MULTIPROCESS SOFTWARE ARCHITECTURE
FOR
AI PROBLEM SOLVING

Richard Dean Fennell

May 1975

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Submitted to Carnegie-Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was supported in part by the Defense Advanced Research Projects Agency of the Office of the Secretary of Defense (Contract F44620-73-C-0074) and monitored by the Air Force Office of Scientific Research. This document has been approved for public release and sale; its distribution is unlimited.
ABSTRACT

This dissertation describes the design and development of a knowledge-based artificial intelligence problem-solving organization that is suitable for efficient implementation on a closely-coupled multiprocessor computer system. The method is a result of formulating the problem-solving organization in terms of the hypothesize-and-test paradigm for heuristic search, with communication between the various hypothesizers and testers being effected by writing intermediate results in a shared blackboard-like data base. These hypothesizers and testers are expressed in terms of knowledge sources which represent bodies of suitably organized subject-matter knowledge pertinent to the task domain of the problem being solved. The various system organization problems connected with such a multiprocessing scheme are discussed, and solutions to these problems are presented. The major contributions of this work lie in the analysis and solution of the various multiprocessing problems that have arisen in the course of specifying this problem-solving organization.

Particular attention has been paid to resolving the data access synchronization problems and data integrity issues arising from the use of asynchronous parallel processes as hypothesizers and testers. A method for efficiently controlling access to dynamically expanding multidimensional shared data bases is developed using the notion of abstract data regions in conjunction with previously created data resource objects; and mechanisms for localized data event retention and global data base tagging are presented by which individual processes may execute in parallel within their own local contexts while being able to keep track of concurrent modifications being made to the shared data base. Mechanisms for data-directed process invocation and goal-directed scheduling are also presented in response to the requirement that, for performance evaluation reasons, hypothesizers and testers be independent (yet cooperating) asynchronous processes. Data-directed process invocation relies on an ability to monitor the blackboard data base in a continuous manner, waiting for data events and data base patterns which will trigger the processing activity of the various knowledge sources; and goal-directed scheduling involves the mapping of the virtual parallel activity created by the data-directed process invocation onto the available hardware resources so as to schedule those processes first which can best help in advancing the overall problem solution.

The Hearsay II speech-understanding system (HSII), which has been developed at Carnegie-Mellon University using the techniques for system organization described herein (this system organizational philosophy also being known as HSII), provides a source of concrete examples.
ACKNOWLEDGEMENTS

I wish to express my sincere appreciation to my thesis advisor, Raj Reddy, for his unfailing encouragement and guidance throughout the course of this research, and to the other members of my thesis committee, Sam Fuller, Lee Gregg, and Victor Lesser, for their criticisms and helpful suggestions.

Special thanks are due all the members of the Computer Science Department Hearsay II speech-understanding research group for submitting themselves (hopefully willingly) to the trials and tribulations of performing their own research within the Hearsay II operating system during the development stages of this system organization. I would like to offer particular thanks to Lee Erman and Victor Lesser for their major roles in the design, implementation, and continuing development of the Hearsay II system, as well as for many fruitful discussions on the various issues of software architecture and system design. Thanks also to Gregory Gill for his endless efforts and assistance in the system implementation.
# TABLE OF CONTENTS

## INTRODUCTION

The Problem ................................................................. 1
- perceptual and intellectual activities
- knowledge-based problem-solving
- knowledge sources
- historical perspective

Goals for a Knowledge-Based Problem-Solving System .................. 4
- employ independent, multiple, diverse sources of knowledge
- knowledge source cooperation via hypothesize-and-test paradigm
- knowledge source cooperation for error detection and correction
- intuitive knowledge source decomposition
- facilitate knowledge source performance evaluation
- provide framework for easy integration of new knowledge
- provide common basis for inter-knowledge source communication (blackboard)
- design for multiprocess and asynchronous parallel processing

Description of Terms .......................................................... 7
- AI problem-solving ....................................................... 7
- software architecture .................................................... 9
- multiprocessing .......................................................... 10

Approaches to a Solution .................................................. 12
- multiprocess-oriented problem-solving ................................ 12
- choosing a search strategy ............................................... 14
  - the knowledge source as an elemental processing unit
  - parallel processing techniques
  - hypothesize-and-test paradigm
- software requirements in problem-solving organizations .......... 19
  - data types and data structures
  - processing elements: specification and control
Parallelism in Knowledge-Based Problem-Solving ........................................ 23
- closely-coupled and loosely-coupled process and processor structures
- parallelism in preprocessing
- parallelism in knowledge source decomposition
- parallelism in hypothesization and verification

Overview of the HSII Problem-Solving Organization ................................. 30
- uniform, multilevel structured data base
- many small independent knowledge sources
- data-directed control using hypothesize-and-test paradigm
- simplified message transaction diagram for HSII
  - blackboard, blackboard handler, and blackboard monitor
  - knowledge sources and preconditions

Organization of the Thesis ................................................................. 33

THE MODEL

An Abstract Model for Problem-Solving .................................................. 34
- production systems
- design choices in production systems
- consequences of making those design choices

The HSII Problem-Solving Organization: A Production System Approach ....... 37
- the HSII design choices
  - their bases
  - their consequences
- detailed message transaction diagram for HSII
  - blackboard, blackboard handler, lock handler, and blackboard monitor
  - knowledge sources and preconditions, with local contexts
  - goal-directed scheduler

THE COMPONENTS OF HSII

Knowledge Sources and Preconditions ................................................. 42
- knowledge representation
  - function
  - structure
  - cooperation
  - attention focusing
- knowledge specification
  - precondition
  - effects
  - algorithm
- preconditions, knowledge sources, and data events
- sample knowledge sources and precondition

The Shared Data Base ............................................... 47
- blackboard design .................................................. 47
  - allows homogeneous representation and anonymous communication of knowledge at all information levels
  - maintains dynamically current global data
  - minimizes duplication of information by suitable structural organizations
  - makes new results, as from modification of a data node, immediately available to all uses of that node
  - minimizes recalculation of information by suitable state-saving mechanisms
  - permits dynamic focus of attention
  - allows data-directed knowledge source activation by blackboard monitoring
- problems to be resolved
  - data access synchronization
  - data deadlock avoidance
  - preservation of process-dependent local state
  - preservation of data event information
  - deletion of data from a shared structure
- blackboard structure ................................................. 54
  - knowledge decomposition into information levels
  - homogeneous data elements: hypotheses and links
- node structure .......................................................... 57
  - attribute classes
  - structural relationships
- associative data retrieval ............................................. 60
  - matching-prototypes
  - data-directed knowledge source activation

HSII MULTIPROCESSING MECHANISMS

Mechanisms ............................................................. 63
- to avoid direct interprocess interference: local contexts
- to avoid indirect interprocess interference: data access synchronization
- to allow direct interprocess communication: no direct messages
- to allow indirect interprocess communication: blackboard monitoring
- primitive data access operations and local context maintenance ......... 100
  - read/write a field: global
  - read/delete old field: local
  - retain old field, receive a message: local
  - create/delete data node: global

Knowledge Source and Precondition Processes ........................................... 104

- knowledge source and precondition specification ................................. 105
  - local context specification
  - algorithm specification and integrity assumptions
- knowledge source and precondition activation and execution .................. 107
  - precondition activation relies on blackboard monitoring
  - knowledge-source activation relies on precondition doing associative
data retrieval from blackboard and its local context
  - information transfer from precondition to knowledge-source process
  - knowledge-source process execution
  - indirect interprocess communication
- knowledge-source and precondition process control .............................. 109
  - goal-directed scheduling
    - scheduler performs evaluation functions and
determines processing power allocation strategy
  - attention focusing
  - policy-type knowledge sources
  - decoupling of focusing strategy from knowledge-source activity
  - decoupling of data environment from control flow
  - attention refocusing for knowledge-source error recovery

EXPERIMENTS WITH AN IMPLEMENTATION

HSII Multiprocessing Performance Analysis ............................................. 115

The HSII Speech Understanding System: The Simulation Configuration ............ 118

Simulation Mechanisms and Simulation Experiments .................................. 120

Primitive Operation Timings ................................................................. 121

Effective Parallelism and Processor Utilization ....................................... 127

Execution Interference Measurements ...................................................... 157
CONCLUSIONS and FUTURE RESEARCH

A Summary .......................................................... 170

- shared data and data access synchronization
- asynchronous processing, data integrity, and local contexts
- data monitoring and data-directed process activation
- process control and goal-directed scheduling
- simulation results

Directions for Future Research and Concluding Remarks ........... 174

SELECTED REFERENCES ........................................... 177

APPENDICES ....................................................... 181
INTRODUCTION

The Problem

Artificial intelligence (AI) offers many interesting challenges in the area of computer emulation of human capabilities. Not only does AI inquire as to the psychological and physiological mechanisms underlying human perception and thought processes, but there is also a desire to emulate merely the external behavior of these processes. Two broad categories of human activity of special interest to AI researchers are intelligent behavior and perceptual activities. Playing chess or proving mathematical theorems are examples of human capabilities that are generally agreed to exhibit intelligent behavior; watching a film or listening to a conversation are examples of perceptual activity, which tasks humans seem to perform without any conscious effort. The question naturally arises as to whether a human uses different mechanisms for perceptual and intellectual activities: how is it that perceptual activities are performed seemingly effortlessly, while intellectual activities are not? While it is not known at present whether a human being uses different types of mechanisms for these two task categories, one conjecture is that he does not: it is just a matter of how effectively he is able to use the available mechanisms. For example, for perception tasks involving speech or vision, the human probably has many different representations of the input data (resulting say, from previous occurrences of a phrase often used in everyday conversation, or from many different observations of the same scene (Minsky, 1974)); the perception activity is perhaps then accomplished by being able to access and process these stored representations in parallel. Perhaps perceptual activities are performed with such facility only after the accumulation of many past experiences and
only when these experiences have been organized so as to form patterns or rules which can then expedite the processing of similar future inputs. If this conjecture is true, then a person should, for example, be able to play master's level chess if he begins to learn to play chess when he is very young and continues to devote a major part of his waking life to playing chess for several years. Thus the difference between perceptual and intellectual activities may be that perceptual activities are characterized by suitably organized subject-matter knowledge which can be quickly accessed as a consolidated source of knowledge rather than having to "derive" this knowledge using more generalized (and weaker) conceptual mechanisms, as might be the case in intellectual activities. Of course, the collection of human activities is not neatly divided into two parts, intellectual and perceptual; but rather these categorizations typify the extremes. In the middle lie activities performed with varying degrees of ease, depending on the individual involved and his accumulated experience with the given task. Thus, a chess problem that might be an extremely intellectual task for a chess novice would be more a perceptual activity for the chess master, the master having previously accumulated and organized the subject-matter knowledge necessary for efficiently performing this task. In order to solve a problem in an effective manner, "elements of general heuristic power are auxiliary to, but not substitutable for, subject-matter knowledge appropriately organized [in productions]." (Simon, 1971).

The object of the research reported herein is to define an AI system organization suitable for expressing knowledge-based problem-solving strategies so that "appropriately organized subject-matter knowledge" may be represented as knowledge sources capable of contributing their knowledge in a parallel data-directed fashion. A knowledge source may be described as an agent that embodies the knowledge of a particular aspect of a problem domain\(^1\) and is useful in solving a problem from that domain by performing actions based on its knowledge so as to further the progress of the overall solution. It is felt that the knowledge source is an appropriate unit for use in the decomposition of a knowledge-intensive task domain. Knowledge sources, being suitably organized capsules of subject-matter knowledge, may be independently formulated as various pieces of the knowledge relevant to a task domain become crystallized. It remains to specify a system organization in which these many

\(^1\) For purposes of this discussion, the specification of a knowledge source can be considered to be static; i.e., whether a knowledge source learns from experience is an issue that is orthogonal to this organization.
independent and diverse sources of knowledge may be specified and their interactions coordinated so they might cooperate with one another (perhaps asynchronously and in parallel) to effect a problem solution. A particular system organization, that used as the basis for the Hearsay II speech-understanding system (HSII), is the subject of this report.

Much of the work presented in this report was based on the development of several generations of software systems for speech-understanding research; but the resultant system organization described herein is also felt to have more general applications for expressing knowledge-based AI problem-solving strategies (as might be found, for example, in vision (Reddy, 1973a; Ohlander, 1975), robotics, chess, natural language understanding, and protocol analysis).

The efforts described here follow from the early work of Reddy (Reddy, 1966) and Reddy and Vicens (Vicens, 1969), whose work in speech-understanding research provided a basis for a knowledge-based model of speech understanding. The report of Newell, et al., (1971) discusses many of the factors influencing the structure and operation of a speech-understanding system. The Hearsay I system (Reddy, et al., 1973b; Reddy, et al., 1973c; Erman, 1974) was a system design intended to respond to many of the organizational issues described in the Newell report. Hearsay I was the first demonstrable connected-speech understanding system, being publicly demonstrated in June, 1972. While incorporating most of the basic mechanisms required for representing a knowledge-based problem-solving system, Hearsay I had several organizational drawbacks, despite its rather impressive capabilities for understanding connected speech. Hearsay II (Erman, et al., 1973; Lesser, et al., 1974; Fennell and Lesser, 1975) was designed to overcome the organizational insufficiencies of Hearsay I by significantly generalizing the capabilities provided by the system for the representation and cooperation of knowledge sources. In particular, the major areas of organizational improvement of Hearsay II over Hearsay I included: a) the ability for knowledge sources to operate and communicate with one another at a level of information representation appropriate to the individual knowledge sources (rather than having all interprocess communication be expressed in terms of a standard data unit, which unit was the "word" in Hearsay I); b) the ability to retain alternative search paths within a global data structure so as to reduce the amount of recalculation effort necessary in pursuing a more classical backtracking search strategy, as was used in Hearsay I; and c) the ability for the knowledge sources to execute in an asynchronous,
parallel manner so as to permit a more flexible data-directed implementation of the
hypothesize-and-test heuristic search paradigm used by both Hearsay I and Hearsay II. 
The knowledge-based problem-solving organization presented in this dissertation is a
result of the design of a structured environment in which to express the system
organization of the Hearsay II speech-understanding system, this more general
problem-solving organization is also referred to as HSII.

Goals for a Knowledge-Based Problem-Solving Organization

The development of the Hearsay II speech-understanding system is part of a
major undertaking at Carnegie-Mellon University involved with the AI problem of the
machine emulation of human perceptual capabilities. The basic system philosophy of
HSII closely parallels that of its predecessor, Hearsay I (Reddy, et al., 1973b); this
system philosophy is based on the belief that the inherently errorful nature of the
information flow during the perception process (and, in particular, during the processing
of connected speech) can be handled only through the efficient use of multiple, diverse
sources of knowledge, cooperating through a generalized mechanism such as the
hypothesize-and-test paradigm (Newell, 1969) for heuristic search. Thus it is the goal
of the HSII design (as well as having been the goal of Hearsay I) to provide an
operating system environment within which a collection of such knowledge sources
could be easily integrated so as to cooperate with one another in efficiently pursuing
the task of emulating the human perceptual process.

The computational grain for a knowledge source is to be intuitively defined,
balancing intuition with the constraints presented by the known hardware resources
upon which the problem-solving system would execute and with the software
constraints arising from the costs of process control and interprocess communication.
The decomposition of the task into knowledge sources is intended to be a natural
decomposition, with each knowledge source representing a collection of knowledge
about a particular area of the task environment. Knowledge sources should be
expressible in an algorithmic format; and the conditions upon which a knowledge source
is to be executed should be specified in terms of the dynamic problem data state. The
application of this data-oriented method of specifying the conditions necessary for

1 Others have also proposed structured environments for speech-understanding
systems. In particular, see Barnett (1973, 1975) and Rovner, et al., (1974); Baker
(1974) proposes a highly structured system based on a simple Markov model.

4
knowledge source execution is referred to as *data-directed knowledge source invocation*. The system environment provided for supporting knowledge source actions should permit an arbitrary number of knowledge sources to be combined, these knowledge sources perhaps being independently written by many different people. For reasons of knowledge source performance evaluation, it is required that it be possible to configure, in a flexible manner, a problem-solving system using any (reasonable) subset of the total available knowledge sources, either statically, at system generation time, or dynamically, as the system is running. Furthermore, the absence of one or more knowledge sources should not have a crippling effect on the performance of the system. On the other hand, the addition of knowledge sources should lead to a greater improvement in system performance (by virtue of the increased knowledge source interaction brought about by the addition of the knowledge source) than is possible to attain by the use of any subset of the sources of knowledge. These objectives require "pluggable" knowledge sources which are quite independent of one another (or clusters of knowledge sources which are independent of the remainder of the knowledge sources). Any given knowledge source should not presume the existence of any knowledge source outside of its own cluster, thereby facilitating the writing of separate knowledge sources by separate people, without the need for undue collaboration between the various people involved. However, in order for such elements of processing capability to cooperate with one another in the anonymous fashion required by their legislated independence, a centralized means of indirect interprocess communication is necessary. A blackboard-like shared data base was selected as the mechanism for providing this centralized and indirect interprocess communication; this blackboard-like data base would also serve as the basis for the data-directed knowledge source invocation facility mentioned above. Knowledge sources must cooperate with one another, not only to progress the state of the problem solution, but also to detect and resolve ambiguities and errors that might be introduced at every stage of processing: the information flow among knowledge sources during the processing must be assumed to be inherently errorful, and means must be provided for achieving adequate cooperation among knowledge sources so that any errors introduced may be detected and resolved. The hypothesize-and-test paradigm for heuristic search represents an elegant way of obtaining this cooperation in a uniform manner, while at the same time allowing for the receiving knowledge-source anonymity necessary for use in an environment where the activation circumstances of the hypothesizers and testers are expressed in a data-directed manner.
Due to the inherent complexity of most AI problem-solving systems and to the many computation steps required to solve most problems of interest, any implementation of an AI problem-solving system has to be efficient in its computation. Not only is this true for a "production" application system, but it is also necessary for development versions of the system to be reasonably efficient because of the experimental way that a knowledge-based system is developed: the development and evaluation of the knowledge sources and control strategies require repeated runs over significant amounts of test data. Despite efforts to achieve an efficient computer implementation, many artificial intelligence problem-solving tasks still require large amounts of processing power in order to achieve solution. The amount of processing power required is directly related to the size of the search space which is examined during the course of problem solution. Exhaustive search of the state space associated with almost any problem of interest is precluded due to the sheer size of the space.\(^1\) In most problem-solving attempts, heuristics are employed which prune the search space to a more manageable size. However, searching even the reduced state space often requires large amounts of processing power. The demand for sufficient computing power becomes critical in tasks requiring real-time solution, as is the case in the speech understanding task with which this research is primarily concerned. For example, a speech understanding system capable of reliably understanding connected speech involving a large vocabulary and spoken by multiple speakers is likely to require from 10 to 100 million instructions per second of computing power, if the recognition is to performed in real time.\(^2\) Recent trends in technology suggest that this computing power can be economically obtained through a closely-coupled network of asynchronous "simple" processors (involving perhaps 10 to 100 of these processors), (Bell, et al., 1973) and (Heart, et al., 1973). The major problem (from the problem-solving point of view) with this network multiprocessor approach for generating computing power is in expressing the various problem-solving algorithms in such a way as to exhibit a structure appropriate for exploiting the parallelism available in the multiprocessor network, for it is only by taking advantage of this processing parallelism that the

\(^1\) As an example, consider the chess-playing task. In an end game situation, there are typically 20 legal moves at each ply (half-move); so for a search depth of 6 plies, the search space will have 64 million branches.

\(^2\) The Hearsay I (Reddy, et al., 1973b) and Dragon (Baker, 1974) speech understanding systems require approximately 10 to 20 mips of computing power for real-time recognition when handling small vocabularies.
desired effective computing power will be achieved. Thus, the problem-solving organization proposed herein is designed to allow as much asynchrony as possible in the specification and control of the collection of knowledge sources. Certainly, the organization has to provide sufficient system facilities to allow the implementation to be carried out in a true asynchronous (multiprocessor) environment. It is also required that the implementation of any heuristic search technique used in coordinating the problem-solving activities be done in a way that can take maximal advantage of any physical asynchronous processing capabilities that might be made available by the hardware configuration.

The HSII design philosophy is the result of trying to satisfy these goals, given the hardware environment of a closely-coupled multiprocessor; and the HSII speech-understanding system is the concrete result of applying this design to the AI task of connected-speech processing.

This dissertation describes the design and development of a knowledge-based problem-solving organization that is suitable for efficient implementation in a closely-coupled multiprocessor environment. The major contributions of this work lie in the analysis and solution of the various multiprocessing problems that have arisen in the course of specifying this problem-solving organization. Contributions have also been made in defining techniques for process decomposition and knowledge decomposition for knowledge-based multiprocess AI problem-solving.

Description of Terms

The more global objectives of this research can perhaps best be described by a discussion of the various terms used in the title of this dissertation, "Multiprocess Software Architecture for AI Problem Solving."

AI Problem Solving

A primary concern of AI is to discover ways to emulate various functions of the human mind so as to perform these same functions without the aid of a human (or perhaps with minimal human intervention). The purpose of such efforts is manifold. Some wish to investigate human thought and perceptual processes by modeling these functions in terms of information processing concepts so as to build up a computational model which exhibits the same characteristics as its human counterpart. The effort to
model the human as a complex information processing system has theoretical psychological implications in that not only is the computational model intended to mimic human thought processing as might be viewed by an external observer, but it is also intended to provide a psychological model for how the human goes about exhibiting such behavior.

A different motivation for developing a machine emulation of human capabilities derives from the desire to improve the productivity of the human by relieving him of mundane tasks (especially tasks to which the human is prone to error or in which he is exposed to physical danger). Human productivity might also be increased by improving the man-machine interface between a human operator and a non-human task processor. With this alternative motivation, the means by which the human function is mimicked is of much less importance than the resulting external behavior of the emulation process. For example, in perception tasks, if a machine is to provide an effective means by which to improve the efficiency of a human operating in such a situation (or even replace the human), then the machine's ability to perform the task of perception (for example, understanding a spoken sentence or analyzing a visual scene) must be externally equivalent to that of a human, insofar as the equivalence is necessary in performing the given task. That is, the task may require that the machine accomplish its perception computation such that its external behavior rivals the facility with which humans could perform the same task. The latter motivation (that of emulating external human behavior without special regard to trying to propose any psychological model to describe the underlying human thought processes) is the one pursued by the problem-solving organization described herein. In particular, a framework is presented which is intended to serve as a generally useful and intuitive way to express and control sets of knowledge source algorithms which serve to mimic the external behavior exhibited by a human problem-solver. The framework is intended to be equally capable of coordinating knowledge sources involved in the solution of problems of perception (at which humans are quite successful and extremely efficient) or problems of thought (such as planning strategies or discovering a solution to a mathematical puzzle, at which humans are not always so facile). If, however, from this attempt to create a rather intuitive computational framework in which to express problem-solving systems as computer algorithms, an underlying psychological model for human behavior arises, then so much the better: a psychological model for explaining external behavior will have been proposed by studying the more apparent structure of
a mechanical information processing system which attempts to mimic external human behavior, and in the process of which perhaps models the psychological functions underlying that problem-solving behavior.

Software Architecture

In a general sense, the word "architecture" involves the formulation and application of various design principles to combine structural components in an orderly fashion so as to constitute a finished whole. These design principles may be determined by aesthetic considerations, as well as by practical or material considerations. Quite often, such design principles are derived by observing (and enhancing or simplifying or generalizing) an orderliness in existing structures. This perception of orderliness may apply to something which is naturally occurring (such as the orderliness of celestial orbits or of atomic structures\(^1\)), thereby producing a perceived architecture of the various aspects of nature (which architectural principles may then be implemented using the same physical laws which describe the orderliness of the natural structures); or the perception may apply to something which is created by man (such as is connoted by the everyday use of the term "architecture" to describe the structure and orderly design of buildings; or as may be used to describe the orderly organization of various structures created by man, whether such structures be concrete in their realizations or of a less tangible nature), thereby producing a perceived architecture of various of the organizational efforts of man. While the emphasis here is on describing a (man-made) architecture in which specific organizations of computer programs may be expressed so as eventually to execute and mimic human problem-solving activities, if this architecture results in perhaps also describing the natural psychological structures of human behavior underlying the problem-solving activity, then a pairing of the man-made architecture of computer software structures with the natural architecture of some human thought processes will be achieved. Like other forms of architecture, "software architecture" is concerned with the application of rational design principles to the creation of organizational structures from more elemental system components. In the

\(^1\) Jacob Bronowski, in his "Ascent of Man" BBC television specials on the growth and development of scientific knowledge, suggested that science merely models the orderliness of things humans cannot actually see or feel by observing the orderliness of those things that actually can be perceived by human faculties; so an orderliness is implied for atomic structures from observations of more humanly tangible occurrences, such as planetary orbits, by modeling the atomic structures based on planetary observations.
case of software architecture, the elemental system components include computer software constructs, such as mechanisms for algorithm specification, program control structure specification, and data structure definition. This report deals with the description of a particular software architecture, an organizational philosophy for computer software designed to aid in the rational creation of particular program structures which are primarily intended to express algorithmic solutions for a variety of problem-solving tasks.

Multiprocessing

Unfortunately, the term "multiprocessing" has two rather distinct meanings within the field of computer science, one being hardware oriented and the other being software oriented. The first definition of multiprocessing is related to the hardware utilization of a multiprocessor computer, in which the various available physical resources are shared among the running programs which are competing for these resources (the central processing units being considered as resources, just as are primary memory and disk channels). The "multi" in this use of "multiprocessing" refers primarily to the availability of multiple physical processing units by means of which several processes\(^1\) may execute in parallel (one process per processor at any given instant of time); that is to say, there are multiple physical program counters. In this first use of the term, "multiprocessing" is principally the application of multiprogramming techniques (that is, the efficient allocation of scarce hardware resources to competing running programs, or processes) to multiprocessor hardware organizations. Note that this definition says nothing in particular about any interrelations among the various running programs which are competing for the resources; such things are not especially an issue from the point of view of a supervisory program which is merely trying to satisfy outstanding requests from processes for hardware resources.

While the software organization under consideration here is indeed concerned with running on multiprocessor hardware configurations, there are additional considerations implied by the term "multiprocessing." If instead of focusing on the hardware aspects, one applies the "multi" of "multiprocessing" to the software concept

---

\(^1\) A process may be taken to be an instantiation of a program; that is, a process is a program plus enough state information regarding this particular execution of the program (e.g., an instruction counter, program status words, execution stacks, local data) such that the instantiation may be executed independently of and potentially concurrently with other similarly constituted processes.
of a process, the notion of a system of interacting program instantiations -- multiple processes -- arises. The desirability of a collection of such processes can be derived from several sources. If, indeed, one has the availability of a multiprocessor structure and would like to take advantage of such a connected machine organization, then it might be appropriate to decompose any given suitable task into a set of relatively independent processes which could be spread over the multiprocessor and run in parallel so as to achieve a (hopefully) more effective solution (in terms of throughput, say) to the given problem. Thus far, the motivation for process decomposition has been presented as mostly hardware driven: given a multiprocessor hardware configuration, decompose a problem so as to adapt it for solution on the given hardware.

There are also other motivations for desiring a decomposition of a task into a collection of processes. Oftentimes, complex tasks are of the nature that they can be divided into subproblems, each of which can be solved fairly independently of the other subproblems of the task. This decomposition of a given task into component subproblems is, of course, task dependent, as well as often being dependent upon the hardware upon which the resulting programs are eventually intended to run. The main point is that higher level motivations (such as at the conceptual level of the task to be solved, or at the problem-solving organizational level, or at the research management level) often influence the desirability and effectiveness of a task decomposition into its subparts. Thus, even without the motivation of having to run in a multiprocessor environment, the concepts of multiprocessing (in the second, software interpretation) might be quite applicable. Of course, the actual execution of a system organized using a multiple process approach (wherein the various processes could conceptually be run in parallel) does not gain anything with respect to problem solution time when it must in actuality be run on a uniprocessor, but the various higher level decisions (such as those regarding conceptual system organization, performance evaluation, or division of labor) might justify pursuing the multiple process approach even in the uniprocessor hardware case. The "multiprocess" used in the title of this dissertation refers primarily to the more general software-oriented definition of multiprocessing; and the multiprocess software architecture described in the remainder of this dissertation may be rationally applied to either uniprocessor or multiprocessor hardware configurations, depending on the reasoning behind desiring a multiprocess approach to any given problem-solving system specification.
Thus, the goal of defining a "multiprocess software architecture for AI problem-solving" involves specifying an organizational computer software framework which takes advantage of multiprocessing concepts (both at the hardware multiprocessor level and at the conceptual design level) in order to derive a scheme suitable for expressing and solving various artificial intelligence tasks.

Approaches to a Solution

Many tasks in AI seem to lend themselves readily to multiprocess-oriented problem-solving techniques. As a result, there has been a recent increase in the number of languages intended for use in writing high-level problem-solving systems (Bobrow and Raphael, 1974). Many of these languages provide mechanisms for multiprocess or coroutine control structures, either explicitly or implicitly. Indeed, in the case of problem-solving systems which employ a search tree, the very nature of problem-tree searching, wherein during the exploration of one area of the tree one needs to retain state information regarding the progress of the search in other regions of the tree for later processing, seems to suggest a process-oriented control structure.

As further evidence that AI tasks often conform nicely to multiprocess-oriented solutions, consider various of the global strategies used in problem-solving. Many problem-solving methods in AI use the notion of trial-and-error search, searching for a solution in a space of possible solutions. Search-based problem-solving methods may be classified according to the way in which the search space is constructed, modified, or examined: that is, the problem-solving method is related to the creation and representation of the search space, as well as to the order in which the space is examined for a solution. Often the search space may be characterized as a tree of problem states; and the search problem is one of applying various operators to a given state in order to move to another state in the tree, so that the problem-solving task becomes one of determining which sequence of operators to apply to the root node in order to achieve a solution node.

A broad class of problem-solving strategies may be described (as by (Simon, 1971)) as model-manipulating systems, wherein the tree search may be viewed as a process of carrying a "model of a system" through possible "histories." That is, by applying various transition operators to nodes of the search tree, one moves to another node of the tree; and each path through the search tree (starting with the initial
problem state, or *root node*) represents a "history," or sequence of events which could lead to a particular terminal node.

Just to contrast for a moment, there are, of course, other global strategies in which to pursue problem-solving tasks. As an example, consider a theorem-proving system which derives its results via the resolution method (Pirotte, 1973). Such a system might be termed a *reasoning system*, to use Simon's words, as opposed to the model-manipulating systems discussed above. The essential difference is that, in methods such as previously described which use search trees, the information at each node represents a possible state of the system during some particular sequence of transition operations: the validity of the information is contingent upon following that particular sequence of operations to that particular node. However, in the case of reasoning systems, which do not use the concept of a search tree in their global strategies, information is accumulated in a non-branching (non-contingent) way and may be interpreted as universally valid: the transformations employed are reasoning steps which derive new information from the old without making tentative decisions, thereby preserving the validity of all past information.

It seems that both model-manipulating systems and reasoning systems are amenable to multiprocess solution techniques. In the case of the model-manipulators, it seems quite reasonable to suggest searching various paths through the search-tree in parallel. The problem becomes one of coordination of the parallel control flow, especially with respect to the sharing of information discovered during the parallel execution. In the case of reasoning systems, such as might be used in theorem-proving tasks, it is again reasonable to suggest that various subtasks be defined and solved in parallel.

Thus, the use of parallelism might be exploited under various circumstances. If the problem presents *alternative choices* in determining the solution path, the alternatives may be explored in parallel (at the expense of requiring a more sophisticated control flow mechanism in order to coordinate this parallel search than would be required if each alternative were explored in turn). If the problem may be reduced to a set of *subproblems*, the various subproblems may be attacked in parallel (again, with an additional expense in coordination supervision if the various subproblems are not independent of one another, either by virtue of the information required during solution or the information gained upon completion of the subproblem, since solving one
subproblem may entail using information being generated in parallel by another subproblem, even to the point of curtailing execution based on external information generated in parallel).

The amount of control necessary to coordinate the various parallel activities is directly related to the amount of independence exhibited by the various individual activities. Such dependency relationships may be used to construct (perhaps only implicitly) an AND/OR graph which expresses the interdependence of the inputs and outputs of one activity relative to the inputs and outputs of the other activities which are to be proceeding in parallel. For example, in the case of parallelism arising from alternative choices, the completion of one of a set of parallel activities may be used to force the early termination of its alternatives. In the case of parallelism arising from subproblem definition, the execution of one of the parallel activities may be affected by the results of the execution of some other activity (e.g., modifications made to a shared data base by one activity might demand the actions of a parallel activity be altered to account for those modification). In general, interdependency relationships among parallel activities arise from the assumptions or conditions that one activity places upon the inputs or outputs of other activities. As stated separately by (Simon, 1962), (Alexander, 1964), (Parnas, 1971a), and (Lesser, 1972), in order to maximize the independence of parallel activities (and thereby minimize the inter-activity coordination necessary), one should seek a decomposition into activity clusters such that the information transfer between clusters is minimized. To best exploit parallelism in such tasks, each activity should minimize the number of assumptions made concerning the behavior of other activities with which the given activity is to be run in parallel. In any case, in order to supervise and coordinate such parallel activities, the various inter-activity assumptions made must be explicitly stated (or derivable), and it is through such a specification that one may devise an AND/OR graph to be used in the task of activity coordination and scheduling.

**Choosing a Search Strategy: The Hypothesize-and-Test Paradigm**

Selecting a problem-solving strategy which can be used in generating and searching a problem state space defines a basis for the organizational framework of a problem-solving system. Having chosen the problem-solving strategy, attention can be paid to identifying and solving the various problems arising from the actual implementation of the problem-solving strategy. In specifying the heuristic search
strategy to be followed by the HSII organizational framework, a generative approach will be taken, based on the organizational goals set forth above.

A search mechanism is desired which can coordinate the efforts of multiple, diverse sources of knowledge, the knowledge source having been chosen as the elemental processing unit responsible for manipulating any given problem-state representation so as to transform it into a successor state representation. Due to considerations of knowledge source performance analysis and the desire to take advantage of multiprocessing techniques and technologies, the search mechanism must be able to treat these knowledge-source units as independent, yet cooperating, processing elements; and as a result of this required independence, the determination of which processing elements are capable of executing at any given step in the search process must be data-directed in nature, being dependent upon the current problem-state representation. The use of data-directed, independent knowledge-source processing elements is amenable to parallel processing solution techniques, and any search strategy which is to guide the progress of the problem solution must be equally amenable to these parallel processing techniques.

Note that although the knowledge sources are to be independent of one another, they must still, of course, cooperate with one another if they are to achieve a joint solution to the given problem. However, equally important to cooperating with one another in order to progress the problem solution, the knowledge sources must cooperate with one another in detecting and resolving ambiguities and errors that might be introduced during the processing. The information flow during the problem solution process must be assumed to be inherently errorful, and any chosen search strategy must be able to allow sufficient cooperation among knowledge sources as to detect and resolve any such errors. This implies the (conceptual) existence of knowledge sources whose function it is to examine the dynamic problem-state representation and to detect and resolve (either by transformation actions of their own or by eliciting the actions of other knowledge sources) the ambiguities, inconsistencies, and errors in that problem-solving representation. Thus, these verifying knowledge sources work in conjunction with the hypothesizing knowledge sources in order to maintain consistency among the

---

1 Errors due to knowledge source actions might arise because of an incomplete or inaccurate understanding of the theoretical basis underlying the knowledge source, or because the knowledge source might be incompletely specified in its implementation, or because the data input to the knowledge source may itself be errorful.
problem-state transformations being executed by those hypothesizing knowledge sources. Hypothesizing knowledge sources perform contingent problem-state transformations, thereby altering the dynamic representation of the problem-solution state (by adding new guesses, or hypotheses, to the solution state); verifying knowledge sources evaluate these efforts, and perhaps modify the previously proposed hypotheses. The efforts of the verifying knowledge sources help to avoid the propagation of error, as well as to detect errors and correct them. Any chosen search strategy must permit this form of checks-and-balances among knowledge sources, allowing for rich connectivity among the various processes to provide the feedforward and feedback necessary in overcoming the errorfulness inherent in the knowledge-source hypothesization process.

Given these various constraints, a rather elegant strategy may be derived in which to express the generation and search of a problem state space which is to be used by a collection of diverse, independent (but cooperating), and inherently errorful knowledge sources; this same strategy is also amenable for solution in multiprocessing environments. The strategy that lends itself so well for use in knowledge-based problem-solving environments, while satisfying the constraints discussed above, is the hypothesize-and-test paradigm for heuristic search (Newell, 1969). Stated in a sequential way and using the the concept of the knowledge source as the elemental processing unit, the basic cycle of the hypothesize-and-test paradigm consists of some knowledge source generating an hypothesis, which is intended to be an educated guess representing the solution state (or some part thereof) of the given problem; other knowledge sources are then called upon to test and then verify or reject the hypothesis based upon their current knowledge and beliefs concerning the problem solution. Given the reactions of the testers to the hypothesis, another (perhaps the same) knowledge source proposes a new hypothesis (which might be a modification of the previous hypothesis) which is intended to be an improved guess representing the updated solution state of the problem. This cycle of hypothesizing and testing continues until the testing knowledge sources are satisfied with an hypothesis presented to them, and the problem is declared to have achieved a solution state; or the cycle may end in a failure to discover a solution if no such hypothesis is ever generated within an allowed processing time.

In actual use, the hypothesize-and-test paradigm need not be so rigid or
sequential in nature. In HSII, the roles of hypothesizers and testers are played by the knowledge sources applicable to a given task domain. It is not necessary that only one hypothesizer be active at any given moment, nor that only one verifier execute at a time, nor that a hypothesizer generate only one hypothesis at a time, nor that a verifier examine only one hypothesis at a time, nor that the cycle of hypothesize-verify-hypothesize be lockstepped across all available hypotheses. Given an adequate representational framework in which multiple hypotheses may be maintained and related to one another in appropriate ways, the hypothesize-and-test paradigm may be applied to the various hypotheses in a parallel fashion, allowing for the concurrent generation of hypotheses and the concurrent verification of these hypotheses (as well as of previously generated hypotheses). In this way, the hypothesize-and-test paradigm allows the knowledge source structure of a problem-solving organization to take maximal advantage of any multiprocessing capabilities (whether hardware or software) that might be provided in an actual implementation. To satisfy the constraints that the individual knowledge sources be maximally independent of one another, while at the same time cooperating with one another in effecting a problem solution, the implementation of the hypothesize-and-test paradigm itself must make no assumptions about any relationships between an hypothesizing knowledge source and any verifying knowledge sources that might respond to its hypotheses; and no assumptions may be made about relationships between concurrently executing hypothesizing knowledge sources and concurrently executing verifying knowledge sources.

The decisions as to which hypothesizing knowledge sources or which verifying knowledge sources should be invoked at any given moment are entirely dependent upon the dynamic data state of the representational framework holding the various active hypotheses; this is in line with the requirement that the selection and activation of all processing elements (knowledge sources) be done in a data-directed manner, so as to preserve knowledge source independence. This data-directed invocation is achieved by having the individual knowledge sources (both hypothesizers and verifiers) specify abstract data conditions upon which they wish to be activated. Then, when such conditions actually occur in the hypothesis data base, the knowledge sources awaiting those conditions are automatically activated. Thus, the determination of which processing elements are capable of executing at any given step in the solution process is entirely data-directed in nature.
Given any actual implementation, there will be restrictions imposed upon the problem-solving organization regarding hardware resources that might be available for use in the problem solution. Thus, in specifying the problem-solving organization it is necessary to account for these restrictions by adding an additional level of control to the basic hypothesize-and-test paradigm. While the data-directed nature of hypothesize-and-test might suggest the set of knowledge sources capable of executing at any given moment, a goal-directed scheduling function must be added to the hypothesize-and-test paradigm in order to allocate the limited available processing resources first to those knowledge sources which can do the most to advance the state of the problem solution. It is the function of such a goal-directed scheduler to apply an evaluation function to the applicability of each knowledge source which is capable of executing and to match these knowledge sources to the areas of the hypothesis database which could benefit most from the actions of the knowledge sources. Through this matching process, which could entail some amount of means-ends analysis (by which the scheduler might decide to provide computing resources to some knowledge source merely as a means of causing further desired activity to be triggered based on the intermediate results of the initial knowledge source), the scheduler can allocate processing resources to knowledge sources in a goal-directed manner. Note that the importance-evaluation function of the goal-directed scheduler can also take into account factors relating to the error-correcting capabilities of knowledge sources (or, conversely, the likelihood that a given knowledge source might introduce error) in scheduling processing resources.

In summary, the search process used in the HSII organizational framework for knowledge-based problem-solving is based on the concept of the knowledge source, which represents an independent processing unit embodying the knowledge of a particular aspect useful in solving a problem of the chosen task domain. Knowledge sources are activated in a data-directed manner, based on the dynamic problem-solution state, and their individual actions are coordinated via the hypothesize-and-test paradigm for heuristic search, which is stated in sufficiently parallel terms as to allow the effective use of multiprocessor hardware and software architectures. The mapping between the virtual parallelism of the software multiprocessor organization and any actual parallelism offered by the hardware organization is accomplished by a goal-directed scheduling mechanism; this same mechanism evaluates and orders the processing resource requests of knowledge sources so as to result in efficient use of processing capabilities (even in uniprocessor hardware architectures).
Software Requirements for Problem-Solving Organizations

Consider now some of the software capabilities that might be of use in approaching an implementation of such multiprocess-oriented problem-solving systems as suggested above. In order to do this, a more general description of AI problem-solving will first be presented.

Conceptually, AI problem-solving may be divided into three parts: a) the task environment, b) the problem-solving organization, and c) the implementation. The task environment provides the problem to be solved, whether the environment be speech understanding, visual scene analysis, theorem proving, or game playing. Also contained within the task environment are various conceptual schemes which may be used in attacking the problem at hand, and various ideas and algorithms useful in determining the solution of the problem (or some of its constituent subproblems). For example, in the domain of speech understanding, phonetic classification of the various segments of the acoustic input may be determined by interpreting sound spectrograms. Or, in trying to understand what a speaker has said, it might be useful to have developed a dynamic user model based on the speaker's past behavior in such situations. In the realm of chess playing, certain board configurations suggest (or even demand) specific next moves, perhaps based upon the strategy of the players, their skill at the game, or the rules of the game itself. In addition to providing the problem to be solved, the task environment contains other necessary and relevant pieces of knowledge, such as various conceptual methods and schemes which might be useful in a machine-aided solution of the task.

However, without an appropriate structure within which to view the available or attainable knowledge, there is little hope of solving any given problem other than by chance. Methods of search and other problem-solving techniques can be fruitfully applied only to knowledge structures which are appropriately organized, and conceptual problem-solving methods which may be found within a task environment are of use only when they are appropriately formulated in precise algorithmic terms. Recalling a previous quotation, "elements of general heuristic power are auxiliary to, but not substitutable for, subject-matter knowledge appropriately organized [in productions]," (Simon, 1971). It is the purpose of a problem-solving organization to provide this necessary structuring of the various pieces of knowledge available within the task environment.
A problem-solving organization is intended to provide a structural basis for the various algorithmic or conceptually general mechanisms chosen and grouped together in order to attack a chosen task environment. In particular, a given problem-solving organization chooses among various alternative conceptual structurings which might be employed to solve the problem at hand, as for example, various problem-solving strategies, which may range from the very general (like hill climbing) to the more specific (like alpha-beta game tree pruning), or various methods of problem representation, again ranging from the very general (like connected graphs of problem states) to the more specific (like bit encodings of game board positions). Primary decisions in specifying a problem-solving organization involve choosing among the possible data representations and control structures suitable for the given task domain. Many of the mechanisms involved in specifying a problem-solving system relate to developing programming capabilities which seem suitable for handling the chosen task environment (or multiple task environments). The requirements of such a problem-solving system include: a) a means by which to encode and store facts and assumptions about the general task environment and the current state of the problem solution, b) a means by which to specify the active processing elements which interpret and manipulate the problem solution state so as to effect the problem solution, and c) a means by which to control the actions and efforts of the active processing elements. For example, choices of data representation may include whether to use linked lists or ordered n-tuples to represent some data structure; choices involving control structures include whether to represent various algorithmic processes as directly-callable routines or as asynchronous routines invoked by monitoring a data base, etc. It is the purpose of a problem-solving organization to provide the elements of information ordering (through the appropriate choice of data representation and control structures) necessary for the successful application of the chosen problem-solving technique.

Given any particular problem-solving organization scheme, an actual machine implementation will be strongly influenced by the hardware and software available for approaching the task. With respect to software capabilities which are desirable in facilitating the implementation of problem-solving systems, several broad categories of concern may be mentioned: a) data types and data structures, b) control structures for specifying processing elements, and c) control structures for controlling processing elements.
Data Types and Data Structures. Regarding data types and data structures, as noted in (Bobrow and Raphael, 1974), programs which are to deal in symbolic manipulation tasks rather than just in numeric quantities require data types suitable for such symbolic manipulation. Hence many computer languages which are used for AI problem-solving include facilities for handling lists, trees, strings, sets, and ordered n-tuples, with appropriate operations defined for the manipulation of these data types. In addition, some languages allow the user to define his own primitive data types (along with defining operations appropriate to these new data types). A need is also felt for providing higher level constructs than just these primitive data types: many applications demand rather large structures to be built from the primitive data types, which structures require mechanisms for their creation, accessing, and manipulation. Such is the trend of data base management systems, where the user is presented a homogeneous interface and the system handles the details involved in managing the arbitrarily large data base.

An additional requirement on any data base management scheme which is to be used in a problem-solving environment is that the data base be content addressable; that is, the user specifies a pattern to be matched, and an associative retrieval mechanism then searches the data base for an instance of the pattern (or a group of such instances), returning the various bindings found in the data base which satisfy the pattern. This feature of associative data retrieval is a necessity for systems in which the underlying data base is dynamically changing and the most current information is sought at any given moment during the computation.

Processing Elements: Specification and Control. The data structure requirements presented above form the information base necessary in which to record the transactions which take place during the course of the problem solution. Information contained in this data base facility may be static in nature (as might be the case in storing tables of rules to cover various situations which might arise during problem solution) or the information may be dynamic (as in recording the progress of the problem solution). Quite often the static information contained in the data base represents an encoding of knowledge regarding the task environment and various methods to be applied in order to transform the input data representing a particular problem so as to achieve a problem solution. To effect these transformations, a problem-solving system requires the specification of active processing elements which are capable of recognizing states in the dynamic sector of the data base to which the
various encoded rules may be applied, and effecting their application, resulting in a transformation of the data base. These processing elements themselves often contain an encoding of knowledge about the task environment, which may be used in conjunction with the encoded knowledge represented in the data base. Knowledge so encoded into the problem-solving processing element is often referred to as "procedural embedding of knowledge," (Winograd, 1971).

Processing elements may be implemented in a variety of ways, depending on the control structure chosen to guide the actions of these elements. For example, the active processing elements might be implemented as subroutines if the chosen control structure is to be centralized in a supervisory process; or a coroutine organization might be chosen if the various processing elements are to pass control from one to another in an explicit manner; or asynchronous parallel processes might be chosen as the means to express the processing elements if the various elements are to cooperate with one another in an anonymous manner; or a combination of these forms of activity control might be used.

The requirement that the active processing elements be responsive to conditions existing in the data base, which conditions may be subject to constant revision or may even be transient in nature, emphasizes the need for an associatively-based data retrieval mechanism whereby the processing elements are capable of specifying the particular data patterns in which they are interested. By applying these various patterns to the dynamic sector of the data base, the data points corresponding to pattern satisfaction provide the processing elements with the information necessary to apply a corresponding transformation rule, which presumably will create a new dynamic data state which will be matched by another transformation rule, until some transformation rule indicates that the problem has been solved.

As an example, a typical problem-solving organization might consist of a set of pattern-based rules (the static sector of the data base) which specify transformations to be performed on the dynamic sector of the data base when the patterns of the particular rules are satisfied by some component of the dynamic data base. The transformation functions of the rules might be procedurally encoded in a processing element, the function being activated based upon the dynamic state of the data base. This pattern-based activation of processing elements has been referred to previously as "data-directed invocation."
This characterization of AI problem-solving describes a class of organizations that might be classified as knowledge-based problem solvers. Such systems (of which HSII is an example) often rely on the actions of processing elements consisting of procedurally encoded knowledge (or some other form of static knowledge) being applied to a dynamic knowledge-oriented data base so as to transform that data base (by extending it or modifying it) until a goal state is achieved. The method of controlling these processing elements is usually one of data-directed invocation, where the processing elements effect their transformations based on the current state of the data base. Such knowledge-based problem solvers exhibit a great potential for the application of parallel processing techniques, as will be shown in the following sections.

Parallelism in Knowledge-Based Problem-Solving

Multiprocessing systems may be characterized according to the degree of interprocess interaction (coupling) of the concurrently executing processes in the system. This coupling is measured by the frequency of communication among processes with respect to the amount of computation time used by processes between their communication with one another. If processes tend to communicate with one another quite frequently relative to the amount of time spent in performing localized operations, then such a process structuring is termed closely-coupled; if the interprocess communication rate is relatively low with respect to the amount of time processes spend doing localized computation, then the process structuring is termed loosely-coupled.\(^1\) The terms closely-coupled and loosely-coupled can be applied to the hardware organization of a multiprocessor system, as well as to the software organization of a multiprocess system, the two classifications obviously being dependent upon one another in any actual multiprocessing system. Closely-coupled multiprocessor systems are those that permit the processes that are concurrently executing on separate processors to communicate with one another quickly, such as by sharing physical address spaces among processors or by directly connecting the processing units of those processes which wish to communicate with one another. Loosely-coupled

\(^1\) Notice that the phrase "interprocess communication" is being used in a general sense to differentiate between local and global activity. Thus, in the HSII system, for example, where direct interprocess communication is not actually permitted, the so-called interprocess communication is accomplished via indirect means, where a sending process can post the information to be communicated in a shared global data base (the blackboard) and a receiving process can poll or wait for such information.
multiprocessor systems are those in which the speed with which two processes executing concurrently on separate processors can communicate with one another is such as to discourage frequent interprocess communication: the various processors may be physically far apart as to increase the interprocessor communication time, or the communication paths between processors may be indirectly linked as to require message switching through the intermediate nodes of the interprocessor network.

Thus, for software multiprocess structures, "closely-coupled" refers to frequent interprocess communication; and for hardware multiprocessor structures, "closely-coupled" refers to low cost (speedy) interprocess communication. An analogous statement holds for the term "loosely-coupled." In choosing a software scheme that is compatible with an available hardware scheme, it is clear that using a closely-coupled or a loosely-coupled software scheme with a closely-coupled hardware scheme would be acceptable, as would a loosely-coupled software scheme with a loosely-coupled hardware organization; but combining a closely-coupled software scheme with a loosely-coupled hardware scheme would be inadvisable in terms of trying to attain processing throughput gains by utilizing the potential parallelism of the given software organization. The focus of the work reported herein is in devising multiprocess software organizations which are suitable for use on closely-coupled hardware configurations. While the use of loosely-coupled hardware organizations is certainly important in providing distributed networks of computing power (for example, the ARPA network (Roberts and Wessler, 1970)), their usefulness is limited in supporting closely-coupled software problem-solving organizations as are described here. As a result, little attention will be paid to resolving the problems arising from the use of such loosely-coupled network hardware organizations, although these problems will be discussed as they relate to the problems arising in closely-coupled systems.

As pointed out previously, many AI problem-solving tasks require large amounts of processing power in order to achieve solution in any given computer implementation of a problem-solving strategy. It has been suggested that closely-coupled multiprocessor organizations are capable of delivering the required computing power in a cost effective manner (Bell, et al., 1973; Heart, et al., 1973); but this computing power can only be used to maximal advantage if problem-solving organizations can be designed to take advantage of the parallelism offered by the hardware multiprocessor structure. A knowledge-based problem-solving organization as described above, which can be decomposed into a set of independent processes...
cooperating via the hypothesize-and-test paradigm, represents a natural structure for exploiting this hardware parallelism.

There are three major areas available for the exploitation of parallelism in the structure of such a knowledge-based problem-solving organization: a) input data preprocessing, b) specification of the processing algorithm for each source of knowledge, and c) utilization of the hypothesization and verification phases of the hypothesize-and-test paradigm.

**Parallelism in Preprocessing.** In perception tasks (such as visual scene analysis or speech understanding), the input data for a particular problem (scene or utterance) is presented in real-time at a very high data rate. A pipeline form of parallelism might be useful in performing data reduction preprocessing on this large volume of raw data. In the case of speech, this *preprocessing* task involves the repetition of a sequence of simple transformations on the acoustic input data (e.g., detection of the beginning and end of speech, amplitude normalization, preliminary phoneme-like labeling, segmentary smoothing), with little interaction between successive computations in the pipeline. The processing parallelism in such preprocessing tasks involves a fairly fine grain of computation: this suggests that such tasks could be best accomplished in a hardware environment specialized to the particular task, since the communication and synchronization costs of a more generalized multiprocessor network would likely overwhelm any gain achieved by the parallel decomposition.

**Parallelism in Knowledge-Source Decomposition.** The potential parallelism of a network of asynchronous processors can probably better be exploited at a more macroscopic level, such as at the level of the problem-solving organization, where the costs of communication and synchronization among the processes executing on the various processors will not overwhelm the benefits of the process decomposition. The issue is in choosing a grain of computation appropriate to the interprocess communication capabilities available (or to reduce the need to use such interprocess communication capabilities) such that the parallel decomposition results in a net gain in processing capability when compared to a uniprocessor solution, given equal economic constraints.

Problem-solving organizations which are based upon the use of *knowledge sources* seem to be capable of making effective use of multiprocessor machine architectures. The knowledge source has been described above as a suitable unit for
the conceptual decomposition of a problem-solving organization from an artificial intelligence point of view; this grain of decomposition, where a knowledge-source process would be the unit of scheduling for processing resource allocation, also seems suitable from a multiprocessing point of view for use on closely-coupled multiprocessor organizations. Given an adequate covering set of these knowledge sources and an appropriate control structure for coordinating the activities of these knowledge-source processes (which are the executing instantiations of the generic knowledge sources), a problem solution may be attained through the combined efforts of the constituent knowledge sources. Although the various knowledge sources are intended to cooperate with one another in order to effect a problem solution (having, in theory, been once decomposed from a more unified whole), they are not presumed to be inherently dependent upon one another; and, in fact, from a multiprocess system performance evaluation standpoint, it is desired that the various knowledge sources be fairly independent of one another so that easy introduction, replacement, and deactivation of the various knowledge source components may be performed so as to measure the contribution of each to the overall problem solution. Of course, the more independent the knowledge sources are, the less they will interfere with the execution progress of one another, which is clearly a desirable characteristic in terms of parallel activity. Hence, the problem-solving strategy becomes one of specifying a set of knowledge sources which are fairly independent of one another (and hence are amenable to execution in an asynchronous multiprocessor environment), and yet cooperate with one another, so as to use the potential parallelism provided by the hardware configuration in order to effect an overall problem solution.

Parallelism in Hypothesization and Verification. The hypothesize-and-test paradigm has been suggested as a suitable method for coordinating the efforts of the various knowledge sources in a knowledge-based problem-solving organization. This hypothesize-and-test paradigm for sequencing the activities of the knowledge sources can be expressed in a very parallel way. This parallel activity is generated by permitting the concurrent (parallel) processing of multiple data hypothesizations and multiple verifications of these hypothesizations (allowing further hypothesization to occur concurrently with the verifications). While the concurrent processing of multiple hypotheses could introduce unnecessary computation, in general it is expected that the parallel activity generated by this concurrent processing will result in a proportional speed-up in the problem-solution process. Correspondingly, concurrent hypothesization
and verification by all sources of knowledge also results in a proportional speed-up of the solution process because each source of knowledge is independent and is designed so that its knowledge contribution is additive. Combining these various forms of potential parallelism would seem to indicate that significant speed-up in the time required for problem-solution is possible, given an appropriate multiprocessor hardware architecture. A major software problem to be investigated in a parallel implementation of HSII is how to map virtual parallelism (process activity) into actual parallelism (processor activity) in an efficient way. This mapping problem in turn centers on three design issues, each of which relates to how processes interact:

1. The design of the interlock structure for a shared data base,
2. The choice of the smallest computational grain at which the system exhibits parallel activity, and
3. The techniques for scheduling a large number of closely-coupled processes.

The first design issue is important because in a closely-coupled process structure many processes may attempt to access a shared data base at the same time. In a uniprocessor system, the sequentialization of access to this shared data base does not significantly affect performance because there is only one process running at a time. In a multiprocessor system, however, if the interlock structure for a shared data base is not properly designed so as to permit as many non-interfering concurrent accesses as possible, then access to the shared data base becomes a significant bottleneck in the system's performance (McCredie, 1972). Of course, the costs of maintaining an intricate interlock structure must be balanced against the gains in processing concurrency achieved by offering such a synchronization structure.

The second issue relates to how closely-coupled processes may interact. If the grain of decomposition is such that the overhead involved in process communication is significant in relation to the amount of computation done by the process, the added virtual parallelism achieved by a finer decomposition can actually decrease, rather than increase, the performance of the system. Thus, the relationship between the grain of decomposition and the overhead of communication is an important design parameter. In the HSII design, interprocess communication is achieved indirectly by performing shared data base accesses, and the data-directed invocation of knowledge sources is a form of process communication. Refining the grain of knowledge source decomposition increases
the costs of data base monitoring (by increasing the number of knowledge sources requiring monitoring services), and hence may decrease the overall performance of the system, despite any increase in virtual parallelism created by the finer decomposition.

The third issue points out the scheduling problems inherent in a system in which processes are instantiated in an asynchronous and data-directed manner: the data monitoring mechanisms are responsible for instantiating knowledge-source processes whenever the data base contains appropriate data patterns, but it is up to a goal-directed scheduling algorithm to allocate the limited processing resources first to those instantiated knowledge sources which can best advance the problem solution. This goal-directed form of scheduling depends on the scheduler being able to make dynamic assessments of the relative importance of the various instantiated knowledge-source processes, as well as being able to employ various analysis techniques in order to predict and guide the progress of the solution state in an effective way.

This scheduling issue is also subject to a phenomenon called the control working set (Lesser, 1972). This phenomenon predicts that the execution of a closely-coupled process structure on a multiprocessor may result in a significant amount of supervisory overhead caused by a large number of process context switches unless adequate attention is paid to the interprocess communication traffic among the various processes. The reason for this potentially high rate of process context switching is analogous to the reason for thrashing within a data working set (Denning, 1968). For example, in a uniprocessor system, if two parallel processes interact closely with each other, then each time one process is waiting for a communication from the other, it would have to be context switched so as to allow the other process to execute. If these processes communicate often, then there would be a large number of context switches. However, if there were two processors available, each containing one of the processes, then there would be no context switching.

The implications of this phenomenon on constructing process structures include the following:

1. Processes should be collected into clusters where communication among cluster members is closely-coupled whereas communication among clusters is loosely-coupled. This process structuring paradigm has also been suggested as a model for the operation of complex human and
natural systems (Simon, 1962). The scheduler of a multiprocessor system must take into account this process clustering and use a strategy that schedules process clusters rather than single processes.

2. The size of a process cluster cannot be chosen independently of the particular hardware configuration that will be used to execute it. For example, a cluster size of eight may be appropriate for a hardware organization consisting of sixteen processors while being inappropriate for a system consisting of six processors.

3. The use of process structures to implement inherently sequential, though complex, control structures (e.g., coroutines) could lead to inefficient scheduling of process structures on a multiprocessor system; the scheduling strategy should be able to differentiate between those processes that can go on in parallel and those that are sequential in nature.

The Hearsay II speech-understanding system (HSII) is an implementation of a knowledge-based multiprocessing AI problem-solving organization based upon the process decomposition and system organization techniques described herein. This system organization philosophy is also known as HSII (having been derived during the course of designing the speech-understanding system). The HSII method of problem-solving system organization is intended to represent a problem-solving organization which is applicable to implementation in a multiprocessing environment, especially when the underlying hardware organization is that of a closely-coupled multiprocessor. An implementation of the HSII speech-understanding system is, in particular, currently being implemented on the C.mmp multiprocessor system (Bell, et al., 1971) at Carnegie-Mellon University. This dissertation explores various of the ramifications of such a problem-solving organization by examining the mechanisms and policies underlying HSII which are necessary for supporting its organization as a multiprocessing problem-solving system. The HSII speech-understanding system, which is being implemented and developed using the organizational principles espoused herein, provides a source of concrete examples. The following section will present an overview of this HSII problem-solving organization.
Overview of the HSI1 Problem-Solving Organization

HSII is based on the views that: a) the dynamic problem solution state of a knowledge-based problem-solving system can be represented in a uniform, multilevel database, and b) statically encoded knowledge which can determine transformations to be performed upon the problem solution state, given the dynamic solution state, can be characterized in a natural manner by defining many small knowledge sources. These knowledge sources will react to certain states of the data base by directing the transformation of this data base so as to progress towards a problem solution state. The hypothesize-and-test paradigm, when stated in sufficiently non-restrictive (parallel) terms, serves to describe the general interactions among these knowledge sources and their interactions with the solution state data base. In particular, changes caused by one or more knowledge sources may trigger other knowledge sources to react to these changes by validating (testing) them or hypothesizing further changes.1 The intent of HSII is to provide a framework within which to explore various configurations of information levels, knowledge sources, and global strategies.

From a more general point of view, the goal of HSII is to provide a multiprocess-oriented software architecture which is to serve as a basis for systems of cooperating (but independent and asynchronous) data-directed knowledge-source processes. The purpose of such a structure is to achieve effective parallel search over a general artificial intelligence problem-solving graph, employing the hypothesize-and-test paradigm to generate the search graph and using a uniform, interconnected, multilevel global data base as the primary means of interprocess communication.

One can derive from the description of the desired HSII problem solution process given above several basic components of the required system structure. First, a sufficiently general structured global data base is needed, through which the knowledge sources may communicate by inserting hypotheses and by inspecting and modifying the hypotheses placed there by other knowledge sources. Second, some means for describing the various knowledge sources and their internal processing capabilities is required. Third, in order to have knowledge sources activated in a data-directed manner, a method is required by which a set of preconditions may be specified.

1 Note that this use of an asynchronous parallel version of hypothesize-and-test leads to a system organization with some characteristics similar to QA4 (Rulifson, et al., 1973) and PLANNER (Hewitt, 1972). In particular, there are strong similarities in the data-directed sequencing of processes.
and associated with each knowledge source. The purpose of these preconditions is to monitor the dynamic problem solution state and activate their associated knowledge sources whenever the data state is such that the knowledge source might contribute to the progress of the solution. Fourth, in order to detect the satisfaction of these preconditions and in order to allow knowledge sources to locate parts of the data base in which they are interested, two mechanisms are needed: a) a monitoring mechanism to detect and record where in the data base changes have occurred and the nature of those changes, and b) an associative retrieval mechanism for accessing parts of the data base which conform to particular data patterns which may be specified by the knowledge sources or their preconditions.

The basic structure and components of the HSII organization may be depicted as shown in the message transaction diagram of Figure 1. The diagram indicates the paths of active information flow between the various components of the problem-solving system as solid arrows; paths indicating control activity are shown as broken arrows. The major components of the diagram include a passive global data structure (the blackboard) which contains the current state of the problem solution. Access to the blackboard is conceptually centralized in the blackboard handler process, whose primary function is to accept and honor requests from the active processing elements to read and write parts of the blackboard. The read-requests may be for explicitly specified data fields in the blackboard; or these read-requests may be implicitly specified, requiring an associative search of the blackboard. Write-requests are always explicitly specified. The active processing elements which pose these data access requests consist of knowledge-source processes and their associated preconditions. Preconditions (which act as relevancy tests for the application of the knowledge source associated with the precondition) are activated by a blackboard monitoring mechanism which monitors the various write-actions of the blackboard handler; whenever an event occurs which is of interest to a particular precondition process, that precondition is activated (via the control path indicated by the broken arrow from the monitoring mechanism to the precondition). If upon further examination of the blackboard, the precondition finds itself "satisfied" (that is, the data patterns of interest to the precondition's associated knowledge source are found to exist in the blackboard), the precondition may then request a process instantiation of its associated knowledge source to be established, passing the details of how the precondition was satisfied as parameters to this instantiation of the knowledge source. Once instantiated, the knowledge-source process
Figure 1. Simplified HSII System Organization
can respond to the blackboard data condition which was detected by its precondition, hopefully requesting further modifications be made to the blackboard, perhaps thereby triggering further preconditions to respond to the latest modifications. This particular characterization of the HSII organization, while certainly overly simplified, shows the data-driven nature of the knowledge source activations and interactions. Knowledge sources are instantiated as processes to respond to changes and data patterns occurring in the blackboard. These responses may result in additional data being introduced into the blackboard in the form of hypotheses; or the responses may be such as to test and validate or modify or reject previously existing hypotheses in the blackboard. Thus, HSII provides a data-directed implementation of the hypothesize-and-test paradigm for heuristic search, with the dynamic problem state being maintained in the blackboard data base.

The following sections of this report will attempt to refine this diagram of the HSII organization by pointing out the difficulties that arise from this oversimplified representation of the organization and by supplementing the various components of this simple diagram to resolve these problems and result in a more complete organization for AI problem-solving in multiprocessing environments. A more complete message transaction diagram for HSII will be presented in a subsequent section.

Organization of the Thesis

As a guide to the remainder of this dissertation, the following comments are offered. First, an abstract description of a class of problem-solving systems is given using the Production System model of (Newell, 1973); the HSII problem-solving organization is then described in terms of this model. The various decisions made during the course of the system design necessitated the introduction of various multiprocessing mechanisms (e.g., mechanisms for maintaining data localization and data integrity), and these mechanisms are discussed. A simulation study is then presented which details the effects of actually implementing such a problem-solving organization for use in a particular application area, that of speech understanding. The concluding comments summarize the current state of this research, discuss some of the remaining problems in multiprocessing AI problem-solving, and suggest some areas of future research into multiprocessing, software architecture, and AI problem-solving.
THE MODEL

An Abstract Model for Problem Solving

In the abstract, the problem-solving organization underlying HSII may be modeled in terms of a "production system," (Newell, 1973). A production system is a scheme for specifying an information processing system in which the control structure of the system is defined by operations on a set of productions of the form 'P → A', which operate from and on a collection of data structures. 'P' represents a precondition which may or may not be satisfied by the information encoded within the dynamically current set of data structures. If 'P' is found to be satisfied by some data structure, then the associated action 'A' may be executed, which presumably will have some altering effect upon the data base such that some other (or the same) precondition becomes satisfied. Productions are executed as long as their antecedent preconditions are satisfied, and the process halts either when no precondition is found to be satisfied or when an action executes a stop operation (thereby signalling problem solution or failure, in the case of problem-solving systems).

The production system model as specified thus far leaves much room for variation with respect to its control structure. By varying any of a number of parameters, many different information processing systems may be modeled. In the "classical" form of production systems, the precondition-action rules are ordered sequentially and precondition testing is accordingly sequential. That action whose precondition is first found to be satisfied is then executed, and the precondition testing begins again at the beginning of the rule sequence. Tables 1, 2 and 3 suggest some of the other possible control structures which may be modeled by a production system by
presenting some of the choices that must be made in specifying the control structure of
the production system model itself. Tables 1, 2 and 3 also point out various choices
regarding where, in the overall production system, the knowledge necessary for solving
any given problem is maintained or accumulated. For example, knowledge concerning
the task environment could reside in an encoded form in the data base (in both static
and dynamic forms), or it could be encoded in the specification of a processing element
(i.e., encoded in the prescribed operations of a precondition or an action), or knowledge
could be encoded in a global supervisory process which could control the execution
ordering of preconditions or actions.

What sort of information is expressible in the data base?
How is knowledge encoded in the data base, and how is it accessed?
Is there any implied information ordering within the data base?
If there is such an ordering, does it vary as preconditions are matched or
actions are executed?
How can the content of the data base be affected by the operations performed
by production actions?
Are there any size limitations (artificial or otherwise) imposed on the data base?
Are there auxiliary data bases, perhaps accessible only to production
preconditions and not modifiable by production actions?

Table 1. Production System Design Choices: Underlying Data Structures

Note that many of the issues mentioned in Tables 1, 2 and 3 have to do with
specifying a control regime under which the preconditions are to be tested and the
actions executed. In making the various choices regarding the specification of the
overall control strategy, corresponding decisions must be made to provide the
mechanisms necessary to effect the chosen control regime in any given machine
implementation. For example, it is often very difficult to guarantee that only one
precondition 'P' will be satisfied at any given time; and often the actions 'A' of
competing satisfied 'P's' will fluctuate in their relative importance as they execute.
Therefore, in the interest of user simplicity and scheduling flexibility, it is often realistic
to desire to accommodate simultaneous precondition satisfaction and concurrent action
execution. More specifically, if the choice is made that precondition evaluation is to be
done on a continuous basis without regard to any ordering on the set of preconditions,
then conflict rules may have to be devised to handle possible race conditions which may
arise when two or more preconditions become satisfied simultaneously (if such an
What kinds of tests are expressible in the preconditions?
How is the matching of preconditions to the data structure actually done, and when or how often is it done?
Is the match operation dependent upon an implied ordering of the data base?
Is the match operation an exact match, or a partial match, or a match involving templates using local variable bindings?
How is the order of precondition testing specified, or does it matter, and can it vary dynamically?
How is knowledge of the task environment and the current problem solution state encoded in such an ordering?
If the order of precondition testing is important, how can this ordering be affected (either statically or dynamically) by detecting a precondition match or executing an action part?
In general, can productions be modified, added, or deleted during the course of execution (as might be desirable in a knowledge acquisition system)?
What sorts of conflict rules are applicable in cases of multiple precondition satisfaction, or how are multiple matches avoided in the first place if they are deemed undesirable (as might be the case in trying to construct a simple deterministic system)?
Can any given precondition simultaneously match multiple pattern instances in the data structure (thereby giving rise to multiple action instantiations), or how are multiple matches by a single precondition avoided?

Table 2. Production System Design Choices: Production Preconditions

What primitive operations may be performed on the data and how may these operations be used in specifying production actions?
How is knowledge of the task environment and the current problem solution encoded in the body of a production action?
How is the order of action execution specified in the case of multiple precondition matches, or does it matter?
If the action execution order is important, how can this order be affected (either statically or dynamically) by detecting a precondition match or executing an action part?
How is the execution of multiple action parts coordinated (if it is even allowed), and how can the execution of one action part affect the executability of concurrent or subsequent action parts?

Table 3. Production System Design Choices: Production Actions
occurrence is deemed to be undesirable). Furthermore, if the execution of an action 'A' is temporally separated from the satisfaction of its precondition 'P', mechanisms must be introduced to save the relevant problem solution state at the time of the precondition satisfaction in order to compare it with the solution state at the time of action execution in case any intervening change to the solution state might affect the applicability of the delayed action. This required state saving raises various issues relating to localized context and context revalidation, since the effects of various events which affect the problem solution state and which are relevant to the delayed action execution must be saved in a state-saving context local to that action for use during the subsequent execution of the action. Similar issues of localized context and context revalidation arise when the execution of production actions is permitted to be interrupted or when the concurrent execution of several production actions is allowed: an action must be able to detect whether intervening or concurrent processing by other actions has invalidated its own precondition, thereby perhaps demanding alternative processing on the part of the affected action. The essence of the problem is that if, in any chosen control regime, it is possible to interrupt or delay the execution of an action once its precondition has been satisfied,\(^1\) then state-saving mechanisms must be provided to maintain the necessary context needed to complete the action execution.

The HSII Problem-Solving Organization: A Production System Approach

Referring again to Tables 1, 2 and 3, the HSII problem-solving organization may be described in terms of a production system by answering the various questions posed. The HSII organization has a centralized data base which represents the dynamic problem solution state. The data base is a multidimensional data structure which is readable and writable by any precondition or knowledge-source process (where a knowledge-source process is the embodiment of a production action).\(^2\) Preconditions

---

\(^1\) An action is considered to be interrupted if the execution sequence from precondition satisfaction to action completion is not effectively enclosed within an indivisible critical region which could assure the integrity of the precondition satisfaction until the action no longer needed such assurance.

\(^2\) As an example, the dimensions of the HSII speech-understanding system data base are informational level (e.g., acoustic level, phonetic level, and word level), utterance time (speech time measured from the beginning of the input utterance), and data alternatives (where multiple hypotheses are permitted to exist simultaneously at the same level and utterance time).
are procedurally oriented and may specify arbitrarily complex tests to be performed on the data structure in order to decide precondition satisfaction. Preconditions are themselves data-directed in that they are tested for satisfaction whenever relevant changes occur in the data base;¹ and simultaneous precondition satisfaction is permitted. Note that testing for precondition satisfaction is not presumed to be an instantaneous or even an indivisible operation, and several such precondition tests may proceed concurrently. Preconditions cannot be created dynamically, although they may be parameterized so as to allow dynamic modification to the testing patterns of existing preconditions. The knowledge-source processes representing the production actions are also procedurally oriented and may specify arbitrarily complex sequences of operations to be performed upon the data structure. The overall effect of any given knowledge-source process is usually either to hypothesize new data which is to be added to the data base or to verify (and perhaps modify) data previously placed in the data base: this follows the general hypothesize-and-test problem-solving paradigm wherein hypotheses representing partial problem solutions are generated and then tested for validity; this cycle continues until the verification phase certifies the completion of processing (and either the problem is solved or failure is indicated). The execution of a knowledge-source process is usually temporally disjoint from the satisfaction of its precondition; the execution of any given knowledge-source process is not presumed to be indivisible; and the concurrent execution of multiple knowledge-source processes is permitted.² The task-oriented knowledge relevant to a particular

¹ In effect, preconditions themselves have preconditions, call them "pre-preconditions." In HSI, knowledge-source preconditions (which correspond to action preconditions in the production system model) may be arbitrarily complex. In order to avoid executing these precondition tests unnecessarily often, they in turn have pre-preconditions which are essentially monitors on relevant primitive data base events (e.g., monitoring for a change to a given field of a given node in the data base, or a given field of any node in the data base). Whenever any of these primitive events occurs, those preconditions monitoring such events are awakened and allowed to test for full precondition satisfaction.

² Detaching the execution of a knowledge-source action from the satisfaction of its precondition introduces an additional degree of control over the progress of the problem solution. The various knowledge-source instantiations (and indeed, even any actively executing preconditions) become schedulable process units to which processing power may be allocated as seen fit by the HSI scheduler. This allows the scheduler to be goal-directed in that it may schedule those processes first which can best advance the progress of the overall problem solution. In this way, processing power is not wasted on processes which are capable of executing but which would

38
problem domain is expressed within the problem-solving organization through the specification of the preconditions and the knowledge-source processes. Dynamic knowledge about the particular problem being solved is held within the central data structure, with the structure of the database itself possessing some semantic content. The only global control over the problem-solving activity is provided by a process scheduler whose function it is to schedule instantiated processes (preconditions and knowledge sources) on the available hardware processing facilities so as to effect a problem solution in a goal-directed manner. The specification as to which processes are to be candidates for scheduling is primarily determined by the data-directed nature of the preconditions (although the goal-directed scheduler may vary the parameters of these preconditions to produce a modified knowledge-source invocation behavior, or the scheduler may allocate processing power to certain knowledge-source instantiations merely to produce the side effects necessary to satisfy further preconditions so they might in turn instantiate knowledge-source actions capable of producing the desired overall effect).

To provide a basis for the discussion in the subsequent sections of this report, Figure 2, depicting the various components of the HSII organizational structure, is offered. The diagram is a more detailed version of the message transaction model presented in an earlier section. The new components of this diagram are primarily a result of addressing multiprocessing considerations, and the rationale and function of these components will be described in detail in the following sections.

As in the earlier, more simplified organizational diagram, the dynamically current state of the problem solution is contained in a centralized, shared database, called the blackboard. The blackboard not only contains data nodes (hypotheses and links), but it also records data monitoring information (tags) and data access synchronization information (locks). Access to the blackboard is conceptually centralized in three processes. As before, the blackboard handler process accepts and honors read and write data-access requests from the active processing elements (the knowledge-
Figure 2. HSII System Organization
source processes and their precondition processes). A lock handler coordinates data-access synchronization requests from the knowledge-source processes and preconditions, with the ability to block the progress of the requesting process until the synchronization request may be satisfied. A monitoring mechanism is responsible for accepting data tagging requests from the knowledge-source processes and preconditions, and for sending messages to the tagging processes whenever a tagged data field is modified. It is also the responsibility of the monitoring mechanism to distribute data events to the various local contexts of the knowledge-source processes and preconditions, as well as to activate precondition processes whenever sufficient data events of interest to those preconditions have occurred in the blackboard.

Associated with each active processing element (knowledge-source process or precondition process) is a local data base, the local context, which records data events that have occurred in the blackboard and are of interest to that particular process. The local contexts may be read by their associated processes in order to find out which data nodes have been modified recently and what the previous values of particular data fields were. The local contexts are automatically maintained by the blackboard monitoring mechanism.

Upon being activated and satisfied, precondition processes may instantiate a knowledge source (thereby creating a knowledge-source process), passing along the reasons for this instantiation as parameters to the new knowledge-source process and at the same time establishing the appropriate data monitoring connections necessary for the new process. The goal-directed scheduler retains the actual control over allocating hardware processing capability to those knowledge-source processes and precondition processes which can best serve to promote the progress of the problem solution.

The following sections will detail the specification and functioning of these various components of the HSII problem-solving organization.
THE COMPONENTS OF HSII

Knowledge Sources and Preconditions

Both HSII and its predecessor, Hearsay I (HSI) (Reddy, et al., 1973b; Erman, 1974), are based on the view that the inherently errorful nature of the information flow during the knowledge-intensive perception process can be handled only through the efficient use of multiple, diverse sources of knowledge (Newell, et al., 1971; Reddy, et al., 1973b). The major focus of the design of HSI was the development of a framework for representing these diverse sources of knowledge and coordinating their individual actions in order to allow cooperation among knowledge sources (Reddy and Newell, 1974). From a conceptual point of view, HSII may be considered as a generalized extension of the framework posited by HSI, adding a data-directed scheme for determining the points of applicability of the various knowledge sources. While both HSI and HSII were developed during the course of investigating the AI problem of connected speech understanding, the resulting system organization evidenced in HSII is believed to have much broader applicability to knowledge-based problem-solving tasks.

There are four dimensions along which knowledge representation in the Hearsay system model can be described: a) function, b) structure, c) cooperation, and d) attention focusing.

The function of a knowledge source has three aspects. The first is for the knowledge source to know when it has something useful to contribute; the second is to contribute its knowledge through the mechanism of making an hypothesis (guess) about some aspect of the problem solution; and the third is to evaluate the contributions of other knowledge sources, i.e., to verify, and perhaps modify or reject, the hypotheses
made by other knowledge sources. Each of these aspects of a knowledge source is carried out with respect to a particular context, the context being some subset of the previously generated hypotheses. Thus, new knowledge is built upon the educated guesses made at some previous time by other knowledge sources.

The structure of each knowledge source is specified so that it is independent and separable from all other knowledge sources in the system. This permits the easy addition of new types of knowledge sources and the replacement of knowledge sources with alternative versions of those knowledge sources. Thus, the system structure can be easily adapted to new task domains which have knowledge sources specific to that domain, and the contribution of a particular knowledge source to the total problem solution effort can be more easily evaluated.

The choice of a framework for cooperation among knowledge sources is intimately interwoven with the function and structure of knowledge. The mechanism for knowledge source cooperation involves hypothesizing and testing (creating and evaluating) hypotheses in a global data base (blackboard). The generation and modification of globally accessible hypotheses thus becomes the primary means of communication among diverse knowledge sources. This mechanism of cooperation allows a knowledge source to contribute knowledge without being aware of which other knowledge sources will use its knowledge or which knowledge sources contributed the knowledge that it is using. Thus, each knowledge source can be made independent and separable.

The global data base that knowledge sources use for cooperation may contain many possible interpretations of the current problem state. Each of these interpretations represents a "limited" context in which a knowledge source can possibly contribute information by proposing or validating hypotheses. Attention focusing of a knowledge source involves choosing in which of these limited contexts it will operate and for how much processing time. The attention focusing strategy is decoupled from the functions of individual knowledge sources. Thus, the decision of whether a knowledge source can contribute in a particular context is local to the knowledge source, while the assignment of that knowledge source to one of the many contexts on which it can possibly operate is made more globally (by a goal-directed scheduler). This decoupling of focusing strategy from knowledge acquisition, together with the decoupling of the data environment (global data base) from control flow (knowledge
source invocation) and the limited context in which a knowledge source operates, permits a quick refocusing of attention of knowledge sources. As an example of the utility of this decoupled control strategy, consider the case of the speech-understanding system. The ability to refocus quickly is very important in such a system, because the errorful nature of the speech data and its processing leads to many potential interpretations of the speech. Thus, as soon as possible after an interpretation no longer seems the most promising, the activity of the system should be refocused to the new most promising interpretation.

A knowledge source is specified in three parts: a) the conditions under which it is to be activated (in terms of the conditions in the global data base in which it is interested), b) the kinds of changes it makes to the global data base, and c) a procedural statement (program) of the algorithm which accomplishes those changes. A knowledge source is thus defined as possessing some processing capability which is able to solve some subproblem, given appropriate circumstances for its activation. A knowledge source is instantiated as a knowledge-source process whenever the global data base exhibits characteristics which satisfy a "precondition" of the knowledge source. A precondition of a knowledge source is specified in terms of a description of some partial state of the total problem solution state which defines when and where its associated knowledge source can contribute its knowledge by modifying that solution state. As has been mentioned previously and will be described in more detail below, this solution state is held in a centralized data base called the blackboard. Blackboard modifications made by any given knowledge-source process are expected to trigger further knowledge sources by creating conditions in the blackboard to which the preconditions of those knowledge sources will, in turn, respond. Through this data-directed interpretation of the hypothesize-and-test paradigm, knowledge sources can exhibit a high degree of asynchronous activity with great potential for parallel execution.1

---

1 Control schemes in which one knowledge source explicitly invokes other knowledge sources are not appropriate because of the design requirement that knowledge sources be independent. Besides, the invocation of a knowledge source may depend on a complex set of conditions which is created by the combined action of several other knowledge sources, the order of execution of those knowledge sources perhaps being non-deterministic. Furthermore, such direct-calling schemes would require that knowledge sources contain explicit information about the knowledge sources which they might call, which is not exactly in the spirit of independence. Similar arguments
It should be noted that the decomposition of the overall task into various knowledge sources is regarded as being a natural decomposition. That is, the units of the decomposition represent those pieces of knowledge which can be distinguished and recognized as being somehow naturally independent. Such a scheme of knowledge source decomposition seems very natural for many problem-solving tasks, and, as has been discussed previously, it fits well into the hypothesize-and-test approach to problem solving. Another reason for the use of the knowledge source as the elemental processing unit is related to the inherently errorful nature in the specification and execution of any given knowledge source. Errors may arise due to an incomplete or inaccurate understanding of the underlying theoretical basis of the knowledge source, or the knowledge source may be incompletely specified in its implementation, or the data input to the knowledge source may itself be errorful, all of which will produce errorful output from the knowledge source. As one knowledge source introduces errors or creates ambiguities in the data base, other knowledge sources must be able to correct or clarify these errors. The use of the knowledge source as the scheduling unit allows such cooperation to be achieved quickly and effectively.

In addition to being a suitable unit for knowledge decomposition, the knowledge source also provides the unit of scheduling in a multiprocessor environment: it represents the grain of computation which must be compared to the cost of inter-knowledge-source communication. Maximizing the independence of the various knowledge sources minimizes the need for inter-knowledge-source communication, thereby supporting the use of the knowledge source as an appropriate unit for decomposition (or as it usually turns out, as the unit of overall system composition, since it is more natural to think of composing a system from a collection of knowledge sources than of decomposing a monolithic system into knowledge sources).

Although conceptually the precondition of a knowledge source is an integral part of the specification of the knowledge source, it was felt appropriate to separate these two functions in the HSII system organization, for various reasons. First of all, just as a knowledge source may specify an arbitrarily complex action be performed in order for the knowledge source to impart its knowledge to the data context in which it was invoked, so too may the precondition upon which a knowledge source is instantiated apply against a centralized control scheme which is explicitly defined for a particular set of knowledge sources. Hence, the need for system reconfiguration flexibility is satisfied by a more data-directed control scheme.
be arbitrarily complex. A precondition is essentially a series of tests performed upon the data contained in the blackboard (which data may be associatively retrieved from the blackboard) to determine whether an associated knowledge source should be instantiated (and thence contribute its additional knowledge to the problem solution state). Not only may the tests of a precondition be based upon the current data values contained in the blackboard, but these tests may involve the occurrence of certain recent data events which may have occurred due to the addition or modification of data in the blackboard, as well as perhaps involving the previous values of modified data fields. It is the function of the precondition to accumulate these data events (in its local context, as will be described later) and then to perform tests on these events and the blackboard to decide whether or not to instantiate a knowledge source to perform an action based on these events. A fundamental difference between preconditions and knowledge sources is that preconditions perpetually monitor the blackboard for data events relevant to its knowledge source, while the active lifetime of knowledge-source processes is limited, the knowledge-source processes being created by their associated preconditions. The ramifications of this difference in processing lifetime will be explored later in conjunction with the discussion of local contexts. Another reason for separating preconditions from their knowledge sources is that such a separation allows an additional degree of scheduling flexibility. Furthermore, this functional separation allows the implementation of preconditions to be more efficient by allowing a single precondition process to handle the testing and instantiation of several related knowledge sources; this multi-functional capability of a precondition also decreases the amount of system overhead in maintaining the local contexts of data events for preconditions by reducing the number of actual precondition processes. Perhaps the most important reason for decoupling precondition evaluation from knowledge source evaluation relates to the issues of attention focusing and goal-directed scheduling mentioned above; these issues will be discussed in more detail in a later section.

As an elementary example of how a precondition and its associated knowledge source(s) are specified in the HSII organization, consider the following pseudo-Algol code which represents two possible precondition/knowledge-source pairs operating in the speech-understanding task domain. One knowledge source (noun.speller) is presumed to be able to phonetically "spell" nouns and the other knowledge source (verb.speller) spells verbs; a common precondition serves to handle the instantiation of both knowledge sources. The precondition process is activated whenever the
blackboard monitoring mechanism detects the creation of a new hypothesis of type "word".

```plaintext
knowledge.source noun.speller (hypothesis word);
  hypothesis phone.hyp;
  foreach phone in noun.dictionary.entry(word) do
    begin
      phone.hyp ← create.hypothesis(phone);
      create.link.between(word, phone.hyp);
    end;
  end.knowledge.source;

knowledge.source verb.speller (hypothesis word);
  hypothesis phone.hyp;
  foreach phone in verb.dictionary.entry(word) do
    begin
      phone.hyp ← create.hypothesis(phone);
      create.link.between(word, phone.hyp);
    end;
  end.knowledge.source;

precondition speller (data.event new word);
  if word.category(word) = noun
    then invoke.knowledge.source noun.speller with word
  else
    if word.category(word) = verb
      then invoke.knowledge.source verb.speller with word
    else error "unknown word category";
  end.precondition;
```

The Shared Data Base

Blackboard Design

A main goal of the HSII design is to permit the representation and cooperation of knowledge at all the levels of information needed in a problem-solving system, while at the same time retaining the advantages of an integrated data base structure. It is important for the various knowledge sources to be able to communicate their results in a form befitting the information level at which they are operating. A judicious choice of a set of information levels adequate for expressing the results of the various knowledge sources, while at the same time permitting the transfer and sharing of information across knowledge sources, is necessary for the successful cooperation of these knowledge sources. In an attempt to satisfy this goal, the primary means for dynamic
data accumulation and consolidation is for the various knowledge sources to record the results of their efforts in a centralized blackboard-like data base; this data base thus contains a dynamic representation of the state of the problem solution. Any knowledge source may contribute pieces of information based on its own computations by merely posting the results of its efforts in the blackboard. Likewise, any knowledge source is free to examine and use any results posted by other (concurrently executing) knowledge sources. This mechanism of cooperation, which is an implementation of the hypothesize-and-test paradigm, allows a knowledge source to contribute knowledge without being aware of which other knowledge sources will use the information, or which knowledge source supplied the information that it had used. The blackboard data base hence also serves as the basis for an indirect means of interprocess communication, where messages and replies are anonymously posted in the blackboard. This anonymous and indirect form of interprocess communication satisfies the performance evaluation and system reconfiguration constraints given in the system goals above. In fact, in an attempt to assure the independence of individual knowledge sources, this form of anonymous message sending, along with the ability to "tag" various pieces of the data structure so as to receive a message should the tagged data element be subsequently modified, represents the primary means of interprocess communication. To further aid in maintaining the independence of knowledge-source processes, the underlying control sequencing of the overall recognition process is data-directed, being based on the dynamic state of the problem solution as specified in the blackboard. Thus the entire system organization is very data-directed, with the various knowledge sources sharing information through the blackboard, as well as virtually all interprocess communication and overall process control being effected through the same data medium. Of course, in order to prevent the centralized role of the blackboard from becoming a bottleneck to problem-solving efficiency during system execution, consideration had to be given to the relationship between the various knowledge sources.

1 The blackboard data base is assumed to be entirely resident in primary memory (as is the case in the HSII speech-understanding system, where the blackboard typically grows to about 20 to 50K, 36-bit words); input/output operations are thence not an issue here, the system being essentially compute-bound. In a paged environment, such as that of the C.mmp multiprocessor, the blackboard should be decomposed according to information levels (and other appropriate dimensions of the data base) and then allocated to memory pages so as to minimize the data working sets of the various knowledge sources (knowledge sources usually being associated with only one or two information levels).
sources and their demands upon the data facilities of the blackboard. The blackboard data base itself is partitioned into information levels (Erman and Lesser, 1975) into which the various executing knowledge-source processes place data units called hypotheses, which units are themselves composed of various data fields (the structure of all hypotheses throughout the blackboard being identical). Any given knowledge source usually will be interested in only one or a few of the information levels of the blackboard. This localization of interest is a result of the decomposition of the problem solution strategy into the various knowledge source components; and the existence of such localization further contributes to the suitability for implementing such a problem-solving organization on a network multiprocessor machine architecture.

Consider now, in a bit more detail, the reasoning behind choosing a blackboard-like global data structure, as well as some of the resultant advantages (and disadvantages) in making such a choice. Given that the knowledge source seems to provide a useful unit of decomposition for dividing a problem-solving system into parts which are amenable for efficient execution on a closely-coupled multiprocessor hardware architecture, and that the hypothesize-and-test paradigm for heuristic search seems to fit nicely into such an asynchronous environment, the components of data communication and process control remain to be specified. From a performance evaluation viewpoint, it is desirable to be able to reconfigure the overall system at the knowledge-source level so that the effectiveness of the various contributing knowledge sources (and subsets of them) may be measured and evaluated, and alternatives or additions to these knowledge sources be proposed and easily inserted into the system structure for further evaluation. This presumes that any given knowledge source should be maximally disjoint from any other knowledge source (or at least clusters of knowledge sources should be disjoint from other knowledge source clusters) in the sense that a knowledge source should not presume the existence of any other knowledge source, except for necessary interfaces and assumptions at a functional level (where, for example, a knowledge source KS₁ might make an assumption about the existence of some other knowledge source KS₂ for the reason that KS₂ would provide KS₁ with various input data, but such assumptions would be sufficiently abstract that a third knowledge source KS₃, functionally equivalent to KS₂, could be substituted for KS₂ without informing or disrupting KS₁). This interchangeability of functionally equivalent knowledge sources also requires a form of interprocess data communication that allows the communication to be made anonymously. Not only should the sender(s) remain
anonymous to the receiver(s), but the receiver(s) should be nameless to the sender(s) as well, for similar reasons. Furthermore, each knowledge source should have free access to all of the results of other knowledge sources, in case such results might necessitate the initiation, alteration, or termination of the execution of the inquiring knowledge source.

All of these points suggest a centralized shared data facility, accessible by every knowledge source for the purposes of reading the computation results of other knowledge sources, posting the results of its own calculations, and sending or receiving indirect interprocess communications which could indicate future execution sequences. The shared data facility of HSI is called the blackboard because of the equal accessibility to all parts of the data base by all executing knowledge sources, and because the usual means of interprocess communication in HSI is for knowledge sources to post messages publicly for other knowledge sources to read whenever they poll the blackboard. The blackboard is intended to represent the dynamically current state of all globally shared information, organizing this information into a connected graph structure accessible to all.

The use of a blackboard data base has several significant advantages over more localized approaches to data sharing. In a blackboard suitably organized as a graph structure, any given piece of information need occur only once (depending, of course, on the structural organization capabilities of the data base); hence, the duplication of shared information is minimized. Minimizing the duplication of information wherever possible also saves processing time: if a field of a particular node in a graph-structured data base is modified, the new information from the modification of that node is immediately available to all uses of that node (represented by the connections to that node in the graph structure), since the node would normally occur only once in the data base (rather than be duplicated for different structural contexts). Furthermore, given appropriate structural capabilities for defining contextual relationships within the graph structure of the data base, processing state information can be maintained within the data structure so as to minimize the need for the recalculation of information as would occur in more conventional backtracking heuristic search strategies (where information is discarded, piecewise invalid or not, as the search mechanism retraces its steps to try an alternative search path). In a more generalized graph structure, where contextual dependencies can be maintained and data can be retained until invalidated (rather than lost upon backtracking, for instance), it is also possible to pursue more generalized
search strategies. The blackboard structure of HSII retains enough contextual information relating to the node structure that knowledge sources capable of working in various parts of the data structure can be scheduled for execution according to the importance attached to investigating those areas of the data structure. This focus of attention mechanism allows the HSII scheduler to allocate processing resources to those knowledge sources capable of developing those areas of the data base which could best benefit the overall problem-solving effort.

The organization and management of a blackboard-like data base can be greatly simplified if the elementary data structures represented by the nodes of the graph (as well as access to those node structures) can be specified in a homogeneous manner. The data nodes (called hypotheses) of the HSII data base share a uniform structure throughout the data base, even though the data base itself is divided into several information levels. In addition, the connected graph structure linking the various hypothesis nodes is also homogeneously specified throughout the data base (this linking structure being specified by auxiliary data objects, called links). Thus, the structural format of all hypothesis nodes is identical and the format of all links is identical, thereby allowing uniform processing procedures to operate upon any hypothesis node or any link node, regardless of its information level within the overall data structure. This structural homogeneity permits the existence of policy-type knowledge sources whose function it is to propagate results through the blackboard by chaining from node to node through the various information levels without regard to the particular information level of the nodes being modified. Similarly, the homogeneous node structure allows all knowledge sources to determine (graph) structural relationships between the various nodes, without having to be concerned with the actual informational content of any nodes being accessed during such a structural search. A further advantage of a homogeneous connected-graph data structure is that access to the data structure may be centralized (and hence centrally controlled). In particular, it is especially easy to monitor for various data events that may occur to fields in the data base if all access to such fields passes through centralized accessing routines; and a node structure that is homogeneous throughout the entire data structure is certainly amenable to such access centralization. Within a data-directed system such as HSII, it is of great importance to be able to monitor for data events in the blackboard; it is this monitoring mechanism that triggers the various precondition evaluations (which in turn instantiate the knowledge-source processes). Whenever any data field is modified, the
monitoring mechanism is also responsible for sending (anonymous) messages to knowledge sources which had tagged those various data fields. The homogeneous node structure and centralized node accessing control of the HSII organization permit these data monitoring functions to be implemented effectively and efficiently.

Of course, the connected-graph blackboard approach to a shared data base is not a panacea, and, as might be expected, various problems must be overcome or avoided if an information processing system is to use such a structure. First of all, since the blackboard is a shared data base and since the knowledge sources are presumed to be concurrently executing (and hence capable of concurrently accessing the blackboard for both reading and writing), the possibility of "destructive interference" resulting from simultaneous overlapping data access by two knowledge sources exists and must be avoided. In particular, any possibility of two knowledge sources simultaneously modifying the same shared data field must be precluded if the results of such a modification would be non-deterministic in nature; such non-determinism would certainly be the case if the modification action involved more than a single memory (hardwired) write operation. The obvious solution to this problem is to provide mutual exclusion semaphores for data elements so that knowledge sources might gain exclusive access to a data element for the duration of a data modification operation. However, this introduces other considerations. What is the proper unit for data access synchronization? Choosing a data unit too large might result in the data base becoming a system bottleneck, leading to reduced effective parallelism by forcing a more sequential data access pattern than might actually be necessary. On the other hand, choosing the unit for data access synchronization to be too small might result in excessive system overheads associated with the management of numerous semaphores (especially if knowledge sources often want access to collections of such data units).

Furthermore, the necessary addition of mutual exclusion semaphores to eliminate race conditions in data accessing also introduces the possibility of the misuse of these semaphores. In particular, mechanisms must be devised to prevent the occurrence of data deadlock situations in which each of two processes becomes blocked waiting for the other to release resources which it would like to acquire, while in the meantime refusing to release its own resources so that the other may proceed. On the other hand, each process desiring exclusive access to data resources must be assured that it will not have to wait indefinitely for its data requests to be fulfilled (assuming any individual process is guaranteed to complete its execution within a finite period of
time). These issues of deadlock and resource allocation arise because of the possibility of concurrent access to the shared data base by two or more processes.

Another problem that arises from the ability of concurrently executing processes to access the centralized blackboard data base is that of preserving process-dependent local state information. The blackboard is intended to be a homogeneous structure, relatively independent of the various processes that access it; as such, the blackboard contains only data of global interest and only the most current state of the problem solution, with no history of data modifications being globally maintained. The reason for not maintaining a history is that such a history is actually process-dependent: knowledge sources are instantiated based on recent events which have occurred in the blackboard, and, as such, these knowledge sources are interested only in those events leading to its instantiation as well as any events which might occur subsequent to that instantiation (but, of course, prior to the termination of the instantiated process). Thus, the time period of interest varies from process to process, as does the variety of events that are actually felt to be relevant to a particular process. As a result of such considerations, a local context is associated with each process instantiated during a problem solution in order to maintain state information and data events of interest to that process. Of course, individual processes may also maintain personalized data structures for their own use, apart from the shared blackboard data base.

One further problem that arises in a dynamically changing shared data base is that of data deletion. Effectively, a reference count method is used to know when it is safe to actually delete a data node that some process had requested be deleted from the shared data base; but the problem is complicated somewhat by the possibility of having processes that are executing concurrently with the deleting process which are able to retain references to the node. Thus, the node can only be marked for deletion (to assure that no further processes will get references to that node), while actually retaining the node in the blackboard structure until all outstanding references (as might be retained in the local contexts of preconditions and knowledge sources, as will be discussed below) to the node are relinquished, whether these references be in the blackboard itself or in the local data bases of the knowledge-source processes.
Blackboard Structure

HSII attempts to attack problems which may be characterized as requiring the accumulation of large amounts of information for their solution. Furthermore, the continued progress of the execution toward problem solution is dependent upon the knowledge thus far accumulated. The blackboard data base of the HSII organization consists of a uniform and integrated multi-level structure which serves to hold the dynamically current data state of the problem solution. Knowledge sources cooperate by creating and accessing elements of the blackboard; and the activation of a knowledge source is data-driven, based on the occurrence of patterns in the blackboard which match templates specified by the knowledge source.

Each level in the blackboard specifies a different representation of the problem space, and the sequence of levels forms a loose hierarchy in which elements at each level can be roughly described as being abstractions of elements at the next lower level. This decomposition into levels can be thought of as an a priori framework for a plan for the problem solution, with each level corresponding to a generic stage in the plan. Examples of levels in the speech problem are the "syntactic," "lexical," "phonetic," and "acoustic" levels of speech knowledge; examples of levels in scene analysis are "objects," "regions," "line segments," and "picture points." Associated with each level is a set of primitive elements appropriate for representing the problem at that level. In the speech problem, for example, the elements at the lexical level are the words of the vocabulary to be recognized, while the elements at the phonetic level are the phones (sounds) of the English language.

The decomposition of the problem space into levels is a natural complement to the decomposition of the knowledge embodied in any overall processing algorithm into knowledge sources. Particular levels for a given task domain may be chosen such that not only may knowledge sources be localized with respect to their requisite knowledge of the overall problem-solving strategy, but they may also be localized with respect to their blackboard communications with other knowledge sources. For many knowledge sources, the knowledge source needs to deal with only one or a few information levels in order to apply its knowledge; in fact, it need not even be aware of the existence of other levels. Thus, each knowledge source can be made as simple as its knowledge allows, with its interface to the rest of the system being in information units and concepts which are natural to it. Also, new levels can be added (just as new knowledge
sources can be added) whenever new sources of knowledge are introduced which need those information levels. As will be discussed later, this localization of knowledge sources according to problem-solving knowledge and data levels permits the activity of the knowledge sources to be efficiently sequenced in a non-deterministic manner suitable for execution in a multiprocessing environment.

The data elements at each level in the blackboard represent hypotheses about some aspect of that level, each element being labeled by a particular member of the set of primitive elements for that level. The structural format of all hypotheses throughout the blackboard is identical, consisting of a fixed set of hypothesis attributes (these attributes being fixed at system generation time). The particular attributes selected (one of these attributes being, of course, the primitive element designator representing the informational content of the hypothesis) are chosen so as to facilitate the efficient implementation of the data-directed mechanisms used in blackboard monitoring, precondition testing, goal-directed knowledge-source scheduling, and associative data retrieval. For example, the attributes of hypotheses are selected so that the preconditions of most knowledge sources may be sensitive to a single, simple change in some hypothesis attribute (such as the creation of a new hypothesis of a particular type, the modification of a certain field of an hypothesis, or the creation of a structural link between certain types of hypotheses).

Knowledge sources may also create structural relationships among hypotheses via the introduction of link nodes. Like hypotheses, the structural format of all link nodes is identical throughout the blackboard data base, with the link attributes being chosen in order to satisfy the same goals as for hypotheses. The structural relationships that may be created among hypotheses serve to represent inferences and deductions made by the knowledge sources about the hypotheses. These structural relationships also allow competing and overlapping partial problem solutions to be represented and handled in the blackboard data base in an integrated manner. These structural relationships may be made between hypotheses at different (but usually adjacent) levels in the blackboard, or between hypotheses within a particular level (thereby allowing intra-level data structures). Furthermore, these relationships can usually be derived (by a knowledge source appropriate to the level) without having to examine the data structure above or below the level of abstraction of the hypotheses involved. This locality of data context simplifies the specification and operation of knowledge sources.
Thus, the blackboard data structure consists of hypotheses placed at the levels within the blackboard, with these hypotheses being joined into a connected graph structure by links. Considering the level structure as decomposing the blackboard into an a priori framework for representing stages in a plan, the goal at each level is to create and validate hypotheses at that level; and the overall goal of the system is to create the most plausible network of hypotheses that sufficiently covers the levels, where the value judgments concerning "plausibility" and "sufficiency" are left to the collection of executing knowledge sources. In speech understanding, for example, the goal at the phonetic level is to produce a phonetic transcription of the utterance; while the overall goal is to create a network which connects hypotheses directly derived from the acoustic input at the lowest level in the blackboard to hypotheses which describe the semantic content of the utterance at the highest level in the blackboard.

The function of some knowledge sources is to create and modify hypotheses at particular levels in the blackboard. If this creation/ modification is based on a context of hypotheses at a lower level (or levels), the resulting action may be considered as a synthesis, or abstraction, of those hypotheses. Conversely, manipulation of hypotheses based on higher level hypothesis contexts may be considered as an analysis, or elaboration, of that context. Both sorts of data actions are desirable in order to overcome the errorfulness of individual knowledge sources or in order to take advantage of redundancies in the informational content of the total set of knowledge sources.

Notice that it is not necessary for any given level to be completely elaborated in order for the system to achieve problem solution. While the context for an analysis or synthesis action is often localized to levels adjacent to the level in which the action takes place, this is not required; actions which skip over several levels can serve to direct the activity of the system and thereby significantly prune the search space. Such a jump over levels is equivalent to constructing a major step in a plan. Furthermore, it may not be necessary that the skipped levels ever be filled in by subsequent analysis or synthesis actions, if the knowledge sources are confident enough in the plausibility of the larger step. Thus, the knowledge sources can dynamically define the granularity in the hypothesis network necessary to assure the desired degree of plausibility; and this granularity may vary in different places in the blackboard, depending on the particular structure present in those places.
As an example of the decomposition of a problem-solving system, Appendix A (extracted from (Lesser, et al., 1974)) contains a description of the blackboard and knowledge-source decompositions for the Hearsay II speech-understanding system:

Node Structure

Consider now, in more detail, the internal structure of the nodes (hypotheses and links) which make up the blackboard. The internal structure of an hypothesis consists of a fixed (at system generation) set of attributes, or named fields; this set is the same for hypotheses at all levels of representation in the blackboard. As mentioned before, these attributes are selected so as to facilitate the data-directed implementation of the hypothesize-and-test paradigm. The values of the attributes are defined and modified by the knowledge sources.

Attributes can be grouped into several classes:

1) The first class of attributes names the hypothesis: it contains the unique name of the hypothesis, the name of its level, and its label from the element set at that level.

2) The next class of attributes is composed of parameters which rate the hypothesis. These include separate numerical ratings derived from a) a priori information about the hypothesis, b) analysis actions performed on the hypothesis, c) synthesis actions, and d) combinations of (a), (b), and (c).

3) Another set of attributes contains information about knowledge source attention to the hypothesis. These include a cumulative measure of the amount of computation that has already been expended on the hypothesis, as well as suggestions for how much more processing should occur and of what type (e.g., analysis or synthesis).

4) One very important set of attributes describes the structural relationships with other hypotheses, as described below.

5) For each problem domain, it is likely that there are other attributes which are basic to the problem and which should be provided in the structure of the hypotheses; these form a problem-specific class of attributes. In speech understanding, for instance, utterance time is a fundamental concept, so the HSII speech-understanding system has a class of attributes for describing the begin- and end-time and the duration of the event which the hypothesis represents. (These attributes include ways of explicitly representing fuzzy notions of the times.) For vision, likely attributes would include the location and dimension of the element and trajectory information for moving objects.

6) The capability for arbitrary knowledge-source-specific attributes is also included. This can be used by a knowledge source to hold arbitrary information about the hypothesis; in this way a knowledge source need not hold
state information about the hypothesis across activations of the knowledge source and allows, for example, the easy implementation of generator functions. If several knowledge sources share knowledge of the name of one of these attributes, each of them can access and modify the attribute's value and thus communicate just as if it were a "standard" attribute; this can be used as an escape mechanism for explicit knowledge-source intercommunication.

7) A unique class of hypothesis attributes, called **processing state** attributes, contains succinct summaries and classifications of the values of the other attributes. For example, the values of the rating attributes are summarized and the hypothesis is classified as either "unrated," "neutral" (noncommittal), "verified," "guaranteed" (strongly verified and unique), or "rejected." Other processing state attributes summarize the structural relationships with other hypotheses and characterize, for example, whether the hypothesis has been "sufficiently and consistently" described synthetically (i.e., as an abstraction of hypotheses at lower levels). The processing state attributes are especially useful for efficiently triggering knowledge sources; for example, a knowledge source may specify in its precondition that it is to be activated whenever a hypothesis at a particular level becomes "verified." These attributes are also used for the goal-directed scheduling of knowledge sources, as described in a subsequent section.

**Structural relationships** between hypotheses in the blackboard are represented through the use of **links**; links provide a means of specifying contextual abstractions about the relationships of hypotheses. A link is an element which associates two hypotheses as an ordered pair; one of the nodes is termed the **upper hypothesis**, and the other is called the **lower hypothesis**. The lower hypothesis is said to **support** the upper hypothesis, while the upper hypothesis is called a **use** of the lower one; in general, the lower hypothesis is at the same or a lower level in the blackboard than the upper hypothesis. Links possess an attribute structure similar to that described above for hypotheses, except that links do not possess a label field for identifying the information within the node as describing a particular element chosen from the set of elements applicable to a given level.

There are several types of links, with the types describing various kinds of relationships.\(^1\) Consider this structure:

---

\(^1\) The particular kinds of relationships described here are some of those that were designed for the speech problem. Although they undoubtedly are not the complete set for all conceivable needs, they do represent the kinds of relationships that need to be and are expressible in the blackboard.
H₁ is the upper hypothesis and H₂, H₃, and H₄ are the lower hypotheses of links L₁, L₂, and L₃, respectively. If the links are all of type OR, the interpretation is that H₁ is either an H₂ or an H₃ or an H₄. This is one way that alternative descriptions are represented. If the links in the figure are of type AND, the interpretation is that all of the lower hypotheses are necessary to support the existence of H₁. (Note that, in general, all of the supporting (lower) links of a hypothesis are of the same type; one can thus talk of the "type of the hypothesis", which is the same as the type of all of its lower links.)

These two types of node represent different kinds of abstractions: the OR-node specifies a set/member relationship, while the AND-node defines a composition abstraction. Variants of the AND- and OR-links are also possible. For example, a SEQUENCE link is similar to the AND-link, except that an ordering is implied on the set of lower hypotheses supporting the upper hypothesis. (For the HSI! speech-understanding system, this ordering usually is interpreted as indicating a time ordering of the lower hypotheses.)

Besides showing analysis and synthesis relationships between hypotheses (e.g., that one hypothesis is composed of several other units), a link is a statement about the degree to which one hypothesis implies (i.e., "gives evidence for the existence of") another hypothesis. The strength of the implication is held as attributes of the link. The sense of the implication may be negative; that is, a link may indicate that one hypothesis is evidence for the invalidity of another. This statement of implication may be bi-directional; the existence of the upper hypothesis may give credence to the existence of the lower hypothesis and vice versa. Finally, these relationships can be constructed in an iterative manner; links can be added between existing hypotheses by knowledge sources as new evidence for support is discovered.

Just as an hypothesis can have more than one lower link, so it can have several upper links. Each of these represents a different use of the hypothesis; the uses may be competing or complementary. The ability to have multiple uses and supports of the same hypothesis, as opposed to creating duplicates for each competing
use and abstraction, serves to keep the blackboard compact and thereby reduces the combinatoric explosion in the search space. Further, since all the information about the hypothesis is localized, all uses and supports of the hypothesis automatically and immediately share any new information added to the hypothesis by any knowledge sources.

A problem with this localization can occur if the interactions between hypotheses span more than one level. In this case, a particular support of the hypothesis (at a lower level) may be inconsistent with one (or more) of the uses of the hypothesis (at a higher level) yet be consistent with other uses (or potential uses) of the hypothesis. In order to avoid duplicating the hypothesis, a mechanism, called a connection matrix, exists in the system. A connection matrix is an attribute of an hypothesis; its value specifies which of the alternative supports of the hypothesis are applicable ("connected to") which of its uses. The use of a connection matrix allows the results of previous decisions of knowledge sources to be accumulated for future use and modification without necessitating contextual duplication of parts of the data base. This kind of reusage and multiple usage of blackboard structures reduces much of the expensive backtracking that characterizes many problem-solving systems.

Appendix B (also extracted from (Lesser, et al., 1974)) contains an example of a structure built in the blackboard of the HSII speech-understanding system.

**Associative Data Retrieval**

While any knowledge source or precondition may access (read or write) any field in the blackboard by explicitly referencing the desired field, this is not sufficient in a data-directed environment where the blackboard is being dynamically modified by various asynchronously executing concurrent processes. Such a situation requires the availability of a mechanism for associative data retrieval from the blackboard; HSII provides such a mechanism. HSII defines primitives for associatively searching the blackboard for hypotheses (or links) satisfying specified conditions (e.g., find all hypotheses at the phonetic level which are vowels and which occur within a particular time range in the utterance). The search condition is specified by a matching-prototype, which is a partial specification of the components of an hypothesis. This partial specification permits a component to be characterized by: a) a set of desired values, b) a don't-care condition, or c) values of components of an hypothesis previously derived by matching against another prototype. A matching-prototype is applied to a
set of hypotheses (or links); the hypotheses whose component values match those specified by the matching-prototype are returned as the result of the search. Associative retrieval of structural relationships among hypotheses is also provided. More complex retrievals can be accomplished by combining the retrieval primitives in appropriate ways.

The existence of this associative data retrieval mechanism is especially important to precondition processes, since it is the purpose of these processes to discover data patterns within the data base that might be of particular interest to their associated knowledge sources. In fact, precondition processes represent the procedural embodiment of the data base conditions necessary for the instantiation of the associated knowledge source. Precondition specifications conceptually form AND/OR trees composed of matching-prototypes and structural relationships which, when applied to the data base in an associative manner, detect the regions of the data base in which the knowledge source associated with the precondition is interested (if the precondition is capable of being satisfied at that time). Alternatively, one might think of the precondition specification as a procedure, involving matching-prototypes and structural relationships, which effectively evaluates a conceptual AND/OR tree. This procedure may contain arbitrarily complex decisions (based on current and past modifications to the data base) resulting in the activation of desired knowledge sources with the chosen contexts. Once invoked, a precondition procedure uses sequences of associative retrievals and structural matches on the data base in an attempt to establish a context satisfying the preconditions of its associated knowledge source (although, for efficiency reasons, in an actual implementation any given precondition procedure may be responsible for instantiating several related knowledge sources). The context corresponding to the discovered data base region which satisfies some knowledge source's precondition is used as an initial context in which to instantiate that knowledge source as a new process. If there are multiple regions in the data base that satisfy the specified conditions, the knowledge source can be separately instantiated for each context, or once with a list of all such contexts.

This set can be derived by the knowledge source from several sources. The HSII speech-understanding system implementation includes the following primitive sources: a) all hypotheses (in the blackboard), b) all hypotheses at a particular level, c) all hypotheses at a particular level whose time attributes overlap a given interval (this provides an extremely efficient, two-dimensional partitioning of the blackboard), and d) all hypotheses whose attributes which are being monitored (by that knowledge source) have changed.
Note that the data-directed nature of precondition evaluation and knowledge-source instantiation is linked closely to the primitive functions that are able to modify the data base, for it is only at points of data base modification that a precondition that was unsatisfied before may become satisfied. Hence, data base modification routines have the responsibility (although perhaps indirectly) of activating the precondition evaluation mechanism; and the precondition evaluation mechanism is then responsible for instantiating any knowledge sources, should a precondition be found to be satisfied.
HSII MULTIPROCESSING MECHANISMS

Given the decision that multiple preconditions may be simultaneously satisfied and that multiple knowledge-source processes may execute concurrently, various mechanisms must be provided to accommodate such a multiprocesssing environment. Mechanisms must be provided to support the individual localized executions of the various active and ready processes in order to keep the processes from interfering with one another, either directly or indirectly. On the other hand, mechanisms must also be provided so that the various active processes may communicate with one another so as to achieve the desired process cooperation. The following sections describe the approach taken in providing the mechanisms which allow the knowledge-based problem-solving organization of HSII to operate in a multiprocessing environment.

1 The decision to allow simultaneous precondition satisfaction and concurrent action processing is based on several objectives. First, HSII is intended to provide a framework within which alternative sets of knowledge-source processes may be compared with respect to relative performance and problem-solving effectiveness. Referring to the production system model described earlier, in such a framework it would be difficult to assure precondition satisfaction and action execution compatibility between every pair of productions so that arbitrary subsets of the total production set could be combined to create an instance of the problem-solving system. Therefore, no explicit compatibility constraints are imposed. Instead, the various component productions are simply assumed to cooperate with one another by not being malicious. The second reason for deciding to permit concurrent processing of productions is the desire to implement a HSII speech-understanding system on the C.mmp multiprocessor. It is believed that such a hardware organization can provide the large amount of computational power necessary to perform the task of speech processing in a cost effective manner; therefore, if the HSII model is to provide the basis for implementing the speech system, the model must face the various multiprocessing issues that arise. Thirdly, the pure research aspects of such a problem-solving organization and its implementation were found to be quite appealing.
Data Access Synchronization

In a multiprocessing organization such as HSII, the various knowledge-source processes are intended to cooperate with one another in achieving the goal of problem solution. However, since these knowledge sources are to have been developed independently, their means of cooperation must necessarily be indirect in nature. As a result of this anonymous form of cooperation and as a result of the desire to use asynchronous parallel processing techniques, multiprocessing mechanisms are required which can prevent the various processes from destructively interfering with the execution sequences and the various accumulated data assumptions of other processes. In the following paragraphs, methods for data access synchronization are presented which prevent such interference from occurring while processes are accessing the shared global data base. These access synchronization methods avoid indirect interprocess interference by controlling the available access patterns to the shared data base, this shared data base providing the primary (albeit indirect) means for interprocess communication.

Data Node Accessing

In a situation where several processes can potentially access shared data concurrently, mechanisms must necessarily be provided to synchronize this access so as to maintain the integrity of the shared data by avoiding race conditions. An example of such a race condition might occur if a given process desires to write a particular field of a node in the shared data base, but only after having examined and tested the value of that field: if the compound operation of examining a field, testing it to determine whether to modify it or not, and then perhaps subsequently modifying it, were not indivisible, then a concurrently running process could, say, modify the field between the testing operation and the modification operation of the first process, perhaps invalidating the previously observed result of the testing operation and invalidating the subsequent modification (which would nonetheless be made, since the first process would not be made aware of the intervening data modification). Clearly, some sort of data synchronization is required to assure the uninterrupted operation of compound data operations. Solutions to such problems may be had by regarding the various components of a shared data base as resources and allocating these resources in an orderly (i.e., well controlled) manner to those processes desiring access rights to the resources so as to preserve the integrity of compound data operations. Various choices
are left open in implementing such a mechanism. First, the shared data base must be divided into allocatable resources. Then a policy must be imposed prescribing the way the resources are to be allocated so as to guarantee "fair" service and avoid deadlocking.

The choice of allocation units into which to subdivide a shared data base should be based on several criteria. If it is known in advance how the data will be shared by concurrent processes, the data base subdivision should certainly take that information into account so as to minimize the amount of potential interference among processes with respect to data sharing; i.e., subdivide the data base so as to permit maximal processing concurrency, while at the same time dividing the data base into only as many allocatable pieces as is necessary to allow this maximal processing concurrency.\(^1\) Note that a static method of maximally effective resource specification is not always possible. More general schemes must be employed in the event that either the contents of the shared data structure are not known in advance or the shared data demands of the various processes are not known in advance, or both.

In the case of the HSII system organization, the shared data structure is a dynamically constructed connected graph of nodes, where each node is in itself a small data structure. Nodes represent the units of data creation in the data base, although subsequent access may be to subfields of a node. Data nodes are of two types in HSII: hypotheses and links. Within each category of node types, all nodes of that category are structurally identical (with an escape mechanism to handle special cases), thereby making the overall graph data structure very homogeneous when viewed as a collection of data nodes. The standard fields which constitute a node have been chosen for various reasons, including facilitating the data base monitoring capabilities to be described later in connection with "tagging" and data-directed process invocation. The

\(^1\) Note the similarities between choosing the grain of data decomposition for data base access synchronization in a shared data base and choosing the grain of knowledge decomposition for active processing elements. In both situations, the chosen decomposition should allow maximal processing concurrency (and minimal interference between processing elements), while not being "over-decomposed" as to create excessive system overheads (as from increased bookkeeping costs, etc.). Obviously, the choice of decomposition for data access synchronization is related to the choice of decomposition for the specification of processing elements (and vice versa). In the HSII organization, the unit of decomposition for processing element definition has been chosen as the knowledge source; the decomposition for data access synchronization follows from that decision.
important point here is that each node has a unique identification index (assigned at the
time of node creation), which thereby imposes a linear ordering across the set of all
nodes in the data base. This ordering will become important in the subsequent
discussion of data base deadlock prevention.

The node was chosen as the primary unit of data resource allocation for data
synchronization purposes, primarily because the node is also the unit of data creation.
Whenever a given process wishes to access or modify a field (or fields) of a node, the
process may request the HSII operating system to grant it exclusive access to that node
for as long as it needs to guarantee the integrity of the node over time (i.e., for as long
as it needs to guarantee that no other process will interfere with the assumptions
generated by the given process during some sequence of data accesses to the node). It
is often the case that a process wants only to examine the contents of a node in a
read-only fashion, but during the course of this examination (as well as perhaps during
subsequent data modification elsewhere in the data base) the process needs to be
assured that no other process will intervene and modify the node (thereby perhaps
destroying the basis for assumptions that the given process has built up through its
read-only examination of the node). Therefore, in addition to being able to request
exclusive access to any node in the shared data base, an executing process may also
request read-only access to any node, this read-only access mode being introduced in
an attempt to reduce potential shared data access interference between concurrent
processes (since several processes may simultaneously attain read-only access to a
given node, whereas only one process could have been granted exclusive access to that
node).

Region Accessing

Given the node as an allocation unit for acquiring rights to access the data
present in the shared data base at any given time during execution, notice that while
such an allocation scheme might be adequate for static data structures, it is not
adequate for acquiring access in a dynamically expanding structure. As an example,
consider a situation where two processes are capable of creating nodes in the data
structure, but neither wants to create a node with certain characteristics if such a node
already exists in the data structure. In the case of the pre-existence of such a node,
the node-creating process would instead merely reference the already existing node.
So before creating a node, each process would examine the data base to see whether a
node matching the one it is about to create already exists. For the sake of example, suppose each of the two processes was about to create a node identical to the one being proposed by the other process, with no such node already in existence in the data base. If the order of concurrent operations were such that one process checked the data base, found no conflicting node already present, and introduced its new node before the second process checked the data base for pre-existence, then the second process would use the node created by the first rather than making a duplicate node. But suppose the sequence of checking the data base, deciding whether or not to create a new node, and creating any new node was not indivisible (in an intuitive notion of “indivisible”); then it would be possible for each process to check for pre-existence concurrently, decide to create a new node, and create the node, unaware of the concurrent creation of an identical node by the other process, thereby resulting in the presumably undesirable situation of having duplicate nodes in the data base. A solution to this form of race condition is not possible if the only form of data access synchronization available is based on allocating access rights according to existing elements of the data base: this situation calls for obtaining access rights to a node before it even exists. As a result, a supplementary form of access allocation must be introduced for handling dynamically expanding data structures. This mechanism is referred to as region-locking in the HSI system organization.

Region-locking is based upon the division of the abstract data base into regions, much like imposing a grid upon a delimited geometric area, where the delimited area represents the potential extent of the data base and the subdivisions of the area produced by the grid correspond to regions in the data base. The ability to specify such regions in the data base depends upon whether or not it is possible to define the data base as a set of discrete points whose coordinates can be associated with values along various dimensions which serve to characterize the data base. In particular, the dimensions chosen to characterize the data base must be either mensurable or denumerable, and the maximum value along any given dimension should be finite (although dimensions of infinite length could be represented finitely by, say, dividing

---

1 This statement should be qualified somewhat. If upon node creation, all nodes were required to be linked to already existing elements of the data base, then the race conditions found in this expanding-data-base problem could be avoided by acquiring exclusive access to the nodes to which the new node is to be linked, before creating the new node. However, the HSI organization allows nodes to be created without specifying any links at creation time; hence, the problem exists and must be resolved.
some finite portion of the dimension into some finite number of divisions and adding one more division to represent the remainder of the dimension's values). In the case of the HSII speech-understanding system, the dimensions chosen to represent the data base for the purposes of region-locking are those of information level (a dimension which is denumerable and finite by definition) and utterance time (which is measurable and finite, by assuming a maximum utterance length). Thus any point in the data base may be defined (for the purposes of region-locking) by a pair of numbers, <time, level>. For the purposes of data access synchronization, regions are then defined to lie completely within one information level (fixed ordinate, so to speak) and extend across one or more time units, the conceptual abscissa being divided into equal divisions up to the maximum allowable utterance length. Using the regional representation of the abstract data base (which, of course, divides the data base without regard to its actual contents, regardless of how long execution has been under way), data access may now be coordinated as if the data base itself were statically subdivided into resource allocation units.

Of course, since the actual data content of the data base is contained in concrete nodes rather than abstract regions, there must be some way in which the locality of data nodes in the data base may be described in terms of regions, if these abstract regions are to be used in coordinating data accesses. This correlation is accomplished by associating with every data node a region descriptor which specifies the abstract region coordinates applicable to the actual data node. Note that, depending on the specification of data fields within the node and depending on the ability of active processing elements to modify these fields, it may be possible for the region descriptor of a data node to vary over time, if the data node contains writable fields which specify any of the dimensional coordinates used in specifying the region descriptor. As an example, using the HSII speech-understanding system, the region descriptor of data nodes are expressed as a set of <utterance-time, level> pairs whose values range over the utterance-times which describe the data node, the information level coordinate being fixed for any given node. However, the data nodes also contain fields for specifying the node's information level index and the bounds of the node's utterance-time. While the level index is fixed (at the point of node creation) for the lifetime of the

---

1 As another example, in a system which analyzes static two-dimensional visual scenes, the data base dimensions might be information level (e.g., line segments, visual regions, geometric shapes, objects), horizontal distance from an origin, and vertical distance from an origin; and points in the resulting data base may be described by triples, <x, y, level>.
node, the utterance-time boundaries are writable fields; hence, the region descriptor for a data node may vary as execution proceeds. This potential variability in the region descriptor (which descriptor is used in region-locking) introduces some problems with respect to data access synchronization which will be discussed below.

Using this regional representation of the abstract data base, processes may lock areas of the data base which may not yet contain any nodes (essentially, region-locking allows a process to lock a "vacuum"); and the locking process may safely expand the actual data structure into that region without any possibility of interference from other processes, thereby avoiding undesirable race conditions relative to node creation. Region-locking may also be used as an efficient means by which to lock the set of nodes which are contained within a data base region without having to specify that set explicitly and without having to perform a node-locking operation on each element of that set. Just as nodes could be allocated for exclusive access or read-only access, so can data base regions. Of course, if data access to the shared data base is to be synchronized using both the nodewise decomposition of the data base as well as the data region decomposition of the data base, then these two allocation mechanisms must be coordinated so as to avoid data base deadlocks, since both schemes represent mechanisms for controlling the access patterns to essentially the same resource (viz., the shared global data base).

Region-locking was introduced as a solution to a data synchronization problem that node-locking alone could not handle. But why could node-locking not be supplanted entirely by region-locking, assuming every node could be associated with some region in the data base? To be able to use the data region concept for efficient and effective data access synchronization (which is to say, such that undue interference due to synchronization is avoided), a process must be able to request access to a region by specifying explicit bounds on the various dimensions of the regions to which it would like access. Specifying arbitrarily sloppy bounds in any dimension increases the possibility for "undue interference," undue because the interference would be caused by the lack of more precise region boundary information, which should have been supplied by the original requesting process. One might suspect that the data nodes, being explicitly placed within the data base would have associated with them the data region which they occupy. This, of course, is true, however the data region associated with any given node need not be very precise in its specification, for various reasons. For one, the structural position of a node within the connected graph structure of the
data base may have greater significance semantically than the association of a data region with that node, and, as a result, little attention may be paid to specifying that data region association precisely. Furthermore, it is oftentimes very difficult to specify the data region of any particular node precisely, especially when one or more dimensions of the data region specification are only loosely related to the specification of the data node. As an example, in the HSII organization for the speech understanding task, the dimensions for data region specification are information level and utterance time. These dimensions are well suited for fairly precise region characterizations at more elemental levels of the utterance representation, such as at the segmental level (where the time boundaries of a speech segment are well specified); but the dimensions are less suitable for precise region specification, say, at the word level, where although the position of a word hypothesis node may be indicated precisely by its structural connections to other word hypotheses, the time region of that node can only be specified very approximately (until sufficient time bounds are propagated upward via links from lower-level supporting hypotheses). Thus, the time regions associated with data nodes are specified with tolerances on the given bounds; and these bounds and tolerances may even be specified as completely "fuzzy." One further reason for possible ambiguity in the specification of the data region associated with any given data node is that the nodes in the data base represent (initially, at least) hypotheses about what may actually exist at that point in the final data base: data nodes represent educated, contingent guesses or estimates about the actual structure which would describe the true solution to the given problem (as specified by the input data). As such, data nodes are very conditional in nature, relying upon the continued support of their contexts for their existence and further refinement. But until such refinement occurs, the estimate as to specification of the data region associated with the data node must be as contingent and as dynamic as the estimate of the nodes likelihood for being correctly placed in the data base in the first place. The result of these rationalizations is that effective and efficient data access synchronization is only as good as the elements of resource decomposition with which it has to work, and since data nodes cannot always provide reliable or precise data regions into which to map themselves for access synchronization purposes, the task of data access synchronization cannot be handled in a sufficiently localized way by data regions alone.¹ Later sections will

¹ Notice, of course, that while nodes with fairly non-descript time bounds (using the speech system example) may be locked without unduly interfering with the locking of
describe various experiments (which have been performed using the HSII speech-understanding system) in which the efficiency and process interference issues relating to the choice of the data access synchronization structure are explored.

Another reason that region locking could not effectively entirely supplant node locking relates to the granularity of region specification. For efficiency reasons, primitive data regions are defined so as to create equivalence classes among the n-tuples possible in the abstract database. The number of these equivalence classes is chosen to provide sufficient subdivision along each abstract data dimension so as to minimize undue process interference during region locking, while also trying to minimize the number of primitive region units in order to reduce the system overhead associated with maintaining the data access synchronization structure necessary for each such region unit. If it becomes necessary to specify rather coarse dimensional subdivisions, then the prospects of excessive process interference may be increased, since locking a region to access certain of the nodes in that region may inadvertently restrict access to other nodes in that region.\(^1\) While this situation certainly arises when region elements are too coarsely defined, a similar, but unavoidable, form of interference can also occur during region locking. Consider the possibility of having several competing alternative data hypotheses (nodes) within a given data region in the database; this is certainly possible due to the inherently contingent nature of data nodes. If a process were to gain access to any subset of these overlapping nodes via region locking, then access to all such overlapping nodes would be denied to any concurrently executing process. Thus, it is easily possible for the careless use of region locking to decrease the amount of effective parallelism in the system; results from experiments dealing with this problem in a particular implementation of the HSII organization will be presented in a later section.

Another potential problem in region locking involves the time-varying nature

---

\(^1\) Rather extreme cases of this coarseness of dimensional subdivision might include treating an entire dimension as having only one subdivision (i.e., not subdividing the dimension at all), or even ignoring a dimension in an attempt to simplify the locking structure. Whether such measures are overly restrictive depends, of course, on the process access patterns along those dimensions, the object being to maximize potential parallelism, while minimizing system overhead and process interference.
of the region descriptors of data nodes. This problem, mentioned above, will be described in more detail below, when the interactions between node locking and region locking are discussed. In any case, despite the potential pitfalls regarding the careless use of region locking, the mechanism is still necessary for access synchronization in dynamically expanding shared data structures.

Deadlock Avoidance: The Data Access Allocation Mechanism

The preceding sections have proposed two methods of data access synchronization for controlling the access of processes to the shared data base. The first method assumes each node existing in the data base to be an allocatable resource, capable of being assigned to a single process for exclusive access (locking) or to more than one process for read-only access (examining), but not both at once. The second method allocates regions of the abstract data base for locking or examining (but not both at once for the same region) without explicit specification of any nodes that might overlap or be contained within that region. Since both schemes allocate access rights to the same data base, they must be integrated and cooperate with one another so as to avoid data deadlock problems, as well as being individually capable of preventing such deadlocks. Considering node locking, and assuming for a moment no interference from region-locks, two processes may become deadlocked\(^1\) if each may request and gain access to data base nodes in an arbitrary order. For example, suppose process \(P_1\) desired exclusive access to nodes \(N_1\) and \(N_2\), and process \(P_2\) desired exclusive access to those same nodes. Now, if \(P_1\) managed to lock \(N_1\) first and \(P_2\) locked \(N_2\) first, then a deadlock situation would arise, assuming neither process will relinquish its acquired resources until its total request has been fulfilled: \(P_1\) will insist upon obtaining access to \(N_2\) (which \(P_2\) now controls) and \(P_2\) will insist upon obtaining access to \(N_1\) (which \(P_1\) controls), and neither process will be able to proceed until the other relinquishes its holdings, but neither will relinquish its holdings until its total request is satisfied -- a deadlock, to be sure. Assuming, either in the interest of resource allocation fairness or for the purposes of reducing system overhead, one wanted to retain the rule that no process will be forced to relinquish its resource holdings once it has acquired them (a non-preemptive allocation policy (Coffman, et al., 1971)), one way to avoid the possibility of data synchronization deadlock is to linearly order the set of resource

\(^1\) Deadlocking is a condition in which neither of two processes can proceed until each relinquishes some previously acquired resource to the other, and in which neither process will relinquish its resources until its total resource request is satisfied.
objects and always request access to those objects according to their assigned order (Brinch Hansen, 1973). Thus, in the example given above, if the nodes were ordered according to their subscripts, and the two processes abided by this ordering when they requested access to the nodes, then each would request \( N_1 \) first, and only one, say \( P_2 \), would succeed (the other being temporarily blocked) and be able to acquire \( N_2 \) (assuming an unknown third process did not have it locked, the third process having no interest in \( N_1 \) and therefore not being blocked waiting for \( N_1 \)). Then, when \( P_2 \) was finished with \( \{N_1,N_2\} \), the nodes would become available and \( P_1 \) could proceed, locking first \( N_1 \) and then \( N_2 \).

This linearly ordered approach to data access synchronization with a non-preemptive data allocation policy is the one followed in the HSII organization. Recall that every node in the shared data base is assigned a unique identification index (called a node-ID) at node creation time. These indices impose a linear ordering on the set of all nodes in the data base (hypotheses and links being treated equally as nodes for data access synchronization purposes). As a further ordering mechanism, each node will exist at exactly one information level throughout its existence in the data base, the set of all possible information levels being linearly ordered. Thus, the nodes of the data base may be divided into subsets according to their information level numbers, and any

---

1 Notice that requests can actually be honored in any order if all outstanding data access requests for all processes are requests for read-only (examine) access only. It is only in the case of requesting exclusive access (locking), or in requesting a combination of read-only and exclusive accesses, that the possibility of deadlock arises. Since access requests may arise at arbitrary points in time (given the asynchronous nature of HSII processes), both forms of data access request are treated uniformly in HSII.

2 The ordering according to information level in the HSII speech system is implemented by associating each data node with a lexicon, the set of available lexicon names being linearly ordered. The lexicon associated with a node defines the semantics of the information index contained within that node; for example, an information index of 23 in the word-lexicon might represent the word "apple." While the concepts are approximately equivalent, a lexicon name is used rather than an information level name because an information level may embody several defining lexicons (the information level being useful for a conceptual decomposition of the knowledge base of the task, while the individual lexicons provide a more detailed decomposition for implementation purposes); thus, using the lexicon name rather than the information level name gives a finer decomposition of the abstract data base. As an example, in the speech system, the segmental information level contains two separate lexicons, one for specifying segment names and segment boundaries, and one for specifying points of maximum/minimum amplitude in the input utterance.
given node may be represented by the pair, <node-ID, information-level>. Similarly, elemental data base regions may be described by n-tuples involving these same information levels and appropriate measures along the other dimensions defining the abstract data base region.\(^1\) Enumerating the data base regions thusly thereby imposes a linear ordering on those regions. However, note that the linear ordering of nodes is not disjoint from the linear ordering of regions, since they are both describing approximately the same composite object, the shared data base. Therefore, the locking mechanism must merge these two orderings appropriately to avoid any possibility of data deadlock when elements of each ordering are simultaneously involved in a data access request, whether the request be from one process or from several processes at once.

The following pseudo-Algol program describes how the merged linear ordering is accomplished to result in a deadlock-preventing non-preemptive data access synchronization scheme, using the example of the HSI speech-understanding system data base dimensions of lexicon name and utterance time to describe the data base regions. Similar nested loop structures would apply for any other chosen set of abstract data base dimensions. Note that the region descriptors associated with every data node provide the necessary node-to-region correspondence by specifying the data node's locality in the data base in terms of a corresponding data base region.

\[
\begin{align*}
\text{by lexicon name do} \\
\text{by time region number do} \\
\quad \text{begin} \\
\quad \text{lock/examine any region starting on this time region number;} \\
\quad \text{by node-ID do} \\
\quad \text{lock/examine any node starting on this time region number;} \\
\quad \text{end;}
\end{align*}
\]

To enforce this ordering, the various processes are not permitted to execute the primitive lock/examine operations directly, lest they violate the proper node/region locking order between successive primitive lock operations. Instead, the processes must first accumulate references to the nodes and regions to which they desire exclusive or

\(^1\) In the HSI speech system, data regions are essentially abstract time regions further qualified by the information level (lexicon) of the region; data regions are described by pairs of the form <utterance-time, lexicon>.\]
read-only access. This set of node and region references is then passed to a HSII operating system function (called LOCK!) which orders the requests appropriately and acquires access to those nodes and regions (according to the loop structure indicated above) on behalf of the requesting process, causing the process to become blocked should any request be currently unobtainable and returning control to the process when the lock operation is completed. The process executes a corresponding unlock system function (called UNLOCK!) when it is finished with its data accessing, thereby relinquishing its access rights to all the nodes and regions which were previously acquired for it. A process is not permitted to issue two system lock requests (via LOCK!) without performing an intervening unlock operation (via UNLOCK!), for the same reason that no process is permitted to execute the actual primitive lock operations directly.

The preceding locking loop structure describes the ordering mechanisms necessary for data deadlock avoidance; the following pseudo-Algol programs describe the various primitive access-granting mechanisms within HSII.

---

2 These references may be accumulated in several ways. The most obvious is for the requesting process to collect the node-ID's of those data nodes to which it seeks access rights, accumulating these node-ID's by direct referencing or through associative retrieval mechanisms. Another way, which is discussed in more detail below, involves tagging, where a process may place markers on various nodes; and when the process wishes to acquire access rights to those nodes, the markers (tags) may be presented to the LOCK! routine which will then order and acquire access to the nodes so tagged. Similar referencing techniques apply to data regions. The problem addressed by these referencing techniques is that quite often a knowledge source does not know in advance the set of nodes to which it would like to acquire access rights: it is possible that this set will become known only after searching the graph structure of the blackboard or after performing an associative data retrieval from the blackboard. In either case, mechanisms must be provided so that node references may be dynamically acquired, while maintaining the necessary data integrity with respect to data assumptions made during this node-reference collection process. Problems relating to data assumption integrity are discussed in more detail in subsequent sections.

1 This unlock function is more inclusive than it really needs to be, since the order of unlocking does not matter with respect to deadlocking issues. The processes could just as well unlock pieces of their locked resource set as soon as they are finished accessing them.

2 In the case of allowing partial unlocking, this constraint would be changed to specify that no system lock request could be issued as long as the calling process still had outstanding data locks.
lock.node:
  repeat
  begin critical section wrt mutex[level(node)];
    if no one else has node locked
      and
      region.descriptor(node) does not infringe on
      anyone else's region-locks
      then begin
        mark node as locked by caller;
        repeat
          if no one else is examining node
            and
            region.descriptor(node) does not infringe on
            anyone else's region-examines
            then return
            else wait for examiner to finish;
        end
        else wait for locker to finish;
    end critical section;
  end;

unlock.node:
  begin
    remove lock mark of caller from node;
    signal first waiting locker or examiner for level(node), if any;
  end;

comment The wait operation releases the critical section mutex semaphore before blocking the waiting process and placing it on a FIFO queue which contains all processes waiting for lock or examine access to a node or a region at the level corresponding to the mutex semaphore;

comment Since the waiting processes are grouped according to level number, rather than according to the explicit resource requested (due to the intermingling and overlapping of nodes and regions), the first process on the wait queue may not necessarily be interested in what is being unlocked (this time), but someone else in the queue might be. Therefore, the signal processing must have a chaining effect: if the first blocked process on the wait queue cannot benefit from the newly released resource (and hence executes another wait), then allow the second blocked process to try to take advantage of the new resource, etc., until either some process succeeds (and is removed from the wait queue, although perhaps not from the front of the queue) or all blocked processes in the queue have had a chance to
resume their locking attempt (but could not use this newly released resource). The latter case can arise in the locking of regions. For the sake of efficiency, regions may be locked according to sequences of values along the various dimensions defining the region (but always with a constant information level index), rather than being simply pointwise defined in the region space. Thus, it may be possible that a node or region be released which is not inclusive enough to allow pending access requests to proceed. Notice that, although this form of "grouped region locking" still avoids deadlocks, there is a possibility (however slight in actuality) that some requesting process might be blocked indefinitely (as, for example, if one process requests a very large region, and everyone else is requesting and releasing small regions, it is possible that the large region will never become simultaneously available in its entirety, and the process requesting that region will be blocked indefinitely). Of course, treating the region requests on a pointwise basis in the region space would eliminate these potential hazards (at an increase in lock processing costs), but the chaining form of wake-up would still be required (since the queue represents requests from the same underlying data resources, but via two separate mechanisms, node-locking and region-locking);

```examine.node:
repeat
  begin critical section wrt mutex[level(node)];
    if no one else has node locked
      and
      region.descriptor(node) does not infringe on anyone else's region-locks
      then begin
        mark node as examined by caller;
        return
      end
    else wait for locker to finish;
  end critical section;

unexamine.node:
begin
  remove examine mark of caller from node;
  signal first waiting locker for level(node), if any;
end;
```
lock.region:
repeat
begin critical section wrt mutex[level(region)];
  if region requested does not infringe on
    anyone else's region-locks
    and
    region requested does not infringe on
    region.descriptor of any locked node
then begin
  mark region as locked by caller;
  repeat
    if region requested does not infringe on
      anyone else's region-examines
      and
      region requested does not infringe on
      region.descriptor of any examined node
    then return
    else wait for examiner to finish;
  end
else wait for locker to finish;
end critical section;

unlock.region:
begin
  remove lock mark of caller from region;
  signal first waiting locker or examiner for level(region), if any;
end;

examine.region:
repeat
begin critical section wrt mutex[level(region)];
  if region requested does not infringe on
    anyone else's region-locks
    and
    region requested does not infringe on
    region.descriptor of any locked node
then begin
  mark region as examined by caller;
  return
end
else wait for locker to finish;
end critical section;
unexamine.region:
begin
   remove examine mark of caller from region;
   signal first waiting locker for level(region), if any;
end;

Notice that the preceding routine descriptions contain references to the region descriptors of data nodes that may be affected during the locking process. The values returned as the results of these references represent assumptions that the lock-requesting process is making about the data base locality of those nodes, and, as such, care must be taken so as not to allow these assumptions to be violated once the locking requests of this process have been granted, lest the possibility of data access race conditions arise (due to more than one process acquiring write-access rights to the same data node). As an example of how such a race condition might occur, consider the case of the HSII speech system, where the region descriptors of nodes are expressed as a sequence of utterance-time segments at a given information level, the sequence of time segments being based on (knowledge-source modifiable) time boundary fields within the node. Suppose process P₁ locks a region specified by utterance times t₁ to t₅ at some information level, thereby implicitly gaining exclusive access to a node N₁ lying within that region; and P₁ then alters the time boundary fields of N₁ to be t₇ and t₉, thereby actually removing it from the region previously locked by P₁. Suppose then another process P₂ were to lock the region t₇ to t₉ at the same level as N₁, thereby enclosing the new position of node N₁. Depending on when the region descriptor of N₁ is updated to reflect the change in data locality specified by the changes made by P₁ in the time boundary fields within the node, a race condition could occur. If the region descriptor update were made simultaneously with the time field changes, then upon performing the field changes, N₁ would no longer be implicitly locked by the region to which P₁ had previously gained access; and upon P₂’s region locking, N₁ would be implicitly locked for exclusive access by P₂.

While this particular situation would not cause a deadlock, it would violate the premises of the locking structure: one of the applications of region-locking is to gain control of the set of nodes within a region without having to undergo the expense of individually referencing and locking each such node. If, in the course of the execution following such a region lock, this set of nodes were to change due to the alteration of
the region descriptors of any of the nodes, the process which had requested the region lock could no longer assume exclusive access to those altered nodes. As a possible solution to this problem, a process could request explicit access to those nodes which might be susceptible to region descriptor changes due to field changes by the requesting process (e.g., in the case of the speech system, those nodes on which time field modifications are to be performed); and all other node access requests could be handled through the implicit referencing of the region lock (including, of course, access rights to those areas of the data base in which new nodes are to be created). However, even in the case where nodes susceptible to varying region descriptors are explicitly locked, the region descriptor must not be permitted to vary for the duration of the lock, because the descriptor could be changed to fall into a region which could be subsequently locked by a competing process which would then think it had exclusive access rights to the same node (even if it did not care to modify the time boundary fields, for the speech system example) -- a definite conflict of interests, and an invitation for data access race conditions.

As a result, in order for node-locking and region-locking to cooperate with one another, the region descriptor associated with any locked node (whether the node be locked explicitly by node-locking or implicitly by region-locking) must not be updated (as might be required due to modifications to any data-locality-defining fields of the locked node) until that node is no longer locked (by node-locking or by region-locking). Thus, the system unlocking function (UNLOCK!) should be responsible for preforming any required updating of any node region descriptor, such updating to be done prior to the actual release of the node or region lock but after the lock-requesting process is finished accessing the node.

Data Base Monitoring

While the focus of the preceding sections has been on avoiding indirect interprocess interference by synchronizing the data access patterns of the various asynchronously executing parallel processes, the present sections will investigate the methods and problems of interprocess communication in this multiprocessing organization. Since the various constituent knowledge sources are to be independently developed and are not to presume the explicit existence of other knowledge sources, communication among these knowledge sources must necessarily be indirect. This
communication takes two primary forms: data base monitoring for the occurrence of data events which violate prior data assumptions (tags and messages), and data base monitoring for collecting pertinent data event information for future use (local contexts and precondition activation). The following paragraphs will discuss these forms of data base monitoring and their relationship to the data access synchronization mechanisms discussed above. The effects of monitoring and access synchronization will also be related to the various data access operations available to executing processes.

Data Integrity

Since precondition and knowledge-source processes are not guaranteed to be executed uninterruptedly, these processes often need to assure the integrity of various assumptions they are making about the contents of the data base; for should these assumptions become violated due to the actions of an intervening process, the further computation of the assuming process may have to be altered (or terminated). One way to approach the problem of data integrity is to guarantee the validity of data assumptions by disallowing intervening processes the ability to modify (or perhaps even to examine) critical data. However, this approach to data integrity preservation is unnecessarily restrictive and would result in a reduction in the potential parallel activity within the system, which otherwise might be used to advantage in a hardware multiprocessing environment. An alternative approach is to provide a means by which data assumptions may be indicated in the data base; and whenever a data event occurs which invalidates an assumption, a message is sent to the process making the assumption. In the meantime, the assuming process can proceed without obstructing other processes, until such time as it intends to modify the data base (since data base modification is the only way one process can affect the execution of another). The process must then acquire exclusive access to the parts of the data base involved in its prior assumptions (which parts will have been previously tagged in the data base to define a critical data set, as will be described below)\(^1\) and check to see whether the assumptions have been violated. If a violation has occurred, the assuming process may wish to take alternative action; otherwise, the intended data base modifications may be made as if the process had had exclusive access throughout its computation. The HSII organization provides mechanisms to accomplish both of these forms of data integrity

\(^1\) Actually, the requirement is that no other process be able to write to these parts of the data base.
assurance: the various data base locking mechanisms described previously provide several forms of exclusive or read-only data access; and the data tagging facility described in the following section allows data assumptions to be placed in the data base without interfering with any process' ability to access or modify that area of the data base (with data invalidation warning messages being sent by data base monitors whenever the assumptions are violated).

Tagging

During the course of the execution of a process, the process will most likely make various read accesses to the shared data base, and based upon the data values that were read, a particular course of action will be chosen and modifications to the shared data base will follow as a result of that action. The resulting modifications that the process performs upon the shared data base quite often assume that the data base has not changed in any significant\(^1\) manner relative to the prior read accesses upon which the modification decision was based. In a multiprocessing environment, where multiple processes may have concurrent access to the shared data base, it becomes necessary to provide a mechanism whereby a process can be assured that no such significant modifications will be made to a given field between the time an assumption is made based on a read access to that data base field and the time an action is taken based on that assumption. If the set of nodes or regions involved in these assumptions and actions is known in advance, then the process could request those data resources be locked (exclusively or for read-only access, as appropriate) prior to forming any assumptions based on their contents and then released when preservation of the integrity of those assumptions was no longer necessary. This solution would certainly provide the degree of data state preservation required by the examining process, but the cost of increased potential interference with the execution progress of other processes might be significant. First of all, the amount of execution time necessary to determine whether the proper data conditions hold for the process to make the final data base modifications might be arbitrarily long. Secondly, the process may be able to make various alternative decisions based on the order and outcome of its data

\(^{1}\) Whether a data field change is significant or not depends, of course, upon the previous field value and the assumptions made with regard to that value. The decision as to which changes are significant, and determining the resulting course of action, is accomplished by an assumption revalidation sequence which is performed prior to making data base modifications based on those assumptions.
observations, thereby perhaps requiring much more of the database to be locked than
may actually be used in the final analysis (recall that, for deadlock avoidance reasons,
this "over-locking" cannot be reduced by cumulatively locking sections of the database
as the need arises). Thirdly, even if the process knew in advance exactly which nodes
and regions it needed to safeguard from intervening alteration, locking just those nodes
and regions might still overly restrict the progress of other processes because the
granularity of the locking structure (having chosen the data node and the abstract data
region as the elemental objects subject to locking, for reasons of efficiency) may well
be coarser than the granularity required for assuring the integrity of assumptions made
about particular data field values of particular nodes. That is, data nodes themselves
are small data structures, and one process' desire to guarantee the integrity of a
particular field of that node over a certain period of time should not prohibit another
process from reading or writing some other field in that node, assuming the individual
fields are independent of one another.¹

As might be expected, a mechanism was created within HSI to alleviate these
problems involving the assurance of data assumption integrity over time, while also
helping to prevent data deadlock and process interference problems. The mechanism is
called tagging, and it consists of a process associating an identifier (a tag-ID) unique to
the process with a data field of a node in the shared database. This tagging operation
is usually done as the process is reading or writing that data field (although the field
may be tagged independently of a read/write operation), with the action of reading (or
writing) and then tagging being an indivisible operation. Unlike the locking operation,
this tagging does not restrict the subsequent access rights of any other process to the
field being tagged, and, as such, a series of tagging operations may be done by a
process in any arbitrary order, without having to consider deadlock problems (since

¹ Of course, one possible solution is to select the data field, rather than the data node,
as the grain of data locking. This was deemed inappropriate due to the observation
that processes will often reference several fields from any given node (in fact, that is
why fields are collected together into the record structures known as nodes); and
using the node as the element of locking avoids the prohibitive expense involved in
locking several fields instead of a single node. Furthermore, if fields were chosen as
the element of locking, then some mechanism analogous to the region descriptor for
nodes would have to be devised to coordinate field accessing and region accessing.
This choice was a design decision in HSI based upon efficiency considerations and
expected database access patterns, and, as such, the decision may not be
appropriate in other contexts.
tagging is not a resource allocation operation). By performing this tagging operation, the process is indicating to future modifiers of the tagged data field that it is concerned about such modifications. Thus, whenever another process subsequently modifies the tagged field, the tagging process will be sent a warning message to that effect so that it may reevaluate its assumptions based upon the new value of the field. It is entirely possible that several processes may tag any given field, in which case each of the tagging processes is sent a message whenever the field is modified. This tagging operation may also be applied at the node level, such that whenever any field of the tagged node is modified, a warning message is sent to all the processes which had tagged the whole node (as well as to those processes which has tagged the modified field). Of course, whenever a process is no longer concerned with monitoring a particular data field (or node), the process may remove the tag (via an untag operation) from that data field (or node) in much the same manner as the tag was originally placed on the field (or node). Furthermore, it is the responsibility of any process which receives warning messages to dispose of such messages when it is no longer interested in their contents. Similar tagging operations are also applicable to abstract data regions: a process may tag a data region so as to receive a message whenever a particular field of any node overlapping the region is modified, or a process may tag a region so as to receive a message whenever any field of any node overlapping the region is modified.

The tag-ID is essentially an alias for the process to which it belongs. Tag-ID’s can be created as needed by executing processes, thereby allowing any given process to have several aliases with which it can tag pieces of the data base. This potential multiplicity of aliases is important if the process wishes to react differently to warning messages received due to different tag-ID’s. Furthermore, any given process may tag any field (or node or region) in the data base with more than one of its tag-ID’s, thereby allowing the possibility of receiving several warning messages upon subsequent modification of the multiply tagged field.

**Tagging and Data Access Synchronization**

While the ability to tag data fields has a usefulness in its own right with respect to data base monitoring so as to reduce data access interference among processes, the principal use of tagging is in conjunction with the various forms of data access synchronization. Recall that individual processes cannot be permitted to
accumulate data nodes or regions for exclusive or read-only access in a random, piecewise fashion, lest they violate the linear ordering constraints necessary to prevent data deadlocking. However, the most natural manner for a process to select which nodes or regions of the data base it might be interested in is for that process to search the data base (perhaps associatively, as described previously, or directly, having been given a starting point from which to search, or both). Since the process certainly cannot predict the order of discovering the nodes or regions in which it might be interested (otherwise, why is it searching the data base?), some way must be provided in which to accumulate these references in an arbitrary order for later presentation to the high-level system locking primitive, such that the proper ordering of the requests may be done at that later time. Of course, in the meantime, the process must be given some assurance that the field which it had read and upon which it had based assumptions depending upon the observed value will not be modified unbeknownst to the assuming process.

Clearly, this is an ideal opportunity for the use of tagging. As a process peruses the data base and finds a data field of interest, if the process wishes to base a decision on the value of that field, then it can tag the field (or its node or its region) as it reads the value (as an indivisible operation) and thereafter be notified (via warning messages) of subsequent changes to that field. It is currently the responsibility of the process receiving a message to poll for the receipt of any warning messages, although a message interrupt facility could also be envisioned. Performing interprocess communication reception by polling allows messages to be received and handled at the convenience of the receiver, thereby avoiding having to handle unstable local data base situations, etc., as would have to be done if message reception were performed at an interrupt level. Thus, as its data base search continues, a process will tag those fields, the preservation of whose integrity is necessary in making all the assumptions in following a particular search strategy. The result of such a data base search is usually the discovery of a node or region (or a collection of such) deserving of modification by the searching process. At this point in its execution, the process will request exclusive access to the nodes and/or regions which it wishes to modify. Also at this time, the process should check for the continued integrity of its assumptions before performing the intended modifications. Thus, upon requesting exclusive access to the nodes or regions to be modified, the process may also present to the high-level data access synchronization primitive (the LOCK! operation previously described) a set of tag-ID's
which were used in tagging the various fields, nodes, and regions, the integrity of whose contained values was deemed necessary to the fulfillment of the intended data base modifications. Then, in addition to acquiring exclusive access to the particular nodes and regions to be modified by the process, LOCK! will also include in the acquired resource set all the nodes or regions tagged by members of the tag-ID set presented to LOCK! (these nodes or regions may be acquired for either exclusive access or read-only access). The nodes tagged by members of the tag-ID set parameters to LOCK! represent the critical set of nodes and regions for this LOCK! operation, so called because the integrity of certain fields within this set may be critical to the successful completion of the part of the caller process bounded by this call to LOCK! and its matching UNLOCK! operation. As a bonus, LOCK! will return to the caller process the number of messages pending for that process based upon the presented tag-ID set, so the caller process can do a quick revalidation check on the integrity of its tagged assumptions. If the number of messages is zero, the integrity of the prior tagged assumptions was maintained, and the intended data base modification can proceed safely, since the further integrity of the tagged fields is now assured by the LOCK! operation itself. The effect is just as if the tagged fields had been locked all along by the tagging process. On the other hand, if the number of messages detected by LOCK! is non-zero, the caller process can enter a revalidation sequence, where, having preserved the tagged fields from further modification by calling the LOCK! routine, the caller process can reevaluate its decision to make its proposed data base modifications in light of the warning messages it has received. If, upon completion of this revalidation sequence, the process determines its assumptions have been violated, it may abort any further plans to modify the data base, or it may pursue an alternative strategy. However, it is possible that the modifications made to the tagged fields by other processes did not actually violate the assumptions of this process (as when this process wanted to assure that a given field value exceeded some threshold, and an intervening process raised the value even higher, thereby preserving the essence of the assumption although triggering a warning message nonetheless\(^1\)), and in such a case, once the

\(^1\) Of course, this superfluous warning message could have been avoided by allowing more complex tags which could embody the essence of the assumption made concerning the field being tagged. The tags used in HSI! are very simple: they contain no specification of the assumptions made, and they cause a warning message to be sent whenever any modification is performed upon the tagged field (or node or region).
flagged changes have been revalidated, the proposed data base modification may be
made as if no messages were received. Thus, the operation of data tagging may be
used in conjunction with the data access synchronization routine (LOCK!) to resolve the
problem of resource ordering so as to prevent data deadlocking: if a process does not
know the contents of the resource set to which it would like to acquire access rights,
then the references to the elements of the set may be accumulated in any order via
tagging, and the set of tag-ID’s used in tagging presented to LOCK! when all necessary
references have been accumulated.

Temporary Locking

Often a process wishes to read or write a single field of a single node at a
time, perhaps with concurrent tagging (as is the case when a process is building up the
set of references to be used in a subsequent LOCK! operation). In such instances,
exception may be made to the rule of disallowing processes to acquire access to
individual nodes without going through the ordering operations implicit in LOCK!. As a
result of implementation efficiency requirements, special consideration is given
processes trying to access a single resource; access rights to that single resource are
granted directly (without having to undergo the ordering overhead of LOCK!), such
access rights being granted only for as long as the elementary data field read or write
takes. This single-resource access is called temporary locking, and is controlled by the
primitive access routines, rather than by the user process itself. For example, if the
process issues a read request for a particular field of the data base, and this process is
not currently in the midst of a LOCK! section (that is, the process does not currently
have anything else locked, either for exclusive or read-only access), then the read
routine will issue a read-only access request for the node containing this field on behalf
of the calling process. Upon having acquired the node, the read operation is performed,
along with any requested concurrent tagging operation; the data read is returned to the
calling process; and the read-only temporary lock is automatically relinquished. The
temporary locking for a write request proceeds similarly, with the calling process never
retaining access rights to the node beyond the completion of the execution of the
accessing routine in either instance. Such locking is necessary for both data field
modification and data field examination, since some composite data fields cannot be
accessed (read or written) in a single (hardware indivisible) machine operation (not to
mention conflict during memory cycles in a multiprocessor); thus even single field
accesses must be protected from intervening modification by other processes.
Data Events

By this time, it should be evident that the underlying organizational structure of HSII is very data-directed. In line with the goal to provide a framework for the easy integration and configuration of knowledge-source processes which are to communicate and cooperate with one another, while at the same time remaining essentially independent of one another, the primary means of coordination of the overall effort is concentrated in the blackboard-like data base. It is in this data base that precondition processes are to find the data patterns matching the conditions necessary to activate their associated knowledge-source processes; and it is into this data base that the instantiated knowledge-source processes are to record the results of their computations, so that other preconditions might be satisfied and further knowledge sources instantiated and executed until a data pattern signifying problem solution is generated (or until failure is indicated). The cycle is one of hypothesize-and-test, where some data patterns satisfy preconditions for knowledge sources whose function it is to create new elements and structures in the data base, and other data patterns satisfy preconditions for knowledge sources whose function it is to test and verify or modify the data elements extant in the data base. But the cycle of hypothesize-and-test is intended to be quite asynchronous in that many hypothesizing processes may be operating concurrently throughout the data base; and many verifying processes may be operating concurrently with the hypothesizing processes, testing previously generated hypotheses.

How is all this concurrent data-directed activity to be initiated and sustained? Given an initial state configuration for the data base (which is, in fact, essentially empty) and an initial process structure configuration (which defines the initially active processes), subsequent activity will be completely dependent upon changes which take place in the data base (except for an initial process or two whose purpose it is to accept problem-defining input from an external source and store that information in the data base in an appropriate form, these processes being themselves activated by an external stimulus, but producing internal data base stimuli which are to serve to activate certain other knowledge sources via their data generation activities). Thus the entire system is activated by data events, changes occurring in the state of the blackboard data base caused by the introduction of new data elements or the modification of existing elements. The preconditions of knowledge sources are expressed in terms of data events: whenever a specified combination of data events has occurred (perhaps
combined with various relationships of these data events to previously existing values or structures in the blackboard data base, then the corresponding knowledge source should be instantiated and executed to respond to those data events (thereby producing further data events). Preconditions themselves are also activated by data events, with the fields of data nodes being selected such that a single field modification is usually sufficient to activate a precondition evaluation; and often that single field change will result in the instantiation of a knowledge-source process which will respond to that data event.

Data events are caused whenever a process executes a primitive system operation which modifies the blackboard data base, whether this be by executing a system routine to write a field in a data node or by executing a system routine to create a new node (hypothesis or link) in the data base. These modification and creation routines are then responsible for propagating the notification of the occurrence of a data event to all interested processes.

Local Contexts

Interprocess communication (and interference) occurs mainly via the global data base, as a result of the design decisions involved in trying to maintain process independence. It is therefore not surprising that the mechanisms necessary to bring about the desired process cooperation and independence are based on global data base considerations. Recall that the global data base is intended to contain only dynamically current information. Since preconditions (being data-directed) are to be tested for satisfaction upon the occurrence of relevant data base changes (which are historical data events), and since neither precondition testing nor action execution (nor the sequential combination of the two) is assumed to be an indivisible operation, localized data bases must be provided for each process unit (precondition or action) which needs to remember relevant historical data events. These localized data bases, called local contexts in HSI, provide personalized operating environments for the various precondition and knowledge-source processes. A local context preserves only those data events1 and state changes relevant to its owner, and the creation time of the local context (i.e., the time from which it begins collecting data events) is also dependent.

---

1 The information which defines a data event consists of the locus of the event (i.e., a data node name and a field name within that node) and the old value of the field (the new value being stored in the global data base).
upon the context owner. Any given local context is built up incrementally: when a modification occurs to the global data base, the resulting data event is distributed to the various local contexts interested in such events. The various primitive data modification routines (or node creation routines) are responsible for the distribution of the data events which it causes, just as these modification routines are also responsible for sending warning messages to those processes which had tagged the data field being modified. Thus, the various local contexts retain a history of relevant data events, while the global data base contains only the most current information.

Consider the following time sequence of events:

- \( t_1 \): start precondition evaluator PRE (triggered by data base changes)
- \( t_2 \): PRE instantiates a knowledge-source process KS
- \( t_3 \): end PRE
- \( t_4 \): start KS
- \( t_5 \): after KS revalidation of precondition
  - \(<\text{computation}>\)
- \( t_6 \): KS modifies global data base
  - \(<\text{computation}>\)
- \( t_7 \): KS modifies global data base again
- \( t_8 \): end KS

PRE is activated to respond to changes (data events) occurring in the global data base. PRE should execute in the context of changes existing at time \( t_1 \) (since that context contains the changes which caused PRE to be activated). KS is instantiated (readied for running) at \( t_2 \) due to further conditions PRE discovered about the change context of \( t_1 \). Hence, PRE should pass relevant pieces from the context of \( t_1 \) as the initial environment in which to run KS.

KS starts to execute at time \( t_4 \), which, in the HSII model, may occur at any arbitrary time after \( t_2 \), being entirely dependent upon the judgment of the HSII knowledge-source scheduler. By time \( t_4 \), further changes could have occurred in the global data base due to the actions of other knowledge-source processes. So KS should examine these new updating changes (those occurring between \( t_1 \) and \( t_4 \)) and revalidate its precondition, if necessary, using the pieces of \( t_1 \)-context passed by PRE. After revalidation, KS assumes the updated context of \( t_5 \), and it proceeds to base its computation on the context of changes as of \( t_5 \).

When KS wishes to perform an actual update of elements of the global data
base at \( t_6 \), it must examine the changes to the global data base that occurred between \( t_5 \) and \( t_6 \) to see if any other knowledge-source processes may have violated KS's preconditions, thereby perhaps invalidating its computations. Having performed this revalidation and any data base updating, KS should update its context to reflect changes up to \( t_6 \) for use in its further computation. At \( t_7 \), KS must look for further possible invalidations to its most recent computations, due to possible changes in the global data base by other knowledge-source processes during the time period \( t_6 \) to \( t_7 \). When KS (which is an instantiation of some knowledge source) completes its actions at \( t_8 \), its local context may be deleted. However, the local context of PRE is not deleted upon PRE's completion at time \( t_9 \), for reasons of event detection continuity discussed in the following paragraphs.

Precondition evaluators and knowledge-source instantiations are alike in many ways within HSII. Both are computational processes which request machine resources from time to time, which requests are fulfilled as seen fit by the goal-directed HSII scheduler; both can access and modify the global data base (although usually the modification capability is exercised only by knowledge-source processes); both can retain a history of pertinent data events in a local context; both can cause the instantiation or reactivation of the other (although usually the instantiation capability is used only by precondition processes, with precondition processes themselves being activated indirectly through the occurrence of specific data base events). However, there is one major distinction between precondition processes and knowledge-source processes: the duration of their existence as processes. Knowledge-source processes lead a transient existence: they are instantiated by a precondition process to perform a particular action upon the data base, being given a specific local context in which to operate. Having performed this action, the knowledge source can make no further contribution until requested to do so by its precondition, at which point in time it will receive an entirely new local context in which to operate. Thus, knowledge-source processes behave as if they were asynchronous, parallel subroutines of their preconditions, being instantiated with a new local context upon demand and having no continued existence from one instantiation to the next. On the other hand, precondition processes must lead a continuing existence, since it is the function of a precondition to be constantly monitoring the data base for events and conditions relevant to the knowledge-source action associated with the precondition. The actual requirement that must be satisfied by a precondition is that it not miss any data base events that might
be relevant to its knowledge-source action. This requirement may be satisfied by assuring for every precondition the continuing existence of a local context which is capable of receiving data base events as they occur (even though an instantiation of the procedural component of the precondition might not exist at all times). Then, when sufficient events have been received by a given local context, the precondition associated with the context may be instantiated and attached to the context to act upon those events accumulated.

Due to differing requirements between precondition processes and knowledge-source processes regarding event detection continuity, local context maintenance for each of the process classes is different. As relevant data base events\(^1\) occur, precondition processes collect the events in localized *dynamic contexts*, the events being distributed to the various contexts by the data base handling process (or routine) which caused the event to occur. When sufficient events have been accumulated in the dynamic context,\(^2\) the precondition process is initiated to operate within that dynamic context. At the instant of initiating the precondition, its dynamic context must be preserved from further modification, since the precondition was initiated based on the contents of the dynamic context at that moment and it should be allowed to execute in exactly that context. Therefore, at the point of precondition initiation, the dynamic context is frozen and becomes the *static context*. At the same time, a new (initially empty) dynamic context is created for the precondition so that future initiations of the precondition will not miss any data events which might occur during the current precondition evaluation.

During the execution of the precondition, the procedural body of the precondition may reference only the global data base and its local static context. In

---

\(^1\) Which data base event types are considered relevant for a given precondition (or knowledge source) are prespecified in a description of the contextual requirements of that precondition (or knowledge source). Then, for any given process instantiation, only those events are collected for its local context which fall into one of the requested event classes.

\(^2\) The determination of when "sufficient" events have been accumulated is accomplished by effectively associating a pre-precondition with every precondition. The pre-precondition is intended as a much simpler (i.e., computationally less expensive) test to determine whether it is worthwhile to initiate the full precondition evaluation (which may be arbitrarily complex). One such simple pre-precondition might be to initiate the full precondition whenever any single data event enters the precondition's dynamic context.
particular, it may not reference its associated dynamic context, which is collecting concurrent data base events for use by future precondition executions (although it may receive messages in response to tags which it places in the global data base). Based on an examination of the event history accumulated in its static context and based on the current state of affairs as reflected in the global blackboard data base, the precondition may find itself to be satisfied and hence instantiate a process instance of its corresponding knowledge source.\(^1\) The instantiation of a knowledge-source process involves creating a schedulable process unit consisting of the code body associated with the knowledge source and its own local dynamic context. The local dynamic context of a knowledge-source process is initialized to contain the contents of those parts of the precondition's static context which overlap the knowledge source's local context specification, combined with the contents of the precondition's dynamic context for the same context specifications, at the moment of instantiation.\(^2\) The invoking precondition may also pass an arbitrary set of input parameters to the new knowledge-source process. The initial dynamic context and the input parameters convey to the knowledge-source process the reason for its instantiation and present it with an initial global data base locus for action. Note that since the local context of the new knowledge-source process is "dynamic," it will begin collecting data base events immediately upon its creation, regardless of when the new knowledge-source process is actually scheduled for execution. Effectively, the historical knowledge of the knowledge-source process will reflect the events that have occurred since the last time its precondition was activated: these events will include why the precondition was activated, perhaps why the knowledge source itself was instantiated, and any events that have occurred since the activation of the precondition (which the precondition itself has not accounted for).

Before describing any further the actions of the instantiated knowledge-source process, consider the situation of the precondition once it has instantiated the knowledge-source process. Having performed its duty, the precondition may become dormant once again, awaiting a reactivation based on data base events which have occurred since the last time it was activated. These events are being collected (as

---

\(^1\) For the sake of program efficiency, an implementation of HSII could allow a single precondition program to govern the instantiation of one or more (usually related) knowledge sources.

\(^2\) This implies that the precondition context specification must include the context specifications of all of the knowledge sources which it might invoke.
always) in the precondition's local dynamic context; and the local static context, having served its purpose, may now be deleted. Depending upon the complexity and thoroughness of the pre-precondition that triggered the activation of the precondition, it is also possible that the precondition executed without being satisfied. For example, the precondition may have been satisfiable only upon detecting the occurrence of two events of a certain type, while the pre-precondition (being very simple) activated the precondition upon the occurrence of any single event of that type. In such a situation, the precondition would want to retain the knowledge of the occurrence of the first event for future reference, so that when activated again due to a second event occurrence, it would be able to recognize that event as the second of two (and thence be satisfied). This event retention may be accomplished by several mechanisms. First, the precondition has the ability to transfer notification of data events from its static context (to which it has free access) to its dynamic context (to which it has no read access), just as if the event had just re-occurred.\textsuperscript{1} Thus, in its next activation, the precondition could discover both event notifications and thereby be satisfied. The second means of information retention available to preconditions is more general. Since the dynamic context of a precondition is considered to have a continuing existence, one might associate with this dynamic context a more general state-saving mechanism, whereby from one activation of a precondition to the next, arbitrary status information may be retained by that precondition. Then, in the current example, the precondition might itself keep track of the detection of the first data event; and when reactivated for the second event, the precondition could thereby determine it to be the second event and be satisfied. Contrast this state-saving capability due to the continuing existence of precondition contexts to the more temporary data structures found during the subroutine-like existence of knowledge-source processes.

Returning now to the situation of the knowledge-source process which the precondition has instantiated, the time at which the knowledge-source process is permitted (by the HSI\textsuperscript{11} scheduler) to begin execution may be an arbitrarily long time from the moment of its instantiation. In particular, events may have occurred in the meantime which could have invalidated the knowledge source's precondition, thereby nullifying any right or reason for the execution of the current knowledge-source instantiation. Thus, it is the task of the knowledge-source process first to revalidate its

\textsuperscript{1} This data event transfer capability is known as data event retention, and is described in more detail below.
reason for existence, which revalidation may mimic the efforts of the precondition or may effect the revalidation by other means (such as aborting the knowledge-source instantiation if any events of a certain kind have occurred at all, regardless of their net effect). One especially effective method of detecting changes which may have occurred to particular pieces of the data base involves the mechanism of data base tagging (which was described in a previous section). In effect, a knowledge source or a precondition may mark (tag) various fields in the data base and be sent a message should any of the fields be subsequently modified by a process other than itself. In the case of the initial precondition revalidation by a knowledge-source process, the instantiating precondition could have placed tags in the data base with instructions to send any messages to the new knowledge-source instantiation. Then, as part of its revalidation effort, the knowledge-source process can check for any messages that may have been received as a result of the precondition's tagging. Having performed the precondition revalidation, the knowledge-source process is free to execute in the environment provided by its local dynamic context and its input parameters. Unlike a precondition, a knowledge-source process has no static context, but only a dynamic one (which was initialized by the precondition at the moment of knowledge-source process invocation). The dynamic context (which, of course, the knowledge-source process is free to read) in combination with the mechanism for data base tagging (by which the knowledge-source process is able to mark arbitrary data fields in the data base so as to receive messages warning of subsequent modifications) provide the knowledge-source process with the ability to keep abreast of concurrent developments in the global data base as it pursues its own activities. Whenever the knowledge-source process is about to modify the global data base and has gained exclusive access to the area to be modified, it can then perform a revalidation check before modifying the data base itself to be sure no other concurrently executing process has interfered with its assumptions regarding the data base. Upon completing its assigned action, the knowledge-source process may be terminated and its local context deleted.

The maintenance of a local context, once it has been created and associated with a particular process, is primarily the responsibility of the data base modification and node creation system primitives. These system routines are the only code bodies which can cause a data event to occur (since all data base modification requests must come through these routines); and, as such, it is natural to associate with these routines the responsibility of communicating the information regarding data events to all current
processes awaiting information on such events. As described above, this data event information is distributed to the appropriate dynamic contexts of currently instantiated processes.

There are basically two types of data events, *value-change events* and *superset events*, with corresponding types of local contexts (both static and dynamic). The first event class involves changes to particular fields of data nodes: in such data modifications, there is a well-defined locus of change (given by the node-ID of the node and the field name within that node), and there is a well-defined value for the change (in particular, the old value of the field being changed). A data event of this type, then, consists of the locus of the event and the old value of the field and may be represented as a triple, <field, node, old value>. When such a data modification is made, this data event triple is distributed to all dynamic local contexts willing to receive the event. In order for a local context to be able to receive this value-change data event, the local context must contain a *change set* (or *cset*) corresponding to the field name of the data event. If such is the case, the node and old value information contained in the event triple is entered into the change set corresponding to the field name for all dynamic contexts which contain this change set. Thus, the field name acts as a classifying mechanism by which to partition data events. While all data events arise from the modification of data fields (except for those events caused by the creation of new data nodes), it is sometimes convenient to group various changes so as to create supersets of events. For example, the data events relating to the modification of various rating fields may all be taken together as a more generalized "rating change," as well as being distributed as separate data events. Thus, it is possible for the local contexts of processes to contain *supersets* (or *sets*) corresponding to these field groupings; supersets are also possible for detecting such events as the creation of new data nodes, which events do not correspond to any field change. Being an aggregate event collection mechanism, supersets contain no old values, but only node-ID's. Except for detecting node creation events, the information provided by supersets is redundant with that possible through change sets; supersets are offered as an efficient way of maintaining several change sets at once, as long as the context owner does not need to distinguish among the various fields constituting the supersets and does not need to access the old field values. A *local context* (static and dynamic) may then be defined as a personalized collection of change sets and supersets for accumulating data events relevant to the owner process.
The owner process of a local context containing change sets which receive value-change data event notifications is assumed to be primarily interested in detecting net changes to a given field of a given node; and as such, if upon distributing a data event triple to a change set, the distributing routine detects the prior presence of the same node name in the same field change set, no alteration is performed to that change set, thereby leaving the previous old value intact. An exception to this policy of not updating old values occurs when the newly written data base field value is equal to an old value previously stored in a change set entry for that node: in this case, the change set entry for the node being modified is eliminated, since the net change over time has been nullified. This policy of reporting net field value changes, and thereby having at most one entry per node in any given change set, is a consequence of the intended asynchrony of the HSII organization. The number of changes to a particular field or the identities of the changers should not be of special import in the design of a knowledge source, because due to the scheduling flexibility allowed the HSII scheduler in allocating physical resources so as to best advance the progress of the overall computation, a particular knowledge source cannot in general depend on any particular execution sequence with respect to processes other than itself. Thus, for example, depending on the current assortment of executing processes, a particular field may be altered many times by many processes during the course of one instantiation of a knowledge source, and perhaps not at all during a subsequent instantiation. Furthermore, just as local contexts are maintained on a net change basis, so, too, are the warning messages sent as a result of modifying tagged fields. If a particular tagged field (or node or region) is altered several times before the tagging process gets around to polling for warning messages, only one message will be sent to represent all those changes to the tagged data object. Tags on data fields and change sets corresponding to those data fields are often used in conjunction: if the tagging process receives a warning message for a field change, it can retrieve the essence of that change by reading the old field value from the corresponding change set (if the process has this change set in its local context). In such usage, one would not want to receive multiple messages corresponding to a single net value change in a change set; hence, the net-effect handling of both local contexts and tag messages.

Since local contexts and tag messages operate on a net-effect basis, mechanisms must be provided whereby an executing process can restart the accumulation of these net changes. Such a mechanism might be invoked, say, during a
revalidation sequence, where a process would read accumulated warning messages and local context values and thence decide upon a course of action based on such data events; but the process would want to be made aware of any events subsequent to the reading. Thus, a means by which a process can simultaneously read and then delete messages is offered as a solution to the message-resetting problem; and a similar mechanism for simultaneously reading the current value of a data field as well as the old value of that field, and deleting the old-value entry, are provided to allow a process to reset the net change indicators of local contexts. The message-resetting mechanism is incorporated within the message receiving mechanism, the details of which are given via a pseudo-Algol program in the next section. The local-read-and-delete mechanism is incorporated within the read-old-field-value operation, which is also detailed in the next section.

Sometimes processes may find themselves to be prematurely activated, as might be the case when the triggering mechanism for precondition execution is very simple-minded (but efficient), and the precondition is activated only to find it is only partially satisfied (that is, perhaps, only some of the requisite set of data events have occurred by the time the precondition was triggered). Due to the way in which dynamic contexts are frozen to create static contexts upon precondition triggering, unless special means are provided, those data events which have been recorded into the static context during the past precondition execution will be lost (deleted) the next time the precondition is triggered. To resolve such situations, a data event retention mechanism is provided which can transfer data events of continuing interest into the dynamic local contexts of specified preconditions, as if the data event had just re-occurred. This mechanism is useful to preconditions in remembering data events which had been prematurely acted upon; it is also useful to knowledge-source processes in recreating data events which it could not handle properly but perhaps which could be handled by other knowledge sources associated with the precondition(s) which is to receive the retained event. The details of this event retention mechanism are given in a pseudo-Algol program in the following section.

There is one additional mechanism by which preconditions and knowledge-source processes can control the selectivity as to which data events are accumulated in their dynamic local contexts. This mechanism, called local context filtering, is similar to data base tagging except that the tags (called filters in this mechanism) are associated with dynamic local contexts rather than global data base fields. Unlike tags, which can
cause warning messages to be issued whenever a modification is made to a tagged field, but which cannot prevent the modification, filters act to select which data events will be permitted to be recorded in a given process’s dynamic local context, given the total set of data events presented to that context. Filters are used to assure relevance of collected data events to a particular local context. For example, a given precondition may be interested in global data modifications which occur to utterance-begin-time fields of nodes at a particular information level. But this standard local context definition may be overly broad for the purposes of the precondition; so the precondition may specify a further filter be placed on that local context to allow only data events caused by write operations possessing a certain filter-ID to be admitted to the local context. This might be useful in allowing a precondition to accumulate only those data events caused, say, by knowledge-source processes instantiated by that precondition. This filtering mechanism may also be specified in a negative manner, such that, for example, the precondition could request only those events be accumulated that were not caused by its own knowledge-source processes.

Local context filtering operates by having a process (precondition or knowledge source) attach a filter-ID (which is a dynamically created tag-like identifier) to a personal change set or superset. This filter may be positive or negative in nature, and multiple filters may be attached to any given change set or superset, thereby creating a predicate which must be satisfied before any data event notification can enter that part of the local context. The filter conditions default so as to specify no restrictions on potential data event notices. In subsequent write operations to the global data base, a process may specify a set of filter-ID’s to be associated with that write operation; and when the data event corresponding to that write operation is distributed to the various dynamic local contexts, these filter-ID’s will act as keys to get through the predicate testing of local contexts which possess filter restrictions. Local context filtering was introduced as an efficiency measure in order to avoid unnecessary processing that might occur as the result of receiving irrelevant data event notifications. The subdivision of local contexts into change sets based on the fields associated with nodes at a given information level, and into supersets of those fields, will satisfy the majority of local context needs in an efficient way; the addition of context refinement by event filtering provides a further mechanism to avoid costly unnecessary processing on the part of the receiving process.
Primitive Data Access Operations and Local Context Maintenance

The purpose of this section is to describe the details of the various data access and context maintenance mechanisms discussed above. This will be accomplished by presenting pseudo-Algol programs for the various operations which may be executed by a precondition or knowledge-source process.

There are effectively three primitive data access operations available to a user process within the HSII organization: reading and writing a field of a global data base node, and reading the old value of a field of a data node from a local context. As has been discussed above, these primitive access operations do more than just read or write data fields during the course of their execution (which execution may be viewed as an indivisible operation from the point of view of the calling process). The sequence of actions for these data access system functions may be characterized by the following pseudo-Algol code:

```algol
read.field.value:
begin
if not inside LOCK!
then temporary-lock node
else if node not LOCK!ed
    then return error;
    perform any tag/untag request;
    read current value from field;
    relinquish any temporary-lock;
    return field value;
end;
```

```algol
write.field.value:
begin
if not inside LOCK!
then temporary-lock node
else if node not LOCK!ed
    then return error;
read field to get old (current) field value;
perform any tag/untag request;
if current value = new value
    then return no action;
write new value in field;
send notification of data event to dynamic local csets
    (subject to local context filtering):
    begin
    lock a receiving dynamic cset;
```
if no previous old value for this node in cset
then put this old value there;
if old value in cset = new value
then remove old value from cset (since no net change);
unlock the cset;
end;
send notification of data event to dynamic local ssets
(subject to local context filtering):
begin
lock a receiving dynamic sset;
if no previous entry for this node in sset
then put reference to this node there;
unlock the sset;
end;
send warning messages to those who had tagged the field,
node, region, or sset;
activate relevant PRE's (those which got the old value inserted
into their csets or got a new sset entry), if their
pre-preconditions are satisfied;
relinquish any temporary-lock;
return old field value;
end;

comment Notice that local change sets (csets) and superset
(ssets) are allocatable shared resources (which may be
written upon by many processes but read by only the
owner process). Local contexts are in a resource class
separate from the allocatable resources (nodes and
regions) of the global data base. Also, local contexts are
only accessed one cset or sset at a time (and never in
groups), so the synchronization structure for local
context access may be done similarly to that of the
single-resource accessing of temporary locking, without
danger of deadlock;
read.old.field.value:
begin
lock your cset (STATIC for PRE's, DYNAMIC for KSI's);
read (via read.field) current (global) value of field;
if cset contains desired old value
then begin
  read old value;
  if deletion requested on old value
  then delete old value from cset;
end
else use the current value as the old value;
unlock your cset;
return both old value and current value;
end;

In addition to these direct data access operations, there are two auxiliary data access operations: data event retention (which is like writing to a dynamic context without the writer ever being able to read what was written), and warning message reception. Both of these operations are used by a process in maintaining and manipulating its local environment. The following pseudo-Algol code characterizes these system functions:

retain.data.event:
forevery precondition process to receive this data event do
begin
lock the dynamic local context element (cset or sset)
  to receive the event;
send notification of event to the locked context element:
  if no previous entry for this node
  then put reference to this node (plus any old value) in context
  else replace any previous old value with the old value to be
      retained (since it will be older than any value that is
      currently being saved);
unlock the context element;
send warning messages to precondition process if it had tagged
  the corresponding field, node, region, or sset;
activate the precondition process (if its pre-precondition is
  satisfied);
end;
receive.message:
begin
if reading a message
then repeat
  if a message with the desired tag-ID, node-ID, and field/sset
description exists (where each of these parameters
may be bound upon searching for a message or they
may be prespecified upon the call)
  then exit repeat
else
  if do not wish to wait for such a message
  then return no message
  else wait for a message to be received;
if deleting a message
then delete the message (after having read it);
return the message <tag-ID, node-ID, field/sset description>;
end;

Now consider the actions of the data node creation and deletion routines, as
shown in the following pseudo-Algol code:

create.node:
begin
  if node is to be of type LINK
  then begin
    if not inside LOCK!
    then temporarily LOCK! nodes to be linked together
    else if nodes to be linked together not LOCK!ed
      then return error;
    allocate space for new data node;
    assign resource sequence number to node (for locking ordering);
    initialize fields of node according to user-supplied values;
    initialize region descriptor for node;
    if inside LOCK!
    then add new node to set of locked resources;
    insert node into data base;
    send notification of node creation to dynamic local ssets
    (subject to local context filtering);
    if node is of type LINK
    then begin
      modify (via write.field.value) linking fields of nodes to be
      linked together to include new LINK node;
    end;
    relinquish any temporary LOCK!
  end;
end;
delete.node:

begin
if not inside LOCK!
then temporarily LOCK! node and any HYPOTHESIS nodes directly
attached to it (as well as intervening LINK nodes if node
to be deleted is of type HYPOTHESIS)
else if any member of the set of nodes described in above
then-clause is not LOCK!ed
then return error;
if node is of type LINK
then
  delete.link: begin
    remove LINK node (via write.field.value) from linking
    fields of nodes being linked together;
    remove LINK node from active data base;
    place LINK node name in set of nodes to be
    reclaimed when no further references remain
    to these nodes;
  end
else begin
  forall LINK leading from the HYPOTHESIS node do
    delete.link;
    remove HYPOTHESIS node from active data base;
    place HYPOTHESIS node name in set of nodes to be reclaimed
    when no further references remain to these nodes;
  end;
  relinquish any temporary LOCK!
end;

Knowledge Source and Precondition Processes

Preceding sections have discussed the data access synchronization
mechanisms by which precondition and knowledge-source processes may execute
without interfering with the assumptions of other processes regarding the contents of
the shared global data state (although these mechanisms could potentially interfere with
the execution progress of processes by interrupting these processes until it was safe to
proceed, this interruption only ever affecting the process actually using the access
synchronization mechanism). Other sections discussed means by which processes could
cooperate with one another and communicate with one another (albeit indirectly)
through the use of various data monitoring mechanisms, such as data tagging and data
event accumulation by local contexts. The present sections will attempt to show how
these mechanisms are established and used by the precondition and knowledge-source
processing elements. Precondition and knowledge source specification will be discussed in terms of local context requirements and assumptions regarding local and global data integrity. Issues of knowledge-source process instantiation and execution will be discussed in terms of the information transfer between precondition and knowledge source and in terms of communication among processes in general. Issues of global problem-solving control will also be discussed, by describing the multiprocessing-oriented goal-directed scheduler of HSII.

Knowledge Source and Precondition Specification

Knowledge-source and precondition processes consist primarily of a code body executing in a specified local environment. For preconditions, this local environment consists of a set of pre-specified local context descriptions which accumulate data events relevant to the precondition. For knowledge-source processes, the local environment also consists of a set of pre-specified local context descriptions, but the initial contents of these contexts are supplied by the precondition process which instantiates the knowledge-source process. A knowledge-source process is also instantiated with a set of input parameters, also supplied by the precondition, which serve to direct the efforts of the knowledge source. Of course, both preconditions and knowledge-source processes may place tags in the global data base; and any messages received as a result of such tagging operations become a part of the local environment of the tagging process.

As an example of how the data environment of a knowledge-source process might be specified, consider the following code segment, using an hypothetical speech system example:

```plaintext
knowledge.source word.speller (hypothesis word; integer word.class);
local context:
  new hyps at phonetic level,
  new links between phonetic level and word level,
  hyps at word level with rating changes,
  hyps at phonetic level with begin-time changes;
  < procedurally encoded knowledge: the code body >;
endknowledge.source;
```

Notice that the first three local context specifications denote supersets, while the begin-time set specification is a change set (with old begin-time values being saved in the context along with the node-ID's of any changed nodes).
The code body for a precondition or a knowledge source is a programmed algorithm representing the knowledge embodied within the processing element. The specification of that algorithm and the effect of executing it is, of course, task dependent; but there are various general comments that can be made regarding the structure of such a code body. Since the preconditions and knowledge sources of HSII are intended to be run in an asynchronous parallel multiprocessing environment, the structure and specification of the knowledge-based algorithms must take that into account. A prime example of the effects of running in a parallel processing environment is the necessity for incorporating revalidation phases (as discussed previously in relation to data tagging) into the processing algorithm. This revalidation effort, wherein a process must re-check the continued integrity of the various data assumptions it has made before it can perform any data base modifications, is a direct result of executing in an asynchronous parallel processing environment (whether the parallelism be real or simulated).

A primary concern in designing an algorithm which is to run in such a multiprocessing environment involves being aware of the grain of process execution: at what points can the execution of a given process be interrupted? In a single processor system, for example, perhaps it can be reasonably assumed that each precondition or knowledge-source process will run to completion without the threat of being interrupted. In such a situation, there would be no need for revalidation checks, since it would be impossible for any process to interfere with (and hence perhaps violate) the data base assumptions being accumulated throughout the execution of the running process. However, this is certainly not the case in a multiprocessor situation, even if each process is run to completion on its assigned processor: revalidation checks are necessary as long as competing processes are capable of concurrently accessing and modifying the data base elements upon which a given process is basing its assumptions.

These data assumption revalidation checks may be taken as representing the global assumptions which a precondition or knowledge-source process makes about how concurrently executing processes can interfere with the successful execution of that precondition or knowledge-source process. Similarly, such processes can also make local assumptions regarding, for example, how long it might be expected to execute before possibly being interrupted. An example of such a local assumption might be the assumption that a primitive data access might be regarded as an indivisible operation from the point of view of a precondition or knowledge-source process; or that the
execution of a conditional statement based on field values of globally locked data nodes may be considered as an indivisible operation from the point of view of the precondition or knowledge-source process. These local and global assumptions are important in that they specify how much validation and revalidation effort must be pre-programmed into the knowledge source code body in addition to the knowledge algorithms themselves.

Knowledge Source and Precondition Activation and Execution

As was described previously, precondition processes are permanently instantiated, their local contexts continually receiving data event notifications. The distribution of these data event notifications to the appropriate local contexts is the responsibility of the primitive data-base-write operation. This primitive operation is also responsible for the activation of precondition processes whenever a data event occurs which satisfies a pre-precondition. ¹

The instantiation of a knowledge-source process is dependent upon the outcome of the tests performed by the precondition. These tests might include direct or associative retrievals from the blackboard data base or from the static local context of the precondition itself. During this testing, the precondition can also tag various fields, nodes, or regions in the blackboard data base for subsequent use by a knowledge-source process which the precondition might instantiate. Having decided to create an instantiation of a particular knowledge source, the precondition calls upon a HSII system primitive, called INVOKE.KNOWLEDGE.SOURCE, to perform the actual process instantiation, passing along the various input parameters which are to specify to the new knowledge-source process the reasons for its creation. During the instantiating sequence, the precondition may also transfer data base tags (and any associated resultant messages) to the knowledge-source process. The local contexts of the knowledge-source process are created according to the context declarations of the generic knowledge source after which this instantiation is being modeled; however, by definition these context declarations are a subset of those of the invoking precondition process. The data events thus far accumulated by the precondition process are copied into the new local contexts of the knowledge-source process (from both the static and dynamic contexts of

¹ Recall that pre-preconditions are assumed to be computationally inexpensive tests to determine whether it is worthwhile to perform the entire precondition evaluation, which could be arbitrarily complex. The simplest form of pre-precondition might be to activate the precondition evaluation whenever any data event is recorded in the precondition's local context.
the precondition process), thereby providing an initial operating context for the knowledge-source process; the effective creation time of the knowledge-source process is thus the same as the activation time of the precondition process.

The details of knowledge-source process instantiation are given in the following pseudo-Algol code:

```
invoke.knowledge.source:
  begin
    create a new process-ID for the knowledge-source process;
    if invoking precondition is transferring tags
      then begin
        create a tag-ID for the knowledge-source process;
        copy the tags to be transferred, using the new tag-ID;
        copy any messages pending as a result of these tags;
      end;
    forevery context declared for the generic knowledge source do
      create a dynamic context for the knowledge-source process
      with initial contents specified by the combined data events
      from the precondition's corresponding static and dynamic contexts;
      create a new process based on the generic knowledge source code
      body, using the new process-ID and substituting the actual
      parameters supplied by the invoking precondition for the
      formal parameters of the generic knowledge source;
      notify the scheduler of the existence of the new process;
    end;
  end;
```

Having invoked the new knowledge-source process, the precondition process may then become dormant, waiting for subsequent data events to reactivate it. When the knowledge-source process is eventually scheduled for execution, it will use the mechanisms of data tagging and locking to accomplish any necessary data base monitoring or data access synchronization actions. Interprocess communication is accomplished indirectly, for reasons previously discussed, by these monitoring mechanisms and by the anonymous interchange of results via the blackboard data base. The results of data monitoring operations are detected by polling for the receipt of messages, rather than by interprocess interrupts, so as to improve the internal control characteristics of processes.\footnote{The use of polling rather than interrupts relates to the previous discussion on the local assumptions which a process makes regarding its continuous execution. If a process is interrupted unexpectedly, it local data state may be in a tenuous condition.}

When a knowledge-source process completes its
execution, the process is terminated and its accumulated local data environment is eliminated, since knowledge-source processes are transient processes.

Knowledge-Source and Precondition Process Control

The preceding sections have described how precondition processes and knowledge-source processes are specified and then activated in a data-directed manner. The current section will describe how the problem-solving organization as a whole manages to map the virtual parallelism of the set of concurrently executing processes onto the available hardware resources so as to allocate these resources in a goal-directed manner. This mapping is the responsibility of the HSII goal-directed scheduler, which is a distributed multiprocess-oriented scheduler which takes as inputs the collection of executable processes plus information relating to the relative importance of executing each of these processes (given by the current problem state as defined by the blackboard data base) and produces, as output, assignments of processors to processes which can best advance the problem state toward its goal of problem solution. Notice that this scheduling process is dynamic in that relative priorities among the set of processes competing for computing resources may vary with time. Furthermore, there may conceivably be many more processes capable of being run than can possibly be granted processing power, given the constraints of attaining a solution (or a failure indication) within a reasonable period of time; in such a case, those processes must be run first which offer the greatest promise for determining a solution within the allotted time, with the hope that it may not be necessary to run all of the remaining processes in order to determine that solution.

The role of the goal-directed scheduler can perhaps best be described by considering its function in terms of expressing the HSII organization as an example of heuristic search. At a superficial level, HSII may be considered as a model-manipulating system (Simon, 1971) in which the blackboard data base as a whole...
represents the dynamic problem solution state and the various knowledge sources are operators capable of transforming this solution state so as to move the solution state through a conceptual search graph, seeking to attain a solution state which matches a goal situation. In such a simplified model of HSII, it is the function of the goal-directed scheduler to decide which operations to apply when, in order to attain a goal state using the available computational resources to the best advantage in the allotted amount of search time. Matters are complicated somewhat in that the dynamic blackboard data base represents the accumulated information of all past processing (i.e., information is not discarded as in more classical backtracking search strategies); and the blackboard also represents the combined solution states resulting from the concurrent search of multiple paths through the conceptual search graph, thereby allowing the free exchange of the (contingent) information gained by following these various search paths in parallel. In this latter sense, HSII might also be characterized as an information-gathering system (Simon, 1971), where, although the information being accumulated in the blackboard is contingent (being based on previously hypothesized information), this accumulated information is used to direct the subsequent search through the problem space. Note that the usual conception of a search strategy is serial in nature: which of the available generated partial solution states should be chosen for further development, and given that a certain solution state in the graph representing the problem search space has been chosen, what operation should be done next to move to an adjoining solution state in the search space? Problem-solving strategies can then be characterized according to how these decisions of next-node and next-operation are made. The information that a problem-solver uses to decide what to do next is of two types: a) information that is available when the problem is first posed (including information particular to this problem instance, as well as more general information relating to the task domain of the problem solver), and b) information that becomes available as the search for a solution proceeds. In the case of HSII, information of the first type includes the task-dependent knowledge algorithmically encapsulated within the various knowledge sources, plus the input data specific to a particular problem to be solved; information of the second type includes the contingent partial problem solutions accumulated in the blackboard data base. While the a priori information encoded in the knowledge sources (and their preconditions) can be used to determine in a data-directed way which areas of the blackboard are possible choices for solution expansion, the heuristic power of the problem-solving system is greatly increased by basing the selection of the locus of control (i.e., the point from which to expand the
conceptual problem-solving search graph) and the operation to be performed (i.e., the knowledge source which is to be applied) on the information that becomes available as the search progresses. In particular, the choice of locus of control for further exploration and the choice of the operation to be performed given that locus may be best made by applying evaluation functions to the candidates for each choice. In HSII, it is the responsibility of the goal-directed scheduler to evaluate these functions and make the decisions appropriate in guiding the search for a problem solution; of course, given the multiprocessing orientation of HSII, it may be possible to make several such choices at any given scheduling point, depending upon the availability of processing resources at that time.

Evaluation functions for selecting loci of control, which in HSII usually refers to selecting nodes or regions of the blackboard database which require further processing, may depend on the properties of the situation at the potential locus. As an example from the speech system, if a particular data node representing a phonemic unit has a very high confidence rating based on its supporting acoustic segments, it might be important to try to link this phone to some word or syllable in an attempt to eliminate having to process extraneous phonetic word spellings (which might occur if only a top-down approach were used and many word units were hypothesized from the syntactic information level, without any regard to existing phonemic units). Alternatively, evaluation functions for selecting loci of control may depend on a relation between the situation at the potential locus and a description of the desired goal state. Using the previous hypothetical example of linking phoneme units to word units, such a locus of control might be chosen for further evaluation not especially because it is important to obtain the immediate results of the linking operation, but rather because this linking would complete a path through the data base along which the confidence ratings of the lower data levels could be propagated (via subsequent processing elements) to the higher data levels, these higher levels perhaps having no other means of obtaining these confidence ratings which are necessary for problem solution.

Evaluation functions for selecting the operations to be performed at a chosen locus of control may also depend on the absolute properties of the chosen locus and the

---

1 Note that in a problem-solving system like HSII, the optimality of the path from start to goal is not nearly so important as the optimality of the search process: the goal is to find a solution with a minimal amount of search, rather than to fine an optimal solution.
particular operator. To continue the phone-to-word linking example, having chosen as a
locus of control an unused phone node, an evaluation function which selects the
operation to be performed upon this node would presumably choose a knowledge
source which was capable of creating a phone-to-word link over a knowledge source
which promised only to make a minor refinement in the confidence rating of that phone
node. Alternatively, just as for locus of control evaluation, evaluation functions for
selecting an operation given a locus might depend on possible relationships between the
chosen locus and the goal state in choosing an operation capable of advancing the
search progress toward the goal state. This latter type of operation evaluation function
is useful in performing a form of means-ends analysis in which a precondition or
knowledge-source process might be chosen to execute in order to prompt further
actions to occur, rather than simply executing that initial action for any direct results it
may obtain.

Summarizing, heuristic search schemes may move a problem-solution state
through a state space based upon evaluation functions which can select loci of control
for information expansion and which can select operations to be performed at these loci;
each form of evaluation function may depend on features of the situation at the
potential locus of control, or the evaluation function may depend on differences between
the properties of the potential locus and the desired goal state.

In HSII, the data-directed method of precondition activation and knowledge-
source process instantiation is primarily responsible for defining the set of potential
operations that may be performed at any given time during the problem solution. By
appropriate precondition definitions, these same mechanisms can provide the set of loci
of control, supplying one or more processes (preconditions or knowledge sources)
capable of executing at each such locus. It is then the task of the HSII scheduler to
take this as input and allocate processing power so as to utilize the multiprocessing
capabilities of the hardware and software organizations to effect an efficient goal-
directed problem-solving search strategy based on the hypothesize-and-test paradigm.

The decisions made by the various evaluation functions of the HSII scheduler
are based on attention focusing parameters associated with the various components
of the blackboard data base. Nodes (hypotheses and links) and regions of the data base
have associated with them various attributes related to how much knowledge-source
attention has been expended on these data elements, as well as suggestions for how
much more processing should occur and of what type (e.g., analysis or synthesis based on some node, or further development of some region). These attention focusing parameters are computed by policy-type knowledge-source processes which are instantiated due to such data events as node validity changes and the creation of new nodes; these policy knowledge-sources are responsible for propagating the effects of such events throughout the data base so that proper attention is paid to these events and processing power is allocated accordingly. Since the structural format of all hypothesis nodes and all link nodes is uniform throughout the data base, these policy knowledge-sources can easily compute and propagate level-independent information such as attention focusing parameters, processing state attributes, and hypothesis validity measures across levels in an equally uniform manner. The goal-directed scheduler may then use the various attention focusing parameters, processing state attributes, and validity measures distributed throughout the data base in order to schedule knowledge-source processes which have been previously instantiated; or the scheduler may use these same focusing factors to detect important areas in the data base which require further work, invoking precondition processes as soon as possible that they might instantiate new knowledge-source processes to work in these important areas. Thus, the attention focusing factors with the global data base serve to schedule both knowledge-source processes and preconditions.

Notice that the implementation of the goal-directed scheduling strategy is separated from the actions of the individual knowledge sources. That is, the decision of whether a knowledge source can execute in a particular data base context is local to the knowledge source (being defined by the precondition of the knowledge source), while the assignment of that knowledge source to one of the (perhaps) many contexts in which it can possibly operate is made more globally (by the goal-directed scheduler). This decoupling of focusing strategy from knowledge-source activity, together with the decoupling of the data environment (blackboard data base) from the control flow (knowledge source activation) and the localized context in which a knowledge-source process operates, permits quick and flexible refocusing of attention of knowledge sources. The ability to refocus quickly is very important in a knowledge-based problem-solving system, because the potentially errorful nature of the knowledge source activity may lead to many incomplete and possibly contradicting hypothesis networks; thus, as soon as possible after a network no longer seems promising, the resources of the system should be redirected and employed elsewhere. It is the
responsibility of the HSII goal-directed scheduler to perform these attention focusing duties in a manner which takes maximal advantage of the multiprocessing capabilities of the system in effecting an efficient search strategy through the problem space, based on a knowledge-based implementation of the hypothesize-and-test paradigm. Further details regarding the HSII goal-directed scheduler may be found in (Hayes-Roth, et al., 1975).
EXPERIMENTS WITH AN IMPLEMENTATION

The preceding sections of this report have presented various of the mechanisms necessary in implementing a knowledge-based problem-solving system such as HSII in a multiprocessing environment. The HSII speech-understanding system implemented at Carnegie-Mellon University is a test of the design principles of this problem-solving organization. The present sections will discuss the various experiments that have been performed in an attempt to characterize the multiprocessing performance of the HSII organization in the speech-understanding task. While the speech task is the first test of the multiprocessing problem-solving organization of HSII, it is believed that the system organization provided by HSII is capable of expressing other knowledge-based AI problem-solving strategies, as might be found in vision, robotics, chess, natural language understanding, and protocol analysis. In fact, proposals are under way which will further test the applicability of HSII by implementing a system for the analysis of natural scenes using the HSII problem-solving organization.

HSII Multiprocess Performance Analysis

Given a problem-solving model which seems to lend itself to multiprocessing in a very natural manner (viz., the hypothesize-and-test paradigm using independent, cooperating knowledge sources) and a hardware architecture which is intended for use as a closely-coupled multiprocessor (viz., C.mmp), several questions of efficiency arise, since the true usefulness of an implementation of a problem-solving organization depends almost as much on the speed with which a particular problem is solved as on the accuracy of the solution. Therefore, various efficiency issues involving multiprocess
problem-solving organizations were investigated, using a simulation model which is incorporated within the uniprocessor version of the HSII speech-understanding system. The HSII problem-solving organization was not itself modeled and simulated, but rather the actual HSII implementation (which is a multiprocessing organization even when executing on a uniprocessor) was modified to permit the simulation of a hardware multiprocessor environment. Certainly, any results presented will reflect the detailed efficiencies and inefficiencies of the particular system implementation being measured, but hopefully the organization of HSII is sufficiently general that the various statements will have a wider quantitative applicability for those considering similar multiprocessor-oriented problem-solving schemes.

By way of summary, the primary characteristics of the HSII problem-solving organization include: a) multiple, diverse, independent and asynchronously executing knowledge sources, b) cooperating (in terms of control) via a generalized form of the hypothesize-and-test paradigm involving the data-directed invocation of knowledge-source processes, and c) communicating (in terms of data) via a shared blackboard-like data base in which the current data state is held in a homogeneous, multidimensional, directed-graph data structure. HSII is intended to provide a software architecture capable of supporting such systems of cooperating (but independent and asynchronous) data-directed knowledge-source processes, allowing the system complete freedom in scheduling the execution of the various precondition and knowledge-source processes so as to achieve effective parallel search over a general AI problem-solving graph, using the hypothesize-and-test paradigm to generate the search graph. By permitting this scheduling flexibility, various other system structures may be simulated. In particular, one can revert to a totally synchronous system in which only one knowledge-source process is permitted to be active at any given time.

In choosing a problem-solving organization which involves multiprocessing and a shared data base, several areas of consideration quickly come to mind, which areas would not be of immediate concern in a synchronous uniprocessor situation. First, one has to be concerned with scheduling the various processes for execution on the available processors, along with the attending problems of maintaining the local states (contexts) of each of the processes. However, the problem is somewhat more than just scheduling ready processes on available processors: from a problem-solving standpoint, certain of the ready processes may no longer need or deserve to run, and certain of the running processes may need to be suspended or even prematurely terminated, depending on the
dynamic execution state of the problem solution. The scheduler, therefore, must not only try to maximize processor utilization, but it must be goal-oriented in order to maximize the utilization of the overall processing time (so as to minimize this time).

A second area of consideration in a multiprocessing problem-solving organization concerns the amount of interprocess execution interference generated due to accessing a shared data base (which interference arises from the data access synchronization mechanisms). Essentially, by observing and measuring the interference patterns which occur during execution, given various machine and knowledge-source configurations, and measuring the various overheads incurred due to scheduling and shared data base maintenance, a measure of effective parallelism might be determined for various scheduling strategies and various shared data base-handling strategies. Of course, an ideal situation might be to be able to specify a problem-solving system as a set of completely independent knowledge-source processes (or at least ones which do not need to communicate directly, except perhaps to signal problem solution) which can be scheduled so as to never interfere with one another's accesses to the shared data base.\footnote{One way a goal-oriented scheduler might help in reducing (or eliminating) global data base access interference is to schedule to run concurrently only processes whose global data demands are disjoint, such data requirement information already being available since this is a "goal-oriented" scheduler. Such a scheduling policy could even be used to supplant an explicit locking scheme, since the global data base locking would be effectively handled by the scheduler (albeit probably on a fairly gross level). Of course, other factors may rule out such an approach to data access synchronization, such as an inability to make maximal use of the available processing resources if only data-disjoint processes are permitted to run concurrently, or the inability to know in advance the precise blackboard demands of each knowledge-source instantiation. Nonetheless, the information relating to the locality of knowledge-source data references is useful in scheduling processes so as to avoid excessive data access interference (thereby improving the effective parallelism of the system).} The objective of the simulation experiments in this area was to measure the effects of various forms of process interference within the problem-solving organization of HSII to determine the multiprocess performance characteristics of the system. In particular, these simulation studies were not directly concerned with the problem-solving ability \emph{per se} of the HSII speech-understanding system, but rather they represented an attempt to measure the advantages and disadvantages of the more general problem-solving organization underlying this particular implementation of the HSII philosophy.
The HSII Speech Understanding System: The Simulation Configuration

The configuration of the HSII speech-understanding system, upon which the following simulation results were based, consists of eight separate generic knowledge sources (each of which may be realized by several active instantiations at any given moment during the problem solution), each of which represents some body of knowledge relevant to the speech-understanding task. Due to the excessive cost of the simulation effort (and due to the limited stages of development of some available knowledge sources), only a subset of the available knowledge sources were actually used in the simulation experiments (Appendix A describes the more complete knowledge source set). The knowledge sources used in the simulation were: the Segment Classifier (called KS!SEG!SEG), the Phone Synthesizer (consisting of two knowledge sources, called KS!PSYN!PSYN and KS!SEG!PSYN), the Phoneme Hypothesizer (called KS!ALO!ALO), the Phone-Phoneme Synchronizer (consisting of three knowledge sources, called KS!UTTBOUNDARIES!PSC, KS!SEARCH!PSC, and KS!TIME!PSC), and the Rating Policy Module (called KS!UV!RPOL). KS!TIME!PSC also functions as a Time Propagation Module to distribute utterance time markers (i.e., time boundaries measured from the beginning of the input data utterance) throughout the blackboard. These knowledge sources are activated by half a dozen precondition processes (which are permanently instantiated in the system), which are continuously monitoring the blackboard data base for events and data patterns relevant to their associated knowledge sources. The preconditions used were: PRE!SEG!SEG (which instantiates the Segment Classifier), PRE!PSYN!PSYN (which instantiates the Phone Synthesizer), PRE!ALO!ALO (which instantiates the Phoneme Hypothesizer), PRE!UTTBOUNDARIES!PSC and PRE!PSYN!PSC (which together instantiate the various parts of the Phone-Phoneme Synchronizer), and PRE!RPOL!RPOL (which instantiates the Rating Policy Module). Both knowledge sources and preconditions may freely access the centralized blackboard data base, which consists of nine lexicon levels.1 The particular levels used were chosen so as to facilitate the information exchange between the various component knowledge sources. The lexicons include SEG and MXN (which contain input data to be used by KS!SEG!SEG), PSEG (which contains packed acoustic segments created by KS!SEG!SEG and read by KS!PSYN!PSYN and

---

1 While there are eight conceptual information levels within the HSII speech-understanding system (see Appendix A), the blackboard is abstractly segmented according to lexicons, rather than information levels, since lexicons allow a finer abstract decomposition of the blackboard.
KSICSEGIPSYN, with KSICSEGIPSYN creating further packed segments), PHON (which contains phonetic units created by KSIPSYNIPSYN from the units in the PSEG level), WORD (which contains lexical units created by KSISEGISEG, which supplies the correct word units for performance analysis purposes, as it also supplies entries for SHDWORD and SHDSSENT, which are "shadow" lexicons for word and sentence units), WRDSURN and SURN (which contain word units representing particular word pronunciations, and the phonemic-like units detailing those pronunciations, respectively, both types of units being created by KSIALOIALO from units in the WORD lexicon level). The three Phone-Phoneme Synchronizer knowledge sources are responsible for creating links between the SURN and PHON levels. Appendix A (which was extracted from (Lesser, et al., 1974)) contains a more detailed description of the blackboard and the various knowledge sources for the more complete HSII speech-understanding system.

This set of knowledge sources and preconditions and the associated operating system facilities provided by the HSII organization were first implemented to execute on a uniprocessor DECsystem-10 computer. The particular implementation represented here was programmed in the Algol-like language, SAIL (Swinehart and Sproull, 1971), using SAIL's multiprocessing facilities (Feldman, et al., 1972) and making extensive use of its LEAP associative data storage facility (Feldman and Rovner, 1969). Thus, while the hardware environment of this version of the HSII speech-understanding system is that of a single processor, the software environment is the multiprocessor structure described throughout this report. The simulation experiments were then run using this HSII configuration, simulating the hardware environment of a closely-coupled multiprocessor hardware organization. The size of the HSII configuration used in the simulations was about 180K, 36-bit words; 70K of this total was the HSII operating system plus the SAIL runtime routines, about 73K was precondition and knowledge source code plus variables, and the remainder (which varied from 20K to 45K depending on the number of processors being simulated and the number of processes being instantiated) represented the blackboard data base plus process activation records and other SAIL working space. The simulations were carried out to determine the efficiencies of the various HSII multiprocessing mechanisms discussed previously, as well as to gain some insight into any problems that might arise in the ensuing implementation of a HSII speech-understanding system for the Carnegie-Mellon C.mmp multiprocessor.1

1 The implementation of the C.mmp version of the HSII speech-understanding system
The following sections will discuss the motivations, implementations, and results of the various experiments which have been performed using the multiprocessor-simulation version of the HSII speech-understanding system.

Simulation Mechanisms and Simulation Experiments

The various multiprocessor simulation results were obtained by modifying the flow of control through the usual HSII multiprocessing organization to allow simulation scheduling points every time a running process could interact in any way with some other concurrently executing process. Such points included blackboard data base accesses and data base access synchronization points (including attempts to acquire data base resources and any resulting points of process suspension due to the unavailability of the requested resource, as well as the subsequent points of process wake-up for retrying the access request). Simulation scheduling points were also inserted whenever a data modification warning message (triggered by modifying a tagged data field) was to be sent, as well as whenever a process attempted to receive such a message. Since each of these points of possible process interaction is centralized within a primitive system operation, the insertion of these simulation scheduling points was relatively simple. The scheduling mechanism itself was also modified to allow for the simulated scheduling of multiple processing units, while maintaining the state information associated with each processor being simulated (such as the processor clock time of that simulated processor and the state of the particular process being run on that processor). The simulation runs were performed so as to keep the processor clock-times of each processor being simulated in step with one another (the simulation being event-driven, rather than sampled), thereby allowing for the accurate measurement and comparison of concurrent events across processors. By selecting the number of processors to be simulated and choosing the usual scheduling parameters and precondition and knowledge-source parameters, a chronological trace of the activity of each process and processor could be obtained. By accumulating statistics during the trace period and by performing various post-processing operations upon this activity trace record, the simulation results presented in the following sections were thus far has been, in fact, essentially a direct mapping of the DECsystem-10 implementation, with additional design being done as necessary to solve the particular problems of running in the Cmpp environment (such as having to resolve the small address space problem, wherein any given process may have at any one moment only a 32K-word window into the centrally located main memory).
obtained.

While instrumenting the HSII implementation to simulate a multiprocessor environment was relatively easy, the resultant grain of simulation was at a very fine level of detail. Although detailed process interaction patterns could be traced, the cost of simulation was extremely expensive (especially given the large size of the system); being on the order of one to two hours of DECsystem-10 (KA-10) processing time per run (or about 50 times the cost of running without the simulation overhead), where a run consisted of setting the various simulation configuration parameters and then allowing the system to execute for a standardized processing period defined by the process activity; as a result, the number of simulation runs was necessarily small. While further future simulations are planned, most of the results presented here were achieved by using a single set of knowledge sources (as described above), with a single speech data input utterance, keeping the data base locking structure and scheduling algorithms essentially fixed, while varying the number of simulated (identical) processors. Several runs were also performed to test the effects of altering the knowledge-source set, altering the locking structure, and altering the mode of data input (the normal input mode being a utterance-time-ordered introduction of input data which simulates real-time speech input).

Primitive Operation Timings

Time measurements of various primitive operations were first made using the 10-microsecond hardware interval timer attached to Carnegie-Mellon's DECsystem-10. Some of the timed primitive operations (such as those involving simple data base access and modification) were not especially subject to the fact that the problem-solving organization involved multiple parallel processes, whereas others (such as those involving process instantiation and process synchronization) were directly related to the multiprocess aspects of the organization (and might even be taken in part as overhead when compared to alternative single-process system organizations). The times for the various system operations, as shown in Tables 4a, 4b, and 4c, should be read as relative values, comparing the multiprocess-oriented operations with the data accessing operations to get a relative feel for the overheads involved in supporting and maintaining the multiprocess organization of HSII. Keep in mind that such time measurements are highly dependent on the particular implementation and can change
fairly radically when implemented differently. In fact, a primary use of such timings is in determining operating system bottlenecks so that such code sections can be rewritten in a more optimal way. As a result, some primitive operations reflect execution times which are a result of extensive optimization attempts, while other operations have not yet been subjected to this optimization.

<table>
<thead>
<tr>
<th>% total runtime</th>
<th>mean time (ms)</th>
<th>number of calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>w/o lock</td>
<td>w/lock</td>
<td>w/o lock</td>
</tr>
<tr>
<td>create.hyp</td>
<td>2.38</td>
<td>1.52</td>
</tr>
<tr>
<td>create.link</td>
<td>4.58</td>
<td>2.63</td>
</tr>
<tr>
<td>read.hyp.integer</td>
<td>3.48</td>
<td>9.64</td>
</tr>
<tr>
<td>read.hyp.list</td>
<td>0.87</td>
<td>2.94</td>
</tr>
<tr>
<td>write.hyp.integer</td>
<td>9.97</td>
<td>5.58</td>
</tr>
<tr>
<td>modify.hyp.integer</td>
<td>8.19</td>
<td>4.68</td>
</tr>
<tr>
<td>write.hyp.list</td>
<td>0.88</td>
<td>0.54</td>
</tr>
<tr>
<td>read.link.integer</td>
<td>0.25</td>
<td>0.57</td>
</tr>
<tr>
<td>read.link.list</td>
<td>0.46</td>
<td>2.53</td>
</tr>
<tr>
<td>write.link.integer</td>
<td>1.99</td>
<td>1.42</td>
</tr>
<tr>
<td>modify.link.integer</td>
<td>5.94</td>
<td>3.07</td>
</tr>
<tr>
<td>delete.link</td>
<td>0.02</td>
<td>0.01</td>
</tr>
</tbody>
</table>

**Blackboard Accessing:**

<table>
<thead>
<tr>
<th>% total runtime</th>
<th>mean time (ms)</th>
<th>number of calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>w/o lock</td>
<td>w/lock</td>
<td>w/o lock</td>
</tr>
<tr>
<td>create.hyp</td>
<td>2.38</td>
<td>1.52</td>
</tr>
<tr>
<td>create.link</td>
<td>4.58</td>
<td>2.63</td>
</tr>
<tr>
<td>read.hyp.integer</td>
<td>3.48</td>
<td>9.64</td>
</tr>
<tr>
<td>read.hyp.list</td>
<td>0.87</td>
<td>2.94</td>
</tr>
<tr>
<td>write.hyp.integer</td>
<td>9.97</td>
<td>5.58</td>
</tr>
<tr>
<td>modify.hyp.integer</td>
<td>8.19</td>
<td>4.68</td>
</tr>
<tr>
<td>write.hyp.list</td>
<td>0.88</td>
<td>0.54</td>
</tr>
<tr>
<td>read.link.integer</td>
<td>0.25</td>
<td>0.57</td>
</tr>
<tr>
<td>read.link.list</td>
<td>0.46</td>
<td>2.53</td>
</tr>
<tr>
<td>write.link.integer</td>
<td>1.99</td>
<td>1.42</td>
</tr>
<tr>
<td>modify.link.integer</td>
<td>5.94</td>
<td>3.07</td>
</tr>
<tr>
<td>delete.link</td>
<td>0.02</td>
<td>0.01</td>
</tr>
</tbody>
</table>

**Blackboard Associative Retrieval:**

<table>
<thead>
<tr>
<th>% total runtime</th>
<th>mean time (ms)</th>
<th>number of calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>w/o lock</td>
<td>w/lock</td>
<td>w/o lock</td>
</tr>
<tr>
<td>and.retrieve</td>
<td>2.72</td>
<td>4.98</td>
</tr>
<tr>
<td>match.prototype</td>
<td>1.65</td>
<td>3.68</td>
</tr>
<tr>
<td>compare.hyp</td>
<td>0.57</td>
<td>1.07</td>
</tr>
<tr>
<td>get.time.adjacent</td>
<td>9.31</td>
<td>15.33</td>
</tr>
<tr>
<td>get.struct.adjacent</td>
<td>3.99</td>
<td>6.31</td>
</tr>
<tr>
<td>get.nodes.in.rgn</td>
<td>2.05</td>
<td>0.87</td>
</tr>
<tr>
<td>get.overlap.hyps</td>
<td>3.11</td>
<td>6.35</td>
</tr>
</tbody>
</table>

Table 4a. Primitive Operation Times
Table 4a gives timing statistics relating to the costs involved in maintaining
the shared, centralized blackboard data base. Two sets of statistics are given, one set
showing the operation times without the influence of data access synchronization
(blackboard locking) and one set with the locking structures in effect. Of course, any
actual multiprocessor implementation must include the access synchronization
mechanism, but these two sets of times give a quantitative feeling for the cost of such
mechanisms in this particular implementation of HSII. The figures given include the
average runtime cost per operation, the number of calls (in this particular timing run) to
each operation (thereby showing the relative frequencies of operation usage), and the
percentage of the overall runtime consumed by each operation. With respect to the
individual entries, create.hyp and create.link are composite operations (involving many
field-writes and various local context updates) for creating blackboard nodes.
Hypotheses and links are made up of integer and list-type data fields, and the various
read and write operations indicated (such as read.hyp.integer and write.hyp.integer) are
used in accessing these field types. Note that included in any given field read or write
operation is the cost of perhaps tagging (or untagging) that particular field (or its node).
The various functions of the blackboard monitoring mechanism are contained within the
field write operations. Thus, also included in the field write operation is the cost of
distributing the data event resulting from the write operation to all relevant
precondition and knowledge-source process local contexts, as well as the cost of
sending tag messages to all processes which may have tagged the field being modified;
these additional costs are also accounted for independently in the send.msgs.and.integers
and the send.msgs.and.lists table entries in Table 4b. Field write operations are also
responsible for evaluating any pre-preconditions associated with the field being
modified and activating any precondition whose pre-precondition is satisfied. A further
cost included in field write operations is that of notifying local context supersets of the
data event; this cost is also separately accounted for in the notify.ssset entry in Table
4b. Included in the cost of reading a data field (e.g., read.hyp.integer) is the cost of
verifying the access right of the calling process to the node being read (which could
involve a temporary-locking operation, the cost of which is also given independently in
the lock.node entry in Table 4c); this access-right checking cost is also separately
accounted for by the read.access.chk operation of Table 4c. To alleviate repetitive
access checking for write operations, which could involve modifying several fields of the
same node, the access checking cost is not included in the field writing operation (such
as write.hyp.integer). Instead, modify.hyp.integer and modify.link.integer are used,
which include the access checking costs for writing data fields (also separately accounted for in the write.access.chk entry of Table 4c), as well as the field writing costs and any associated bounds checking on the new field value and any necessary local context superset notifications (also accounted for separately by the notify.sset item in Table 4b), in addition to the notifications normally performed within the field writing operations themselves.

Additional blackboard operation costs are described in the Associative Retrieval section of Table 4a. Associative retrieval is based on specifying partial node descriptions (called matching-prototypes) which serve as a means of retrieving the set of blackboard nodes fitting that partial description. And.retrieve, match.prototype, and compare.hyp represent the various retrieval operations possible using these matching prototypes. Retrieval from the blackboard may also be done by requesting the nodes which are time-adjacent (according to the utterance-time dimension of the speech-understanding blackboard) or structurally adjacent (according to the blackboard graph structure) to a given node (or set of nodes); get.time.adjacent and get.struct.adjacent perform these operations. Furthermore, retrieval may be done by requesting the set of nodes contained within a certain region of the blackboard (by get.nodes.in.rgn) or requesting the set of nodes which overlap a specified time interval (by get.overlap.nodes).

Table 4b relates the costs of process handling within HSII. Process invocation and process creation are separated (the former being a request from a precondition or knowledge-source process to the scheduler to perform the latter), and the costs are accounted separately, as in invoke.ks and create.ks.prcs. Ks.cleanup is the cost of terminating a knowledge-source process; preconditions never get terminated.

Additionally, local context maintenance costs are given in Table 4b, since they are also a cost of having asynchronous parallel processes. While individual tag creation and deletion is handled by the primitive field read and write operations, tags may be transferred from one tag-ID to another via transfer.tags; clear.tags clears the tags of a given tag-ID from a particular data node; and delete.all.tags destroys all tags belonging to a particular tag-ID. Reset.local.cxt is an operation performed by the system upon setting up a knowledge-source process or upon activating a precondition. Receive.msg is the operation used by precondition or knowledge-source processes to receive a tagging message (or perhaps wait for one, if one does not yet exist); and
<table>
<thead>
<tr>
<th>Operation</th>
<th>w/o lock</th>
<th>w/ lock</th>
<th>mean time (ms)</th>
<th>number of calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>total runtime</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>w/o lock</td>
<td>5.29</td>
<td>2.30</td>
<td>22.64</td>
<td>23.64</td>
</tr>
<tr>
<td>w/ lock</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mean time (ms)</td>
<td>345</td>
<td>345</td>
<td></td>
<td></td>
</tr>
<tr>
<td>number of calls</td>
<td>342</td>
<td>342</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Process Handling:**

<table>
<thead>
<tr>
<th>Operation</th>
<th>w/o lock</th>
<th>w/ lock</th>
<th>mean time (ms)</th>
<th>number of calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>invoke.ks</td>
<td>0.75</td>
<td>0.31</td>
<td>3.21</td>
<td>3.22</td>
</tr>
<tr>
<td>create.ks.prcs</td>
<td>8.20</td>
<td>5.24</td>
<td>35.06</td>
<td>53.94</td>
</tr>
<tr>
<td>ks.cleanup</td>
<td>0.10</td>
<td>1.04</td>
<td>10.44</td>
<td>10.59</td>
</tr>
<tr>
<td>invoke.pre</td>
<td>0.42</td>
<td>0.40</td>
<td>8.53</td>
<td>19.57</td>
</tr>
<tr>
<td>create.pre.prcs</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Local Context Maintenance:**

<table>
<thead>
<tr>
<th>Operation</th>
<th>w/o lock</th>
<th>w/ lock</th>
<th>mean time (ms)</th>
<th>number of calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>transfer.tags</td>
<td>7.12</td>
<td>2.99</td>
<td>9.12</td>
<td>9.17</td>
</tr>
<tr>
<td>clear.tags</td>
<td>0.01</td>
<td>0.00</td>
<td>2.05</td>
<td>2.10</td>
</tr>
<tr>
<td>delete.all:tags</td>
<td>0.52</td>
<td>0.22</td>
<td>2.01</td>
<td>2.03</td>
</tr>
<tr>
<td>reset.local.ctx</td>
<td>0.20</td>
<td>0.09</td>
<td>1.15</td>
<td>1.18</td>
</tr>
<tr>
<td>notify.sset</td>
<td>6.52</td>
<td>3.01</td>
<td>2.63</td>
<td>2.92</td>
</tr>
<tr>
<td>send.msgs.and.integers</td>
<td>4.04</td>
<td>2.12</td>
<td>3.68</td>
<td>4.68</td>
</tr>
<tr>
<td>send.msgs.and.lists</td>
<td>0.47</td>
<td>0.20</td>
<td>2.67</td>
<td>2.69</td>
</tr>
<tr>
<td>receive.msg</td>
<td>0.36</td>
<td>0.15</td>
<td>1.00</td>
<td>1.01</td>
</tr>
<tr>
<td>read.cset.or.sset</td>
<td>0.11</td>
<td>0.05</td>
<td>0.84</td>
<td>0.84</td>
</tr>
</tbody>
</table>

Table 4b. Primitive Operation Times

**read.cset.or.sset** is the operation for retrieving the set of node-ID's associated with a local context change-set or superset.

Finally, Table 4c gives the costs associated with the data access synchronization mechanism. **Lock!** and **unlock!** represent the overhead costs of the LOCK! and UNLOCK! operations, which costs do not include the time spent in performing the actual primitive locking operations. The primitive lock costs are given by **lock.node** (lock a node for exclusive access), **exam.node** (lock a node for read-only access), and **lock.rgn** (lock a region for exclusive access). **Lock.local.ctx** is the operation used by **sendmsgs.and.integers** and **sendmsgs.and.lists** for gaining exclusive access to the various local contexts during the distribution of data events. The access-checking operations (**write.access.chk** and **read.access.chk**) are used by the blackboard accessing routines discussed above.
<table>
<thead>
<tr>
<th></th>
<th>% total runtime</th>
<th>mean time (ms)</th>
<th>number of calls</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>w/o lock</td>
<td>w/ lock</td>
<td>w/o lock</td>
</tr>
<tr>
<td>lock! (overhead)</td>
<td>---</td>
<td>7.78</td>
<td>---</td>
</tr>
<tr>
<td>lock! (overhead)</td>
<td>---</td>
<td>3.22</td>
<td>---</td>
</tr>
<tr>
<td>lock.node (overhead)</td>
<td>---</td>
<td>2.32</td>
<td>---</td>
</tr>
<tr>
<td>exam.node</td>
<td>---</td>
<td>9.34</td>
<td>---</td>
</tr>
<tr>
<td>lock.rgn</td>
<td>---</td>
<td>0.11</td>
<td>---</td>
</tr>
<tr>
<td>lock.local.cxt</td>
<td>---</td>
<td>0.61</td>
<td>---</td>
</tr>
<tr>
<td>write.access.chk</td>
<td>---</td>
<td>0.41</td>
<td>---</td>
</tr>
<tr>
<td>read.access.chk</td>
<td>---</td>
<td>14.45</td>
<td>---</td>
</tr>
</tbody>
</table>

Data Access Synchronization:

Table 4c. Primitive Operation Timings

These timing statistics can be used to determine the amount of system overhead incurred in running precondition and knowledge-source processes under the HSII operating system. The following summary statistics are offered, given as percentages of the total execution time, the percentages being calculated so as to avoid overlapping between categories (as, for example, factoring blackboard reading costs out of blackboard access synchronization):

- Blackboard reading 30%
- Blackboard writing 4%
- Local context maintenance 10%
- Associative retrieval 7%
- Blackboard access synchronization 13%

It should be noted that these blackboard reading and writing percentages include the cost of any access-rights checking required for the associated access (which amounts to 14% for reading and 0.4% for writing, the reading-check cost being proportionately high due to the increased frequency with which nodes are read as opposed to written -- an obvious area for possible implementation optimization).
Effective Parallelism and Processor Utilization

The problem-solving organization underlying HSII was designed to take maximum advantage of any separability of the processing or data components available within that organization. Knowledge sources were intended to be largely independent and capable of asynchronous execution in the form of knowledge-source processes. Overall system control was to be distributed and primarily data-directed, being based on events occurring in a globally shared blackboard data base. The intercommunication (and interdependence) of the various knowledge-source processes was to be minimized by making the blackboard data base the primary means of communication, thereby exhibiting an indirection with respect to communication similar to the indirect data-directed form of process control. Such a problem-solving organization was believed to be particularly amenable to implementation in the hardware environment of a network of closely-coupled asynchronous processors which share a common memory. Given sufficiently many completely non-interfering processes (i.e., processes which do not interfere in any way with the execution progress of one another), one would expect the achieved parallelism (speed-up) of that set of processes executing on \( n \) identical processors to be a factor of \( n \), as compared to the same set of processes executing on a single processor (assuming the same scheduling and multiprocessing overheads). While the HSII organization attempted to allow the various knowledge sources to be as independent as possible, the various processes were to cooperate with one another (primarily via the blackboard data base) in the effort to effect the problem solution. This necessary cooperation (and the various forms of execution interference resulting from it) was expected to result in the achieved parallelism in a multiprocessor environment being somewhat less than the potential parallelism without interference. Several experiments were run to measure the parallelism achieved in this particular implementation of the HSII problem-solving organization using varying numbers of identical processors, the results of which are summarized in the following figures. These same effective parallelism experiments also provided data which was used to measure processor utilization. Recall that any given part of the blackboard data base is assumed to be (time-wise) equally accessible to all processors, and all parts of the data base are also (time-wise) equally accessible to any given processor; that is, the blackboard is centrally located in an homogeneous storage medium.\(^1\)

\(^1\) Note that the size of the HSII blackboard is expected to grow to only several
The graphs of Figures 3 to 12 summarize the results of the simulation experiments which were run to determine effective parallelism and processor utilization. Each of these experiments was run using the knowledge-source set described previously, using the same input data (introduced into the data base so as to simulate real-time speech input), the same blackboard locking structure, and the same scheduling algorithm, while varying the number of (identical) processors. To comment on these activity plots, the "# runnable processes" plots give the number of processes either running or ready to run at each simulation scheduling point; the "# running processes" plots give the number of actively executing processes at each scheduling point; the "# ready processes" plots show the number of processes awaiting assignment to a processor at each scheduling point; and the "# suspended processes" plots give the number of processes blocked from executing because of data access interference or because they are waiting on the receipt of a tagging message. As might be expected, for example, the running-processes plot for the uniprocessor case is a constant plot where the single processor is always active; and its suspended-processes plot is empty (no one ever gets blocked because no other process can interfere with the executing process).

A feeling for the amount of processor utilization may be gained by comparing the various running-processes plots across the various processor configurations. For one, two, four and eight processors, the plots show all processors usually being kept busy; but in the sixteen-processor case (Figure 11b), only once are all sixteen processors in use simultaneously. Due to the limited state of development of the total set of knowledge sources, the set of knowledge sources used in the simulation was necessarily limited; so the fact that these plots indicate that not more than about eight processors are being effectively utilized is not to say that the full HSII speech-understanding system needs only eight processors -- much more parallelism is possible given a more complete knowledge source set, as will be discussed further below.

Referring to Figure 3c, notice the spiked nature of the ready-processes plot. This is a result of delaying the execution of a precondition (due to the limited thousand nodes (hypotheses and links), at, say, 25 field entries apiece, depending, of course, on the task domain. Thus, it is assumed (for the purposes of the current investigations, at least) that the blackboard is entirely resident in primary memory; input/output operations are not an issue here, the system being essentially compute-bound.
Figure 3. 1 Processor
Figure 4. 1 Processor
Figure 5. 2 Processors
Figure 6. 2 Processors
Figure 7. 4 Processors
Figure 8. 4 Processors
Figure 9. 8 Processors
Figure 10. 8 Processors
Figure 11. 16 Processors
Figure 12. 16 Processors
processing power available) beyond the point in time at which its pre-precondition is first satisfied: the longer a precondition is delayed, the more data events it is likely to accumulate in the meantime, and the more knowledge-source processes it is likely to instantiate once it does get executed; hence the spiked nature of the resultant ready-processes plots for configurations of few processors. As parallel processing power increases, preconditions can more often be run as soon as their pre-preconditions are initially satisfied, and the spiking phenomenon subsides. In the case of sixteen processors (where Figure 11b indicates low processor utilization), the small spikes in Figure 11c are a phenomenon of the grain of simulation: when a precondition instantiates a knowledge-source process, that process first becomes "ready," and is accounted for immediately on the ready-processes plot; shortly thereafter, the knowledge-source process will be scheduled (since the sixteen processors are under-utilized) and become "running," then being transferred to the running-processes plot. Hence, the running-processes plot lags slightly behind the ready-processes plot, and peaks occur on the ready-processes plot even when plenty of processing power is available. The ready-processes plots can thus be used to observe the rate at which processes enter the system, since all processes are initially assigned to the ready-processes plot.

As an example of how these activity plots have been used in upgrading the performance of the implementation, compare Figure 13a to Figure 13b. Figure 13a depicts the process activity under the control of a scheduler which did not attempt to perform load balancing with respect to ready preconditions; and as a result of not increasing the relative priority of preconditions as they receive more and more data events, the activity spike phenomenon referred to above became predominant, to the extent of reducing process activity to a synchronous system while the long-time waiting precondition instantiates a great many knowledge-source processes all at once.\(^1\) Figure 13b (which is the same plot as Figure 9c) shows the activity on the same number of processors, but using a somewhat more intelligent scheduling algorithm, with reduced spiking phenomena. This improved scheduling strategy is the one used for all plots presented herein, except Figure 13a.

\(^1\) This can be inferred from Figure 13a by noting that the sample points (vertical tick marks) are taken at each simulation scheduling point, and the lack of samples between times 220 and 380 indicate that the process that started running at 220 had no concurrently running processes competing with it until time 380, when there were suddenly 25 new processes contending for computing resources.
Figure 13a. 8 Processors, old scheduling strategy
Figure 13b. 8 Processors, current scheduling strategy
The various suspended-processes plots of Figures 3 to 12 indicate the
dynamic process interference activity over the various processor configurations. These
plots will be described in more detail in the next section, as will the accompanying plots
on descheduling activity and context swap activity.

<table>
<thead>
<tr>
<th>number of prcrrs</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32 (special)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(all times in secs)</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>--------------</td>
</tr>
<tr>
<td>activity Fig. #’s</td>
<td>3.4</td>
<td>5.6</td>
<td>7.8</td>
<td>9.10</td>
<td>11.12</td>
<td>16.17</td>
</tr>
<tr>
<td>KS instantiations</td>
<td>355</td>
<td>401</td>
<td>423</td>
<td>421</td>
<td>415</td>
<td>434</td>
</tr>
<tr>
<td>PRE activations</td>
<td>82</td>
<td>126</td>
<td>173</td>
<td>213</td>
<td>200</td>
<td>229</td>
</tr>
<tr>
<td>hyps created</td>
<td>157</td>
<td>159</td>
<td>163</td>
<td>170</td>
<td>177</td>
<td>179</td>
</tr>
<tr>
<td>links created</td>
<td>126</td>
<td>132</td>
<td>138</td>
<td>151</td>
<td>160</td>
<td>165</td>
</tr>
<tr>
<td>cpu secs</td>
<td>2058</td>
<td>2468</td>
<td>3167</td>
<td>4219</td>
<td>n.a.</td>
<td>3998</td>
</tr>
<tr>
<td>sim sched points</td>
<td>62879</td>
<td>71868</td>
<td>78578</td>
<td>81767</td>
<td>80248</td>
<td>35835</td>
</tr>
<tr>
<td>multiprcr clock time</td>
<td>1076</td>
<td>634</td>
<td>389</td>
<td>350</td>
<td>351</td>
<td>43</td>
</tr>
<tr>
<td>total idle time</td>
<td>9</td>
<td>15</td>
<td>37</td>
<td>380</td>
<td>2608</td>
<td>867</td>
</tr>
<tr>
<td>total lost time</td>
<td>0</td>
<td>5</td>
<td>34</td>
<td>900</td>
<td>1546</td>
<td>0</td>
</tr>
<tr>
<td>avg prcr utilization</td>
<td>99%</td>
<td>98%</td>
<td>95%</td>
<td>54%</td>
<td>26%</td>
<td>37%</td>
</tr>
<tr>
<td>effective # prcrrs</td>
<td>0.99</td>
<td>1.96</td>
<td>3.80</td>
<td>4.32</td>
<td>4.16</td>
<td>11.84</td>
</tr>
<tr>
<td>utilization speed-up</td>
<td>1.00</td>
<td>1.98</td>
<td>3.84</td>
<td>4.36</td>
<td>4.20</td>
<td>11.96</td>
</tr>
</tbody>
</table>

Table 5. Processor Utilization

In addition to the plots described above, various other measures were made
to allow an explicit determination of processor utilization and effective parallelism for
varying numbers of processors. Referring to Table 5, one can get a feeling for the
activity generated by employing increasing numbers of processors (the 32-processor
column represents an experiment which was run under special conditions, to be
explained below, and should not be compared directly to the other columns of the table).
All simulations were run for equivalent amounts of processing effort with respect to the
results created in the blackboard data base by the knowledge source activity. Notice
that as more processing power is made available, the time between pre-precondition
satisfaction and precondition activation will decrease, resulting in an increase in the
number of precondition activations (although the activated preconditions are not always being satisfied, as is seen by the approximately constant number of knowledge-source processes being executed).

The resultant blackboard size for the simulation run is given in terms of the number of hypotheses and links created over the run; under non-simulation production conditions (when the system runs for a much longer period of time, activity-wise), the number of blackboard nodes created is often on the order of a thousand nodes per utterance (which is still small enough to be entirely contained within a central main memory). The actual cpu seconds and number of simulation scheduling points are shown to give an idea of the expense of running such a simulation. The final clock time of the multiprocessor configuration being simulated is given in simulated real-time seconds, and the accumulated processor idle and lost times are also given. *Idle time* is attributed to a processor when it has no process assigned to it and there are no ready processes to be run; *lost time* is attributed when the process on a processor is suspended for any reason and there are no ready processes which could be swapped in to replace the suspended process. As might be anticipated by the previous activity plots, processor utilization (calculated using the final clock time and processor idle and lost times) decreases as the number of processors increases. The processor utilization results are given in Table 5 and are plotted in Figure 14.

Using these processor utilizations, one measure of effective parallelism may be computed; these results are also given in Table 5 and are plotted in Figure 15. As expected from previous plots (especially the running-processes plots), the speed-up for this particular selection of knowledge sources is appreciable up to four processors, but drops off substantially as one approaches sixteen processors. In fact, a rather distressing feature of this effective parallelism plot is that the speed-up actually decreases slightly in going from eight processors to a sixteen-processor configuration (from a speed-up of 4.36 over the uniprocessor case, down to 4.20). This may be explained by noting that both the eight- and sixteen-processor runs had approximately equal final clock times; but in the sixteen-processor case, the number of runnable processes never exceeded sixteen processes, so any ready process could always be accommodated immediately. As a result, the number of knowledge-source instantiations and precondition activations fell off a bit from the eight-processor case, because the preconditions were more likely to be fully satisfied the first time they were activated (since all ready-processes, knowledge-source processes in particular, could be executed
Figure 14. Average Processor Utilization
Figure 15. Effective Parallelism According to Processor Utilization
immediately and complete their intended actions sooner, so that when a precondition came to be activated, it would more likely find its full data pattern to be satisfied; thus, preconditions would not often be aborted, having to be re-tested upon receiving a subsequent data event. However, running fewer preconditions resulted in much more idle time for the sixteen-processor configuration (the increase in lost time is an artifact of having too many processors available, since suspended processes would tend to remain on otherwise idle processors rather than being swapped off the processor -- note the rather dramatic decrease in context swaps indicated by Table 8 for the sixteen-processor case). The result is a lower proportionate utilization of the processor configuration, and hence a decrease in the effective parallelism from the eight-processor configuration to the sixteen-processor configuration.

Referring to the "effective * prcrs" line of Table 5, one might ask that if only 4.16 processors of the sixteen-processor configuration are being totally utilized, what is the maximum potential effective parallelism, given this set of knowledge sources? To answer this question, an experiment was performed in which effectively infinite processing power was provided to this knowledge-source set and all data access interference was eliminated (by removing the locking structure overheads and blocking actions); the scheduling algorithm was kept unchanged, as was the input data, although the input data stream was entered so as to be instantaneously available in its entirety (rather than being introduced in a simulated real-time, "left-to-right" manner). The results of this experiment are summarized by the 32-processor column of Table 5 (32 processors was an effective infinite computing resource in this case, since eight of the processors were never used during the simulation). Notice that no lost time was attributed to the run, due to the lack of locking interference; and the resultant processor utilization was 37% of 32 processors, or 11.84 totally utilized processors. From such a utilization factor, the effective parallelism (speed-up) over the uniprocessor case for this knowledge-source set is 11.96; so if the use of the locking structures could be accomplished in a non-interfering manner, the speed-up indicated by the eight- or sixteen-processor configurations could be increased substantially.

Table 6 presents some other system configurations to show effective processor utilizations under varying conditions. The first row repeats the statistics of the sixteen-processor case of Table 5; the second row is a summary of the 32-processor case of Table 5, as just described. Three further data points are offered to indicate the effects of increasing the size of the knowledge-source set. The last three
<table>
<thead>
<tr>
<th>experiment description</th>
<th>multiprcr clock</th>
<th>total idle</th>
<th>total lost</th>
<th>% util</th>
<th>effective # prcrs</th>
<th>activity Fig. #’s</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 KS’s, 6 PRE’s</td>
<td>351</td>
<td>2608</td>
<td>1546</td>
<td>26%</td>
<td>4.16</td>
<td>11,12</td>
</tr>
<tr>
<td>16 prcrs, w/ lock l-to-r input</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 KS’s, 6 PRE’s</td>
<td>43</td>
<td>867</td>
<td>0</td>
<td>37%</td>
<td>11.84</td>
<td>16,17</td>
</tr>
<tr>
<td>32 prcrs, w/o lock instantaneous input</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9 KS’s, 7 PRE’s</td>
<td>148</td>
<td>854</td>
<td>726</td>
<td>33%</td>
<td>5.28</td>
<td>18,19</td>
</tr>
<tr>
<td>16 prcrs, w/ lock l-to-r input</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9 KS’s, 7 PRE’s</td>
<td>155</td>
<td>839</td>
<td>784</td>
<td>35%</td>
<td>5.60</td>
<td>20,21</td>
</tr>
<tr>
<td>16 prcrs, w/ lock instantaneous input</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9 KS’s, 7 PRE’s</td>
<td>13</td>
<td>226</td>
<td>0</td>
<td>46%</td>
<td>14.72</td>
<td>----</td>
</tr>
<tr>
<td>32 prcrs, w/o lock instantaneous input</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6. System Configuration Variations

rows of Table 6 involve experiments using an expanded knowledge-source set consisting of the knowledge sources of all the previous runs plus the Syntactic Word Hypothesizer (see Appendix A) and its precondition. Using this expanded knowledge-source set, simulations were performed to evaluate the effects of this knowledge-source set on a sixteen-processor configuration with the locking structure in effect, presenting the input data in the usual "left-to-right" manner, as well as in the instantaneous manner used in the infinite-processor test. Comparing the results (in Table 6) to the original sixteen-processor run, the "left-to-right" input scheme achieved a processor utilization of 33%, up 7% from the smaller knowledge-source set case; and by presenting all input data simultaneously, the utilization rose to 35%. Figures 11, 12, and 16 to 21 relate the activity plots for the configurations given in Table 6. The fifth row of Table 6 represents the results of providing effectively infinite computing power (only 25 processors were ever used during the run) to the expanded knowledge-source set and
eliminating all data access interference, in the same manner as for the experiment of the second row. In this "optimal" situation for the expanded knowledge-source set, processor utilization was measured at 46%, or 14.72 totally utilized processors. Again, it may be noted that a more effective (less interfering) use of the locking structures can result in substantial increases in processor utilization and effective parallelism.

The addition of the Syntactic Word Hypothesizer was able to achieve the increases in utilization noted in Table 6 because it operates on lexicons that are used by only one other knowledge source (the Phoneme Hypothesizer) in the basic knowledge-source set; hence, the process interference introduced by adding this knowledge source was minimal. Unfortunately, the development of knowledge sources at lexicon levels which more directly conflict with those of existing knowledge sources is limited, so direct experimentation on the interfering effects of such knowledge sources could not be performed; but based on the observations comparing the 32-processor without-lock experiments to the original sixteen-processor with-lock runs, substantial interference due to ineffective use of the locking structure would be expected in such cases of adding "competing" knowledge sources. One mitigating circumstance which could alleviate such interference was noted in the "instantaneous" input case of the expanded knowledge-source set case, as compared to the "left-to-right" input case: if process activity can be spread across the utterance-time dimension of the blackboard, process interference would decrease -- but interference due to data access synchronization interference can easily overwhelm this improvement. Further experiments along these lines will be attempted as the appropriate knowledge sources become available for use.

Another measure of effective parallelism may be obtained by using the measures of total process runtime given in Table 7, which is an extension of Table 5. This effective parallelism measure is calculated by measuring the increase in throughput based on the number of knowledge-source processes executed, after subtracting out the pro rata runtime cost of the precondition processes and normalizing according to knowledge source activity. For example, on one processor, the 355 knowledge sources used 66% of the clock time of 1076, for an average of 2.00 secs/knowledge-source instantiation; similarly, the two processor case yields 1.04 secs/knowledge-source instantiation. Thus, the resultant speed-up for two processors by this measure of knowledge-source throughput is 1.92. This method of measuring the speed-up is summarized in Figure 22, using the figures for average process throughput (which
Figure 16. 32 Processors, optimal case
Figure 17. 32 Processors, optimal case
Figure 18. 16 Processors, expanded KS set, I-to-R input
Figure 19. 16 Processors, expanded KS set, l-to-r input
Figure 20. 16 Processors, expanded KS set, instantaneous input
Figure 21. 16 Processors, expanded KS set, instantaneous input
Table 7. Process Runtime Characteristics

<table>
<thead>
<tr>
<th>number of prcprs (all times is secs)</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>KS instantiations</td>
<td>355</td>
<td>401</td>
<td>423</td>
<td>421</td>
<td>415</td>
<td>434</td>
</tr>
<tr>
<td>PRE activations</td>
<td>82</td>
<td>126</td>
<td>173</td>
<td>213</td>
<td>200</td>
<td>229</td>
</tr>
<tr>
<td>multipprcr clock time</td>
<td>1076</td>
<td>634</td>
<td>389</td>
<td>350</td>
<td>351</td>
<td>43</td>
</tr>
<tr>
<td>avg runtime/KS</td>
<td>2.00</td>
<td>1.98</td>
<td>2.21</td>
<td>2.10</td>
<td>2.19</td>
<td>0.61</td>
</tr>
<tr>
<td>total KS runtime</td>
<td>709</td>
<td>794</td>
<td>937</td>
<td>885</td>
<td>909</td>
<td>264</td>
</tr>
<tr>
<td>KS thruput speed-up</td>
<td>1.00</td>
<td>1.92</td>
<td>3.28</td>
<td>3.85</td>
<td>3.64</td>
<td>29.85</td>
</tr>
<tr>
<td>avg runtime/PRE</td>
<td>4.48</td>
<td>3.21</td>
<td>2.75</td>
<td>2.50</td>
<td>2.64</td>
<td>0.55</td>
</tr>
<tr>
<td>total PRE runtime</td>
<td>367</td>
<td>404</td>
<td>477</td>
<td>533</td>
<td>528</td>
<td>126</td>
</tr>
<tr>
<td>PRE thruput speed-up</td>
<td>1.00</td>
<td>2.64</td>
<td>5.89</td>
<td>7.23</td>
<td>7.00</td>
<td>73.44</td>
</tr>
<tr>
<td>avg thruput speed-up</td>
<td>1.00</td>
<td>2.05</td>
<td>3.78</td>
<td>4.47</td>
<td>4.32</td>
<td>37.85</td>
</tr>
</tbody>
</table>

considers both knowledge-source and precondition activity). Notice that since the sixteen-processor configuration executed only about as many preconditions and knowledge-source processes as the eight processor configuration, and it took the same amount of clock time to do it (the sixteen-processor configuration being very under-utilized), the speed-up according to knowledge source throughput is negligible in going from eight to sixteen processors (and in this case actually dropped off slightly, as is seen in Figure 22, for reasons similar to those of the speed-up curve based on processor utilization, Figure 15).

From the total process runtimes given in Table 7 and from statistics on precondition and knowledge-source process runtime cost and frequency, it may be estimated that knowledge-source processes consume about 65% of the total execution time of the system (including all the HSI| operating system functions they might execute) and preconditions consume about 27% of the execution time, leaving about 8% of the execution time being used in pure system overhead. If the costs of blackboard reading and writing, local context maintenance, associative retrieval, and blackboard access synchronization summarized previously are attributed to the operating system instead, knowledge sources and preconditions end up with about 30% of the total execution time for their own purposes.
Figure 22. Effective Parallelism According to Process Throughput
Execution Interference Measurements

In addition to the primitive operation timings and achieved parallelism measurements given above, various other measurements were made to determine the various aspects of system performance as related to multiprocessing. As has already been mentioned, a major concern in a multiprocess environment in which the various processes are not entirely independent is that of execution interference. Execution interference may arise whenever any given process enters a critical section within which it requires the integrity of a given data structure be maintained (thereby necessitating a means by which to disallow access to others until the original critical section is exited). Execution interference may also arise whenever processes must synchronize their activities and perhaps cause themselves to wait on an event based on an action which is to be performed by some external process. Thus execution interference may arise due to causes external to the process being delayed (as in the case of trying to access a data structure which is currently held for exclusive access by another process), or the interference may arise due to causes internal to the process being delayed (as when a process delays itself by waiting for the occurrence of an externally caused event). As a result of the HSII design philosophy, which states that the various knowledge-source processes should be as independent as possible in specification and execution, most of the execution interference experienced in HSII is of the external variety, wherein a process is delayed due to external causes unknown to itself (and the delay itself is transparent to the process being delayed).

In an attempt to measure the external execution interference, several experiments were performed and memory contention in the form of data base access interference was measured. The shared data base of HSII consists of hypothesis nodes existing within various information levels. The hypothesis nodes may be linked across levels or within levels by specifying link nodes. Both hypothesis nodes and link nodes consist of various pre-defined fields which contain the actual data values. Accessing the shared data base usually consists of accessing a field or group of fields within a specified node. For data integrity reasons, various pieces of the data base, including and auxiliary to the actual fields being accessed, often need to be reserved for the exclusive use of the process while it is accessing the given data fields. It is in providing for this exclusive access that execution interference may occur, should two or more processes request simultaneous exclusive access to some portion of the shared data base. Various simulations were performed to measure the amount of data access
interference while varying the number of contending processes (by varying the number of simulated processors available for knowledge source scheduling). Additional locking flexibility was introduced by allowing knowledge-source processes to request read-only access to data fields; this reduces possible contention by permitting multiple readers of a given field to coexist, while excluding any writers of that field until all readers are finished. The overlapping of areas of data reference locality is what causes the data access interference.

Table 8 contains various measures of the amount of data access and data access interference experienced by precondition and knowledge-source processes, for varying numbers of processors. Essentially, Table 8 is an extension of Table 5, which was discussed above (i.e., the underlying simulation runs were the same for both tables). As might be expected, the average process runtime for a knowledge-source process is about constant across processor configurations, individual processors being assumed to be identical. However, note the general decrease in process runtime for preconditions as the number of processors increases. This phenomenon may be explained by recalling the ready-processes plots of Figures 3 to 12: for situations of high processor utilization (few processors), the spiked nature of the the plots was due to delaying the execution of preconditions (necessitated by the limited processing power available), which resulted in each precondition instantiating more knowledge-source processes (due to the additional data events the preconditions received while awaiting execution). This spiking nature is also reflected in the values for the average numbers of knowledge-source processes instantiated by each precondition, as given in Table 8. The trends displayed by the average numbers of primitive lock operations and the average numbers of blackboard accesses by preconditions as a function of the number of processors available may be similarly explained. Note that the number of primitive lock operations for preconditions is equal to the number of blackboard accesses (from the precondition process averages of Table 8): preconditions do not usually need a long-lasting locked environment (since they do not modify the blackboard except to place tags into it), thus each access is individually protected by the HSSI operating system (via temporary-locking), rather than having the precondition perform an explicit LOCK! operation before each access.

Execution interference was measured by recording the amount of process suspension (also called descheduling), which usually resulted from processes being temporarily blocked in their attempts to gain access to some part of the blackboard data
<table>
<thead>
<tr>
<th>number of prcrs (all times in secs)</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>KS instantiations</td>
<td>355</td>
<td>401</td>
<td>423</td>
<td>421</td>
<td>415</td>
</tr>
<tr>
<td>PRE activations</td>
<td>82</td>
<td>126</td>
<td>173</td>
<td>213</td>
<td>200</td>
</tr>
<tr>
<td>avg KS's/PRE</td>
<td>4.34</td>
<td>3.18</td>
<td>2.44</td>
<td>1.98</td>
<td>2.08</td>
</tr>
<tr>
<td>avg runtime/KS</td>
<td>2.00</td>
<td>1.98</td>
<td>2.21</td>
<td>2.10</td>
<td>2.19</td>
</tr>
<tr>
<td>avg runtime/PRE</td>
<td>4.48</td>
<td>3.21</td>
<td>2.75</td>
<td>2.50</td>
<td>2.64</td>
</tr>
<tr>
<td>avg LOCK!'s/KS</td>
<td>1.39</td>
<td>1.36</td>
<td>1.38</td>
<td>1.26</td>
<td>1.30</td>
</tr>
<tr>
<td>avg LOCK!'s/PRE</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>avg LOCK! duration/KS</td>
<td>1.00</td>
<td>1.33</td>
<td>1.91</td>
<td>1.72</td>
<td>1.93</td>
</tr>
<tr>
<td>avg LOCK! duration/PRE</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>avg prim locks/KS</td>
<td>27.9</td>
<td>27.4</td>
<td>28.0</td>
<td>25.7</td>
<td>26.9</td>
</tr>
<tr>
<td>avg prim locks/PRE</td>
<td>96.7</td>
<td>68.7</td>
<td>55.7</td>
<td>48.2</td>
<td>51.1</td>
</tr>
<tr>
<td>avg BB accesses/KS</td>
<td>54.4</td>
<td>52.8</td>
<td>54.5</td>
<td>53.9</td>
<td>56.4</td>
</tr>
<tr>
<td>avg BB accesses/PRE</td>
<td>96.7</td>
<td>68.7</td>
<td>55.7</td>
<td>48.2</td>
<td>51.1</td>
</tr>
<tr>
<td>avg dscheds/KS</td>
<td>0</td>
<td>0.55</td>
<td>1.74</td>
<td>1.42</td>
<td>1.42</td>
</tr>
<tr>
<td>avg dscheds/PRE</td>
<td>0</td>
<td>0.64</td>
<td>1.43</td>
<td>2.16</td>
<td>2.04</td>
</tr>
<tr>
<td>avg dsched duration/KS</td>
<td>0</td>
<td>5.08</td>
<td>5.69</td>
<td>1.75</td>
<td>1.90</td>
</tr>
<tr>
<td>avg dsched duration/PRE</td>
<td>0</td>
<td>3.95</td>
<td>1.91</td>
<td>1.35</td>
<td>1.86</td>
</tr>
<tr>
<td>ctx swaps</td>
<td>409</td>
<td>834</td>
<td>1538</td>
<td>1012</td>
<td>624</td>
</tr>
<tr>
<td>total swap cost</td>
<td>29</td>
<td>37</td>
<td>72</td>
<td>23</td>
<td>11</td>
</tr>
<tr>
<td>cost/swap</td>
<td>0.07</td>
<td>0.04</td>
<td>0.05</td>
<td>0.02</td>
<td>0.02</td>
</tr>
</tbody>
</table>

Table 8. Data Access Characteristics

The dynamic nature of this process suspension activity is shown in the suspended-processes plots for the various processor configurations (Figures 3d, 5d, 7d, 9d, 11d, 16d, 18d, and 20d). Comparing the first five of these plots, note that the number of suspended processes for the uniprocessor case is zero, since every process runs to completion in this case. The number of suspended processes for the two-processor case is generally comparatively low since that configuration can only accommodate two active processes at a time, and the chance of interference is thence
fairly minimal. In the four-processor case, processor utilization is still quite high, so about four processes will always be active, increasing the chance of execution interference (as verified by Figure 7d). As processor utilization decreases, it becomes more and more likely that as soon as a suspended process becomes unblocked, it will be able to be allocated to a processor to resume its execution. Thus, the plots for the eight- and sixteen-processor configurations show a reduced suspended-process rate, when compared to the four processor case; in the four processor case, even if a process were to become unblocked, it would often have to wait to be reassigned to some processor so it might resume its execution, and the longer that waiting period, the more likely it is that some other process would become blocked and join the ranks of the suspended, thereby increasing the suspended-process activity statistics (as shown by Figure 7d).

Execution interference is also indicated by the results on process descheduling (suspending) given in Table 8, as well as in Figures 4a, 6a, 8a, 10a, 12a (and 17a, 19a, 21a). As might be expected, as process activity increases with increasing numbers of processors, the possibility of execution interference increases (unless the global scheduling policy guards against this by scheduling processes which will not conflict with one another, such information being available in some form to a goal-directed scheduler), and more processes are descheduled per second, as the plots verify. At the same time, with more and more processing power available, the likelihood of suspended processes being unblocked and becoming available for further processing increases as the number of processors increases, as was pointed out in the discussion of the suspended-processes plots; thus Figure 12a shows a decrease in descheduling activity for sixteen processors over the eight-processor case of Figure 10a.

This discussion on process suspension, or descheduling, may be summarized by looking at Figures 23 and 24. Figure 23 indicates that the average number of deschedulings for any given knowledge-source process increases dramatically from one to four processors, but drops off as the number of processors increases beyond that; as processor utilization drops off (for eight and sixteen processors), the reasons that a process might become suspended (e.g., by data access interference) can be resolved more quickly (by allowing interfering processes to complete their data accesses), thereby reducing the possibility of process descheduling. Similarly, Figure 24 shows the average duration of a descheduled period for any given knowledge-source process. Again, as processor utilization decreases, it becomes more likely that the reason a
Figure 23. Average Number of Deschedules / KS Process
Figure 24. Average Duration of Deschedule Period for KS Processes
process became suspended will be resolved more quickly, thereby allowing the suspended process to become unblocked and reassigned to a processor for continued execution in a shorter period of time than if the processors were always busy.\textsuperscript{1}

A related set of execution interference plots is given in Figures 4b, 6b, 8b, 10b, 12b (and 17b, 19b, 21b). These figures indicate the relationship between one process being descheduled (suspended) and the availability of some ready-to-run process which can take the place of the suspended process. As the number of ready processes increases, the possibility of context swapping a suspended process to replace it with a ready one increases. Notice that the swapping activity decreases as processor utilization decreases, since the number of ready processes also decreases. These context swap plots may be correlated with the statistics on the number of context swaps given in Table 8. Note that the initial start-up of each knowledge-source process involves a context swap to get the new process onto a processor; hence, for example, the swap activity for the sixteen-processor case (Figure 12b) is entirely involved in these initial swaps, as might be expected due to the very low utilization of the sixteen-processor configuration.

One further measure on execution interference (or perhaps execution cooperation, to be more positive about it) was accumulated. Figures 25 to 28 represent snapshots of the blackboard locking structure taken at a few random points during the execution of the simulation. The grid structure represents the two-dimensional abstract data structure, the dimensions being lexicon level and region element number. At the

\textsuperscript{1} The number of deschedules attributed to a process is related to the inner workings of the locking mechanism. Not only is the granularity of the locking structure important (i.e., how small a piece of the blackboard data base can be requested for access allocation), but the granularity of the process blocking mechanism is important. For example, processes could be blocked upon trying to gain access to a node and then relegated to waiting in a set of processes which are waiting on any node at the level of the requested node; or the wait set could be divided according to the individual nodes being waited upon. If, in an attempt to conserve semaphore structures, the former strategy is chosen, it could become quite expensive to determine whether, upon receiving an unlock wake-up signal for the wait set, a particular member of the wait set is really re-schedulable as a result of that wake-up signal; hence, it may be cheaper to release all waiting processes in the set, even though all but one will just become descheduled again. If the single-node wait set is used, the costs of maintaining separate semaphores for every possible data object may become prohibitively expensive, although process re-scheduling would not be done unnecessarily in such a scheme.
Figure 25. 16 Processors, blackboard lock map at time 150.9

<table>
<thead>
<tr>
<th>Region Locks</th>
<th>Node Locks</th>
<th>Region Waits</th>
<th>Node Waits</th>
<th>Tags</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>H16, H20, H142, L14, L99</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>k</td>
<td>H20</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>l</td>
<td>H16</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>m</td>
<td>H16</td>
<td></td>
<td>7</td>
</tr>
</tbody>
</table>

Running Processes:
A/1: PREIPSC!PSC
B/2: PREIUTTBOUNDARIES!PSC
C/3: PREIPSYN!PSYN
D/4: !UV256
E/5: !CSEG258
F/6: !CSEG259
G/7: !CSEG260
H/8: !PSYN261
I/9: !PSYN262
J/10: !PSYN263

Suspended Processes:
K/11: PREIRPOL!IRPOL
L/12: !TIME253
M/13: !UV257
Figure 26. 16 Processors, blackboard lock map at time 155.1

<table>
<thead>
<tr>
<th>Region Locks</th>
<th>Node Locks</th>
<th>Region Waits</th>
<th>Node Waits</th>
<th>Tags</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>H150, H151, H152</td>
<td>b: H146</td>
<td>1: H72, H75, H75, H77</td>
<td></td>
</tr>
<tr>
<td></td>
<td>L101, L102</td>
<td>c: H142</td>
<td>4: H149</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>d: L100</td>
<td>7: H69, H70, H70,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>e: H141</td>
<td>H72, H141</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>f: H141</td>
<td>g: H69</td>
<td></td>
</tr>
</tbody>
</table>

Running Processes
A/1: IPSE263

Suspended Processes
B/2: PREIPSCIPSC
C/3: PREIUTTBOUNDARIESIPSC
D/4: PREIRPOLIRPOL
E/5: PREIPSYNIPSYN
F/6: ICSEG259
G/7: IPSE262
Figure 27. 16 Processors, blackboard lock map at time 163.6

Running Processes
A/1: PRE!PSC!PSC
B/2: PRE!UTTBOUNDARIES!PSC
C/3: !PSYN262
D/4: !UV268
E/5: !TIME269

Suspended Processes
F/6: PRE!RPOL!RPOL
G/7: PRE!PSYN!PSYN
H/8: !UV270

<table>
<thead>
<tr>
<th>SHDSENT</th>
<th>SHDWORD</th>
<th>WORD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>WRDSURN</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>D</td>
<td>D</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>D: H72,H149,L100</td>
<td>f: L101</td>
<td>3: H69,H70,H70,</td>
<td>H72,H141</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g: H123</td>
<td>6: H150</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>h: L102</td>
<td>7: H61,H62,H62,</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>H64,H123,H123</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Figure 28. 16 Processors, blackboard lock map at time 183.1

Running Processes
A/1: PREIPSC!PSC
B/2: PREIPOL!IPOL
C/3: PREIPSYN!IPSYN
D/4: ISEARCH279
E/5: !TIME280
F/6: !UV282
G/7: !UV283
H/8: !UV284

Suspended Processes
1/9: PREIUTTBOUNDARIES!PSC
J/10: !TIME281

Region Locks | Node Locks |
---|---|
E | F: H123,H155,L106 |
| G: L107 |
| i: H143 |
| j: H154 |
| 3: H88,H92,H92,H92, |
| H94,H94,H94,H94, |
| H98,H98,H98,H98, |
| H106,H112 |
| 4: H27,H27,H38 |
point of each snapshot, the outstanding node and region locks are indicated, as well as the areas requested (but not yet obtained) by suspended processes. The various (non-interfering) tags placed throughout the data base are also indicated. The key indicates the sets of active and suspended processes (the names referring to the precondition and knowledge source names given previously, and the numbers in the names indicating a process instantiation index unique to that particular process). These particular snapshots were taken from the sixteen-processor simulation run with the smaller knowledge-source set. Figure 26 shows an interesting case demonstrating that the indiscriminate use of region-locking can obstruct the execution progress of many processes and thereby temporarily reduce the effective parallelism of the system. Notice that PSYN263 has locked regions at the PHON, MXN, and PSEG lexicon levels for its exclusive access; the nodes locked by PSYN263 (hypotheses being indicated by H<sequence number>, and links by L<sequence number>) within these regions are those being created by PSYN263, hence the reason for the region locks. Unfortunately, this locking action resulted in the suspension of six other processes awaiting access to parts of the PHON and PSEG lexicon levels which overlap PSYN263's region locks. Each of these suspended processes (see Figure 26) is waiting to acquire access-rights to a node in these locked regions; in fact, PREPSYNPSYN and CSEG259 are both waiting on the same node (H141). The diagram also shows the various (non-interfering) tags which were placed on the various nodes at the PHON and PSEG lexicon levels by three of the processes at some previous time.

The reason all the locking structure plots are localized in the lower left-hand corner of the blackboard structure is that the construction of the data base in the speech-processing task is initially left-to-right due to the time-sequential nature of the speech input. Also, the particular set of knowledge sources chosen for use in the simulation experiments happened to be an effectively bottom-up speech recognition system (some of the top-down knowledge sources having not yet been developed to a stable enough state to have been used in the simulations); hence, activity starts in the lower left-hand corner of the blackboard. Further simulations are planned which will work in a combined top-down and bottom-up fashion, thereby increasing the potential parallelism (since the top-down knowledge sources will presumably not interfere with the execution of the bottom-up knowledge sources as much as additional competing bottom-up knowledge sources would). The expanded knowledge-source set experiments presented above were a first step in introducing such top-down knowledge; as more
knowledge sources become available, their various interference effects will be investigated. Also, other tasks which could use the HSII organization might not necessarily have the left-to-right input characteristics of speech, so future simulations will also test a more distributed input pattern, thereby also increasing the potential parallelism by spreading the process activity across the breadth of the blackboard; the several experiments presented above which introduced the input in an "instantaneous" manner were the initial attempts in this direction. Since the input distribution pattern primarily affects the interference patterns of the lower level knowledge sources, as more low-level knowledge sources are developed, these interference patterns will be tested and reported in future reports.

Finally, it is once again admitted that the results presented here are derived from a rather limited selection of knowledge-source processes, the coding style of which may be affected by the various efficiencies and inefficiencies of the particular implementation of the HSII system organization. In particular, since the HSII speech-understanding system is under constant development, various code sections involving the system operations have been subject to extensive optimization attempts, while other sections have not yet had the benefit of such optimization. Additionally, the results are biased by the task domain (viz., speech understanding) and the data structure chosen to represent the dynamic solution state of the task. However, it is hoped that the system organization (including the data base design) is of sufficiently general character that these particular results at least give a feeling for the results that might be expected using a different set of knowledge-source processes to solve the same or different problems.
CONCLUSIONS AND FUTURE RESEARCH

A Summary

This dissertation has presented a design for the organization of knowledge-based AI problem-solving strategies which is felt to be particularly applicable for implementation on closely-coupled multiprocessor computer systems. The method of design is a result of formulating the problem-solving organization in terms of the hypothesize-and-test paradigm for heuristic search, where the various hypothesizers and testers are represented as knowledge sources applicable to the task domain of the problem being solved. A knowledge source may be described as an agent that embodies the knowledge of a particular aspect of the problem domain and is useful in solving a problem from that domain by performing actions based on its knowledge so as to further the progress of the overall problem solution. The hypothesize-and-test paradigm provides the conceptual means of coordinating these various knowledge source activities by suggesting that it is the function of some knowledge sources to create hypotheses representing a possible (perhaps partial) solution state for the given problem. Hypotheses are created in the form of blackboard data base entries which are available for inspection by all knowledge sources. It is the responsibility of other knowledge sources to evaluate these hypotheses in light of their own knowledge of the task domain, and either accept or reject the hypotheses, or propose their own alternative hypotheses (by either modifying the existing hypotheses or creating entirely new ones). The HSII organization provides the facilities necessary for allowing the exchange of knowledge source action through the hypothesize-and-test paradigm to be carried out in a highly asynchronous and parallel manner, where knowledge sources are
specified as independent processing entities capable of asynchronous activity; the activities of any given collection of such knowledge sources are coordinated by the hypothesize-and-test paradigm through the use of a shared blackboard-like data exchange facility.

In specifying the blackboard as the primary means of interprocess communication, particular attention has been paid to resolving the data access synchronization problems and data integrity issues arising from the asynchronous data access patterns possible from the various independently executing parallel knowledge-source processes. A non-preemptive data access allocation scheme was devised in which the units of allocation could be linearly ordered and hence allocated according to that ordering so as to avoid data deadlocks. The particular units of data allocation were chosen as being blackboard nodes (hypotheses and links), these nodes also representing the units of data creation within the blackboard. Since the blackboard data base is a dynamically expanding structure, the mechanism of data access synchronization according to existing data objects is not sufficient to provide the access synchronization required when multiple knowledge sources are capable of simultaneously hypothesizing identical hypotheses into the blackboard without being required to link these hypotheses to previously existing data nodes. Assuming it is undesirable to have identical (duplicate) nodes in the blackboard (as is the case in the HSII organization, since one of the design goals was to minimize the duplication of information within the blackboard so as to minimize the duplication of processing which would result from such replicated information), a mechanism had to be provided whereby a knowledge source could acquire access to a section of the blackboard which did not yet exist. The region locking mechanism satisfies these requirements by viewing the potential blackboard as an abstract data space in which access rights to abstract regions could be granted without regard to the actual data content of these regions. However, since both the node accessing mechanism and the region accessing mechanism have the capability of allocating access rights to essentially the same data structure, the two forms of data access allocation must be closely coordinated so as to avoid data deadlocking and data access race conditions.

Another area of concern relating to the use of a shared blackboard-like data facility relates to the assumptions made by the various executing knowledge sources concerning issues of data integrity and localized data contexts. Since the blackboard is intended to represent only the most current global status of the problem solution state,
mechanisms were introduced to allow individual knowledge sources to retain recent histories of modifications made to the dynamic blackboard structure in the form of local contexts. Knowledge sources are also permitted to mark (tag) arbitrary fields (or nodes or regions) of the blackboard itself (without requiring continuing access rights to the field being tagged) so as to be able to monitor (in a non-interfering way) those locations for subsequent changes and be sent a message should any modification be performed upon a tagged field. Local contexts provide knowledge sources with the ability to create a local data state which reflects the net effects of data events which have occurred in the data base since the time of the knowledge source activation (actually, since the time of precondition activation, if one considers the operation of precondition testing to be separable from the action of the knowledge source, as HSII does). These historical local data bases are important to knowledge sources whose actions depend upon the effects of data events rather than just the occurrence of such events. Combined with the blackboard data tagging capabilities, local contexts also provide a means by which knowledge sources can execute quite independently of any other concurrently executing knowledge sources (and without interfering with the execution progress of any of these processes). When a knowledge source is about to modify the blackboard and has acquired exclusive access rights to the necessary data fields, it can request the receipt of any tagging messages that may have been sent to it; and by interrogating its local context for the effects of these changes, a revalidation check may be performed on the advisability of proceeding with the intended blackboard modifications.

In an attempt to improve the problem-solving efficiency of a multiprocessor implementation of the system by increasing the amount of potential parallelism from knowledge source activity, the logical functions of precondition evaluation and knowledge source execution are split into separate processing entities (called, of course, precondition and knowledge-source processes). A precondition process is responsible for monitoring and accumulating blackboard data events which might be of interest to the knowledge source associated with the precondition; and when the appropriate data conditions for the activation of the knowledge source exist in the blackboard, the precondition will instantiate a process based on its associated knowledge source, giving it the data context in which the precondition was satisfied. Two primary mechanisms are provided to support the asynchronous form of precondition and knowledge source interaction that results from allowing preconditions and knowledge-source processes to
execute concurrently. The first of these mechanisms relates to the way preconditions become activated; the second responds to the problems involved in having to schedule the many processes that may be capable of running so as to best serve the objectives of efficient problem-solving.

The process activity of HSI is intended to be very data-directed in nature, basing the decisions as to whether a knowledge source action can be performed on the dynamic data state represented in the blackboard data base. It is the responsibility of a precondition to test this data state for conditions which would warrant the instantiation of the knowledge source associated with the precondition. The activation of the precondition itself is also data-directed, being based on monitoring for the more primitive blackboard modification operations which knowledge-source processes may invoke to effect the results of their computation. This blackboard monitoring is implemented by having the various blackboard modification operators be responsible for the activation of preconditions which are monitoring for data events being caused by the modification operation.

While precondition activity might be requested as the result of a blackboard monitoring operation, and knowledge source activity might be requested as a result of precondition satisfaction, some care must be exercised in allocating processing power to these possible sources of activity. In particular, it is quite likely that there will be many more processes capable of executing or requesting computing resources than can be serviced within the constraints of a reasonable problem solution time. Even if there is not an excess of requested processing activity, system performance can often be improved by the use of a goal-directed scheduler whose responsibility it is to allocate processing resources so as to execute those processes first which can best promote the progress of the overall problem solution. The process evaluation functions used within the goal-directed scheduler are based on attention focusing parameters associated with the various components of the blackboard data base; policy knowledge-sources are used in calculating these attention focusing parameters based on the occurrence of various important blackboard data events, and such policy knowledge-sources are also responsible for propagating the effects of such events throughout the blackboard, so that proper attention is paid to these events and processing power may be allocated accordingly.

Simulation results were presented to indicate the nature of the performance
of the HSII organization when run in a closely-coupled multiprocessor environment. While the results are admittedly based on a small (but computationally expensive) set of sample points, generally they have favorably indicated the applicability of this system organization to such a hardware architecture. Given the knowledge-based decomposition of a problem-solving organization as prescribed by the HSII structure, effective parallelism factors of four to six were realized even with a relatively small set of precondition and knowledge-source processes, with indications that up to twelve processors could be totally utilized, given appropriate usage (or structuring) of the data access synchronization mechanisms. Measurements were also made of the various types of interprocess interference resulting from the use of a centralized shared data facility, as well as measuring the direct costs of providing mechanisms to support such a shared data facility. Notice that the individual operation costs (as for data access acquisition or blackboard field accessing) are certainly implementation dependent (and hence amenable to optimization). The interactions among the amount of process interference activity and the numbers of running and ready processes and the amount of processing power available were discussed; and various snapshots were presented to show the dynamic nature of the blackboard locking structure. While all these results are of a preliminary nature (and hence are subject to variation as various components of the given implementation are improved in their relative efficiencies), they seem to indicate that the HSII organization is indeed applicable for efficient use in a closely-coupled multiprocessor environment.

**Directions for Future Research and Concluding Remarks**

Certainly, there are many interesting areas of investigation remaining in understanding the behavioral characteristics of the HSII organization. The simulation results in this report describe the initial phases of this investigation; future reports will present additional details regarding HSII's performance characteristics as additional knowledge sources are developed and further experiments are performed. With respect to the multiprocessor simulation, the effects of varying any number of the many design parameters would produce further data which would be useful in determining the proper tuning of these parameters. Areas under current continuing investigation involve varying the locking strategies and the placement of the data base given various memory organizations, varying the knowledge-source set and the availability rate of input data, varying the goal-directed scheduling strategy to test various forms of
attention focusing, and varying the knowledge-source invocation strategies and scheduling strategies to account for issues associated with the blackboard data locality of the possible points of process activity. Experiments thus far have indicated that careful use of the locking structure is required in order to approach the optimal utilization of any given processor configuration (unless there exist so many ready processes that the number of suspended processes does not matter much, as is the case in configurations of four processors or less). Furthermore, the instantaneous availability of the complete input data is of only marginal use, unless low-level knowledge sources can be developed which can operate without locking inordinate amounts of this input level in the blackboard. An extended use of non-interfering tagging seems to be indicated, along with a reduction in the use of region-locking (perhaps substituting region-examining or node-locking wherever possible). Hopefully, the additional simulation experiments will be useful in understanding some of the multiprocessor issues which will arise as the implementation of the C.mmp HSII speech-understanding system progresses. Of course, parallel efforts are under way to measure and improve the actual problem-solving capability of the HSII speech-understanding system, which issues were not of direct concern in this report.

Perhaps it should be pointed out, in case it is not obvious by now, that this dissertation has concerned itself with trying to solve a broad range of problems that have arisen in the course of specifying an actual AI problem-solving system, rather than being satisfied with investigating only a few specific details of such an organization. As such, the results have been a long time in coming (so it seems to me, at least). Having had completed an initial design, many man-months went into the actual implementation of the HSII system for speech-understanding, including several periods of system re-design and re-implementation; and long hours were spent assisting users (knowledge-source writers) in their use of the underlying HSII operating system, not to mention the many weeks spent in program debugging. Indeed, much of the implementation effort went into providing adequate debugging aids and user interfaces within the HSII operating system, not only at the knowledge-source programming level, but also, and perhaps more importantly, at the level of conversational user interaction with the running system. In any case, the mechanisms provided by the HSII problem-solving system and described in this dissertation have withstood the test of time through actual system use in their capacity of supporting the multiprocess environment necessary for the speech-understanding system. While individual improvements are constantly being
made to the various pieces of the system structure, the underlying organizational framework has remained intact over time. Hopefully, this system organization for knowledge-based AI problem-solving systems, and the results arising from its use, will help in advancing the state of AI problem-solving by clarifying some of the issues involved in the use of multiprocessing problem-solving techniques.
SELECTED REFERENCES


Appendix A:  
HSII BLACKBOARD AND KS DECOMPOSITION

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conceptual</td>
<td></td>
</tr>
<tr>
<td>Phrasal</td>
<td></td>
</tr>
<tr>
<td>Lexical</td>
<td></td>
</tr>
<tr>
<td>Syllabic</td>
<td></td>
</tr>
<tr>
<td>Surface-phonemic</td>
<td></td>
</tr>
<tr>
<td>Phonetic</td>
<td></td>
</tr>
<tr>
<td>Segmental</td>
<td></td>
</tr>
<tr>
<td>Parametric</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1. The Levels in Hearsay II.

Figure 1 shows a schematic of the information levels of Hearsay II.

Parametric Level - The parametric level holds the most basic representation of the utterance that the system has; it is the only direct input to the machine about the acoustic signal. Several different sets of parameters are being used in Hearsay II interchangeably: 1/3-octave filter-band energies measured every 10 msec., LPC-derived vocal-tract parameters, and wide-band energies and zero-crossing counts.

Segmental Level - This level represents the utterance as labeled acoustic segments. Although the set of labels may be phonetic-like, the level is not intended to be phonetic -- the segmentation and labeling reflect acoustic manifestations and do not, for example, attempt to compensate for the context of the segments or attempt to combine acoustically dissimilar segments into (phonetic) units. As with all levels, any particular portion of the utterance may be represented by more than one competing hypothesis (i.e., multiple segmentations and labelings may coexist).

Phonetic Level - At this level, the utterance is represented by a phonetic description. This is a broad phonetic description in that the size (duration) of the units is on the order of the "size" of phonemes; it is a fine phonetic description to the extent that each element is labeled with a fairly detailed allophonic classification (e.g., "stressed, nasalized [l]").

Surface-Phonemic Level - This level, named by seemingly contradicting terms, represents the utterance by phoneme-like units, with the addition of modifiers such as stress and boundary (word, morpheme, syllable) markings.

Syllabic Level - The unit of representation here is the syllable.

Lexical Level - The unit of information at this level is the word.

Phrasal Level - Syntactic elements appear at this level. In fact, since a level may contain arbitrarily many "sub-levels" of elements using the AND and OR links, traditional kinds of syntactic trees can be directly represented here.

Conceptual Level - The units at this level are "concepts." As with the phrasal level, it may be appropriate to use the graph structure of the data base to indicate relationships among different concepts.
As examples of knowledge sources, Figure 2 shows the first set implemented for Hearsay II. The levels are indicated as horizontal lines in the figure and are labeled at the left. The knowledge sources are indicated by arcs connecting levels; the starting point(s) of an arc indicates the level(s) of major "input" for the knowledge source, and the end point indicates the "output" level where the knowledge source's major actions occur. In general, the action of most of these particular knowledge sources is to create links between hypotheses on its input level(s) and: 1) existing hypotheses on its output level, if appropriate ones are already there, or 2) hypotheses that it creates on its output level.
The *Segmenter-Classifier* knowledge source uses the description of the speech signal to produce a labeled acoustic segmentation. For any portion of the utterance, several possible alternative segmentations and labels may be produced.

The *Phone Synthesizer* uses labeled acoustic segments to generate elements at the phonetic level. This procedure is sometimes a fairly direct renaming of an hypothesis at the segmental level, perhaps using the context of adjacent segments. In other cases, phone synthesis requires the combining of several segments (e.g., the generation of [t] from a segment of silence followed by a segment of aspiration) or the insertion of phones not indicated directly by the segmentation (e.g., hypothesizing the existence of an [h] if a vowel seems velarized and there is no [h] in the neighborhood). This knowledge source is triggered whenever a new hypothesis is created at the segmental level.

The *Word Candidate Generator* uses phonetic information (primarily just at stressed locations and other areas of high phonetic reliability) to generate word hypotheses. This is accomplished in a two-stage process, with a stop at the syllabic level, from which lexical retrieval is more effective.

The *Semantic Word Hypothesizer* uses semantic and pragmatic information about the task (e.g., news retrieval or chess) to predict words at the lexical level.

The *Syntactic Word Hypothesizer* uses knowledge at the phrasal level to predict possible new words at the lexical level which are adjacent (left or right) to words previously generated at the lexical level. This knowledge source is activated at the beginning of an utterance recognition attempt and, subsequently, whenever a new word is created at the lexical level.

The *Phoneme Hypothesizer* knowledge source is activated whenever a word hypothesis is created (at the lexical level) which is not yet supported by hypotheses at the surface-phonemic level. Its action is to create one or more sequences at the surface-phonemic level which represent alternative pronunciations of the word. (These pronunciations are currently pre-specified as entries in a dictionary.)

The *Phonological Rule Applier* rewrites sequences at the surface-phonemic level. This knowledge source is used: a) to augment the dictionary lookup of the Phoneme Hypothesizer, and b) to handle word boundary conditions that can be predicted by rule.

The *Phone-Phoneme Synchronizer* is triggered whenever an hypothesis is created at either the phonetic or the surface-phonemic level. This knowledge source attempts to link up the new hypothesis with hypotheses at the other level. This linking may be many-to-one in either direction.

The *Syntactic Parser* uses a syntactic definition of the input language to determine if a complete sentence may be assembled from words at the lexical level.
The primary duties of the Segment-Phone Synchronizer and the Parameter-Segment Synchronizer are similar: to recover from mistakes made by the (bottom-up) actions of the Phone Synthesizer and Segmenter-Classifier, respectively, by allowing feedback from the higher to the lower level.

In addition to the knowledge source modules described above, all of which embody speech knowledge, several policy modules exist. These modules, which interface to the system in a manner identical to the speech modules, execute policy decisions, e.g., propagation of ratings and calculation of processing-state attributes.
Appendix B:
EXAMPLE OF A BLACKBOARD FRAGMENT IN HEARSAY II

Figure 3. An Example of a Fragment in the Blackboard.

Figure 3 is an example of a fragment that might occur in Hearsay II's blackboard. The level of an hypothesis is indicated by its vertical position; the names of the levels are given on the left. Time location is approximately indicated by horizontal placement, but duration is only very roughly indicated (e.g., the boxes surrounding the two hypotheses at the phrasal level should be much wider). Alternatives are indicated by proximity; for example, 'will' and 'would' are word hypotheses covering the same time span. Likewise, 'question' and 'modal-question', 'you1' and 'you2', and 'J' and 'Y' all represent pairs of alternatives.
This example illustrates several features of the data structure:

The hypothesis 'you1' at the lexical level, has two alternative phonemic "spellings" indicated; the hypotheses labeled 'you1' and 'you2' are nodes created, also at the lexical level, to hold those alternatives. In general, such sub-levels may be created arbitrarily.

The link between 'you1' and 'D' is a special kind of SEQUENCE link (indicated here by a dashed line) called a CONTEXT link; a CONTEXT link indicates that the lower hypothesis supports the upper one and is contiguous to its brother links, but it is not "part of" the upper hypothesis in the sense that it is not within the time interval of the upper hypothesis -- rather, it supplies a context for its brother(s). In this case, one may "read" the structure as stating "'you1' is composed of 'J' followed by 'AX' (schwa) in the context of the preceding 'D':" (This reflects the phonological rule that "would you" is often spoken as "would-ja.") Thus, a CONTEXT link allows important contextual relationships to be represented without violating the implicit time assumptions about SEQUENCE nodes.

Whereas the phonemic spelling of the word "you" held by 'you1' includes a contextual constraint, the 'you2' option does not have this constraint. However, 'you1' and 'you2' are such similar hypotheses that there is strong reason for wanting to retain them as alternative options under 'you' (as indicated in Figure 3), rather than representing them unconnectedly. A connection matrix is used here to represent this kind of relationship; the connection matrix of 'you' (symbolized in Figure 3 by the 2-dimensional binary matrix in the node) specifies that support 'you1' is relevant to use 'question' (but not to 'modal-question') and that support 'you2' is relevant to both uses.

The nature of the implications represented by the links provides a uniform basis for propagating changes made in one part of the data structure to other relevant parts without necessarily requiring the intervention of particular knowledge sources at each step. Considering the example of Figure 3, assume that the validity of the hypothesis labeled 'J' is modified by some knowledge source (presumably operating at the phonetic level) and becomes very low. One possible scenario for rippling this change through the data base is given here:

First, the estimated validity of 'you1' is reduced, because 'J' is a lower hypothesis of 'you1'.
This, in turn, may cause the rating of 'you' to be reduced.
The connection matrix at 'you' specifies that 'you1' is not relevant to 'modal-question', so the latter hypothesis is not affected by the change in rating of the former. Notice that the existence of the connection matrix allows this decision to be made locally in the data structure, without having to search back down to the 'D' and 'J'.
'Question,' however, is supported by 'you1' (through the connection matrix at 'you'), so its rating is affected.
Further propagations can continue to occur, perhaps down the other SEQUENCE links under 'question' and 'you'.

Notice that all of these modifications are "speech-knowledge independent" and can be accomplished uniformly at all levels of the blackboard by a single policy knowledge source. This policy knowledge source does not need to access or trigger any other knowledge source but can directly derive all the information it needs from the hypothesis and link fields that are uniformly present and from the implicit semantics of the structures in the blackboard.