Massachusetts Institute of Technology
Laboratory for Computer Science

Computation Structures Group Memo 168

Computation Structures Group Progress Report 1974 - 1975

This research was supported in part by the National Science Foundation under grants GJ-34671 and DCR74-21822, and in part by the Advanced Research Projects Agency of the Department of Defense under contract N00014-75-C-0661.

October 1978

COMPUTATION STRUCTURES GROUP

Academic Staff

Prof. J. B. Dennis
Dr. V. Kotov
Prof. B. H. Liskov
Prof. S. S. Patil
Prof. J. E. Stoy
Prof. H. Weber

Sponsored Research Staff

P. S. Thiagarajan

Graduate Students

J. M. Aiello
S. N. Amersinghe
R. R. Atkinson
S. A. Borkin
J. D. Brock
R. E. Bryant
D. J. Ellis
F. C. Furtek
M. H. T. Hack
D. A. Henderson, Jr.
D. L. Isaman
D. Kapur
P. R. Kosinski
M. S. Laventhal
C. K. Leung
D. P. Misunas
J. E. Qualitz
J. E. Rumbaugh
J. C. Schaffert
L. A. Snyder
K. S. Weng
S. N. Zilles

Undergraduate Students

C. N. Anderson
A. P. Holt
R. G. Jacobsen

Support Staff

M. Nieuwkerk
A. L. Rubin

COMPUTATION STRUCTURES GROUP

A. INTRODUCTION

In the past year, work conducted in the Computation Structures Group has emphasized the study of asynchronous systems, the design and application of data-driven computer languages and architectures, and the development of new programming methodologies using data abstractions.

The work on concurrent systems has concentrated on the analysis of certain properties of Petri nets along with the use of nets to study aspects of formal language theory. The study of data-driven languages and architectures has examined the development of computer languages to exploit concurrency of program parts and the structure of computer systems which can efficiently execute programs expressed in parallel form. A number of computer and memory system architectures which execute programs represented in data-driven form have been investigated. Such systems are constructed of large numbers of modules which communicate asynchronously through the transmission of information packets. To study the feasibility of these system structures, we have developed the initial specification of a microprocessor-based simulation facility and are engaged in its further design.

The study of programming methodologies based on data abstractions involves the study of tools and techniques to enhance the effectiveness of programmers in producing quality software -- software that is reliable, has comprehendable structure, and is relatively easy to modify and maintain. Two major directions are being followed. A programming language/system, CLU, is under development. CLU enhances program quality by permitting direct expression of the kinds of program structures arising from promising design methodologies. In addition, a study of specification techniques well-suited to the program structures of CLU is under way; using these techniques, the programmer will be able to express and investigate properties of his program design in advance of actual implementation, and to prove the correctness of his implementation once it exists.

B. THE CLU LANGUAGE/SYSTEM

Barbara Liskov and her group have been developing the CLU programming language and system. The motivation behind the development of CLU is discussed in [3,4]. Briefly, CLU is intended to simplify the design and implementation of quality software by providing linguistic constructs that allow the kinds of modules identified during design to be written naturally as CLU programs. The most important such construct, and the one that is original in CLU, is the cluster. The cluster permits a program module to be written that implements a data abstraction, or abstract data type, consisting of both a set of objects or values belonging to the type, and a set of operations that completely determine the behavior of the type's objects. Data abstractions are a particularly valuable sort of program module: they occur widely, since the manipulation of data is a primary concern of programming, and although much sharing of information and resources takes place within the data abstraction (particularly important is information about how objects of the type are represented in storage), this sharing is limited to the implementation of the data abstraction, and is not visible to the abstraction's users. The advantages of such an organization have been

discussed in [1,2,5]; for example, the interface of a data abstraction module is very simple, and the hiding of information within the module means that the implementation of the abstraction can be changed without requiring recoding of the programs using the abstraction. One such change that might be made is to choose an alternative representation for the data abstraction's objects.

CLU differs from other languages in its emphasis on constraints, enforced by the language and its compiler, that guide the programmer's search for a good design by removing his or her freedom to violate certain precepts of good programming practice. A common constraint, found in almost all higher level languages, protects the local variables of a procedure from manipulation outside of the procedure's code. CLU extends this idea of constraints to permit a group of procedures to share a local environment: this is done through the cluster, which limits information about a type's implementation to the operations belonging t o the type. Conventional languages (e.g. FORTRAN, PL/I) provide no mechanism like the cluster. Even advanced extensible languages (e.g. EL1[6]), which provide data type extensions and even permit some operations to be defined along with the type, still do not constrain access to the type to just the operations, so the advantages mentioned above can be obtained by the programmer only by extra-language means. The issue here is not whether well-structured programs can be written, since they can be written even in assembly language. However, in languages other than CLU, such programs can be written only in spite of the language. Our goal is to simplify the writing of programs by having the language provide guidance about what constitutes good programming practice.

CLU is a language/system. A CLU program consists of a number of modules; each module implements an abstraction identified as useful during program design. The CLU system includes a description unit for each module, containing all in-computer information about the module. The description unit is created as soon as the module interface is known (before the module is implemented); formal specifications would also be entered at this stage. Modules are compiled separately; the CLU compiler makes use of the interface information to check that modules refer to other modules correctly. The CLU system is also used to control the loading and execution of programs.

Our major activity during the past year has been the implementation of a first version of CLU. The implementation is divided into three pieces: the CLU compiler, inter-module type-checking, and the CLU system. The CLU compiler was implemented first and has been running for several months; it translates CLU modules into a LISP-like language called MDL [7]. The type-checker has been implemented and debugged, and is awaiting integration with the CLU system (which contains the information about the type requirements of modules, and also establishes the meaning of types). The CLU system is currently being designed and implemented. A very interesting problem arising in the design of such a system concerns how to cope with multiple implementations of a data type. We are working on providing multiple implementations of a type in a very flexible way: different users can select different implementations, and even within the same program, different implementations can be used for different data objects of the same type.

The implementation of CLU was undertaken for two reasons: to establish the soundness of our design, and to permit us to gain experience in using CLU. No

problems with the design of CLU were uncovered during the implementation; some minor modifications have been made to the language to make CLU programs easier to write. We have used CLU both for the CLU implementation, and to write many smaller programs. The results have been encouraging; programmer productivity is high, and the resulting programs have a good structure and are easy for others to understand. We have discovered that data abstractions are indeed very valuable for structuring programs, and that we design programs by identifying data abstractions, and then specifying the properties of their operations, in advance of any implementation. The transition from design to implementation is particularly simple, since each design unit becomes a CLU program module.

The language being implemented is only an initial version of CLU, and we have continued to work on the design of CLU itself. One accomplishment of the past year has been the design of the structured exception handling mechanism described below.

C. STRUCTURED ERROR HANDLING

In designing the exception handling mechanism of CLU, our primary concern was the support of "robust" or "fault tolerant" programs, i.e., programs that are prepared to cope with the presence of errors by attempting various error recovery techniques. Note that it is the programs themselves that must recover from errors. We do not assume that a person helps in the error recovery (although this will sometimes happen), and therefore the mechanism need not facilitate person-computer interaction. In particular, the mechanism is not intended to support interactive debugging.

Successful handling of errors involves two separate activities. First the errors must be detected. After an error has been detected, it may then be possible to recover from the error. If it were always possible to recover from an error in the same local context in which the error was detected, no special error handling mechanism would be needed. However, often recovery must occur at a very different point in the program from where the error was detected. Thus the purpose of the mechanism is to permit information about errors to be communicated from one part of the program to another. Note that we are not concerned here with how error detection and recovery are accomplished (through redundancy) except to recognize that paths permitting communication of information about errors are required.

The use of the word "error" in the above discussion is somewhat misleading because what may appear as an error to one part of a program may be considered as reasonable behavior in another part. For example, an attempt to read from an empty file raises an "end-of-file" error; to the user of the read command, this merely means that all data has been read. Therefore, in the remainder of this section, we will use the more neutral term "exception" to refer to the occurrences of interest.

Our study of exception handling has led to the following analysis of how such a mechanism should behave:

1. Information about exceptions always results from a procedure invocation and flows from the called procedure to its caller. Whenever a procedure is invoked, it is invoked to perform a certain action or cause a certain effect. If the procedure is unable to do this, then it must notify its caller that something

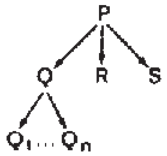
exceptional has occurred.

It is important to recognize that information about an exception results from an invocation even if the invocation does not actually occur. For example, the integer divide-check exception may result from the invocation of the integer division operation. In CLU, as in most languages, division may be written as part of an expression:

$$y/x$$

and the code to do the division occurs in-line in the procedure containing this expression. Nevertheless, conceptually the exception is detected in a lower level procedure invocation, and the procedure containing the expression is notified of the exception.

2. The precepts of structured programming require that only the procedure performing an invocation can handle exceptions arising from that invocation, as the following discussion shows. CLU programs have a hierarchical structure. Consider, for example, the following graph:



Each node in this graph represents a CLU module, which implements an abstraction: P is programmed in terms of abstractions Q, R, and S; Q makes use of abstractions Q_1, ...Q_n; and so on. The graph is a static view of the program structure; each module makes use of the modules connected to it by the outward pointing arcs. Dynamically, a module makes use of another module by invoking one of its procedures; invocations can only occur in the direction specified by the arc between two modules. Information about exceptions flows in the opposite direction; a procedure notifies procedures higher in the call chain of the existence of an exception.

Some rules about structured exception handling can be derived from considering the relationships between modules in the graph. An important precept of structured programming is that a module knows only about the abstractions it uses; it knows nothing about the abstractions used in implementing those abstractions. Thus P knows nothing about Q_1, ...Q_n. But if a module knows nothing about the implementation of a module it uses, it cannot possibly respond intelligently to exceptions detected by procedures called in the course of that implementation. The only module that can respond to the exceptions detected by a procedure is that procedure's caller. E.g., only Q can respond to the errors detected by Q_1, ...Q_n; P cannot.

3. A principle of modularity is that a module should be programmed to know nothing about the module using it, so that it can be used in many different places. Thus, a procedure knows nothing about its caller, and should not report an exception by a mechanism that assumes something about the caller's environment. In particular, a jump to a non-local label (as in PL/I) is not a satisfactory mechanism, nor should information about the exception be communicated using non-local variables. Instead the mechanism should permit a procedure to communicate information about an exception to its caller without making any assumptions about who its caller is. Note that this is very similar to the way that procedures return control to their callers under normal conditions.

   The use of a non-local goto as an exception mechanism is unsatisfactory for another reason. Even if a program is unable to recover from an error, it should restore its non-transient data to a consistent state; this is necessary to prevent an error from causing many other, unrelated errors later on. The exception mechanism must ensure that active programs, whose data may be inconsistent, will have the opportunity to respond to exceptions before becoming inactive. Thus, a mechanism that automatically terminates procedure activations is unsatisfactory.

4. Finally we come to the question of what actions the caller of a procedure can perform when notified about the occurrence of an exception. The simplest view, which is the one we take, is that one of two actions can occur: If the procedure is unable to recover from the exception, it may notify its caller that an exception (different from the one detected) has occurred; or, if the procedure is able to recover, it may continue its normal flow.

   What is explicitly forbidden here is the ability for the calling procedure to resume processing in the procedure that detected the exception. Although we recognize that the ability to resume is sometimes convenient, we believe resuming is not necessary (provided the mechanism is sufficiently general, as ours is) and that resuming necessitates a much more complicated model of computation, in which the exception-reporting procedures act like coroutines.

The result of our analysis of exception handling is the following simple model of how the exception handling mechanism should behave: Each procedure can terminate execution in one of several states; one of these states is the "normal" state, while the others represent exceptional conditions. Each state is given a symbolic name (the normal state is implicitly named "normal"). Finally, in each state, values may be returned to the calling procedure; these values can differ in type and number from one state to another. Allowing parameters to be returned eliminates the need for global variables to hold error information.

We believe the above model strengthens the abstraction power of the language. Each procedure is expected to be defined over all possible values of its input parameters and all possible actions of the procedures it calls. However, it is not expected to behave in the same way in all cases. Instead, it may respond appropriately in each case.

We consider that an abstraction is not meaningful unless all the exceptions that

can arise from its use are identified. Thus, the specification of integers must indicate that divide-check can occur, and (in most languages) that integer-overflow can also occur; without this information, the user of integers will not fully understand their behavior. Similarly, stacks have associated overflow and underflow exceptions.

The requirement that an abstraction identify all exceptions applies whether the abstraction is language or user defined. For all the primitive types in CLU, exceptions have beeen identified and made part of the abstraction. The inventor of new abstractions (e.g. stacks) should do the same. As an added benefit, the process of identifying exceptions can be quite valuable during program design: an abstraction with many exceptions and special cases can probably be improved by redesign. CLU arrays reflect this concern with limiting exceptions; CLU arrays have an "index out of bounds" exception, but not the "undefined element" exception that arises in array abstractions in which space for array elements can exist in advance of values to store in the elements.

We have a partial design of how the exception handling mechanism is to be incorporated in the CLU language. The mechanism is completely defined as far as reporting exceptions is concerned.

1. The mechanism for reporting exceptions is the signal, which is a special type of return. Since the signal is a return, the activation of the signalling module will disappear; therefore the procedure must ensure that all its non-transient data objects are in consistent states before signalling. The best method of ensuring this is to detect exceptions before any objects are modified, but this is not always possible.

   A signal always specifies a particular exception name; thus a procedure may have several exceptions associated with it. In addition, some values may also be returned by a signal. For example,

   signal foo(x)

   terminates execution of the procedure containing the signal statement with an indication that the "foo" exception has occurred; the current value of x is returned as a result.

2. All the exceptions to be reported by a procedure must be specified as part of the header of that procedure. For example,

   ```
   push = oper(s:stack)returns(int)
              signals(underflow)
   ```

   This information is also included in the CLU system library as part of the description unit of the abstraction that the procedure implements. If the exception returns values, the types of these values must always be specified, and these types must agree with the types of values actually being returned. For example, the operation to read the nth character of a string has interface description

```
cn = oper(s:string, n:int)returns(char)
                signals(bounds(int))
```

The bounds exception returns the value of the out-of-bounds integer.

As was mentioned earlier, the calling procedure is required to respond to, or "catch," all exceptions arising from procedures it invokes. The hardest part of designing an exception handling mechanism, once the basic principles are worked out, is to provide good human engineering for catching exceptions. Flexibility in placement of the exception handlers is essential; otherwise the readability of programs will be compromised. Exceptions can arise from the evaluation of every expression, but requiring handlers inside of expressions would make programs unreadable. At present, we are investigating the semantic and syntactic issues that arise from the requirement of flexible placement of exception handlers within the calling procedure.

D. SPECIFICATION TECHNIQUES FOR DATA ABSTRACTIONS

One of the properties of data abstractions mentioned earlier is that their interface is particularly simple, since so much information is hidden within the module. Therefore, we can hope that data abstractions will have simple specifications, since it is precisely the interface that a specification must describe. A study of specification techniques for data abstractions was undertaken this year by B. Liskov and S. Zilles and is reported in [8]. Included here is a brief summary of this work.

A formal specification for a functional abstraction describes the effect of a single operation; a convenient way to do this is by an input/output specification. A specification for a data abstraction, which contains many operations, must describe the effects of all the operations, and also the behavior of the objects belonging to the type. New techniques are being defined for specifying the behavior of data abstractions. Using these techniques, the entire data abstraction is specified as a unit, with the advantage that a more minimal specification results, describing just the externally observable behavior.

The information contained in a specification of a data abstraction can be divided into a semantic part and a syntactic part. Information about the actual meaning or behavior of the data abstraction is described in the semantic part; the description is expressed using a vocabulary of terms or symbols defined by the syntactic part.

The first symbols that must be defined by the syntactic part of a specification identify the abstraction being defined and its domain or class of defined objects, and, in this case, it is conventional to use the same symbol to denote both the abstraction and its class of objects. Thus, the objects belonging to the data abstraction, stack, are referred to as stacks.

The remaining symbols introduced by the syntactic part name the operations of the abstraction, and define their functionality -- the domains of their input and output values. An example describing the functionality of the operations of the data abstraction, stack, is shown below.

CREATE: -> STACK
PUSH: STACK X INTEGER -> STACK
POP: STACK -> STACK
TOP: STACK -> INTEGER

(TOP returns the value in the top of the stack without removing it, while POP removes the value without returning it.)

Note that more than one domain appears in the specification; this is true for almost all interesting data abstractions. Normally, only one of these (the domain of stacks in the example) is being defined; the remaining domains and their properties are assumed to be known. Given this distinction, the group of operations can be partitioned into three blocks. The first block, the primitive constructors, consists of those operations that have no operands in the domain being defined, but which yield results in the defined domain. This block includes the constants, represented as argumentless operations (for example, the CREATE operation for stacks). The second block, the combinatorial constructors, consists of those operations (PUSH and POP in the example) that have some of their operands in and yield their results in the defined domain. The third block consists of those operations (TOP for stacks) whose results are not in the defined domain.

The semantic part of the specification uses the symbols introduced in the syntactic part to express the meaning of the data abstraction. Two different approaches are used in capturing this meaning: either an abstract model is provided for the class of objects and the operations defined in terms of the model, or the class of objects is defined implicitly via assertions of properties of the operations.

In following the abstract model approach, the behavior is actually defined by giving an abstract implementation in terms of another data abstraction, one whose properties are well understood. The data abstraction being used as the model also has a number of operations, and these are used to define the operations of the new data type.

The approach of defining the objects implicitly via descriptions of the operations is much closer to the way mathematical theories are usually defined. Axioms are given that describe the behavior of the operations. The domain or class of objects is determined inductively. Usually it is the smallest set closed under the operations. Only those operations identified above as constructors are used in defining this closure. The closure is the smallest set containing the results of the primitive constructors and the results of the combinational constructors when the appropriate operands are drawn from the set. For example, with stacks, the only primitive constructor is the constant operation CREATE, which yields the empty stack, and the class of stacks consists of the empty stack and all stacks that result from applying sequences of PUSH's and POP's to it. One difficulty with the implicit definition approach is that if the specifications are not sufficiently complete, in the sense that all the relationships among the operations are indicated, several distinct sets may be closed under the operations. The distinct sets result from different resolutions of the unspecified relationships.

In [8] a number of specification techniques for data abstractions were surveyed and compared. Two abstract model approaches were considered: use of a single fixed

modelling domain (e.g., graphs or sets) and use of an arbitrary fixed modelling domain. When using a single fixed domain, the specifications are usually easily understood and easily constructed by someone familiar with the modelling domain, if they describe concepts within the range of applicability of the chosen domain. However, a fixed modelling domain usually has a somewhat limited range of applicability; only certain abstractions are expressed easily within the domain. Using such a technique is similar to writing programs in a programming language that provides a single data structuring method; although a single method can be powerful enough to implement all user-defined data structures, it does not follow that all data structures are implemented with equal facility. This limitation is somewhat mitigated by allowing the specifier to make use of an arbitrary fixed modelling domain. However, the number of domains available for use is not large, and, in addition, if a completely free choice of domains could be made, it is doubtful that the resulting specification would be comprehensible. Thus, in reality, the specifier must choose among a small number of domains. This situation is analogous to writing programs in a language providing several data structuring facilities; programming experience indicates that there will always be (problem oriented) abstractions that cannot be ideally represented by any of the data structuring methods. Thus, it appears unlikely that all data abstractions can be given minimal specifications by choosing among a small number of modelling domains.

Included among the implicit definition approaches are the state machine model approach of Parnas [9], and the algebraic approach of Zilles [10] and Guttag [11]. The state machine model approach as originally described by Parnas is not a formal technique: English is used to describe behavior when all else fails. Two techniques are being investigated to correct this: the hidden function approach at S.R.I. [12], and a new approach by Parnas [13]. The hidden function approach appears to introduce an abstract model to describe behavior, while the approach of Parnas uses axioms to express the behavior of the abstraction as a whole, and appears fairly close to the algebraic approach. Both approaches require more development before their properties will be known.

The algebraic approach of Zilles appears to be quite promising. From the syntactic part of the specification (the functionality of the operations) the set of legal, finitely constructible expressions in the operations can be defined; these expressions are the words of a word algebra. Then axioms are given that specify when two words are equivalent; an example of such an axiom, for the stack example given earlier, is:

$$pop(push(s,i))=s$$

where s is a stack, and i is an integer. All words whose equivalence does not follow from the axioms are taken to be distinct.

The algebraic approach can be used to construct minimal specifications, containing no extraneous information, and there is no limit on the range of applicability. The main problem is that it may prove difficult to construct and comprehend these specifications, because they are so abstract. It is difficult to be certain that a set of axioms is complete and consistent. However, our experience in using the technique indicates that it is reasonably easy to use. In addition, tools can be devised to help the specifier determine the consistency, completeness, and meaning of these specifications [11].

As was explained in the introduction, the study of specification techniques was motivated by the desire to enhance the quality of software. Specifications are not only useful in proving the correctness of programs; they are a valuable aid during the process of system design. When a program is developed by stepwise refinement [14, 15], the problem of concern at a program level is solved by introducing abstractions that provide useful primitives for that problem domain. The original problem is solved in terms of the abstractions; each abstraction then becomes a new problem to be solved. Specifications provide a way to make this process precise: if each abstraction is defined completely by means of a specification, then we can be sure the implementor of a program using an abstraction and the implementor of the abstraction agree about the meaning of the abstraction. This is particularly important for large programs in which the abstractions may be implemented by different people.

In addition, we have found the actual writing of the specifications to be a valuable addition to the design process: if an abstraction has a complicated specification, often a better form of the abstraction with a simpler specification can be found. The provision of tools for examining properties of specifications in advance of implementation, which will be possible when specifications are added to the CLU system, also appears promising.

E. THEORY OF PETRI NETS

1. Representational Power of Nets

While last year's effort was mainly directed at the Liveness and Reachability Problems, a large part of this year's research has focused on the representational power of the Petri net model. We have been investigating the sets of firing sequences generated by a Petri net and the languages that can be obtained from such sets of firing sequences by assigning labels, which are symbols from some alphabet, to some or all of the transitions in the Petri net. By analogy with finite-state machines, the firing of a transition in the Petri net corresponds to the reading of the corresponding symbol from an input tape (language recognizer), or to the printing of the symbol on an output tape (language generator). Two main classes of such Petri net languages are distinguished, depending on whether we consider all firing sequences from an initial marking, or only terminal firing sequences, i.e. sequences which reach a given final marking, usually the zero marking, akin to an accepting state of an automaton. A further three-fold subdivision depends on whether all transitions have distinct labels (free labelling), whether all transitions are labelled, but not necessarily distinctly, (λ-free labelling), or whether there may be unlabelled transitions (λ-transitions, whose firing does not show up in the language). Table 1 shows the resulting six language families.

A summary of the closure properties of these language families, their relation to other formal language families, and their various decision problems is presented in Table 2. Some results about the family $\mathcal{L}0$, in particular the closure properties and the relation to Regular, Context-Free and Context-Sensitive languages, had been obtained previously by J. L. Peterson [16], who called this family "Computation Sequence Sets".

All Petri net languages can be obtained from the free Petri net languages by homomorphisms, and the free languages can be obtained by intersection, concurrency

	terminal firing sequences only	all firing sequences
all transitions labelled distinctly	$\mathcal{L}_0^f$	$\mathcal{L}^f$
all transitions labelled	$\mathcal{L}_0$	$\mathcal{L}$
may have λ-transitions	$\mathcal{L}_0^\lambda$	$\mathcal{L}^\lambda$

Table 1

and inverse homomorphism from the Regular languages and the simple parentheses languages (one pair of parentheses only). This research about Petri net languages is reported in detail in [17]. That report also demonstrates how Petri nets can be extended to generate all recursively enumerable languages, either by explicitly testing for zero (Inhibitor Nets) or by submitting the firing of transistions to a Priority Rule (Priority Nets); similar results have been reported by T. Agerwala [18].

Last year we reported that the Equality Problem for Vector Addition Systems was still an open problem. In October 1974 Michel Hack finally settled the conjecture of its undecidability by reducing the Inclusion Problem -- which Rabin proved to be undecidable in 1967 -- to the Equality Problem. In Petri net terms, it is thus undecidable whether two Petri nets with the same number of places have the same set of reachable markings. This result, as well as a Petri net version of Rabin's proof, is reported in [19], and will appear in the Journal of Theoretical Computer Science.

These recursive reducibility techniques have also produced two more problems equivalent to the Reachability Problem. A transition is said to be persistent iff it cannot be disabled by firing any transition other than itself. The question of the persistence of a transition can be reduced to the submarking reachability problem (this was also proved independently by L. Landweber), and thus to the Reachability Problem. But it turns out that the Reachability Problem is reducible to the special case of zero-reachability in a single place, which in turn can be reduced to the persistence of a single transition. Therefore persistence of a given transition and reachability of zero in a given place are two problems recursively (actually efficiently) reducible to the Reachability Problem.

These recent results will be published in Michel Hack's forthcoming Ph.D. thesis [20]

2. Prompt Nets

In related work, P. S. Thiagarajan and S. S. Patil have been studying an interesting behavioral property of nets termed as promptness. Informally speaking, if we identify certain transitions of a Petri net as external transitions and the other

Closure under:	$\mathcal{L}_0^f$	$\mathcal{L}_0^\lambda$	$\mathcal{L}_0$	$\mathcal{L}^f$	$\mathcal{L}$	$\mathcal{L}^\lambda$
Union	no	yes	yes	?	yes	yes
Intersection	yes	yes	yes	yes	yes	yes
Concatenation	?	yes	yes	?	yes	yes
Concurrency	?	yes	yes	?	yes	yes
Regular Substitution	no	if λ-free	yes	?	if prefix	if prefix
Inverse Homomorphism	yes	yes	yes	yes	yes	yes
Kleene Star	?	no	no	?	?	?
Complementation (up to λ)	?	would imply the undecidability of RP		closure included in $\mathcal{L}_0^\lambda$	no no	no no
Relationship to:						
Regular Languages		λ-free $\subseteq$	$\subseteq$		prefix $\subseteq$	prefix $\subseteq$
Context-Free Languages	symmetric difference is non-empty					
Context-Sensitive Languages	$\supseteq$	$\supseteq$	?	$\supseteq$	$\supseteq$	?
Szilard Languages of Matrix Context-Free Grammars	same					
Decision Problems:						
Membership	D	D	RP	D	D	D
Emptiness	RP	RP	RP	D	D	D
Finiteness	RP reducible to it			D	D	D
Equivalence and Inclusion	RP	U	U	reducible to RP	U	U

RP = Reachability Problem D = decidable U = undecidable

Table 2

transitions as internal transitions, then the net is said to be prompt if there can be at most a bounded number of firings of internal transitions between successive firings of external transitions. Thus, if a Petri net is viewed as a black box and the firing of an external transition interpreted as an interaction with the external world, then promptness demands that there be no more than a bounded number of occurrences of events within the box between successive interactions with the external world.

Such a notion of promptness leads us to conclude that non-promptness of a net implies uncertainty about the response time and lack of a priori knowledge concerning the extent of resource consumption by the system that is represented by the net. Conversely, promptness under suitable interpretation would guarantee that the external world has an adequate amount of "control" over the activities initiated by the system.

From the theoretical standpoint, it turns out to be useful to identify two kinds of promptness, as explained below:

Let G be a Petri net with initial marking M whose set of transitions has been partitioned into external transitions T_E and internal transitions T_I. If there exists an integer K_G such that starting from any reachable marking M' can fire internal transitions alone at most K_G times, then we will say that G is strongly-prompt. Given a net G, such an a priori bound K_G might not exist. However, corresponding to every reachable marking M', there could be an integer bound $K_{M'}$ such that, starting from M' we can fire internal transitions alone, at most $K_{M'}$ times. In this case, we will say that the net is weakly-prompt. Clearly, if a net is strongly-prompt then it is weakly-prompt also.

Figure 1 shows a net that is not strongly-prompt but is weakly-prompt. This net is not strongly-prompt because given any integer k (as a possible candidate for the a priori bound K_G), we can first fire t_1 k times. Then starting from the marking thus reached, a single firing of t_2 followed by k firings of t_3 will yield k+1 firings of internal transitions alone. The net is weakly-prompt since for any reachable marking M', $K_{M'}$, defined as $K_{M'} = M'(p_2) + 1$, will satisfy the required condition.

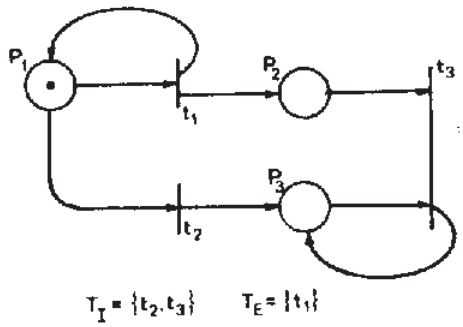


Figure 1. An example of a net which is weakly-prompt, but not strongly-prompt.

We have shown that strong-promptness of a Petri net is decidable. In doing so, we have also been able to reduce the problem of deciding the weak-promptness of a net to an interesting open decision problem in net theory -- namely, the problem of deciding whether a given transition in a net is hot in the sense of Keller [21], i.e., if there exists a firing sequence in which the transition appears an infinite number of times.

At present, however, we are mainly interested in studying the promptness of bounded Petri nets. This is due to our feeling that for representing physically-realizable concurrent systems, only bounded Petri nets would be required. To begin with, we have shown easily that for bounded Petri nets, strong-promptness and weak-promptness are equivalent notions. Hence for bounded nets, our definitions provide a formal counterpart to the intuitive notion of promptness. Secondly, we have carried out a detailed study on the promptness of an interesting class of bounded nets, namely, the class of live and bounded free-choice nets. This class of nets was first studied by Hack [22].

We shall introduce the following terminology to aid in stating our main result. Let G be a Petri net and let T be its set of transitions. If $T_1 \subseteq T$, then $\langle T_1 \rangle$ denotes the Petri net (a subnet of G) consisting of transitions T_1, all places that are input or output places of transitions in T_1, and all arcs of G connecting these places and transitions.

Now let G be a live and bounded free-choice net and let , a set of subnets of G, contain each subnet of G that is a strongly connected marked graph. Then is said to be the set of mg-components of G. One main result may now be stated as follows:

Theorem: Let G be a live and bounded free-choice net with transitions T. Let $T_1, T_2, \ldots, T_k$ be subnets of T such that $\{\langle T_1 \rangle, \langle T_2 \rangle, \ldots, \langle T_k \rangle\}$ is the set of mg-components of G. Then G is prompt with respect to a set T_E of external transitions iff for $1 \leq i \leq k$, $T_i \cap T_E \neq \emptyset$.

In other words, if a live and bounded free-choice net has an mg-component all of whose transitions are internal transitions, then and only then is the net non-prompt. Consequently, the task of checking the promptness of a net belonging to the above class of nets reduces to determining its set of mg-components. Moreover, using the above result, we can easily transform a non-prompt live and bounded free-choice net into a prompt one by introducing a minimum number of additional external transitions to the appropriate mg-components.

F. PROGRAM SCHEMAS

We previously reported [23] work on progress in comparative schematology and equivalence problems of data flow schemas. This work has been concluded with the completion of two theses during the past year.

In his dissertation, J. Qualitz conducted studies of the decidability of equivalence for a class of monadic schemas [24]. This class, called iteration schemas, consists of schemas whose programs comprise assignment statements, conditional statements, and iteration statements. They correspond to program schemas which are structured and

are therefore less powerful than the monadic program schemas.

Schema equivalence is formalized as the functional equivalence of schemas under free interpretations; that is, interpretations which represent symbolically the set of all interpretations of a schema. The work shows that the equivalence problem for iteration schemas is unsolvable, even if certain highly restrictive conditions are imposed. On the other hand, equivalence is found to be decidable for several interesting subclasses of iteration schemas.

Our work in program schematology has also lead us to study the class of open and complete well-formed data flow schemas. This class of program schemas is proposed as a model for well-constructed computer programs. The following undecidability results about such schemas are proved in [25]:

(i) the equivalence problem for open and complete wfdf's is undecidable, and
(ii) openness is undecidable for complete schemas.

G. DATA FLOW LANGUAGES

A continuing objective of our research is the formulation of a suitable base language for use in guiding architectural studies for advanced computer systems. Current work is directed at the design of a base language using the principles of data flow, which are consistent with recent findings about good program structure and expose concurrency of program parts for exploitation by computers of appropriate design. The data flow language described in [26], while sufficiently expressive to encompass programs in Algol 60 or pure Lisp, is incomplete in several ways (just as Algol 60 is itself similarly incomplete): there is no program unit or module able to communicate with other modules by sending a stream of values; and there is no means of representing nondeterminate computations such as those involving updating of a shared data base. Progress has been made towards finding good approaches to both problems.

The language TDFL under development by K. Weng approaches the former problem in a manner which guarantees determinacy and allows the exploitation of the potential parallelism in stream-oriented computation. Parallelism in TDFL is implicit in that no primitives are explicitly used for invoking concurrent processes; this implicit parallelism is exploited by providing a translation rule for converting TDFL programs into data flow schemas in which the parallelism is represented explicitly. The translation rule is greatly simplified by the absence of GOTO's and non-local variables, and by incorporating additional semantic constraints.

An example TDFL program is given in Figure 2 to illustrate a module definition in the language. The program deletes all integers from the input stream "in" which are multiples of the input integer "base".

```
delete : rmodule (base:int, in:st int; out:st int)
        if empty (in) then [ ] -> out
                else get (in) -> head: int, tail:st int;
                        mod (head, base) -> residue: int;
                        delete (base, tail) -> temp: st int;
                        if residue = 0
                            then temp -> out
                            else cons-s (head, temp) -> out
                            end
                end
        mend
```

Figure 2. An example TDFL program.

An identifier in TDFL can be of type integer (int), Boolean (bool), stream of integer (st int), or stream of Boolean (st bool). A module definition is of the form:

<name>: module (<inputs>; <outputs>) <body> mend

or

<name>: rmodule (<inputs>; <outputs>) <body> mend

The rmodule is used to define a recursive module, such as the module "delete" of Figure 2.

Each statement of a TDFL program defines an identifier (or identifiers) in the sense that the identifier appears either as the target of an assignment, as an output of a module call, or is defined in both branches of a conditional statement. (If an identifier is defined in only one branch of a conditional statement, it is considered local to that branch; for instance, the identifiers "head", "tail" and "residue" are considered local to the else branch of "delete" and are not defined outside of the conditional.) The semantic constraints require that all referenced identifiers must be defined by some preceeding statements. Furthermore, the single assignment rule requires that no identifier be defined twice within the same syntactic unit <body>.

An integer (or Boolean) stream is denoted by an ordered sequence of integers (Boolean values) enclosed by brackets. Examples of stream constants are [1, 2, 3] and [true, false, false]. An empty stream is denoted []. Possible operations on a stream x are:

empty (x)
first (x)
rest (x)
cons-s (y, x)
get (x)

The predicate empty (x) tests if x is an empty stream; if so, its value is true, otherwise false. The operations first (x), rest (x), and get (x) are defined only on non-empty streams: first (x) yields the first item in the stream x; rest (x) yields the

stream x with the first item removed; and get (x) returns two outputs corresponding to first (x) and rest (x). The operation cons-s (y, x) requires y to be of a type corresponding to the type of the stream x. The result of cons-s (y, x) is the stream x with y attached to the beginning.

To illustrate the exploitation of parallelism in stream-oriented computation, we present in Figure 3 an example program that generates a stream of primes using the seive of Eratosthenes. The module "generate" in Figure 3 produces a stream of integers consisting of a '2' followed by all odd integers between two and 'n'. The module "delete_np" simply performs the deletion of non-prime numbers by removing the first item in the input stream and using it as the base to remove all multiples through calls to "delete". The main module "prime" comprises simply invocations of the "generate" and "delete_np" modules.

The structure of the program is readily seen in a snapshot of the computation (Figure 4) in which gates and other Boolean operators are not shown for simplicity. Note that parallelism is exhibited by the concurrent firing of data flow operators in different activations of the module "delete_np".

The language TDFL provides a program construct, called a perform-group, for representing a set of modules communicating in an arbitrary configuration. An example of such a system is shown in Figure 5. The module S_1 receives an integer value on one input and a stream value x on the other input. The module S_2 receives a stream y and produces two streams, x and output. When properly designed, the modules S_1 and S_2 cooperate by passing integer values along the "channels" x and y. The system S of Figure 5 can be expressed in TDFL as follows:

```
S: perform (input:int; output:st int)
        S1 (input, x:st int; y:st int);
        S2 (y:st int; x:st int, output)
        pend
```

In a system with cyclic connections, undesirable properties such as "hung-up" (in the sense that a subset of modules may be waiting indefinitely for some inputs) may exist. From the research in marked graphs we have been able to define a subset of the modules for which these properties are decidable, and furthermore, can translate the cyclically connected system into recursive data flow schemas without cycles. Examples of programming applications for this restricted subset of interconnected systems can be seen in such areas as digital filtering.

H. DATA FLOW COMPUTER ARCHITECTURE

Two approaches to the architecture of a processor for data flow programs have been investigated during the past year. One approach, which was initially described in last years report [27], derives from conventional multiprocessor organization: each "activation processor" of the machine performs the execution of one activation of a data flow procedure held in its local memory. Procedure call instructions cause the

(a)

```
generate : module (n:int; out:st int)
           if n < 2 then [ ] -> out
                 else every_other (3,n) -> odd_seq;
                      cons-s (2, odd_seq) -> out
                 end
           mend
```

(b)

```
every_other :  rmodule (lb:int, up:int; out:st int)
               if lb > up then [ ] -> out
                     else lb + 2 -> next;
                          every_other (next,up) -> temp;
                          cons-s(lb, temp) -> out
                     end
               mend
```

(c)

```
delete_np : rmodule (in:st int; out:st int)
            if empty (in) then [ ] -> out
                  else get (in) -> prime, tail;
                       delete (prime, tail) -> new;
                       delete_np (new) -> temp:st int;
                       cons-s (prime, temp) -> out
                  end
            mend
```

(d)

```
prime : module (input:int; prime_stream:st int)
        generate (n) -> integer_stream:st int;
        delete_np (integer_stream) -> prime_stream
        mend
```

Figure 3. An example program to generate a stream of primes.

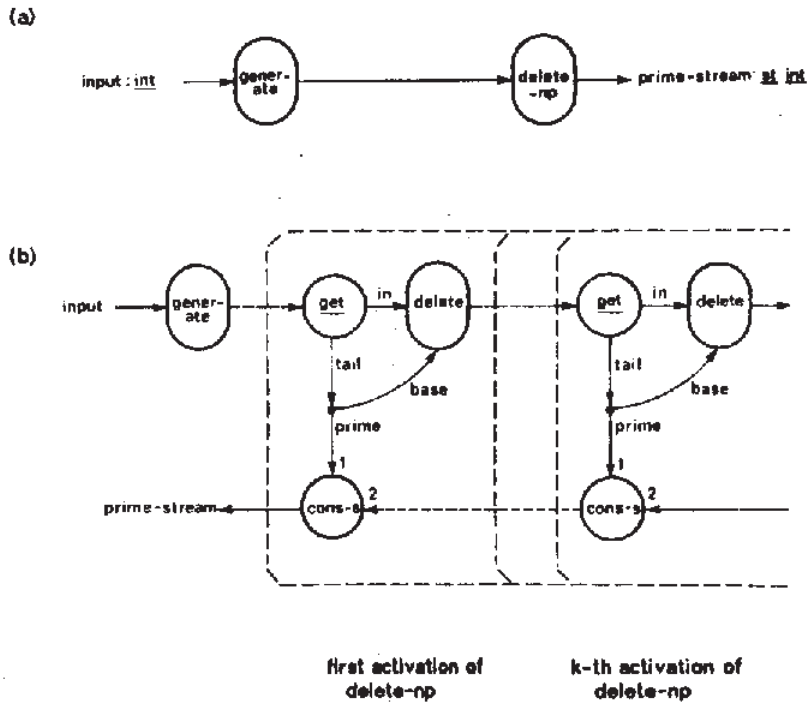


Figure 4. A snapshot of "prime."

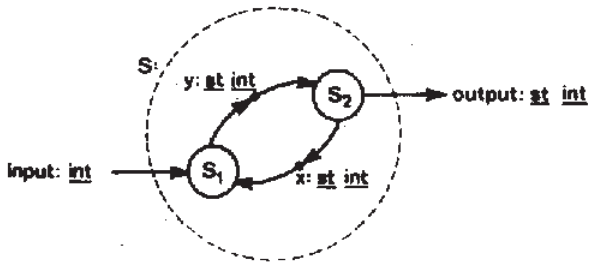


Figure 5. An example of a system of communicating modules.

creation of new activations in other processors. Operations on data structures are performed by specialized processing units in response to requests from activation processors. Since data flow procedures have no side effects, the activation processors may operate independently without need for synchronization. Furthermore, buffer memories may be readily incorporated into the activation processors -- there is no "multiprocessor cache problem." The behavior of this machine is specified by a description in a formal description language, and the machine has been shown to correctly implement the data flow language [28].

The second approach to a general data flow processor is through generalization of the concepts developed earlier for processors that implement restricted versions of the data flow language [29, 30]. These machines do not have processors in the usual sense, as the functions of memory and of instruction sequencing and decoding are intermixed in the units of the machine. The extensions developed by D. Misunas incorporate a general class of data structures and a form of procedure invocation [31].

A data structure within the data flow processor is represented as an acyclic directed graph having one root node with the property that each node of the graph can be reached by a directed path from the root node. A node of the graph is either a structure node or an elementary node. A structure node serves as the root node for a substructure of the structure and represents a value which is a set of selector-value pairs

$$\{(s_1, v_1), \ldots, (s_n, v_n)\}$$

where

$s_i \in$ {integers} $\cup$ {strings}

$v_i \in$ {elementary values} $\cup$ {structure values} $\cup$ {nil}

and s_i is the selector of node v_i. An elementary node has no emanating arcs; rather, an elementary value is associated with the node. A node with no emanating arcs and no associated elementary value has value {nil}.

A structure value is represented by a data token carrying a pointer to the root node of the structure. In Figure 6 the structure α_1 contains three elementary values a, b, and c, designated by the simple selector L and the compound selectors R·L and R·R respectively. Structure node α_3 of structure α_1 is shared with structure α_2 and is designated by a different selector in α_2 than in α_1.

The data-flow program of Figure 7 transposes the elements of the four-element structure presented on its input. Initially, the input link of the program is enabled and, upon firing, creates four copies of the token conveying a pointer to structure α and places the copies on the inputs of the four select operators. Each select operator retrieves the value (either an elementary value or a structure value) at the end of the path specified as its argument. The resulting value is associated with a token placed on the output arc of the operator.

A construct operator is enabled when it has a token on each input arc and, upon firing, creates a new structure of the values associated with the input tokens. In the program of Figure 7, each input arc of a construct operator is labelled with a symbol designating the selector to be associated with that input in the resulting structure.

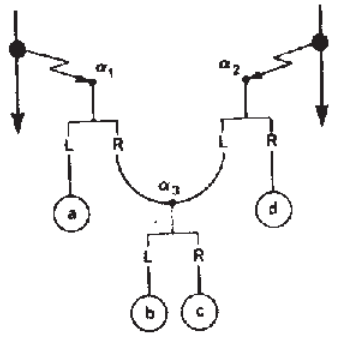


Figure 6. An example of two structures sharing a common substructure.

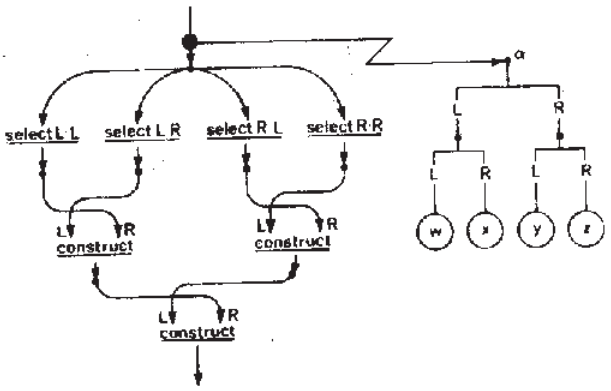


Figure 7. A simple data-flow program to transpose a four-element structure.

The processor described in [31] is composed of two sections, one of which performs instruction processing in a manner similar to that presented previously in [29] and [30], and the other which stores data structures and performs operations upon the data structures it contains. The structure processing section shown in Figure 8 consists of a Structure Operation Unit and a Structure Memory with attendent Distribution and Arbitration Networks. This section of the processor is viewed as a functional unit by the instruction processing section; that is, instructions specifying structure operations are sent to the section, and any resulting values are returned to the instruction processing section.

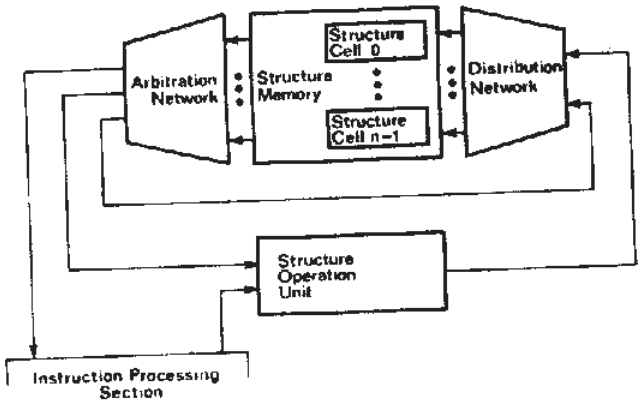


Figure 8. Organization of the structure processing section of the data-flow processor.

The Structure Memory is composed of a number of Structure Cells, each capable of holding one node of a structure. Commands specifying structure operations are received by the Structure Cells from the Structure Operation Unit. The Structure Operation Unit interprets instructions received from the instruction processing section to issue the correct sequence of commands. The Distribution Network provides the means for distributing commands among the individual Structure Cells, whereas the Arbitration Network serves to collect any results. The details of operation of such a structure processing section are described in [32].

The study of procedure invocation in a data-flow processor has proved to be quite interesting. Initial investigations yielded the implementation presented in [31]. This implementation assumes a data flow processor with multi-level memory, such as the basic machine described in [30]. An activation of a procedure is uniquely designated by the identifier of its argument structure. One copy of each procedure is maintained in the lower level memory, and the identifier of an instruction in a particular activation is formed of the activation identifier and the instruction identifier in the one copy. This combined identifier is used to map instructions into the Instruction Cells for execution, hence there is no interference within the Instruction Cells between separate activations of a procedure. The possibility of conflict arises, however, due to the fact that the Instruction Cells must act as a cache and, when full, may have to return some instruction to the lower level memory. It is then necessary to allocate space for the discarded instruction in that memory and to note that it was discarded from the cache so the next time a reference is made to the instruction, the correct copy is readily retrieved.

Data flow procedures in the processor are represented as acyclic directed graphs; that is, as data flow data structures, allowing both the instruction processing section and the structure processing section to be structured with multi-level memory

systems, sharing a common lower-level memory.

I. PACKET MEMORY SYSTEM ARCHITECTURE

The data flow processors just described are composed of many units that operate asynchronously and communicate only through transmission of information packets. We have found that this architectural principle also leads to attractive organizations for large memory systems capable of simultaneously processing large numbers of concurrent memory transactions.

Just as the function of a data flow processor is specified by the formal semantics of a data flow language, a packet memory system is specified by a formal memory model which defines the kinds of transactions that may be requested of a memory system, and defines the change in memory system state and the information to be returned in response to a request. Memory models under consideration have been designed as specifications of memory systems for use with the data flow processors described above.

The memory system shown in Figure 9 is connected to a processing system by four channels. Command Packets sent to the memory system at port cmd are requests for memory transactions, and specify the kind of transaction to be performed. Items to be stored are presented as Store Packets at port store, and items retrieved from storage are delivered as Retrieval Packets at port rtr. A unique identifier is associated with each item stored in the memory system, and these identifiers are used in command and data packets to reference items held by the memory. Unique identifiers not currently bound to stored items are available at the unid port of the memory system for allocation to items entered in the memory system by store commands.

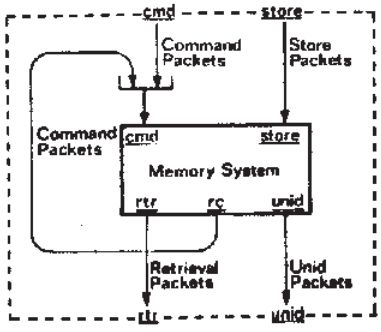


Figure 9. Structure of a memory system.

Two ways of structuring memory systems from independent memory systems using asynchronous packet communication have been developed. Modular Structure (Figure 10) is a way of organizing a packet memory system as a collection of many

simpler packet memory systems to which requests for items are routed by sorting them (in a Distribution Network) according to some property of the item identifier. Such an organization supports highly concurrent processing of memory transactions by the separate memory modules. Hierarchical Structure (Figure 11) organizes a memory system as two or more packet memory systems that represent different trade-offs of speed, capacity and cost, and arranged so that information is automatically redistributed among levels of the memory according to intensity of access. We have shown how correctness of a complete memory system (i.e. that it implements the specified formal memory model) follows from the structure of the memory system and the specifications of its constituent subsystems [33].

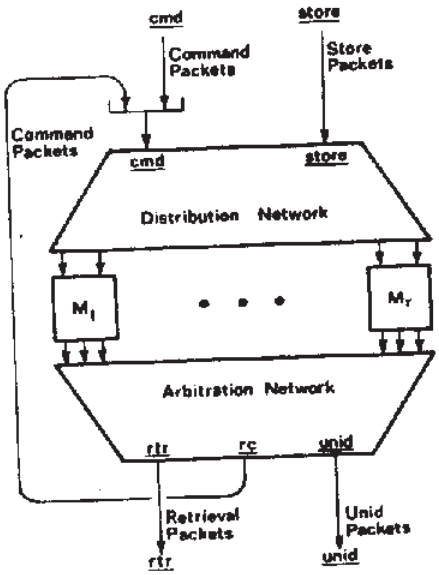


Figure 10. Modular memory system structure.

J. EVALUATION OF ARCHITECTURAL CONCEPTS

Enough interesting architectural proposals have emerged from our recent work that it is essential to begin evaluation of their soundness and performance potential. After considering alternative approaches, we concluded that the modelling of the physical structure of a packet communication system by microprocessors is a feasible means of simulation, and is worthy of investigation.

The simulation facility shown in Figure 12 is composed of a host computer, a number of microcomputer modules, each consisting of a microprocessor and a number of

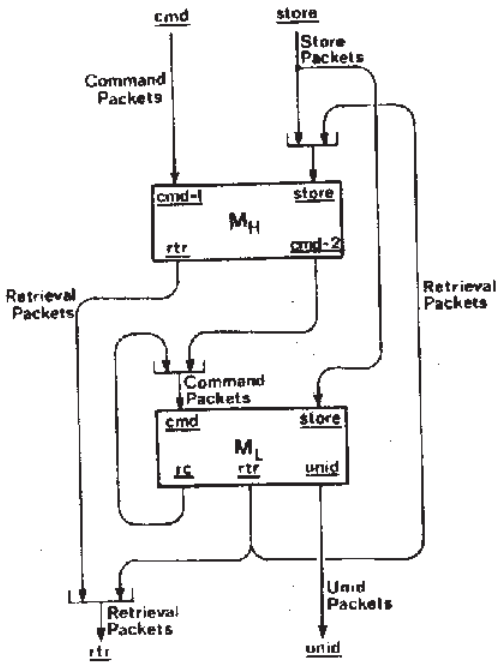


Figure 11. Hierarchical memory system structure.

memory modules, a control bus for host-microcomputer communication, and a routing network for transmitting packets between microcomputer modules. The host computer loads simulation programs into microcomputer modules, monitors and controls the progress of a simulation, and collects statistical data for performance evaluation. The control bus transmits commands, addressing information and data from the host to the microcomputer modules, and transmits acknowledge signals and memory word contents from the microcomputer modules to the host. Under control of the host, microcomputer modules execute programs which simulate the operation of units of a simulated system. In addition to communicating with the host via the control bus, each microcomputer module is connected by an input port and an output port to the routing network, through which the module sends or receives packets to or from other modules during the course of a simulation.

Concurrent with the development of the hardware for the simulation facility, we are designing an Architecture Description Language for the description of Packet Communication Systems. A system to be simulated upon the facility is described in the Architecture Description Language, and that description is translated into microprocessor object code for execution upon the simulation facility. It is intended

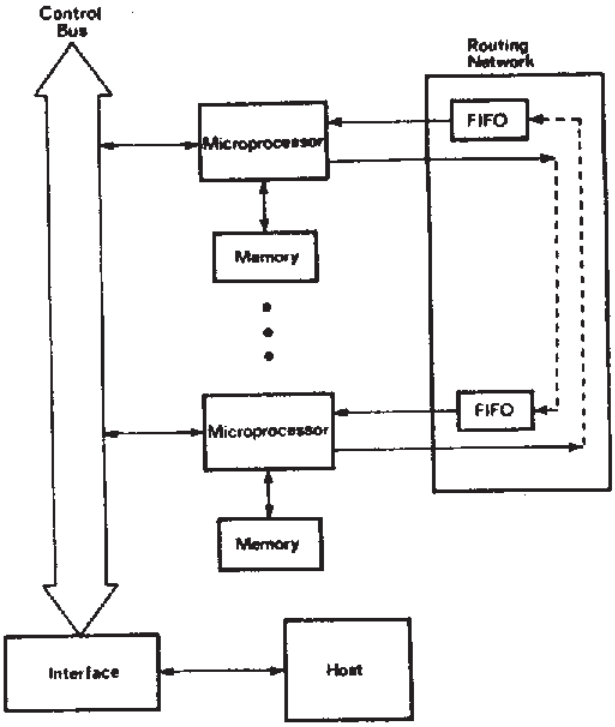


Figure 12. Organization of the simulation facility.

that the Architecture Description Language also serve as a formal description of a Packet Communication System for design purposes.

REFERENCES

1. Liskov, B.H., "A Design Methodology for Reliable Software Systems," 1972 Fall Joint Computer Conference, AFIPS Conference Proceedings, volume 41, 1972.

2. Parnas, D.L., "On the Criteria to be Used in Decomposing Systems Into Modules," Communications of the ACM, volume 15, number 12, December 1972.

3. Liskov, B.H., and S. Zilles, "Programming With Abstract Data Types," Proceedings of the ACM Conference of Very High Level Languages, SIGPLAN Notices, volume 9, April 1974.

4. Liskov, B.H., A Note of CLU, Computation Structures Group Memo 112-1, Laboratory for Computer Science, M.I.T., November 1974.

5. Parnas, D.L., "Information Distribution Aspects of Design Methodology," Proceedings IFIP Congress, August 1971.

6. Wegbreit, B., "The Treatment of Data Types in EL1," Communications of the ACM, volume 17, number 5, 1974.

7. Galley, S.W., and G. Pfister, The MDL Language, Programming Technology Division Document SYS.11.01, Laboratory for Computer Science, M.I.T., in progress.

8. Liskov, B.H., and S.N. Zilles, "Specification Techniques for Data Abstractions," IEEE Transactions on Software Engineering, volume SE-1, number 1, March 1975.

9. Parnas, D.L., "A Technique for the Specification of Software Modules with Examples," Communications of the ACM, volume 15, number 5, May 1972.

10. Zilles, S.N., Algebraic Specification of Data Types, Computation Structures Group Memo 119, Laboratory for Computer Science, M.I.T., March 1975.

11. Guttag, J.V., The Specification and Application to Programming of Abstract Data Types, Ph.D. thesis, University of Toronto, CSRG-59, 1975.

12. Robinson, L., K. Levitt, P. Neumann and A. Saxena, "On Attaining Reliable Software for a Secure Operating System," Proceedings of the International Conference on Reliable Software, SIGPLAN Notices, volume 10, number 6, June 1975

13. Parnas, D.L. and G. Handzel, More on Specification Techniques for Software Modules, Fachbereich Informatik, T.H. Darmstadt, 1975.

14. Dijkstra, E.W., "Notes on Structured Programming," Structured Programming, APIC Studies in Data Processing Number 8, Academic Press, New York, 1972.

15. Wirth, N., "Program Development by Stepwise Refinement," Communications of the ACM, volume 14, number 4, April 1971.

16. Peterson, J. L., Modelling of Parallel Systems, Ph.D. Thesis, Department of Electrical Engineering, Stanford University, Stanford, Calif., December 1973.

17. Hack, M., Petri Net Languages, Technical Report TR-159, Laboratory for Computer Science, M.I.T., in progress.

18. Agerwala, T., A Complete Model for Representing the Coordination of Asynchronous Processes, Hopkins Computer Research Report 32, Johns Hopkins University, Baltimore, Maryland, July 1974.

19. Hack, M., The Equality Problem for Vector Addition Systems is Undecidable, Computation Structures Group Memo 121, Project MAC, M.I.T., April 1975.

20. Hack, M., Decidability Questions for Petri Nets, Ph.D. Thesis, Department of Electrical Engineering and Computer Science, M.I.T., June 1975.

21. Keller, R. M., Vector Replacement Systems: A Formalism for Modelling Asynchronous Systems, TR117, Computer Science Laboratory, Princeton University, Princeton, N.J., December 1972.

22. Hack, M., Analysis of Production Schemata by Petri Nets, Technical Report TR-94, Project MAC, M.I.T., February 1972.

23. Project MAC Progress Report XI, 1973-1974, Project MAC, M.I.T., pp.71-78.

24. Qualitz, J. E., Equivalence Problems for Monadic Schemas, Technical Report TR-152, Project MAC, M.I.T., June 1975.

25. Leung, C. K. C., Formal Properties of Well-Formed Data Flow Schemas, Technical Memorandum 66, Project MAC, M.I.T., June 1975.

26. Dennis, J. B., First Version of a Data Flow Procedure Language, Technical Memorandum 61, Project MAC, M.I.T., May 1975. Also, Proceedings of Symposium on Programming, Institut de Programmation, University of Paris, France, April 1974, pp. 241-271.

27. Project MAC Progress Report XI, 1973-1974, Project MAC, M.I.T., pp.86-90.

28. Rumbaugh, J. E., A Parallel Asynchronous Computer Architecture for Data Flow Programs, Technical Report TR-150, Project MAC, M.I.T., June 1975.

29. Dennis, J. B., and D. P. Misunas, "A Computer Architecture for Highly Parallel Signal Processing," Proceedings of the ACM 1974 National Conference, ACM, New York, November 1974.

30. Dennis, J. B., and D. P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor," Proceedings of the Second Annual Symposium on Computer Architecture, IEEE, New York, January 1975.

31. Misunas, D. P., A Computer Architecture for Data-Flow Computation, S.M. Thesis, Department of Electrical Engineering and Computer Science, M.I.T., June 1975.

32. Misunas, D. P., "Structure Processing in a Data-Flow Computer," Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing, IEEE, New York, August 1975.

33. Dennis, J. B., "Packet Communication Architecture," Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing, IEEE, New York, August 1975.