Module Linkage and Communications in Large Systems*

Jeffrey A. Barnett†
System Development Corporation
Santa Monica, California 90202

Abstract

This paper was originally intended to address only the problems associated with implementing Artificial Intelligence programs. However, an examination of the trends in Al shows that project teams are growing in size and becoming multidisciplinary, as they are in other program development projects. Even though the object of Al research may be the discovery of an appropriate system organization, the presence of a large implementation staff strongly suggests the use of highly modular techniques, the minimization of the number of dependencies among system components, and the development of techniques for ensuring that the remaining dependencies are visible.

This paper advocates the use of a formal, executable language to specify and make visible the dependencies among the modules of a system. Such a language has been developed to organize the control monitor for a speech understanding system. In the course of developing this language, which is called Control Structure Language (CSL), several techniques that aid in organizing systems were explored: *Parnas* modules for representing permanent and semi-permanent data, software bus structures for dynamic data communication among modules, and non-guaranteed parallelism. These techniques are reviewed, and the properties and the use of the language are discussed.

*Originally appeared in *Speech Recognition: Invited Papers Presented at the 1974 IEEE Symposium on Speech Recognition*, D. Raj Reddy, editor, Academic Press, 1975, pp. 500–520, ISBN 0–12–584550–2. A second appendix is added to the original paper.

†Email: jbb@notatt.com

Trends in AI Systems

The field of Artificial Intelligence encompasses a wide variety of activities: natural language processing, game playing, speech understanding, theorem proving, visual scene evaluation, symbolic integration, etc. These activities share the characteristic that they are normally considered to require some thought or intelligence. However, determining whether a particular activity is truly AI is not easy; in fact, an old bromide says that if a particular activity has been well modeled by a computer or if the solution space is well understood, then the activity is no longer considered a part of AI.

Many early AI efforts were conducted by individual investigators or small groups concentrating on very specific problems such as game playing. The major tools developed were heuristic searching and tree-pruning algorithms. Also, many of the problematic issues were highly computer intensive; for example, some of the key problems with constructing a chess-playing machine were such things as how the board is stored internally or how a trial move sequence or strategy is represented. Such issues led to the development of many programming languages and systems in which it was easy to represent a variety of structures: IPL, LISP, COMIT, etc.

Recently, research and development activities undertaken by the AI community have changed. Perhaps the most important difference is in the mode of operation. Large groups and organizations are working on joint projects. The Speech Understanding Research (SUR) project, organized and sponsored by ARPA, is becoming a model (see [9]). Also, the kind of problem drawing attention is more general. The goal is no longer to automate a class B chess player, but to construct a speech understanding system or a functioning robot. In these cases, the initial objective for the system is not final; the ultimate objective may be defined as a system with large bases of diverse knowledge, having learning (adaptive) capabilities rivaling a human's. Along with this open-endedness of goal is a change in primary task to modeling of human functions: speech, vision, motion, communication (language), general problem solving, and combinations of these. In many of these areas, because so much is known outside the computer and AI community, research teams have become more interdisciplinary, and techniques other than heuristic search have been brought to bear. In speech and vision systems, for example, the body of results from signal processing are being incorporated. Work in computational linguistics is drawing on machine-readable versions of Webster's 7th Collegiate Dictionary and Roget's Thesaurus, etc. These

developments, taken together, suggest a trend in AI systems work toward highly interdisciplinary endeavors that involve many people cooperating on the same project.

Communications in AI Systems

A system that models human intelligence can be said to comprise a set of Knowledge Sources (KSs). In a computer, a KS is represented as a combination of data and procedure. Further, the allowable communication paths to and from a KS must be adequately defined to allow an implementation. For example, a KS in a chess-playing program may be a module that maintains the board position; the board itself is represented as some data structure. The communications with this module are the control actions necessary to interrogate the present position, update the present position, and save (restore) the present position.

In general, the set of allowable communications with a KS will be more complicated than this and should also be more precise. (For instance, does this module legality check proposed position updates? Are positions named by squares or pieces? etc.) In normal system building, a lack of precision is unforgivable; in an Al system, however, it may be unavoidable because the discovery of an adequate system organization, and hence communication network, may be the whole essence of the research. Such questions as "what should be the interaction between syntax and pragmatics?" or "what is the contribution of a position evaluator to the total system?" become critical. In fact, the problem of necessary and sufficient communications, and the problems of instrumentation and measurement of the contributions of each KS to total system performance, are becoming the main issues in Al. If one can accurately specify answers to the above types of questions, then the activity is no longer considered Al. (New bromide.)

What is a System?

A system (or program) is a collection of procedure and data specifications, called modules, written in some formal language(s) plus a control structure for performing an interpretation. In [1], Fisher defines a control structure as follows:

> By control structure we mean programming environments or operations which specify the sequencing and interpretation rules for programs and parts of programs.

Fisher goes on to say that any control structure can be described by the use and amalgamation of six types of control primitives:

> 1) There must be means to specify a necessary chronological ordering among processes and 2) a means to specify that processes can be processed concurrently. There must be 3) a conditional for selecting alternatives, 4) a means to monitor (i.e. nonbusy wait) for given conditions, 5) a means for making a process indivisible relative to other processes, and 6) a means for making the execution of a process continuous relative to other processes.

The terms module, process, and data are used to describe systems. A module is a collection of forms written in some programming language and is therefore a lexical representation. Included in the collection may be constant data and specifications for storing dynamic data structures that are either computed or obtained from the external environment. A module may comprise, besides its data storage, a set of entry points in the traditional sense of functions, subroutines, coroutines, etc. Access may also be provided through a macro facility. Thus, procedural knowledge such as algorithms and heuristics is embedded into a system as modules.

The term process, on the other hand, refers to a thing in execution; or, stated differently, a program counter operating an interpretation upon a module with an associated state is a process. Thus, if our computer has more than one CPU (program counter), we may have parallel processes. Note that, if more than one program counter is simultaneously acting on the same module, then each program counter is associated with a separate process. Many systems, for example time sharing and multiprogramming systems, partially achieve the effect of multiple program counters by time slicing a single processor.

(The distinction between data and procedure in a system is not always sharp. It may depend upon which level of system description is viewed. For example, consider the structure of two different recognizers for the same grammar: One system represents each grammar equation as a tree (a data structure) that is operated upon by a generalized interpreter. In the other, each grammar equation is represented as a procedure. In detail, there is

a definite distinction between these two recognizers: the difference between data and procedures But viewed as processes in execution, the two are not distinguishable. And so it is for most procedures; the module may at some level of interpretation be viewed as data. However this observation should not inhibit us from noting that there are many instances in which data is not conveniently or adequately described as procedure. Examples of such data would be the parameter values passed to an ordinary subroutine, the results of a computation not conveniently described by a table lookup, or dynamically derived control parameters such as the value of a program counter. It may be concluded that, even though the distinction between procedure and data could almost be erased, it is desirable at some levels of system interpretation.)

Dependencies Among System Components

A system is built from two kinds of components, data and modules. We do not distinguish *private* data from the module that owns it. Therefore, only data (or data paths) visible to more than one module are considered as system components. A dependency is defined as an order relationship in time. To say that y depends on x is to say that if the state of x reaches one of a specified set of configurations, then there will be some effect on y. There are four general categories of dependencies: (1) process p_2 depends on process p_1, (2) data structure d depends on process p, (3) process p depends on data structures d, and (4) data structure d_2 depends on data structure d_1.

Dependency category (1), process p_2 depends on process p_1, is the best-understood category of system dependency. The most important examples of this relationship are

- p_1 decides whether p_2 should execute,
- p_2 needs results (data) produced by p_1,
- p_3 needs result computed by p_1 before result computed by p_2,
- p_1 is more *urgent* than p_2,
- both p_1 and p_2 need a common resource whose access methods cannot tolerate simultaneous use.

In the last case, it may not matter whether p_1 or p_2 operates first, only that some ordering be imposed.

Dependency category (2), data structure d depends on process p, normally occurs when p calculates an update for d. Category (3), process p depends on data structure d, has the popular names *demon* and *monitor*. In many ways it resembles an interrupt; that is, if d achieves some unusual shape or value, then a process is initiated.

Dependency category (4), data structure d_2 depends on data structures, d_1, is interesting because it describes a time relationship between data without an explicit, intervening calculation. There are two slightly different cases of category (4) dependencies. First, the same information is kept in multiple copies; thus, if the information is updated at one location, then it may need to be updated at its other locations. Second, some information may be stored as a common substructure of several data structures, and any update to the common portion will be simultaneously reflected in all the parent structures.

Management of Dependency

In the above discussions, we have talked about dependencies involving processes and dependencies involving modules without making a distinction. Strictly speaking, we should have mentioned only dependencies involving processes, because the ordering was on execution, a property of processes. However, this misses an important problem with system construction. It is that people code modules not processes. And, if a group is working on a program together, they will make assumptions about and use each others' work. In [2], Parnas says,

> The connections between modules are the assumptions which the modules make about each other.

These assumptions may cover many types of dependencies not included in the four categories outlined above. Some examples are

- Subroutine s in module m performs a specific task.
- Data structure d has a certain format.
- The name of an entity is n.

- Subroutine s may be used recursively.
- Module m has an unfixed bug. (!)

There are two opposing viewpoints on the desirable amount of inter-module dependencies (or connections). The first, attributed to Minsky is called *verticality* and has special significance to AI systems. It assumes that to model a complex process on a computer, it is necessary to have as much flexibility as possible in interconnecting sources of Knowledge (KSs) at whatever level of detail is necessary. (This kind of organization is sometimes called a heterarchy.) The other viewpoint states that a system with too many interconnections is hard to change and impossible to debug. Therefore, it is important to compartmentalize knowledge in a system, thus minimizing interdependencies. This approach is often called *modularity.* The argument for verticality is based upon modeling of complex processes, while the argument for modularity is based upon practical considerations of system development.

In recent years, several programming language systems have been created to better support vertically integrated systems: PLANNER, MICRO PLANNER, Q-LISP, and CONNIVER are a few. So there are some definite feelings that verticality is important. However, most reported uses of the above systems or concepts have involved a single investigator or a small implementation team. (See [3] and [4] for some examples.) The advantages of verticality are that knowledge is not hidden and that real-world activities can be modeled within a natural structure. The problem is that the degree of dependency is large, and therefore the resulting system is hard to instrument or change. Further, the control structure may be quite complex, making it very difficult to follow program flow.

There are several arguments for modularity: the initial design task is relatively straightforward; it is easy to change the details of implementation; the control structure, on the level at which the system is modularized, can be fairly simple; and, as a side effect of the latter, it should be possible to document the system easily. The disadvantages of the modular approach are that the system architecture may be or may become rigid and therefore mask essential knowledge and communication. This happens when too much attention is given to the modularization and not enough to the problem at hand. Also, subcomponents may be duplicated.

The choice between the use of a modular or vertical strategy can be most difficult. The difficulty is that the virtues of each are desirable and complementary. Verticality is a more flexible structure, and modularity is a

better organization for working. However, if a system is to be implemented by a large group of people, the modular approach is a clear winner. The ability to divide the work effort and specify the available interfaces is essential when the efforts of more than two or three people are involved. With the trend in AI systems towards larger groups working on the same project, we will see growing use of modularity. Verticality will be reserved for use by individuals or small groups searching for appropriate representation strategies, and it will probably be their efforts that point out some new directions for the development of larger systems.

In [2] and [5], Parnas suggests criteria for decomposing systems into modules. The basic idea is that a module should comprise the total knowledge of the system about some closely related set of design decisions. In the first section of this paper, a module for maintaining the status of a chess game was briefly mentioned. The external interfaces were functional, but using them made no assumptions about the actual representation of the board. Thus, the design decision hidden by this module was the selection of the board's representation. The merit of the Parnas criterion is that it reduces the amount of global knowledge necessary to program parts of a system because communications, and hence interdependencies, are well defined.

The idea of hiding a design decision is useful when applied to data but not so useful when applied to a procedural module. If the hidden design decision is the module's algorithm, then we have not really hidden very much. (How often can the user of a fast Fourier transformation routine make use of the fact that the computation is radix 4 instead of radix 2?)

Maintenance of data, on the other hand, is the prime candidate for the use of the Parnas approach. Examples of some good decisions to hide away are

- Is data kept in core or on disk?
- Is data tabular or computed?
- What is the format of the data?
- Is structure shared?
- Is the representation unique?

If communication with data is through functionals, the above kinds of design decisions do not have to become public, and we avoid global trouble when one of the decisions is changed.

The kinds of functionals necessary to communicate with a data store are update functionals to change, add, and delete items, and interrogation functionals to retrieve items or relationships about the data. These functionals can be implemented as functions, as macros, or even parametrically, as long as their exact inner workings do not need to be known. Two additional functionals should be provided for completeness: an initializer and a closer, both of which may perform no operation for many data stores. Since the possibility of storing data outside main memory is allowed, and since many operating systems require open and close catalogue calls for each file used, the initializer and closer can be used for these purposes. Even for data kept in main memory, the initializer can be used to lay out storage or compute initial values. The programming system in which the data is embedded calls the initializers for all stores at the start of program execution and the closers at the end.

Features of Data

In this section, some features of data will be examined. In particular, we will look at the usage, longevity, ownership, and visibility of data. (The structuring and formatting problems covered by Standish in [6] are not discussed.) The particular features selected for examination were chosen because they may have a major impact on the organization and modularization of a system.

In a system, a collection of data is usually known by name. Many times, the name of the collection is the same as the name of the module that provides the access paths to the members of the collection. We will assume that knowing the name of a collection is equivalent to begin able to access members of the collection. A distinction is to be made between a data name (or collection) and a structure. A structure is a single, static entity. A name may refer to different sets of structures at different times, e.g., the name, `PRESENT.WEATHER.CONDITIONS`, will refer to different weather conditions at different times. Thus, the access function, `TEMPERATURE`, does not refer to a particular datum, but is more correctly a data path into the collection.

Data structures and collections are used for many purposes. A partial list of usage categories is

- control information,

- argument passing,
- result passing,
- accumulation of state of computation,
- representation of procedure,
- information pooling (data bases).

Data longevity refers to the amount of time a collection holds data structures. Thus, longevity is a property of a collection, not a structure. There are three useful categories of longevity: dynamic, permanent, and semi-permanent. Examples of dynamic data are control information and argument and result passing. In most system organizations, dynamic data does not receive a name (except through bound variables local to a particular module). It is created and passed to its ultimate destination then it disappears from the system. Permanent data usually appears in the form of lexicons or data bases that are not ordinarily updated by operating the system. Semi-permanent data collections have the property that updates may occur, but entrance of a now member does not automatically delete old members. An example of a semi-permanent data collection in a speech system would be a memory of each word that has been successfully recognized in an utterance. As new words are found, they are added to the collection and remain there for the duration of the processing of the utterance. Before work is started on the next utterance, the collection is cleared.

Data ownership and data visibility are discussed together. To say that data collection d is visible to system component c (c knows d's name), can mean either that d depends on c or that c depends on d. If we use the modularity approach for maintaining data in a system, then the owner of the data is the module that allocates storage and provides the access paths. We can now say that a collection is private (or local) if it is visible only to its owner, and a collection is public (or global) if it is visible to at least one module that is not its owner. Being public does not necessarily grant access privileges to all system components. In fact, it is a basic part of the modularity philosophy that the system design should severely restrict the visibility (hence dependencies) of any components to those that have a *need to know*. As defined here, ownership and visibility are properties of modules.

In general, permanent and semi-permanent data are global and are appropriately implemented as modules. A problem arises when one attempts

to define ownership and visibility of dynamic data, because it is not natural to implement it as a module. (For example, the control and state parameters of a process belong to the process, not to a particular module.) Moreover, the unit of interest is a data structure, not a collection. (This is the case for subroutine arguments and values.) Further, because knowing the name of data is equated with its visibility, we can conclude that most dynamic data is invisible in the system, and private to its sender and receiver (or its single user).

Dynamic Data Visibility

There are times when it is appropriate to make dynamic data globally visible. A particular case is broadcasting information and requests among Knowledge Sources (KSs). Above, an example was given of a semi-permanent data collection in a speech system that remembers recognized words in an utterance. As various modules locate words in the utterance, they ship the words along the `LOCATED.WORD` data path to the storage module. In normal use, the KSs could interrogate the store to see if a particular word has already been located. It is also appropriate to broadcast the information placed on the `LOCATED.WORD` data path to various KSs. For instance, this would be a reasonable way to wake up (or create) a bottom-up parsing process. As another example, a top-down parsing process could broadcast a request for a particular word without consideration of who should respond. If the word had not been previously located, a word recognizer would examine the acoustic data and report results along the `LOCATED.WORD` data path.

The broadcasting technique described above closely resembles an important control structure capability provided by some program language systems for implementing vertically integrated systems. In particular, request broadcasting is the major linkage method, e.g., goal specification in PLANNER. The request receiver (as opposed to the request) is named. Thus, the catalogue of data paths is the set of routine names. Changes to the system at almost any level of detail change this catalogue. Such changes represent a major type of design decision. Obviously, this should not be allowed to happen on a day-to-day basis when the implementation staff is large.

An alternative can be used to preserve some of the flexibility inherent in the broadcast invocation strategy: name a few major data paths; for each path, specify the KSs that should operate whenever an item traverses

that path; augment the control to create a process for each specified KS whenever the path is used. There are two important differences between this and the goal-invocation approach. First, the data paths as well as the receiving KSs are named, making them global. Second, a *process depends on a process* dependency has become a *process depends on data* dependency. If the path name is made global, hence visible, the inter-KS communication can become a specified and integral part of the system design and a candidate for instrumentation and measurement studies. Also, the *process depends on data* dependency seems more natural.

An analogue to this strategy for making dynamic data visible is implemented in computer hardware as buses. A bus is a data path to which modules are connected. It is usually the case that the modules do not have to know each other's names. Requests and information are broadcast on the bus. Each module examines the data and can do something with it or disregard it. Therefore, we may call the above system technique for broadcasting dynamic data *software busing*.

Parallelism

Intuitively, parallelism means that two or more processes are operating simultaneously. Figure 1 shows a single process splitting into two parallel processes, *a* and *b*, which then rejoin to make a single process. Process *a* comprises the four sequential steps `a1`, `fa=a1`, `wait until fb eq 1`, and `a2`. Process *b* comprises the four sequential steps `b1`, `fb=1`, `wait until fa eq 1`, and `b2`. The numbered steps are assumed to be total; that is, `a1`, `a2`, `b1`, and `b2` are all finite calculations for any set of external conditions. Clearly, the processing scheme shown in Figure 1 will not itself be total unless there is some guarantee of parallelism. If a sequential interpreter attempts to operate one process to completion (run *a* to end, then *b*, or vice versa), Then a block will occur on the wait operation because the other process has not been able to unlock the control flag, either `fa` or `fb`.

The facade of guaranteed parallelism may be achieved by a variety of techniques. If multiple program counters exist, they may be organized in a network or as a multiprocessor where each CPU shares a memory and other resources. The method of performing parallel processing on a sequential computer with a single program counter is called interleaving. The concept of parallelism in such an arrangement is guaranteed at only some levels of

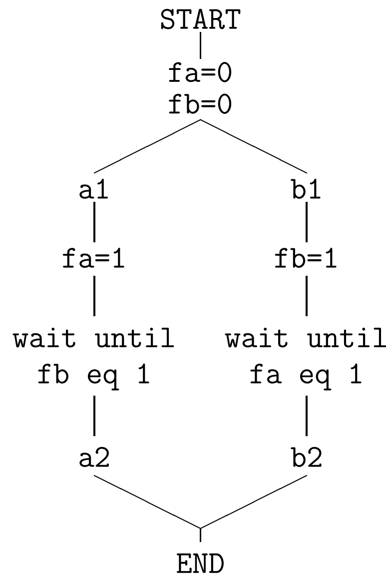

Figure 1: Example Flow

interpretation. Thus, in Figure 1, a single processor could switch back and forth from process *a* to *b* after completing one of the four steps in either process. In fact, process switching is necessary only when an unsatisfied wait operation is encountered. Thus, at the level shown in Figure 1, we must guarantee parallelism. However, there need not be a guarantee that `al` and `bl` will be operated simultaneously. The control structure needs to guarantee only the facade of parallelism, and then only at particular levels of interpretation. The levels are only those in which synchronization operations (such as the `wait`) are present.

Two issues raise the question of whether parallel processing is worth thinking about in system design. First, almost no one has actual equipment available that is capable of doing real multiprocessing. Second, even if the equipment were available, the use of a parallel control structure introduces extra dependencies into a system in the form of synchronization operations. These extra dependencies are of an implementation variety and may have little to do with the problem-level intercommunications of the KSs. Since our general conclusion is that the number of dependencies (communications paths) in a system should be kept to a reasonable minimum, it may be inferred that parallelism is inappropriate in systems developed by large groups.

Non-guaranteed parallelism is a technique that has advantages in a control structure and does not run into the above objections. As the name implies, non-guaranteed parallelism does not promise to operate active processes simultaneously. The only guarantee is that, when an unsatisfied wait is encountered, some other process(es) not at an as yet unsatisfied wait will continue, start, or resume processing. The only legal inter-process order dependencies are those that occur in conjunction with explicit synchronization operations. Thus, guaranteed parallelism in the control regime of one level of system description cannot be hereditary to lower levels unless these levels also have synchronization primitives. When guaranteed parallelism is confined to higher levels of description, many inter-dependencies are "bubbled" outward and become visible or disappear.

The advantages of this approach are numerous. The number of explicit dependencies is reduced and the implicit, undocumentable dependencies are forced to disappear or become explicit at a higher level. Unexpected, implicit dependencies will eventually show themselves as software "race" conditions when parallelism is used in a system. The manifestation is usually that the same initial state produces different results. Even though this is not desired, it has a good effect. It forces the implementers and designers of a system to look for a cause, and this may lead to the discovery of a necessary but overlooked relationship among the KSs. If parallelism, of some sort had not been used, the implementers would have selected some arbitrary sequential ordering and the problem may or may not have appeared. If it had not, then a discovery experience would have been missed. If the problem had occurred, it would have been solved by switching the order of a few statements rather than by analyzing the problem in sufficient depth. Thus, the use of parallelism is a motivator to understanding a system. Non-guaranteed parallelism preserves this characteristic and has the advantage of reducing the number of implicit and explicit dependencies. It allows efficient simulation on single CPU computers. (The use of a randomizing technique for selecting among the eligible execution paths ensures that implicit dependencies, if present, have an opportunity to appear.)

Levels of System Description

Throughout this paper, reference has been made to levels of system description, levels of representation, and levels of interpretation. The relationship

between these three concepts and the concepts themselves are vague because they have many different intuitive connotations. My personal preference defines level of description in terms of representation and interpretation as follows:

> A level of system description is a representation of the system which allows an accurate simulation of the system to some level of detail. The simulator or interpreter for a level of description is the realization of the control structure for the representation.

Thus, a level of description implies that the system can be simulated, which in turn implies the existence of a language for the representation and a control structure for that language.

The concept of level implies a hierarchy. For something as complex as a large program, there exist a variety of choices for the set of description levels. Some choices are more natural than others because they arise as steps in the evolution of the system design. Normally, the first level of system description is a diagram of the major components and dependencies. The control structure for interpreting the diagram is usually prose and informal.

The next iteration of the system design identifies the KSs that the system comprises. Data and process dependencies are also identified. At this level, a KS receives a name that reflects a problem-level label such as pragmatics, parser, eye, etc. Following this, the most crucial system description is attempted: the definition of modules. The importance is obvious because each defined module becomes a work assignment for one or perhaps two people. It is also at this level that all global dependencies must be defined and external module specifications created.

The bottom-most set of levels comprise program language forms organized into layers of subroutines and functions. These are the only levels for which there generally exist a formal representation and a control structure. The KS and module level are not formalized. Probably the best representation of the KSs and their interdependencies is a diagram with adequate explanation. However, this lack of accurate representation should not exist at the module level. In [2], Parnas argues strongly that all global dependency decisions be documented at this level and form the basis for individual work assignments. However, this is not enough. For a variety of reasons, the module level should be described in a formal, executable language that acts as the system control monitor. The most important reasons for formalizing the module level are the enhancements of the abilities to instrument and measure the system's

performance, debug the system, change the system in a way that affects the least number of people, and test the system for compliance with the design specifications. All of these benefits accrue because a control monitor can create and order executable processes and can supervise the global data paths. It is at this level that the system description can best be augmented with test facilities.

A Module-Level Description Language

Ritea in [7] and Barnett in [3] describe a speech understanding system implemented at System Development Corporation in Santa Monica, California. The system accepts spoken utterances in a formal data management language that has a vocabulary of approximately 150 words. Users' questions are interactively answered, reports may be formatted, and the user may request help in understanding the system or the data base.

The system operates on an IBM 370/145 computer with the acoustic preprocessor operating on a Raytheon 704 computer. The organization of the system at the module level was accomplished using a control monitor language. The language is named Control Structure Language, CSL. CSL facilitated the use of the techniques mentioned above such as software buses, parallelism (guaranteed in CSL, non-guaranteed in the modules), visible declaration of dependencies, and "Parnas" modules for data storage. Three kinds of system components may be declared: buses, common (global) storage modules, and processing modules. The functional data paths to a common storage module are also specified and include: initializer, closer, updater, and interrogators.

A processing module has a single entry point, receives no arguments, and returns no value. All external communication must be over the buses or through the common storage modules. A processing module declaration includes the names of the buses and common storage modules it reads, and the names of the buses and common storage modules it writes. This is sufficient information to construct a data-dependency graph of the system. Alias names may also be declared for processing modules to allow a variety of usages in the system.

Besides passive declaration of major components, CSL provides for a description of program flow and order dependencies. The set of invocation primitives are; conditional, sequential, parallel, and fork. `FORK L` creates

a process starting each place the label `L`, appears. Additional flow synchronization is accomplished using `AFTER` forms, `BEFORE` forms, and labels. `AFTER` waits for a specified set of processing modules to complete execution. `BEFORE` causes a specified sequence of preceding events to happen each time a specified module is activated or before a fork to a specified label is completed. `BEFORE` is normally used for debugging and instrumentation. Labels may be placed singly or in groups. Processing does not start at a label until a fork to each label in the group has occurred.

Two data-synchronizing operations are present in CSL. First, `CYCLE` clears visible data structures from the specified buses and places new items, accumulated by operating processing modules, on the buses. Second, `UPDATE` passes the data structures accumulated since the last update for this common storage module to the update functional given in the common declaration.

Debugging facilities include trace and breakpoint capabilities. Also, a system monitor written in CSL may bind variables and operate expressions written in LISP. CSL was implemented as a language extension of the SDC LISP Infix Language and operates in three phases: compile, graph linking, and run-time control monitor execution. The appendix to this paper contains a brief description of CSL.

Experience with the use of CSL for organizing a speech system has confirmed the desirability of having a formal description level for modules and their interdependencies. Several experiments have been performed where one or more modules were deleted from the system and performance measured to determine their contribution. This was accomplished without disturbing the interior of any module; only the CSL system description was modified. It has been possible to debug each module in isolation by using CSL to configure a test facility specifically suited to testing that module. Another benefit has been the ability to borrow modules from the system in order to manufacture other programs quickly. Also, as the system has been used, the dependencies and flow have been altered to improve performance. In no case was a module reprogrammed; the changes were reflected only at the CSL description level.

Acknowledgements

I wish to thank Douglas Pintar who worked on the implementation of CSL and who made many valuable contributions to the design of the language. I also wish to thank John Luke for his help in organizing and improving this

paper.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract Number DAHC15–73–C–0080.

References

[1] Fisher, D. A., *Control Structures for Programming Languages*, Doctoral Dissertation, Carnegie-Mellon University, May, 1970.

[2] Parnas, D. L., Information Distribution Aspects of Design Methodology, *Information Processing* 71, North-Holland Publishing Company, 1972, pp. 339–344.

[3] Miller, P. L., *A Locally-Organized Parser for Spoken Input*, Doctoral Dissertation, Massachusetts Institute of Technology, March, 1973.

[4] Winograd, T., *Procedures as a Representation for Data in a Computer Program for Understanding Natural Languages*, Doctoral Dissertation, Massachusetts Institute of Technology, February, 1971.

[5] Parnas, D. L., On the Criteria to be Used in Decomposing Systems into Modules, *Communications of the ACM*, December, 1972, Vol. 15, No. 12, pp. 1053–1058.

[6] Standish, T. A., *A Data Definition Facility for Programming Languages*, Doctoral Dissertation, Carnegie-Mellon University, May, 1967.

[7] Ritea, H. B., A Voice-Controlled Data Management System, *Contributed Papers of IEEE Symposium on Speech Recognition*, April, 1974, Carnegie-Mellon University, Pittsburgh, Pa., pp. 28–32.

[8] Barnett, J. A., A Vocal Data Management System, IEEE Transactions on Audio and Electroaccustics, Volume AU–21, Number 3, June, 1973, pp. 185–188.

[9] Newell, A., et al., *Speech Understanding Systems: Final Report of a Study Group*, Published for Artificial Intelligence by North-Holland / American Elsevier, 1973.

APPENDIX: CSL Language Description

By example, several kinds of CSL statements and language forms are introduced. Declaration statements are presented first, followed by the flow and synchronization primitives.

Declaration Statements

Buses, common storage modules, and processing modules can be declared in CSL.

Bus Declarations

Bus declarations are introduced by the word, `BUS`, followed by a sequence of bus names. To declare `B1`, `B2`, and `B3` buses, write:

```
BUS B1,B2,B3;
```

Common Storage Module Declarations

Common storage module declarations are introduced by the word, `COMMON`, followed by a declaration for each common storage module:

```
COMMON X(XI,XU,XC,XA1,XA2),
       Y(YI,YU,UC,UA1,YA2,YA3);
```

In this example, `X` and `Y` are declared as common storage modules. `XI`, `XU`, and `XC` are the names of the initalizer, updater, and closer functions, respectively, for `X`. `XA1` and `XA2` are the name of the interrogators. `YI`, `YU`, and `YC` are the names of the initalizer, updater, and closer functions, respectively, for `Y`. `YA`, `YA2`, and `YA3` are the interrogators. All access functions are specified by function or macro names.

Processing Module Declarations

Processing module declarations are introduced by the word, `MODULE`, followed optionally by the specification of alias names, data input-names, and data output names:

```
MODULE M ALIAS(MI,M2) READ(B1)(X,Y) WRITE(B2,B3)(X);
```

The module, `M`, may also be referenced as `M1` or `M2`. `M` reads from bus `B1` and writes on buses `B2` and `B3`. `M` reads and writes common storage module `X` and reads common storage `Y`.

Flow and Synchronization Forms

Flow and synchronization control forms are built up from the names of buses, common storage modules, processing modules, labels, and expressions written in LISP Infix language. The interpretation of a processing module name is an execution of that module. Label definitions may be made either singly or in groups. The label name(s) is surrounded by asterisks. Thus, to place the group of labels `L1`, `L2`, and `L3` at some spot, you write

```
*LI,L2,L3*
```

In order for a process to be initiated at or continue through a label group, a fork to each label must be executed.

The `AFTER` primitive forces a wait until a specified group of processing modules have been executed. To wait on modules `M1` and `M2`, write:

```
AFTER(MI,M2)
```

`FORK`

A fork to label `L` is written

```
FORK L
```

`FAKE`

A `FAKE` form tells the control monitor to initiate all activity dependent on a processing module without first executing the module. To "fake" the execution of `M`, write

```
FAKE M
```

Conditional

A conditional form is written in standard if-them-else format with the else-clause optional. The predicate is any expression written in LISP Infix language. The conditional,

```
IF P EQ Q THEN M1 ELSE M2
```

executes M1 if the value of P equals the value of Q. Otherwise, M2 is executed. The then-clause and the else-clause may be any flow or synchronization form except a BEFORE statement.

Parallel and Sequential

To operate a set of forms in sequential order, write the forms enclosed in parentheses. To operate a set of forms in parallel, write the forms enclosed in square brackets.

```
[(A,B), (C,D,E)]
```

The above will initiate two parallel processes. One process consists of the sequential execution of the processing modules A and B in that order. The other process is the sequential execution of the processing modules C, D, and E in that order.

Top-level CSL statements are assumed to specify a sequential ordering within the statements so that the outer parentheses may be dropped. No ordering among statements is implied by their lexical order of appearance; therefore, each control flow statement must begin with a label definition or an AFTER form. The control monitor always begins execution by forcing a FORK START.

BEFORE Statement

A BEFORE statement specifies a sequence of events that should happen before a processing module is executed or before a fork to a label is completed.

```
BEFORE M X,[Y,Z];
```

When M would normally be executed, the sequence of events is: execute X, operate Y and Z in parallel, and then execute M.

CYCLE

Execution of a CYCLE form discards data items presently on the named buses and makes new data items visible. To cycle buses B1 and B2, write

```
CYCLE(B1,B2)
```

UPDATE

Execution of an UPDATE form passes their accumulated updates to the named common storage modules. To update data collections X and Y, write

```
UPDATE(X,Y)
```

BREAK

A debugging breakpoint may be specified using the BREAK form. The body of the BREAK is an expression written in LISP Infix language. When executed, all processes are immediately suspended, the expression is evaluated and its value is printed on the interactive terminal, identification of the suspended processes is printed, and a debugging supervisor is entered for interaction with the user. To create a breakpoint which outputs the message HELLO, write:

```
BREAK "HELLO
```

STOP

Execution of a STOP form immediately terminates all processing and return to the LISP supervisor. A STOP form is just the word, "STOP".

Data Functions

The CSL run-time package provides modules with three functions for accessing the data paths: GETBUS, PUTBUS, and PUTCOM. Other accesses are made through the interrogators declared with the common storage modules.

GETBUS(*bus-name*)

The argument to GETBUS is the name of a bus. The value is a list of the data structures on the bus.

PUTBUS(*bus-name, data-structure*)

The arguments to PUTBUS are the name of a bus and a data structure to put on that bus. The new value becomes visible the next time the bus is cycled.

PUTCOM(*common-storage-module-name, update*)

The arguments to PUTCOM are the name of a common storage module and an update (data structure) for the named module. The accumulated updates are passed to the update function specified in the module declaration the next time an UPDATE form addresses the module.

APPENDIX: Postscript September, 1999

The 1974 paper did not include an example of the CSL language. That omission is rectified in this supplementary appendix. The example below is an encoding of the control monitor for the Predictive Linguistics Constraints (PLC) speech understanding system described by Barnett [8] and Ritea [7]. A reasonably accurate figure, showing *data flow* paths, appeared in [8] and is reproduced here as (the second) figured numbered 1.

The example CSL code was typeset by using an HP 5100 scanner to convert a 25 year old listing into ASCII, then that data was hand edited. Three language features not described in the prior appendix appear: The first is GO which does the obvious, i.e., it acts just like FAKE to a label that is not a module. The second feature is EVAL which evaluates the following form, written in LISP Infix language, for effect. The third is a BIND statement which binds local variables for use within the CSL form.

Figure 2 shows the *control flow* paths that emanate from the label post. Of special interest, is the handling of the AFTER form and its interaction with nested series and parallel computations. That piece of the PLC system does post-utterance recognition tasks including answering the user's query, updating the information available to the cross-sentential semantics modules, and interacting with the user as required. Note, when multiple arrows point at a computation, control must traverse *all* of them before that computation can commence; thus, the situation represented is a join operation.

The PLC example follows next. Note, the control monitor, like a normal subroutine, can receive arguments. Here, these arguments are used to control user-interface interactions and allow for the processing of canned scripts.

```
CONTROL plcvdms(talk.p, dcnt, dialogue)
  BIND sud;

  BUS pmeta, pword, pclass, phole;
  BUS fmeta, fword;
```

Addendum to the original added in September, 1999.

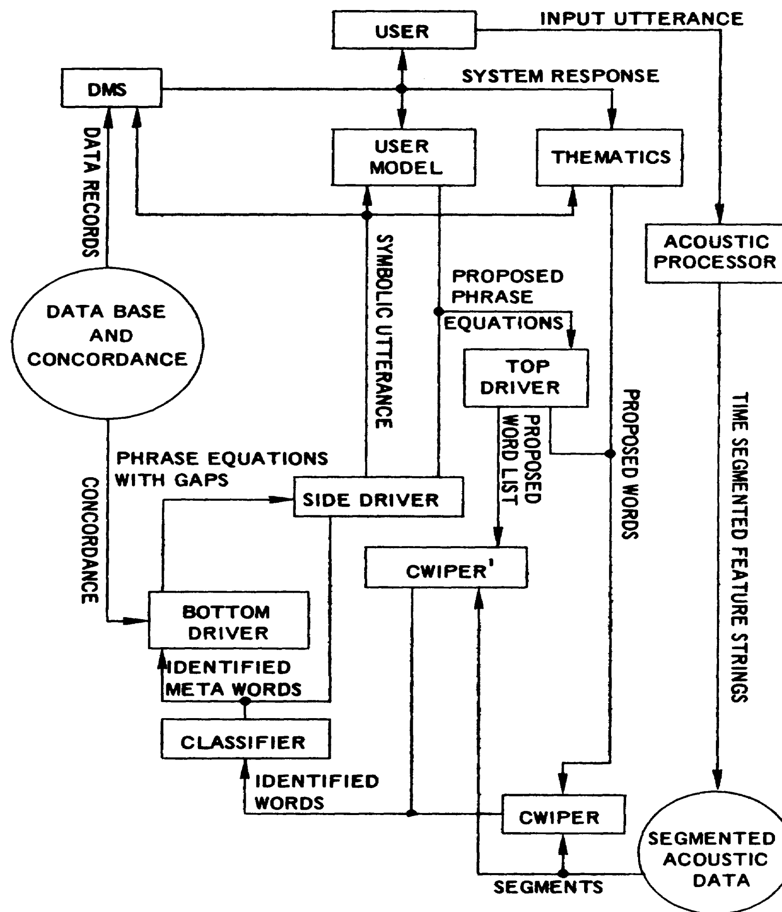

Fig. 1. PLC model for VDMS.

```
BUS sud, response;
BUS message;

COMMON cequ(cequ.i, cequ.u, cequ.i, cequ.g),
       sequ(nop, nop, nop, sequ.g, r2e);
COMMON lwrd(lwrd.i, lwrd.u, lwrd.i, lwrd.g),
       lmta(lmta.i, lmta.u, lmta.i, lmta.g);
COMMON wnxt(wnxt.i, pwnxt.u, wnxt.i, wnxt.g),
       mnxt(mnxt.i, mnxt.u, mnxt.i, mnxt.g);
COMMON vocab(nop, nop, nop, spell, cwiper.g),
       phone(nop, nop, nop, phone.g);
COMMON conc(nop, nop, nop, dbi.g, dbv.g,
```

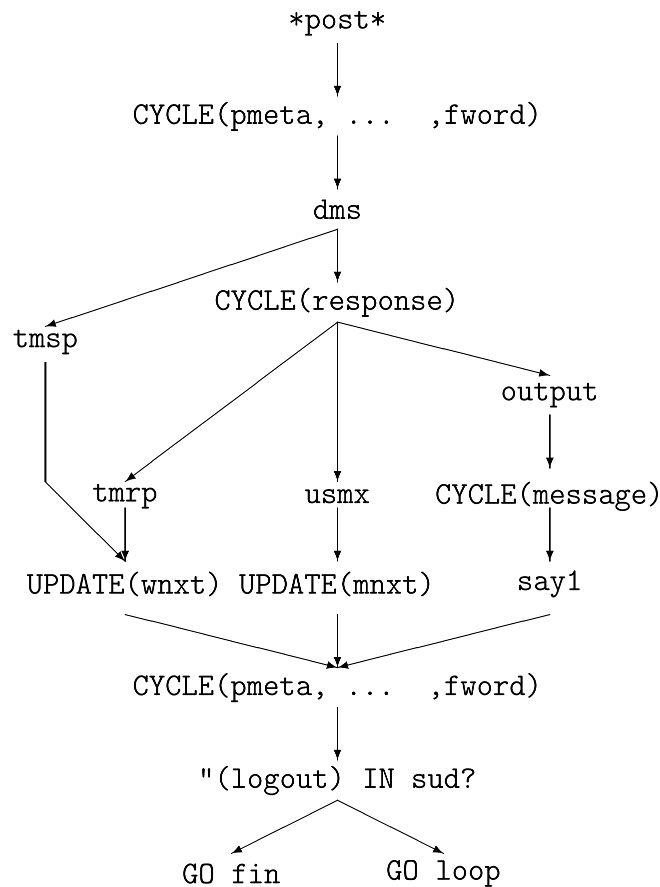

Figure 2: Control flow after label `*post*`

```
              dbd.g, aname.g, sname.g);
COMMON talk(talk.i, talk.u, talk.e),
       db(db.i, nop, db.i);

MODULE dms READ(sud)(sequ, db, conc) WRITE(response)();
MODULE synt READ(pmeta)(sequ, lmta)
            WRITE(fmeta, pmeta, pclass, pword)();
MODULE syns ALIAS(xsyns) READ(phole)(sequ, conc)
            WRITE(fmeta, pmeta, pclass, pword, sud)(lmta);
MODULE synb ALIAS(xsynb) READ(fmeta)(sequ, cequ, conc)
                         WRITE(phole)();
MODULE clsf ALIAS(xclsf) READ(fword)(conc)
```

Addendum to the original added in September, 1999.

```
                         WRITE(fmeta)(lmta);
  MODULE tmrp READ(sud, response)() WRITE()(wnxt);
  MODULE tmsp READ(sud)(sequ, conc) WRITE()(wnxt);
  MODULE tmxx ALIAS(tmpp) READ()(wnxt) WRITE(pword)();
  MODULE usmp READ()(mnxt) WRITE(pmeta)(cequ);
  MODULE usmx READ(sud, response)(sequ) WRITE()(mnxt);
  MODULE cwiw ALIAS(cwiper)
              READ(pword)(lwrd, talk, phone, vocab)
              WRITE(fword)(lwrd);
  MODULE cwix ALIAS(cwipex)
              READ(pclass)(lwrd, talk, phone, vocab)
              WRITE(fword)(lwrd);
  MODULE output READ(sud, response)() WRITE(message)();
  MODULE say ALIAS(say1,say2) READ(message);


*start* EVAL wcnt=initcount(), GO loop;

*loop* IF NODEP(talk.p) OR dcnt LQ 0 THEN UPDATE(talk),
       EVAL [lwrd.i(), lmta.i()]
       [ (usmp, CYCLE(pmeta), UPDATE(cequ)),
         (tmpp, CYCLE(pword)) ],
       IF dcnt EQ 0
          THEN (BREAK "cwiper.start.up, EVAL dcnt=-1, GO norm)
          ELSE IF dcnt LS 0 THEN GO norm
                  ELSE (EVAL [PUTBUS("sud, CAR(dialogue)),
                              sud=LIST(CAR(dialogue)),
                              dialogue=CDR(dialogue),
                              dcnt=dcnt-1],
                        CYCLE(sud),
                        GO post);

*norm* [synb, synt, syns, clsf, cwiper, cwipex],
       CYCLE(pmeta, pword, pclass, phole, fmeta, fword, sud),
       UPDATE(lwrd, lmta),
       GO check;

*check* IF sud=GETBUS("sud)
```

```
            THEN (BREAK "(you said)@CAR(sud)), GO post)
            ELSE IF GETBUS("fword) OR
                        GETBUS("fmeta) OR
                        GETBUS("phole)
                    THEN ( [xsynb, xsyns, xclsf],
                            CYCLE(fword, fmeta, phole, sud),
                            UPDATE(lword, lmta),
                            GO check)
                    ELSE IF GETBUS("pmeta) OR
                                GETBUS("pword) OR
                                GETBUS("pclass)
                             THEN GO norm
                             ELSE ( CYCLE(pmeta, pword, pclass),
                                    IF GETBUS("pmeta) OR
                                          GETBUS("pword) OR
                                          GETBUS("pclass)
                                       THEN GO norm
                                       ELSE GO erro );

*erro* BREAK "no.comprendo,
       CYCLE(sud),
       IF NOT (sud=GETBUS("sud))
          THEN GO erro
          ELSE GO post;

*post* CYCLE(pmeta, pword, pclass, phole, fmeta, fword),
       [ (tmsp, AFTER(tmrp) UPDATE(wnxt)),
         (dms, CYCLE(response),
          [ tmrp,
            (usmx, UPDATE(mnxt)),
            (output, CYCLE(message), say1)] )],
       CYCLE(pmeta, pword, pclass, phole, fmeta, fword),
       IF "(logout) IN sud
           THEN GO fin
           ELSE GO loop;

*fin* EVAL [FOR x IN "(mcnt bcnt ccnt wcnt), i=10 STEP 10
                DO prx(x, i), TERPRI()],
```

Addendum to the original added in September, 1999.

```
    EVAL [prx(mcnt,10), prx(bcnt, 20), prx(ccnt,30),
          prx(wcnt, 40), TERPRI()];
    STOP;
END;
```

Addendum to the original added in September, 1999.