# SYSTEM ENGINEERING TECHNIQUES FOR ARTIFICIAL INTELLIGENCE SYSTEMS

Lee D. Erman and Victor R. Lesser[1]

Department of Computer Science[2]
Carnegie-Mellon University, Pittsburgh, Pa. 15213

## ABSTRACT

It is impossible to develop a large knowledge-based artificial intelligence system successfully without careful attention to issues of system engineering. A set of principles is presented for organizing the design and implementation of such a system. Problems of maintainability and configuration control, human engineering, performance analysis, and efficiency must be faced. Tools used to solve these problems are described, along with examples of their use in the Hearsay-II speech understanding system.

## INTRODUCTION

In the last several years, there has been a trend in Artificial Intelligence (AI) research to develop high-performance systems specialized for particular problem domains (e.g., medical diagnosis, chemical analysis, image understanding, and speech understanding). In order to attain the high performance desired, these systems need to use a large amount of problem-specific knowledge and, often, large numbers of heuristics. These systems are commonly called "knowledge-based" systems, which differentiates them from the earlier AI systems that used a small number of general heuristics and little problem-oriented knowledge.

Several of the characteristics common to high-performance knowledge-based systems have significant implications for their development:

- The systems are <u>large</u>, because of the large amount of knowledge; they are often structurally <u>complex</u>, because of the diversity of their knowledge and the complexity of the heuristics required for applying this knowledge.
- The development of such systems is marked by much <u>experimentation</u> and <u>redesign</u>. This occurs because the problem is not well enough understood to enable the knowledge needed or the methods of its application to be pre-specified. In addition, because the knowledge and heuristics interact in a complex way, models cannot be built to predict the system's performance -- instead, the system itself must be run.
- The systems are <u>computationally expensive</u> (in time and/or space). Part of the expense is inherent in the problem domain and in the demands of high-performance. The expense is increased because the knowledge is imperfect and the strategies for applying it are not optimal. Much of the cost often manifests itself as search for a problem solution within a large space.
- <u>Many researchers</u> may be needed to develop such a system and they may have diverse expertise. The system must provide a structure for coordinating their individual efforts.

<u>Thesis:</u>
*We believe that it is impossible to develop a high-performance, knowledge-based system successfully without careful attention to system engineering considerations derived from the characteristics listed above.* These considerations dictate the tools needed for the system. Many of these tools are missing in even the most sophisticated general-purpose computing environments and must be built. Many of the tools that already exist must be modified to make them suitable for the special demands of these systems. Thus, there must be a flexibility and a willingness to experiment with <u>all</u> levels of the computing facility -- hardware, firmware, operating system, programming system, terminal system, etc.

In expanding this thesis, which is based primarily on our experience in a long-term effort in building speech understanding systems, we will address the following four problems that we have found to be crucial[3]:

<u>The Maintainability and Configurability Problem:</u>

The system is constantly evolving in an asynchronous manner, with various components in different states of development. No component is ever "finally" stabilized, but is subject to further modification and reimplementation. In the face of this flux, the system must be usable by each researcher when he wants it, independent of the ongoing modifications being made by others to components not directly of interest to him. Modification of a component should require minimal if any modifications to other components.

While developing one component of the system, it is often desirable to be able to experiment with that component in relative isolation, i.e., with just enough of the other components present to provide the necessary context. Thus, it should be easy to configure a subsystem consisting of that necessary subset of components.

---

1 V. R. Lesser's current address: Department of Computer and Information Science, University of Mass., Amherst, Mass. 01003.

2 This work was supported at Carnegie-Mellon University by the Defense Advanced Research Projects Agency (F44620-73-C-0074) and is monitored by the Air Force Office of Scientific Research.

3 These problems are faced in the design and implementation of any large programming system. Research in software engineering particularly relevant to the techniques discussed in this paper include Parnas [72] on modular decomposition; Dijkstra [68], Liskov [72], and Habermann et al [76] on hierarchical structuring; Liskov and Zilles [74], Flon [75], Guttag [76], and Wulf et al [76] on abstract data types; and Habermann [78] and Cooprider [78] on configuration management techniques.



Copyright © 1978 by Academic Press, Inc.
All rights of reproduction in any form reserved.
ISBN 0-12-323550-2

In addition, the software system should allow the user to configure a system with the most up-to-date tested versions of components produced by the other researchers, as long as they are consistent with the version of his component that he would like to test. Thus, the system should automatically check that the configuration being assembled is constructed from compatible versions of the various components.

The Human Engineering Problem:

Unless the system is relatively easy to use, it will be a confusing morass for the researcher. When we speak of "the" researcher experimenting with and modifying the system, we really mean many researchers with diverse interests and often significantly different levels of expertise with computers and programming. Thus the system design must be sensitive to these varying interests and skills. For example, a researcher will in general not be intimate with more than some fraction of the system. Thus, the system must allow him to interact effectively without having a detailed knowledge of the components he has not considered.

The Performance Analysis Problem:

The major part of the experimentation is the refinement and augmentation of the various pieces of knowledge and their interactions. To accomplish this, the researcher must be able to determine the effects of each piece of knowledge. The complex interactions between the large amounts of knowledge makes this analysis difficult. Thus, the system must provide tools which make it easier for the researcher to isolate and trace the effects of particular knowledge.

The Efficiency Problem:

The overall computational efficiency is important for producing a usable performance system, as well as for experimenting with the system during its development. In a large and changing system, it is difficult and inappropriate to optimize across the whole system. Rather, an evolutionary approach is called for, in which those aspects of the system that become bottlenecks are selectively optimized.

Efficiency is a relative issue, based on the ways that the system is being stressed at each stage of development. The critical bottlenecks change as use of the system evolves, as the system is changed in response to the changing use, and as bottlenecks are removed through optimization. In order to support selective optimization, the system must make it convenient to measure costs and re-implement the components responsible for bottlenecks at the appropriate level (e.g., in micro-code, in machine language, within the operating system, at some level within the programming language, or at some level within the implementation of the system). Such modifications should require minimal modifications to other system components.

The next section presents a brief overview of the Hearsay-II speech understanding system, to provide a context for what follows. Although Hearsay-II is used as the motivating example for this paper, we believe that the problems and solutions presented are relevant to other AI system efforts, and, in fact, to many other large system efforts as well. Following the Hearsay-II overview is a description of several principles for organizing the system implementation. Finally, there is a listing of many of the key tools used in the Hearsay-II system.

## OVERVIEW OF HEARSAY-II

In 1971-72, the Hearsay-I speech understanding system was developed at Carnegie-Mellon University -- the first of a series of such systems. Hearsay-I [Reddy et al 73 and Erman 74] was a successful attempt to solve the problem of machine understanding of speech in specialized task domains. In this early system, the size of the vocabulary (fewer than 100 words) and complexity of the grammar were very limited. Experiences with Hearsay-I led to the more generalized Hearsay-II architecture [Lesser et al 75, and Erman and Lesser 75] in order to handle more difficult problems (e.g., larger vocabularies and less-constrained grammars). The Hearsay-II system has been successful: it came close to the original ARPA performance goals set out in 1971 to be met by the end of 1976 [Newell et al 73]. Its performance in September, 1976, was 90% correct semantic interpretation of sentences over a 1011-word vocabulary and constrained syntax [CMU 77].

The Hearsay-II Architecture[4]

At the beginning of the Hearsay-II effort in 1973, based on our experiences with Hearsay-I, we expected to need types of knowledge and interaction patterns whose details could not be anticipated. (As mentioned above, this uncertainty is characteristic of the development of knowledge-based systems.) Instead of designing a specific speech understanding system, we considered Hearsay-II as a model for a class of systems and a framework within which specific configurations of that general model could be constructed and studied. One can think of Hearsay-II as a high-level system for programming speech understanding systems of a certain type -- i.e., those that conform to the Hearsay-II model.

In the Hearsay-II model of knowledged-based systems, each of the diverse types of knowledge needed to solve a problem is encapsulated in a knowledge source (KS). For speech understanding, typical KSs incorporate information about syntax, semantics, acoustic-phonetics, prosodics, syllabification, coarticulation, etc. The current Hearsay-II system configuration has about ten KS modules. KSs are kept separate, anonymous, and as independent as possible, in order to make the creation, modification, and testing of KS modules as easy as possible.

The KSs interact to solve the problem (i.e., interpret a spoken utterance) by communicating via a shared global data-base called the blackboard. The blackboard is partitioned into distinct information levels (e.g., "phrase", "word", "syllable", and "phone"); each level holds a different representation of the problem space. The current state of problem solution is represented in terms of hypotheses on the blackboard. An hypothesis is an interpretation of a portion of the spoken utterance at a particular level (e.g., an hypothesis might be that the word 'today' occurred from millisecond 100 to millisecond 600 in the utterance). All hypotheses, no matter what their level, have a uniform attribute-value structure. For example, each hypothesis has attributes containing its level, begin- and end-time within the utterance, and plausibility ratings. Hypotheses are connected through a directed graph structure, usually across levels.

4 Little attempt is made here to motivate this architecture from an AI viewpoint. For more information on Hearsay-II, see Lesser et al [75], Erman and Lesser [75], and Lesser and Erman [77].

Each knowledge source is activated in an asynchronous manner, based on the occurrence on the blackboard of patterns of hypotheses specific to its interests. Once activated, a KS may examine the blackboard, typically in the vicinity of the hypotheses that activated it. Based on its knowledge, the KS may then modify those hypotheses or other hypotheses, or create new hypotheses. Such actions establish new patterns on the blackboard; these potentially cause other KSs to be activated. This mechanism for KS activation implements a data-directed form of the hypothesize-and-test paradigm.

### The Hearsay-II Implementation

Based on the model just described, a high-level programming system was constructed to provide an environment for programming KSs, configuring groups of them into systems, and executing them. All interactions of KSs are via the blackboard -- triggering on patterns, accessing hypotheses, and making modifications. Because the blackboard has a uniform structure, KS interactions are also uniform. Thus, one set of facilities can serve all KSs. Facilities are provided for:

- defining the levels on the blackboard,
- configuring groups of KSs into runnable systems,
- accessing and modifying hypotheses on the blackboard,
- activating and scheduling KSs.

These facilities, along with other utilities to be described below, are called the Hearsay-II 'kernel'. The kernel is the high-level environment for creating and testing KSs and configurations of them.

Hearsay-II is implemented in the SAIL programming system [Reiser 76], an Algol60 dialect which has a sophisticated compile-time macro facility and a large number of data structures (including lists and sets) and control modes which are implemented fairly efficiently. The Hearsay-II kernel provides a high-level environment to KSs at compile-time by extending SAIL's data types and syntax through declarations of procedure calls, global variables, and complex macros. This extended SAIL provides an explicit structure for the specification of a KS and its interaction with other KSs (through the blackboard). The high-level environment also provides mechanisms that enable KSs to specify to the kernel (usually in non-procedural ways) a variety of information which the kernel uses when configuring a system, scheduling KS activity, and controlling user interaction.

The knowledge in a KS is represented in SAIL data structures and code, in whatever form the KS developer finds appropriate. The kernel environment provides the facilities for structuring the interface between this knowledge and the other KSs, via the blackboard. For example, the syntax KS contains a grammar for the specialized task language that is to be recognized; this grammar is in a compact, network form. The KS also contains procedures for searching this network, for example, to parse a sequence of words. The kernel provides facilities (1) for triggering this KS whenever new word hypotheses appear on the blackboard, (2) for the KS to read those word hypotheses (in order to find the sequence of words to be parsed), and (3) for the KS to create new hypotheses on the blackboard, indicating the structure of the parse.

There are two aspects of KS specification in the Hearsay model which appear to drive an implementation in opposite directions -- one for diversity and the other for uniformity:

- Because the KSs are diverse, the kinds of data and control structures that are natural and efficient for their implementation are also diverse. No single pair of structures (e.g., lists as data structures and a production system as a control structure) is optimal. Thus, this first aspect pushes the implementation in the direction of diversity (i.e., to provide a diverse set of KS-specific data and control structures).
- Because the KSs are to cooperate, they must interface with the rest of the system. Given that one needs to be able to add new KSs easily and construct a set of utilities applicable to all KSs, it is desirable to have this interface be uniform. In addition, if the interface declarations are designed properly, it should be possible to make extensive modifications to the implementation of these facilities without having to modify the KSs. Thus, this second aspect pushes the implementation in the direction of a uniform set of data and control structures.

By having a uniform high-level framework for KS interaction while still permitting KS developers to code the knowledge in whatever form is found to be most convenient, an appropriate balance has been struck between diversity and uniformity.

### Development of Hearsay-II

The active development of Hearsay-II extended for three years. About twenty KS modules were developed during that time, each a one- or two-person effort lasting from two months to three years. The modules range from about 10 to 200 pages of source-code (with 60 pages typical); additionally, each KS has up to about 100K bytes of information in its local data base.

The kernel is about 300 pages of code. About one-third of that is made up of the declarations and macros that create the extended environment for KSs. The remainder is made up of the code for implementing the architecture -- primarily activation and scheduling of KSs, maintenance of the blackboard, and a variety of other standard utilities (to be described below). During the three years of active development, an average of about two full-time research programmers were responsible for the implementation, modification, and maintenance of the kernel. Included during this period were a half dozen major reimplementations and scores of minor ones; these changes usually were specializations or selective optimizations, designed as experience with the system led to a better understanding of the usage of the various constructs.

Implementation of the first version of the kernel began in the autumn of 1973, and was completed by two people in four months. The first major KS configuration, though incomplete, was running in early 1975. The first complete configuration, called "C1", was running in January, 1976. This configuration had very poor performance -- less than 10% sentence accuracy over a 250-word vocabulary. Experience with this configuration led to a substantially different KS configuration, "C2", that performed much better (90% accuracy over a 1011-word vocabulary), coming very close, in September, 1976, to the original goals of the project.

## ORGANIZING PRINCIPLES

Four design principles were used in organizing the Hearsay-II kernel and surrounding facilities:

- The design should start with a general framework which is then tailored as needed.

Two of the principles are derived from the observation that a system is naturally implemented as a series of levels, which form a loose hierarchy:

- In order to implement a complex system, all the supporting levels (hardware through programming language) must be subject to change.
- The application to be implemented on top of the programming system level should be implemented as a series of levels, rather than as a single level.

The fourth principle deals directly with the problem of experimentation:

- Facilities for analysis and debugging must be an integral part of the system design from the beginning.

### Tailoring a General Framework

The approach taken in the project was one of starting with a general model for a class of speech systems and then specializing this general model based on experience. Three ways of tailoring can occur:

- The elimination of excess generality, based on the ways the system is actually used. For example, Hearsay-II began with a set of complex primitives for interconnecting hypotheses across levels; as experience was gained with real KSs, it was discovered that these facilities could be simplified.
- The addition of new features, as needed. For example, the invocation of KSs was expanded to include, for scheduling purposes, an abstract description of the potential action of the KS.
- The reimplementation of existing features, for efficiency. For example, the internal representation of hypotheses went through several reimplementations, based on changing usage patterns of the blackboard primitives.

The notion of a general framework provides a context for tailoring so that the overall system retains a coherence, rather than evolving in a haphazard manner with one change piling on the next.

Such an approach has a potential disadvantage: the start-up cost is relatively high; the danger is that the model may be inadequate for keeping the high-level system usable long enough to amortize those costs. However, if the framework is suitable, it can be used to explore different configurations within the model more easily than if each configuration were built in an ad hoc manner. Additionally, a natural result of the continued use of any high-level system is its improvement in terms of enhanced facilities, increased stability and efficiency, and more familiarity on the part of the researchers using it.

Hearsay-II has been successful in this respect; we believe that the total cost of creating the high-level system and using it to develop KS configurations C1 and C2 was less than it would have been to generate C1 and C2 in an ad hoc manner. It should be stressed that the construction of even one configuration is itself an experimental and evolving process. The high-level programming system provides a framework, both conceptual and physical, for developing a configuration in an incremental fashion. The speed with which C2 was developed is some indication of the advantage of the system-design approach used in Hearsay-II. And, we are still far from exhausting the possibilities of the existing Hearsay-II framework.

### Levels

The usual approach to building an application system is to take as given a language system and its supporting levels (e.g., hardware, firmware, and operating system) and construct the application directly on top of the language system. However, for the efficient implementation of a complex application, it is often necessary to modify these lower levels. For example, the Hearsay efforts led to these modifications:

- At the hardware level:[5] Special-purpose audio devices were constructed. A hardware device-poller was constructed because the existing software polling on the PDP-10 was not fast enough to support the real-time audio devices. Graphics terminals and printers were added to the system.
- At the operating system level: Highly optimized service routines were needed to handle the audio and graphics devices. The scheduler was modified to handle real-time jobs and very large jobs in special ways. Changes were made in I/O handling to support overlaying efficiently.
- At the language system level: The compile-time macro facility was greatly expanded, the memory allocator was modified to handle the demands of large, long-running jobs, and an overlaying facility was added. In addition, the compiler was modified so that it could be pre-initialized with the kernel-provided declarations, significantly decreasing the time to compile each KS.
- At the utilities level: The loader was modified to handle overlaying. A cross-referencing program was implemented which could dynamically search through the many source-code files which make up the kernel and the KSs.

In order to make such modifications, it is necessary to have both the expertise and appropriate control over the computing facilities.

The second principle concerning levels deals with the highest levels, those usually lumped together and called the application level.[6] As this level becomes more complex, the implementation distance between it and the underlying language system increases. Rather than trying to bridge this distance in a single jump, one or more intermediate levels should be built. These layers should reflect the components and structure of the application. Such a layered

5 The system used has no microstore; this level, however, is becoming increasingly attractive for application-dependent modification.

6 The fact that the highest level is conventionally called the "application" level is indicative that the principle of modifying any necessary level is not usually adopted. That is, most people consider only the level above the language-system as application-dependent; all the lower levels are taken as immutable.

implementation is easier to understand, debug, and modify because the complexity of interactions between components is reduced.

For example, a KS in Hearsay-II modifies an attribute of an hypothesis on the blackboard through four layers of procedures and macros: (1) At the highest level, each attribute has its own modification macro, part of whose function is to append the name of the modifying KS. (2) These macros call one of the modification procedures; these routines are generalized over a set of the hypothesis attributes and do parameter checking (e.g., to insure that the thing pointed to is really an hypothesis and that the new value is appropriate for the attribute). (3) These procedures in turn call a lower level procedure which is attribute-independent; this procedure is also called directly by other kernel routines in places where the parameter checking is not required. (4) Finally, this procedure calls a macro which actually modifies the attribute.

By hiding, through such layers of macros and procedure calls, the details of how an hypothesis on the blackboard is stored and accessed, it has been possible to re-implement these aspects of the blackboard with only minimal changes required elsewhere; it has also allowed the addition of specialized options to the standard retrieval commands for the needs of specific KSs without affecting other KSs. For instance, during the lifetime of the project, the hypothesis storage and accessing functions went through four major implementations:

- Initially, all attribute/value pairs of an hypothesis were stored as an attribute/value list structure.[7]
- This storage scheme was then modified so that integer-valued attributes of an hypothesis (e.g., its rating and begin- and end-times) were stored as compacted bit-fields in a linear block of memory, thus permitting these fields to be accessed by a simple index and byte extraction operation.
- Next, the storage of standard list attributes of an hypothesis (e.g., the list of connected hypotheses) was changed into a compacted list structure with each list directly accessible through an index operation.
- Finally, optional attributes of an hypothesis were stored in a compacted attribute/value list structure.

The net result of these optimization steps was an improvement of about an order of magnitude in both storage space and accessing time. Similarly, other aspects of the system, such as retrieval functions and their associated data bases, the scheduler, and the activation records of KS processes were successively reimplemented as we gained a better understanding of how these system functions were actually used, both in terms of time and space. Each of these changes resulted in significant increase of system-wide efficiency, in some places orders of magnitude, without requiring any changes to the KS source-code, and only localized changes within the kernel.

7 The values of attributes can be integers of varying numbers of bits or variable-length lists or sets, each element being the name of some hypothesis or a small (12-bit) integer.

Integrated Analysis and Debugging Facilities

The principle of having analysis and debugging facilities as an integral part of the system design at all levels follows from several observations:

- The size and complexity of the system require automatic analysis facilities within the system; one cannot understand the internal functioning of the system just by observing its external behavior.
- The experimental nature of a knowledge-based system results in modification of the system throughout its lifetime. No part of the system is ever "final"; any part is subject to modification at any time and hence may need analysis and debugging facilities at any time.
- Debugging and analysis facilities are needed at all levels within the system. In particular, facilities appropriate to the application levels built on top of the language system are crucial. Facilities provided within the language system may be helpful but are not sufficient for debugging and analyzing the higher levels. Data and control structures at higher levels should be displayed in terms of concepts appropriate to the level rather than in terms of their implementation in the base language (which might be several levels removed). For example, the "rating-state" attribute of an hypothesis in Hearsay-II can take on values from the set "unrated", "accepted", "rejected", etc. Internally, these values are small integers; the KS developer, however, usually does not want to see the integers, but rather their names.[8]

It is important to consider analysis and debugging facilities as an integral part of the system design. Pragmatically, they are especially important when the system is young, but must also endure as the system evolves. If their design is ad hoc and not well-integrated, they will be difficult to modify to keep pace with the system modification.

Good debugging and analysis facilities need not be computationally costly if implemented with selective enabling. In particular, compile-time disabling of such a facility permits it to be retained in the system at no run-time cost; when needed again, it can be re-enabled without code modification.

## SYSTEM-BUILDING TOOLS

The organizing principles outlined in the preceding section provided a context for developing many of the tools in the Hearsay-II system. Following is a description of these tools, classified roughly in three groups: configuration management, user interaction, and performance analysis. We feel that similar tools are required in any large, knowledge-based system. Obviously, many other tools are possible; those presented here were found useful for Hearsay-II and indicate the broad range of facilities required.

8 Notice that the external representation (i.e., the name) is independent of the internal representation; one can optimize the internal representation without sacrificing the natural representation used for interaction if appropriate facilities are included at each implementation level.

Configuration Management Tools

As described earlier, the configuration control problem is one of selecting pieces to comprise a system. An important aspect of this problem is ensuring the consistency of those selections. In a complex system, there are a large number of selections that need to be made in order to create and execute a configuration. For Hearsay-II, some of these include selecting:

- A version of the kernel.
- A set of KSs and a version of each.
- The local knowledge base(s) for each selected KS.
- Settings for the many local parameters which fine-tune the knowledge in the KSs.
- The set of optional facilities (tracing, timing measurement, graphical display, special-purpose analysis, etc.).
- The vocabulary and grammar of the spoken language to be recognized. (Languages of several different sizes and grammatical complexities were developed).
- The set of test data (utterances, either live or "canned") to be run through.

A simplified diagram of the sequence of events necessary to create a Hearsay-II configuration is given in Figure 1. The processors, represented in the figure by nodes, include the SAIL compiler, the system linking loader, special-purpose compilers for data-structures to be used during system execution, and a runnable Hearsay-II system. For each process, a set of input objects must be specified, represented in the figure by labeled arrows. The result of the execution of the process with the inputs is one or more output objects.

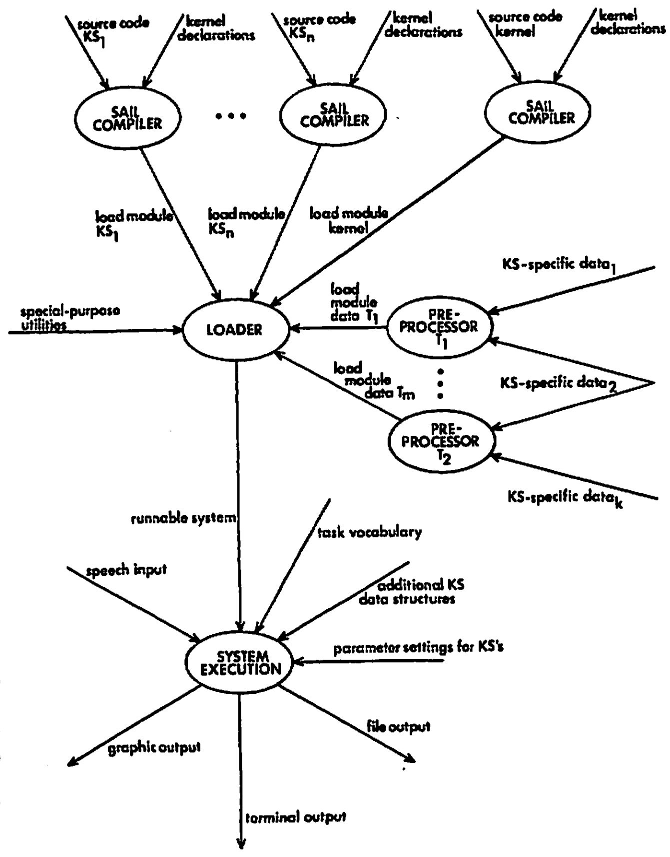


Figure 1. Creating and executing a Hearsay-II system (simplified).

In order to make the system usable, configuration management tools must be provided which make it easy to select the input objects for each process. Mechanisms must also be provided for automatic consistency checks across the specified input objects. For example, referring to the figure, such checking should make sure that all the KS modules that are loaded together with the kernel module were originally compiled with kernel declarations that match that version of the kernel load module. Similarly, each data object module should be consistent with the accessing procedures contained in the corresponding KS object module that is loaded with it. There are a variety of other types of consistency checks that need to go on. These occur during all phases: compile-time, load-time, and execution-time.

In order to accomplish this consistency checking, each output object of a processor is labeled with a header indicating the type of processing that occurred, the version number of the program performing the processing, and a condensed form of the header information of all the input objects and key parameter settings of the processing. Before processing, the header information of the input objects is checked in order to determine if those objects are consistent with each other and appropriate for the processor and mode of processing selected.

Sometimes information in the headers of one or more of the input objects can be used by the processor to calculate the names and versions of the other input objects that are needed. This mechanism reduces the amount of information that the user needs to specify explicitly (because it is redundant in the header information in these cases), simplifying the task and reducing the likelihood of error in the specification.

Another way to reduce the amount of external specification needed is by building into the processors default values for selecting their input objects and parameter settings. Thus, only exceptions to the defaults need be specified. This allows the researcher to build added flexibility into a processor without complicating its control for those users not needing that flexibility.

Although these techniques for automatic selection do reduce the total number of parameters to be set and options to be selected, there are still many left to be specified explicitly by the user. In general, command files for processors greatly simplify this task by clustering the specifications for "standard" configurations and allowing them to be named. In addition to standard types of command files for the compiler and loader, a tailored "job control language" has been constructed in Hearsay-II for configuration control and user interaction at execution time. This language permits a user to construct files, called cliche files, which contain sequences of parameter settings and selection control for the various "processes" within an executable Hearsay-II system. Cliche files can contain commands to execute other cliche files and to override parameter settings accomplished in previously executed cliches. Nesting and resetting together provide a simple mechanism for constructing specialized configuration control files from a set of standard sub-configuration cliche files.

The need for consistency checking at all phases of the system cannot be emphasized strongly enough. Errors resulting from inconsistent configurations are the most difficult to detect and trace to their sources. Early versions of the Hearsay-II system did very little such checking, resulting in many hours and days lost.

User Interaction Tools

Since this research is based on the philosophy of the experimental paradigm (build a system to try out ideas), the ease with which the researcher interacts with the system is crucial.

When we speak of the "researcher" experimenting with and modifying the system, we really mean several researchers with diverse interests and, perhaps, significantly different levels of expertise with computers and programming. A user has different goals at different times, thus providing the need for different interaction modes for even a single researcher. The system design must be sensitive to these varying interests and skills. In particular, the interaction facility must permit the user to interact with the system at a high level, i.e., in terms of the information units most natural to the way he is thinking about the problem.

In Hearsay-II, the primary mode of user interaction is via a facility that permits the user at the terminal to display and modify in a high-level manner both kernel-implemented data structures (e.g., the blackboard) and those specific to each KS. Since the meanings and structures of KS-specific data bases are highly individualized, a mechanism that allows for the display of data structures only in terms of their implementation is inadequate. Rather, a debugging interface tailored for each KS is necessary.

This interface should be integrated into the terminal interaction facility in a coherent manner so that the researcher does not suffer from major changes in context as he moves from one interface to another; similar actions by the user in different interface packages should cause similar reactions by the system.

Each KS module in the Hearsay II system has its own package of routines for interfacing with the researcher at his terminal. Each interface package is tailored to the module, but they all share a common syntax. While "talking" with a module's interface, the researcher may display and alter data, set break-points in the module, and execute actions of the module. In addition, he may set switches that enable data dumping at prescribed points during the module's processing; this data may be dumped at the terminal and/or to a file for later processing.

Multiple special-purpose interfaces might seem to require much more programming than a single interface. In fact, this is not true. The kernel provides macros and associated procedures which enable a user to build this interface easily. For example, a macro, called MakeVariable, is provided which permits a user to declare a SAIL variable in his KS while also specifying routines to be used to display and update the value of the variable at the terminal. In addition, the user has the choice of building his own display and update routines or choosing from a predefined set in the kernel. An example of a declaration using built-in display routines is the following:

```
MakeVariable( VwlAThrsh, CvS,
              GetInteger( 20, 40, 33,
                          "Vowel amplitude threshold" ))
```

This declares a variable with the name "VwlAThrsh". Whenever the user asks to display its value, the CvS ("convert an integer to string") procedure will be used; this is a standard SAIL run-time procedure. Whenever the user asks to change the value, the kernel-supplied GetInteger procedure will be used. This procedure will prompt the user for a new value for VwlAThrsh; if the user asks for the default, it will be 33. If the user specifies a new value, GetInteger will guarantee that it will be not less than 20 nor greater than 40. Also, if the user responds with a "?", he will be further prompted with the comment string.

The terminal interaction facility has a number of additional features which make it convenient to use without remembering a vast amount of detail: cliche files, default values, a spelling corrector and automatic abbreviation recognizer for variable and command names, user-defined prompts (comments on the meaning and range of variables and commands), range checking on parameter values, and menu display capabilities for determining the variables and commands available within a specific KS and at the system level. These features make it possible for the system to be somewhat self-documenting. This is important because it is difficult to keep manuals up-to-date in a complex, evolving system.

Another requirement for the use of the system is that the user be able to regain control from an executing system when certain types of events occur. There are three kinds of interrupts that can stop the system and place the researcher in communication with the terminal interaction facility: The user can cause an asynchronous interrupt from his terminal any time the system is executing; this causes the system to halt and allows him to access the interface routines. Breakpoints may be set which will cause the system to halt at predetermined points in its execution. Internal error conditions also cause interrupts. After processing an interrupt, the user can cause the system to continue or can abort execution.

Analysis Facilities

As described in the introduction, understanding and improving the system comes largely from analyzing experiments made with the system. The kinds of data that need to be analyzed include both the final results of running the system (i.e., its "outputs") and the intermediate results (i.e., the internal processing that causes the final results). For example, the final output of an execution of the Hearsay-II system on a spoken utterance is the system's interpretation of that utterance; an analysis of this could mean determining to what degree the result is correct (e.g., matched the intention of the speaker). Intermediate results of Hearsay-II most often of interest are hypotheses on the blackboard; analyses of these might include a determination of how correct the hypotheses are, as well as assignment of credit or blame for their existence to individual KSs. Analyses often need to be aggregated over multiple runs, either to compare the performance of different versions of the system or to provide a statistical validity of performance measurement.

In order to understand the internal processing of the system, a tracing facility has been developed. This facility dumps, at selectable levels of detail, snapshots of the state of processing. Because of the large amount of internal activity in the system, system behavior is difficult and time-consuming to understand from just the trace output, even though that output is carefully formatted and at a high level of abstraction. In a typical run of one utterance in Hearsay-II, 300 to 600 KSs activations occur. Each activation

creates on the average two to three new hypotheses. A typical data set contains about twenty-five such utterances.

In order to make the trace information more comprehendible, a mode has been added which allows the trace facility to distinguish correct hypotheses from incorrect ones and to mark the hypotheses on the trace output accordingly. This correct mode requires that the system be supplied with a file specifying, for each utterance being processed, the characteristics of the correct hypotheses at one or more levels of representation on the blackboard. As new hypotheses are created, they are marked "correct" if they match those pre-specified characteristics or are correctly derived from correct hypotheses. Hypotheses labeled "correct" can be highlighted in the trace output, as can descriptions of KS activations that are working on correct hypotheses. Thus it becomes easy to distinguish in the trace output between areas of correct and incorrect processing. The ability to distinguish between correct and incorrect hypotheses also permits the automatic computation of analyses such as the average rank order of the correct hypotheses.

Some KS problems can also be detected by using the automatic labeling of correct hypotheses in another way, called automatic pruning. In this mode, any incorrect hypotheses that a KS attempts to create are discarded: this guarantees that only correct hypotheses will be on the blackboard and, thus, that all KS activations will have only correct hypotheses to work on. In a system in which the knowledge is incomplete and errorful, there is no guarantee that a KS will produce correct output given correct input. Automatic pruning identifies such problems very quickly by eliminating all consideration of incorrect inputs on all KS activations.

The ability to create and execute partial configurations of KSs is also useful for making analysis more efficient. With this ability, a system which is smaller and requires less computation can be used if only a small number of KSs are to be tested. One can also save the results of a run of a (partial) configuration by dumping the blackboard; the saved state can be loaded subsequently into a system with a different configuration of KSs, saving the cost of regenerating the state. The Hearsay-II system is often run in two non-overlapping configurations of KSs: the first partial configuration does preprocessing of the speech signal and the second uses those results. With some simple changes to the command files only, a complete system (i.e., containing the KSs in both partial configurations) can be created and executed.

Facilities are also required to make it convenient to run and analyze a large number of test cases. Often it is possible to do these analyses in an automatic manner by post-processing the system trace output. Thus, it is important to format the output so that it is machine-readable. Appropriate header information must also be included in the output in order to identify it; this is important because many different trace files are produced and need to be distinguished. For Hearsay-II, typical header information includes the name and creation date of the KS configuration, settings of key parameters, the task vocabulary and grammar, and the name of the utterance being recognized.

It is often convenient to have several concurrent destinations for trace output, e.g., an interactive terminal, a comprehensive log file, and a file with trace information about a particular KS. Hearsay-II provides a mechanism so that a user is able to select those output channels to which each kind of trace information should be routed.

Much effort goes into making the system work well interactively. However, it is often desirable to run in batch-mode, in order to analyze large numbers of test cases. A special-purpose facility has been implemented within the Hearsay-II kernel to facilitate batch processing. Traditional batch facilities are not adequate for handling the errors that occur often while running large, experimental systems. Typical errors are running excessively long, exhausting some resource (e.g., memory space), and the execution of an illegal operation (e.g., array subscript out of bounds or parameter mismatch on a procedure call). Normal batch systems react to such errors by aborting execution, perhaps with some low-level dump of the system state. In order to produce meaningful dumps, the experimental system *itself* needs to regain control; it can then use its high-level display and trace facilities to provide information for subsequent analyses. Often, the system can recover from such errors, either continuing to process the same test case, or, at worst, going on to the next. In order to retain this control, the experimental system must be able to detect such errors before the controlling batch-system detects them; it must also be able to interpret the errors and take appropriate action.

In order to help analyze and improve the efficiency of system execution, a timing facility has been integrated into the kernel for use of KS and system developers. This timing facility is a compile-time option and thus (like debugging statements) incurs no overhead when not used. This facility allows the programmer to name blocks of code that are to be timed. For each such timed block, statistics are accumulated on the number of block entries, time spent in the block, and time spent in the block but excluding time spent in any timed block (dynamically) nested within it. The timing package is very economical to use: typical costs for extensive use are 4% overhead in computation time (which the package also keeps statistics on) and 6% increase in job size. The package has seen considerable use in selectively optimizing the Hearsay-II system.

## CONCLUSION

The sections on tools for configuration management, user interaction, and analysis have indicated the large amount of thought and effort involved in system engineering aspects of Hearsay-II. This effort was well spent -- without it the project would not have been successful. We feel such an effort is necessary for the success of any large, complex knowledge-based system.

A set of organizing principles found helpful in accomplishing this system engineering has been described:

- The design should start with a general framework which is then tailored as needed.
- In order to implement a complex system, all the supporting levels (hardware through programming language) must be subject to change.
- The application to be implemented on top of the programming system level should be implemented as a series of levels, rather than as a single level.
- Facilities for analysis and debugging must be an integral part of the system design from the beginning.

By following these principles, we have found that a complex system can be constructed and can evolve without becoming an unmanageable kludge.

## ACKNOWLEDGMENTS

Rick Fennell, Greg Gill, Gary Goodman, Richard Neely and others have contributed to the tools developed in the course of the Hearsay efforts. Bill Broadley, George Robertson, Jim Teter, Howard Wactlar, and many others have expertly made the hardware and operating system changes changes described here. The developers and maintainers of SAIL at Stanford University have been very responsive over the years -- Jim Low, John Reiser, Hanan Samet, Bob Sproull, and Dan Swinehart. This paper has benefited greatly from readings by Lee Cooprider, Mark Fox, Don McCracken, John McDermott, and Jack Mostow.


## REFERENCES


CMU Computer Science Speech Group (1977) Summary of the CMU Five-year ARPA effort in speech understanding research. Tech. Report, Comp. Sci. Dept., Carnegie-Mellon Univ.

Cooprider, L. W. (1978, to appear) Representation of families of systems. Ph.D. thesis, Comp. Sci. Dept., Carnegie-Mellon Univ.

Dijkstra, E. W. (1968) The structure of the "THE"-multiprogramming system. *Comm. ACM, 11*, (5), 341-346.

Erman, L. D. (1974) An environment and system for machine understanding of connected speech. Tech. Report, Carnegie-Mellon Univ. (Ph.D. Dissertation, Comp. Sci. Dept., Stanford Univ.).

Erman, L. D. and Lesser, V. R. (1975) A multi-level organization for problem solving using many diverse cooperating sources of knowledge. *Proc. 4th Inter. Joint Conf. on Artificial Intelligence*, Tbilisi, USSR, 483-490.

Flon, L. (1975) Program design with abstract data types. Tech. Report, Comp. Sci. Dept., Carnegie-Mellon Univ.

Guttag, J. V. (1976) Abstract data types and the development of data structures. *Comm. ACM, 20*, (6).

Habermann, A. N., Flon, L., and Cooprider, L. W. (1976) Modularization and hierarchy in a family of operating systems. *Comm. ACM, 19*, (5), 266-272.

Habermann, A. N. (1978, to appear) On system development control.

Lesser, V. R., Fennell, R. D., Erman, L. D. and Reddy, D. R. (1975) Organization of the Hearsay-II speech understanding system. *IEEE Trans. on ASSP 23*, 11-23.

Lesser, V. R. and Erman, L. D. (1977) A retrospective view of the Hearsay-II architecture. *Proc. Inter. Joint Conf. on Artificial Intelligence*, Cambridge, MA, 790-800.

Liskov, B. (1972) The design of the VENUS operating system. *Comm. ACM, 15*, (3), 144-149.

Liskov, B. and Zilles, S. (1974) Programming with abstract data types. *SIGPLAN Notices, 9* (4), 50-59.

Newell, A., Barnett, J., Forgie, J., Green, C., Klatt, D., Licklider, J. C. R., Munson, J., Reddy, R. and Woods, W. (1973) *Speech Understanding Systems: Final Report of a Study Group*. North-Holland.

Parnas, D. L. (1972) On the criteria to be used in decomposing systems into modules. *Comm. ACM, 15*, (12), 1053-1058.

Reddy, D. R., Erman, L. D., Fennell, R. D. and Neely, R. B. (1973) The Hearsay speech understanding system: an example of the recognition process. *Proc. 3rd IJCAI*, Stanford, CA, 185-193.

Reiser, J. F. (1976) SAIL. Stanford Artificial Intel. Lab., Memo AIM-289.

Wulf, W. A., London, R. A., and Shaw, M. (1976) An introduction to the construction and verification of Alphard programs. *IEEE Trans. Software Eng., 2*, (4), 253-265.