```

```
     ELSE
          IF THEREEXISTS y: concept_access SUCHTHAT
               prim_instance_of (x.contents_ptr, y) THEN
               RETURN (y);
          ELSE RETURN (NULL);
          ENDIF;
          || If the db ref is of contents flavor, then we want
          || the concept associated with that contents.

     ENDIF;

END;


FUNCTION resolve_file (x: db ref) RETURNS file access;
     || This function takes a db_ref of arbitrary form and
     || returns the associated file entity.

     IF x.ref_type = by name THEN
          RETURN (resolve_file (look_up (x)));
          || if the db ref is of by name flavor, then we
          || must look it up first.

     ELSEIF x.ref type = file THEN
          RETURN (x.file_ptr);
          || The file associated with a file is itself.

     ELSEIF x.ref_type = contents THEN
          IF THEREEXISTS y: file_access SUCHTHAT
               prim stored in (x.contents_ptr, x) THEN
               RETURN (y);
          ELSE RETURN (NULL);
          ENDIF;
          || The file related to a contents is the file
          || in which the contents is stored, if any.

     ELSE RETURN (resolve_file (resolve_by_context (x)));
          || If the db ref is to some concept, then we
          || use context information to select a related
          || the related contents, from which we can
          || infer the related file.

     ENDIF;

END;
```

```
FUNCTION resolve_by_context (x: db_ref) RETURNS db ref;
     || This function takes a db ref of concept flavor and
     || returns a db ref of contents flavor.  This function
     || is specified non-deterministically, in that the
     || instance_of relationship can relate one concept to
     || many contents.

     ASSERTIONS
          x.ref type = concept;


     IF THEREEXISTS z: contents access SUCHTHAT
          prim instance_of (z, x.concept_ptr) THEN
          RETURN ( <contents, z> );
     ELSE RETURN ( <contents, NULL> );
     ENDIF;

END;



FUNCTION make db ref ( aname: STRING, aspace: ref_space)
     RETURNS db ref;
     || This function whips up a db ref from its arguments.

     ASSERTIONS
          aspace = file OR aspace = concept;
          || These are the only types of database entities
          || which have user-available names.

     RETURN ( <by name, aname, aspace> );

END;

PROCEDURE create file (x: db ref, y: contents kind);
     || This procedure, which is a state changing procedure,
     || enters a new contents into the database.  If the file
     || in which the contents is stored was not previously
     || known, then a file entity is created; otherwise, the
     || stored_in relationshipd for the file is updated.

     ASSERTIONS
          x.ref type = by name AND x.space = file;

     EFFECTS
          z = NEW (contents);
          NEW_VALUE'z->contents.kind = y;
          NEW_VALUE'z->create time = PRESENT TIME();
          || Allocate a new contents entity and initialize;
```

```
            lx = look up (x);

            IF lx.file ptr = NULL THEN
                 w = NEW (file);
                 NEW_VALUE'w->file_name = x.name;
                 lx.file ptr = w;
                 || If there is no such file, then create one now.
            ENDIF;

            NEW_VALUE'prim_stored_in (z, lx.file_ptr);
            || Record that the new contents is stored in this file.

END;

PROCEDURE create concept (x: db ref, y: concept kind);
      || This procedure, which is a state changing procedure,
      || enters a new concept into the database if the
      || concept is not already known.

      ASSERTIONS
           x.ref_type = by name AND x.space = concept;

      EFFECTS
           IF NOT exists (x) THEN

                z = NEW (concept);
                NEW_VALUE'z->concept.name = x.name;
                NEW_VALUE'z->concept.kind = y;

                || IF the concept has not yet been encountered,
                || then we create it now and initialize its
                || constituent fields.

           ENDIF;

END;



FUNCTION exists (x: db ref) RETURNS BOOLEAN;

      IF (x.ref type = by name) THEN
           RETURN (exists (look-up(x)));
           || IF the db ref is of by name flavor, then
           || we must look it up first.

      ELSEIF (x.ref_type = file) THEN
           IF x.file ptr = NULL THEN
                RETURN (FALSE);
           ELSE RETURN (exists (resolve_contents (x)));
           || A file exists if both it exists and its
           || contents exist.
```

```
     ELSEIF (x.ref_type = contents) THEN
          RETURN (IF x.contents ptr = NULL THEN FALSE ELSE TRUE);
          || When the db ref is of contents flavor, then the
          || contents exists if the database pointer is not null.

     ELSE
          RETURN (IF x.concept ptr = NULL THEN FALSE ELSE TRUE);
          || When the db ref is of concept flavor, then the
          || concept exists if the database pointer is not null.
     ENDIF;
END;


FUNCTION kind (x:db ref) RETURNS contents_kind;
     || The kind function returns the kind of the
     || contents associated with the db ref given as
     || input.

    ASSERTIONS
          IF x.ref_type = contents THEN
              NOT x.contents_ptr = NULL;

     IF NOT x.ref_type = contents THEN
          RETURN (kind (resolve_contents(x)));
          || In this case, we must first find the contents
          || associated with this db ref.

     ELSE x.ref type = contents THEN
          RETURN (x.contents ptr-> contents.kind);
          || once we have the associated contents, we can
          || return its kind directly.
     ENDIF;
END;


FUNCTION assoc name (x:db ref) RETURNS STRING;

     IF x.ref type = by name THEN
          RETURN (x.name);
          || if the database reference is by name,
          || then return the associated name.
     ENDIF;
```

```
     z = resolve file (x);

     IF NOT z.file ptr = NULL THEN
          RETURN (z.file name);
          || The file associated with the given db_ref has
          || the desired name.
     ENDIF;

     z = resolve concept (x);

     IF NOT z.concept_ptr = NULL THEN
          RETURN (z.concept name);
          || The name concept associated with the given db ref
          || is the desired name.

     RETURN (FALSE);
     || The name of the associated file is not the name desired
     || nor is the name of the associated concept.

END;


|| Each of the following functions is defined in terms of
|| its primitive level counterpart; for example, successor
|| is defined in terms of prim successor.  In each case,
|| the db ref arguments are transformed into the
|| appropriate type of database entity accesses by the
|| use of resolve_contents or resolve concept or resolve_file.


FUNCTION successor (x:db ref, y:db ref) RETURNS BOOLEAN;

     RETURN (prim_successor (resolve contents (x),
                             resolve contents (y)));
END;


FUNCTION tested (x:db ref, y:db ref, z:db ref) RETURNS BOOLEAN;

     RETURN (prim tested (resolve_contents (x),
                          resolve_contents (y),
                          resolve_contents (z));

END;
```