LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

ADL:An Architecture Description Language for Packet Communication Systems

Computation Structures Group Memo 185
October 1979

Clement K-C. Leung

An abbreviated version of this paper was published in the Proceedings of the 1979 International Symposium on Computer Hardware Description Languages and Their Applications.

This research was supported by the National Science Foundation under grant MCS-7902782, and by the University of California, Lawrence Livermore Laboratory under contract 8545403.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Abstract

A packet communication system (PCS) consists of a number of modules which communicate only by sending information packets to each other. Modules in a PCS can operate concurrently in an asynchronous fashion. ADL is an architecture description language designed for studying PCSs, and is intended to serve as a medium for system documentation and human communication, as a formalism for design verification, and as a language interface to a design automation and simulation facility. It complements many existing computer hardware description languages in that it is designed for describing the system structure and algorithmic behavior of a computer architecture. The features of ADL include a type facility for defining system structure; the adoption of data-flow as a basis for its operational semantics; state variables for implementing functions on data streams; and monitors for sharing data objects, input streams and output streams.

1. Introduction

Several commonly used descriptive tools in computer hardware design are listed in Table 1. Currently there is also much interest in hardware description languages and their applications to hardware design and construction.

Descriptive Tool	Information Content	Usage
Logic Diagram	physical layout of logic elements	hardware construction maintainence
State Diagram	gross dynamic system behavior	subsystem design with SSI
Register Transfer	sequencing control and data paths	subsystem and system design with MSI
System Block Diagram	bus structure, memory modules, I/O controllers	system design with LSI

Table 1 Levels of Hardware Description

We are interested in these descriptive tools from the point of view of a computer architect. In particular, we are studying processor organizations [4] appropriate for executing data-driven computations and are interested in expressing our design proposals concisely and precisely. We find most available descriptive tools inadequate: some are not completely defined; others seem to restrict us to express our designs assuming certain implementation strategies when we are not ready to be so committed. In view of these inadequacies, we have formulated the following two

guidelines for the design of an *architecture description language*, ADL:

- provide adequate support for the top-down or bottom-up design of multi-level systems
- based on a suitable semantic model that facilitates understanding and development of verification and automated implementation techniques

This paper reports on the current status of this language design effort. It describes a set of language features that we have found useful in describing the various modules of a basic data-flow processor [4]. These features provide a basis for designing a complete architecture description language ADL. Since we are interested in evaluating its potential as a basis to guide implementation, ADL is also designed to allow natural expression of the inherent parallelism in a chosen algorithm.

2. Packet Communication Systems

A packet communication system (PCS) consists of a number of hardware modules which communicate only by sending information packets to each other. Modules in a PCS are intended to operate concurrently in an asynchronous fashion, which may be supported by the adoption of a hand-shaking asynchronous communication protocol for packet transmission in a hardware implementation.

The class of packet communication systems is closed under interconnection. A number of PCS's can be connected together to construct another PCS. A PCS can also be decomposed into an interconnection of PCS's. This closure property is useful for hierarchical structuring of PCS's.

An example of a PCS is shown in Figure 1. The architecture unit illustrated in this figure consists of a number of identical arithmetic logic units managed by a control unit. Requests for arithmetic processing arrive at the controller in the form of *operation packets*. Each operation packet contains the opcode, the operands and addresses of other units to which results of the operation should be returned. The controller receives operation packets, dispatches them to free arithmetic logic units, and generates *result packets* by tagging the results returned by the arithmetic logic units with the appropriate destination addresses. We shall use this example to illustrate the various ADL constructs introduced in later sections. For convenience we shall refer to this unit as an arithmetic logic processor (ALP).

We note that in designing ALP, the following levels of description are useful:

(i) the input/output behavior of ALP, in terms of the operation packets received and the result packets generated;

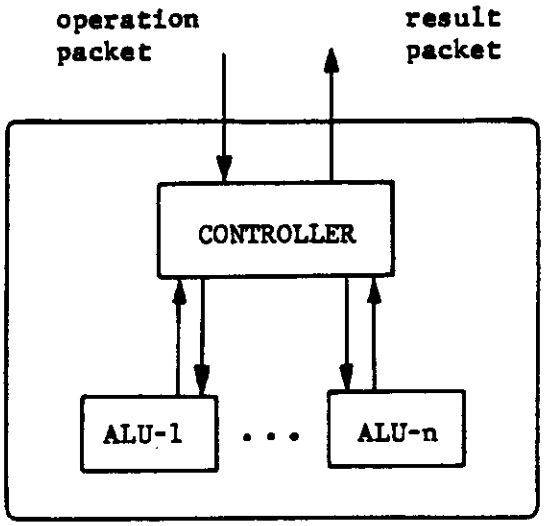


Figure 1. The Arithmetic Logic Processor

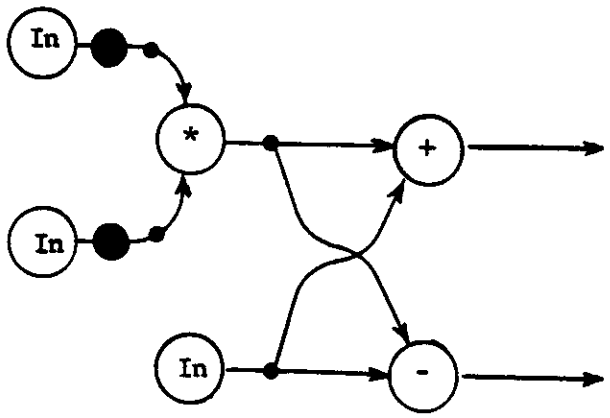
(ii) the decomposition of ALP into a controller and several arithmetic logic units, together with the input/output behavior of each of these submodules;

(iii) any further refinement on the controller and the arithmetic logic units.

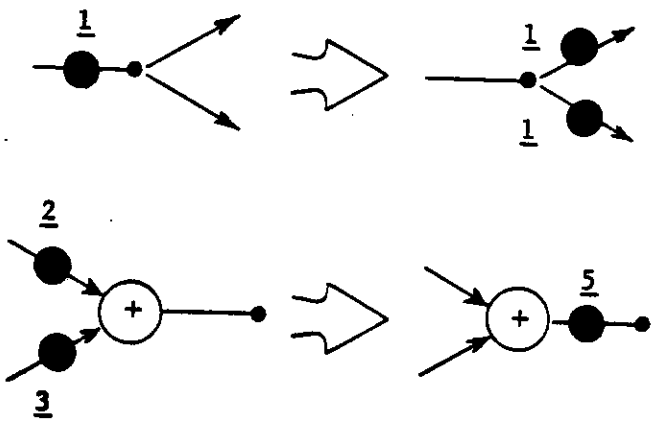
ADL allows precise description at each of these levels. In addition, the information contents of an operation packet and a result packet can also be defined explicitly.

3. Data Flow Concepts

Data flow provides an operational semantics for algorithmic behavior in ADL. An algorithm in data flow form is expressed as a directed bipartite graph. The two types of nodes are links and operators. An example of a data flow graph is shown in Figure 2a.



(a) A data flow graph



(b) Enabling conditions and firing rules for links and operators

Figure 2. Data Flow Semantics

To initiate a computation, inputs are placed on the input arcs of a data flow graph (Figure 2a). A node is **enabled** if its **enabling conditions** are met. An enabled node may **fire**. The enabling conditions and firing rules for links and data operators are shown in Figure 2b. A set of control operators, whose duty is to route data according to data-dependent decisions and to support iteration, are shown in Figure 3. A T-gate passes a data token from its data input arc to its output arc when it receives the value **true** on its control input arc. It will absorb a data token from its data input arc and place nothing on its output arc if it receives the value **false**. An F-gate has similar behavior, but with the sense of the truth value reversed. A merge actor has T- and F- data input arcs, and a truth value input arc. When a truth value is received, the merge actor places a token on its output arc bearing the next data value received on the corresponding data input arc. A token on the other data input arc is unaffected. Computation proceeds by firing enabled nodes. Outputs of the computation are available from the output arcs of a data flow graph. A conditional construct and an iteration construct are shown in Figure 4. The reader is referred to [2] for a more detailed discusssion on data flow

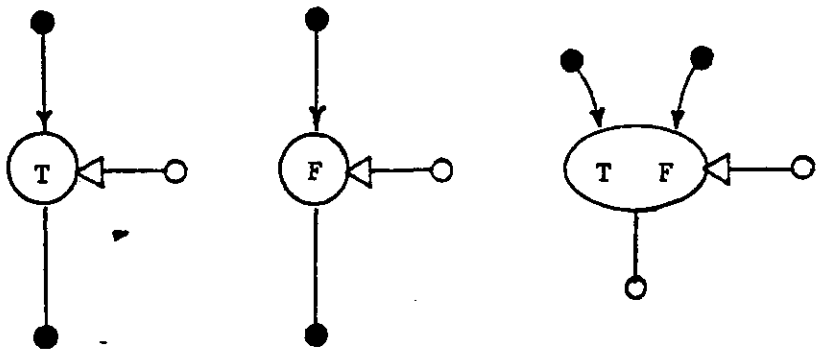
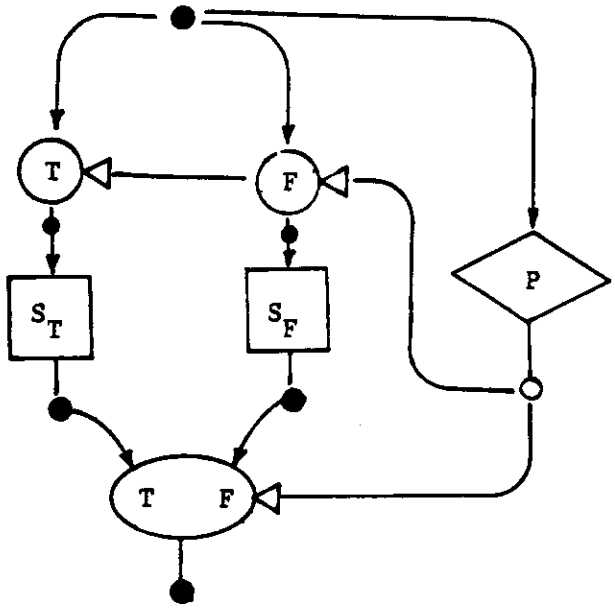
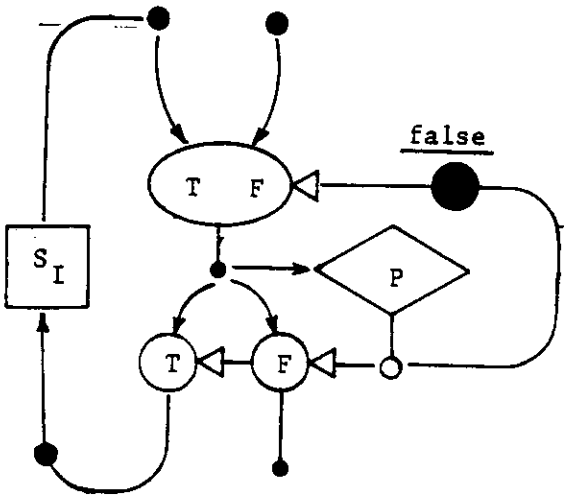


Figure 3. Actors for Data Routing



(a) A Conditional Schema



(b) An Iteration Schema

Figure 4. Branching and Iteration in Data Flow Graphs

concepts.

An important point to note about the firing rule for data flow graphs is that no actor is enabled unless its output arc is empty. We also note that at each firing of a data link with multiple output arcs, a **copy** of the data (elementary or structured) on its input arc is made and put on each output arc. This copying operation eliminates a source of obscure side-effects.

The flow of data from operator to operator in a data flow graph and the flow of packets from module to module in a PCS are different views of the same fundamental activity. This observation prompts us to define a connection between PCS modules as a data link connecting an output arc of one module to an input arc of another module. We retain this dichotomy to distinguish between intermodule and intramodule activities in the following discussion.

4. Behavior Description in ADL

There is presently much interest in the development of module specification techniques [8], but no obvious candidate has yet emerged for incorporation into an architecture description language. Behavior description in ADL is constructed instead from expressions, state variables, monitors and input/output activities. We will present these constructs in their textual form. Each is designed so that its semantics can be vigorously specified in data flow. We emphasize that behavior descriptions should be organized conceptually in terms of expression evaluation, interaction between state variables and input/output, and sharing via monitors. The data flow graph formalism is useful for defining these concepts more vigorously and, as discussed in the last section, as an intermediate representation for translation into hardware.

ADL is expression oriented. In describing hardware, it is appropriate to view an ADL expression as a hardware submodule which receives inputs and generates outputs. The most general form of expression is shown in Figure 5. An expression may receive as input: an input packet, an output from other expressions, a component of the current module state, or a component of a data object which it shares with other expressions, accessed via monitor procedures. (Module state and monitors are further explained later in this section.) An output of an expression may be used: as an input to other expressions, as an output packet of the module, to update a component of the module state, or to update a component of a shared data object via a monitor procedure.

ADL also allows user-defined functions. In hardware description a user-defined function is analogous to a macro in an assembly language. Viewing an expression E as a hardware submodule S, a call to function F in E simply denotes that S contains a copy of the hardware denoted by F. Expressions containing calls to the same function do not share any hardware.

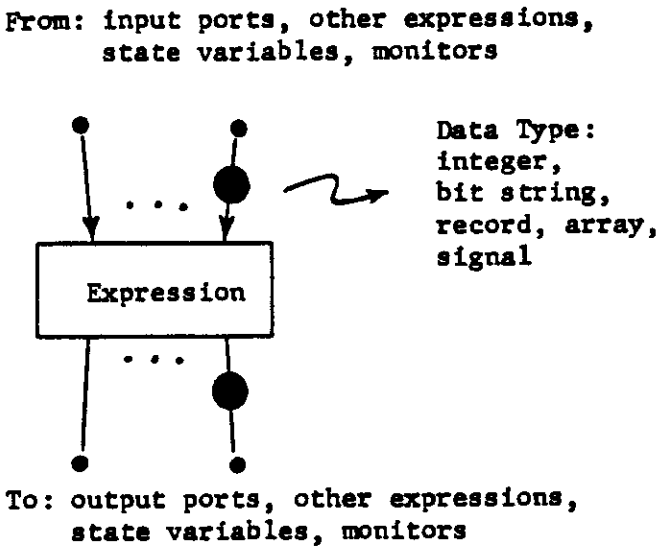


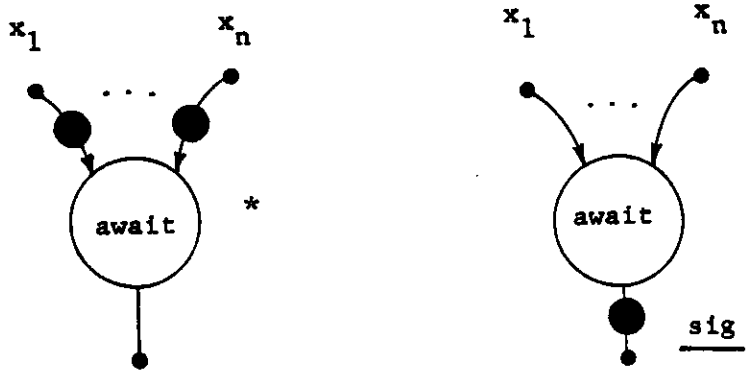
Figure 5. A general expression

Sharing in ADL is achieved through a single mechanism: monitors.

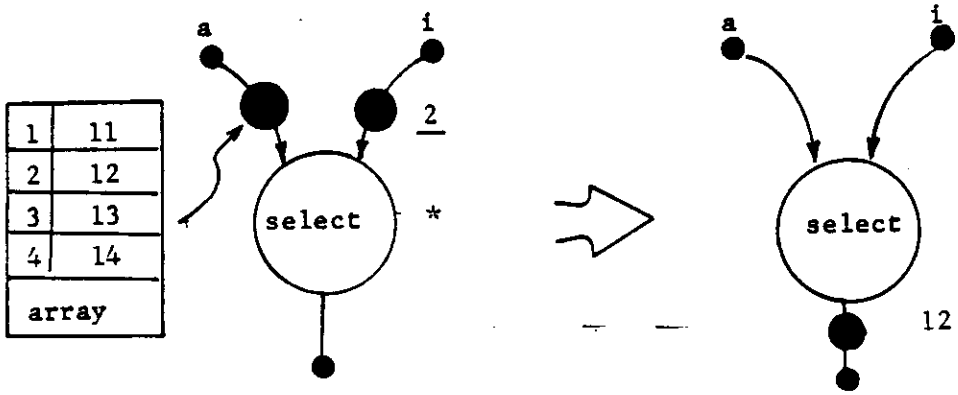
In Figure 5 we have also listed the available data types. We assume the usual primitive operators for the more common data types. In Example 2 we will also use the **await** operator (Figure 6), which delivers a **signal** upon firing. The **acons** and **rcons** operator construct arrays and records from their components, while the **mod** operator is convenient for constructing new structured values from old ones.

4.1 Expressions and Link Definitions

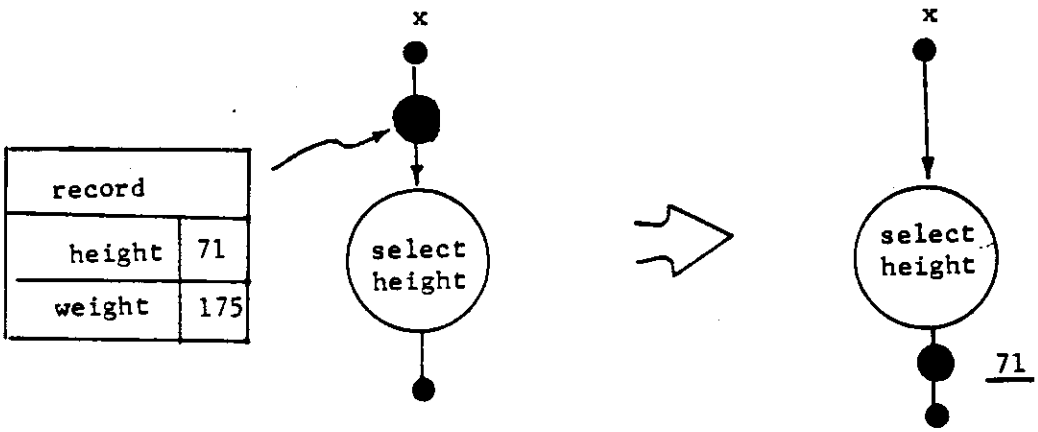
Module behaviors or algorithm specifications are synthesized from expressions. In many algorithmic languages, results of expression evaluations are stored in cells from which they are later retrieved for evaluating other expressions. In ADL, the result of an expression evaluation is made available for evaluating other expressions through links (as in data flow graphs). The



(a) **await** syntax : **await**(x_1, ..., x_n)

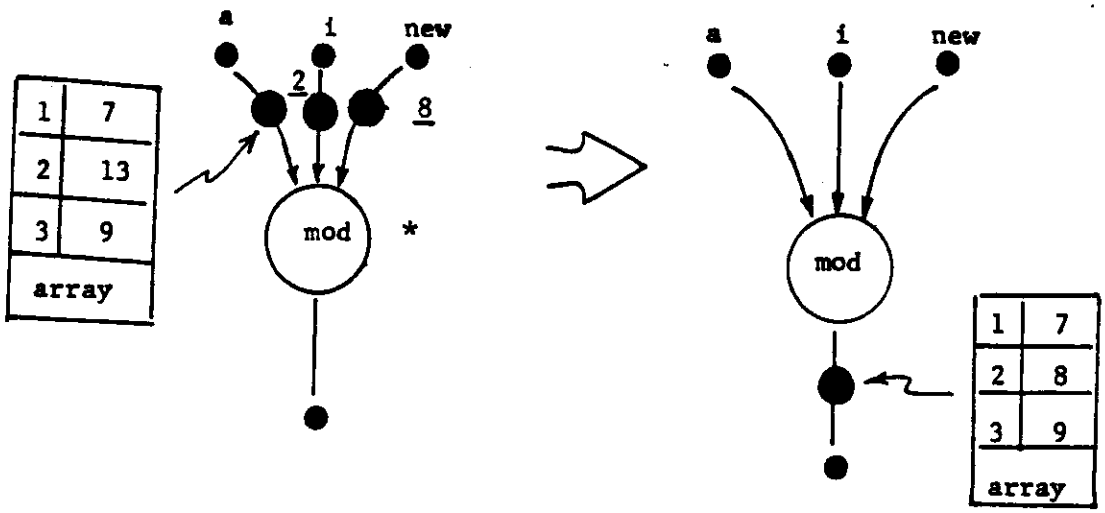


(b) array select syntax : a[i]



(c) record select syntax : x.height

Figure 6. ADL operators



(d) **mod** syntax : a **mod** [i : new]

Figure 6. (continued) ADL operators

---

output links are defined for expressions through **link definitions**. It is also convenient to allow a tuple of expressions in a **multiple link definition** specifying the generation of a set of outputs for each set of inputs. While we will not give precise rules for determining the scope of link identifiers, it is obvious that within such a scope, a link identifier must be **uniquely** defined. An example of link definitions and its data flow semantics are given in Figure 7.

In addition to simple expressions involving the application of primitive operators or user-defined functions to data objects, there are several types of **compound** expressions.

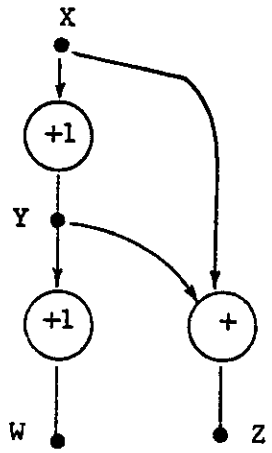
(i) conditional expression

Example

(a) if x > 1 then 2 else 3

(b) **if** destaddr-in-use **then**

rcons [result: res. result, destaddr: x]



Y: **integer** = X + 1;
Z: **integer** = X + Y;
W: **integer** = Y + 1;

Figure 7. Data flow semantics of link definitions

The semantics of (b) is that if the destination address is not in use (Section 2), no result packet will be generated, i.e. the expression evaluation produces no output object.

(ii) iteration expression

Example

for y: **integer** = 1, p: **integer** = N

if p = 0 **then iter** y, p - 1 **else** y

y and p are loop links which are not accessible outside the iteration expression. Evaluation of this expression returns the result of applying the following recursively defined function to 1 and N:

f(y,p) = **if** p=0 **then** f(y*p, p-1) **else** y

The iteration expression in its full generality is given in Appendix A.

(iii) case expression

A case expression is an extension of the condition expression, allowing the expression to take on different values in different cases.

Example

Let day be a link identifier of scalar type (monday, ..., sunday):

case day **of**

monday: <expression 1>;

...

sunday: <expression 7>;

otherwise: <expression 8>;

(iv) block expression

A block expression allows grouping together several expression and treating them as one single expression:

begin uses $inexp_1$, ..., $inexp_n$

<link definitions>

resultis $outexp_1$, ..., $outexp_m$

end

A block expression introduces a new scope for link declarations. Viewed as a data flow graph, it has the outputs of $inexp_1$,...,$inexp_n$ for input links and has m output links delivering the results of evaluating $outexp_1$,...,$outexp_m$.

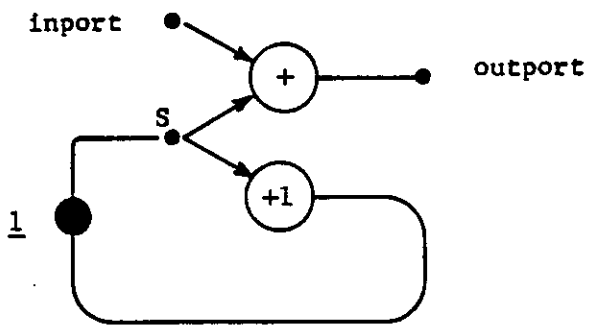
4.2 State Variables

State variables are necessary to implement functions from input sequences to output sequences. State variables are initialized, accessed and updated.

Suppose we want to implement the following function:

$f(x_i) = x_i + i$ where x_i is the i^{th} input to f

A straightforward implementation of this function uses a state variable which is initialized to 1, as shown in Figure 8.




```
state s: integer  := 1;
      outport ← s + inport;
update s  := s + 1;
```

Figure 8. Behavior Specification using State Variables

In an ADL behavior description, each state variable must be initialized, as follows:

state <state variable identifier>: <data type> := <initial value>;

State variables may be used in expressions, just as data objects available from data links. In a behavior description, however, each state variable is **updated** exactly once in an update statement:

update <state variable identifier> := <expression>

State variables in ADL are analogous to feedback variables in discrete-time systems. Structurally they are links on a feedback path. No new firing rules are necessary, but the above rules for their proper use must be added. The data flow graph for a behavior description utilizing state variables has the general structure shown in Figure 9.

This particular choice of semantics for state variables has the following attractive property:

If a state variable S is an input link of an expression e, then the $i+1^{st}$ evaluation of e is performed with a data object generated by evaluating the unique **update** expression of S for the i^{th} time. The data object generated by evaluating this update expression for the 0^{th} time is the initial value of S.

4.3 Resource Sharing and Monitors

In a PCS information, hardware and data streams may be shared. Information is shared when several hardware operations may access and modify the stored information. Hardware is shared when it is economically attractive to do so. Data streams may be shared in two different ways. A stream of input data may consist of resource requests to be serviced by a pool of

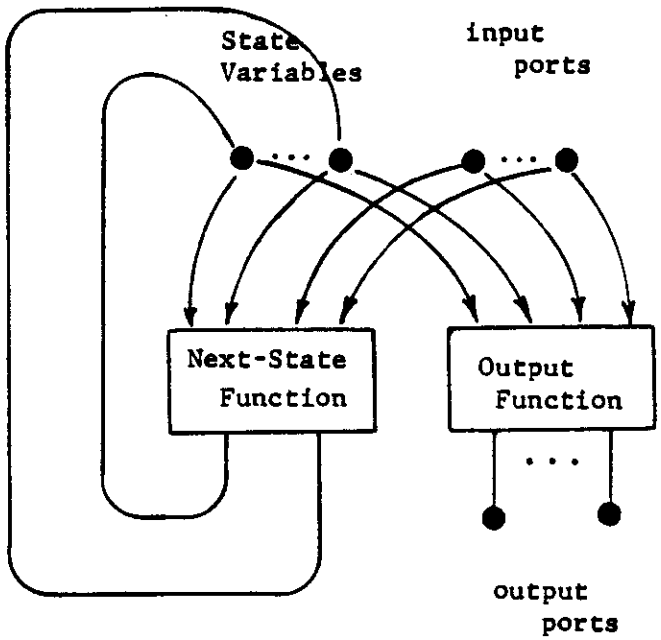


Figure 9. Structure of a Module using State Variables

hardware modules. A stream of output data may be the result of merging together the output streams of several modules. In each of these cases, it is necessary to provide some mechanism for arbitrating among competing requests and to allow implementation of efficient and consistent sharing policies. At present we propose a monitor mechanism for handling sharing [5].

A monitor in ADL consists of the shared objects, a set of user-defined procedures and an arbitration mechanism. The shared object may only be accessed via these procedures. It is sufficient to use a nondeterministic **merge** operator in addition to other operators previously given to give a data-flow semantics to the monitor construct as proposed. The details are given in Appendix B. An example using monitors for data stream sharing is given in Section 5. More examples can be found in [6].

In this section we give a simple example to illustrate data sharing with monitors (Figure 10). Information is shared through the monitor state variable *sum*, which is initialized to 0. Two monitor procedures are provided for manipulating it: *add* and *sub*. *Add* is invoked to add elements of a data stream to *sum* while *sub* is invoked to subtract elements of another data stream from *sum*. Hardware operations performed by *add* and *sub* are serialized. While an implementation of a monitor may allow monitor procedure invocations to be overlapped, or pipelined, the net effect of interaction with the shared resource must be as if such invocations

---

```
                    . . .

m: monitor
   state sum: integer := 0; /* initialized to 0 */

      /* procedure to increment sum */
      add: procedure(n: integer) result ( integer )
            x: integer = sum + n;
            update sum := x;
            return x
       end add;

      /* procedure to decrement sum */
      sub: procedure (n: integer) result ( integer );
            x: integer = sum - n;
            update sum := x;
            return x
       end sub;
end m;

                    . . .

i: integer = I1;
j: integer = m.add(i);

                    . . .

p: integer = I2;
q: integer = m.sub(p);

                    . . .
```

Figure 10. An example of a monitor

take place one after another. The data flow semantics of this monitor is given in Figure B.4.

4.4 Module and Input/Output

At this point it is appropriate to introduce the notion of a behavior module type, which forms a unit of behavior description in ADL. A behavior module type defines the behavior of a hardware module using expressions, state variables and monitors, along with its external interface. This external interface consists of **input** ports and **output** ports through which the module receives and transmits packets, respectively. The type of packets transmitted at each port is also specified in a description.

The syntax for a complete behavior description, as encapsulated by a module type definition, is:

type m = **module**

inlet il: ptype i_1;

...

in: ptype i_n;

outlet kl: ptype k_1;

...

k0 ptype k_0;

<state variables, monitors, expressions>

end m;

Input and output ports are external data links through which a module receives and sends packets. Within a module, an input port may be used just like an output link of an expression or a state variable. In ADL packets are not distinguishable from data objects that are generated and manipulated within a module. We use the term "packet" to emphasize the fact

that it is transmitted across module boundaries.

The following syntax is used for outputing:

output-port ← expression;

Note that ":=" is used in state variable initialization and update, "=" is used in link definitions and "←" is used for outputing.

Input ports and output ports may also be shared by making them accessible only through monitor procedure. This is the operational meaning of data stream sharing alluded to above.

A behavior description of a PCS module P in ADL is a procedural specification of the input/output behavior of P. The gross behavior of P is as follows. Initially the state variables of P, if any, are initialized. As inputs are received, outputs are generated from the module state and the inputs, and new states are generated.

An ADL description that does not contain monitors exhibits *functional* behavior, i.e. the output sequences of the module are uniquely determined by the input sequences. This type of behavior is sufficient to characterize a large class of architectural units. Some architectural units of interest, typically those performing arbitration in resource allocation, exhibit non-functional behavior, supported in ADL through the use of monitors. Monitor procedure invocations produce side effects visible to subsequent monitor procedure invocations. A monitor procedure may be invoked at any point inside a behavior module, and is executed with exclusive access to the shared objects. It is most useful to construct monitors so that even though the order in which the shared objects of a monitor are accessed via monitor procedures is *non-determinate*, certain properties of these state variables are kept invariant.

The organization of a behavior description given in a module can be visualized as in Figure II. There are a number of *sections*. Each section receives input from its own set of input ports, generates output at its own set of output ports and maintains its own state. Sections are *independent* of each other in that the set of ports and states manipulated by differing sections are disjoint. In addition to sections, a behavior module may maintain shared data and ports through a set of monitors. These data and ports are shared by otherwise independent sections through the set of monitor procedures.

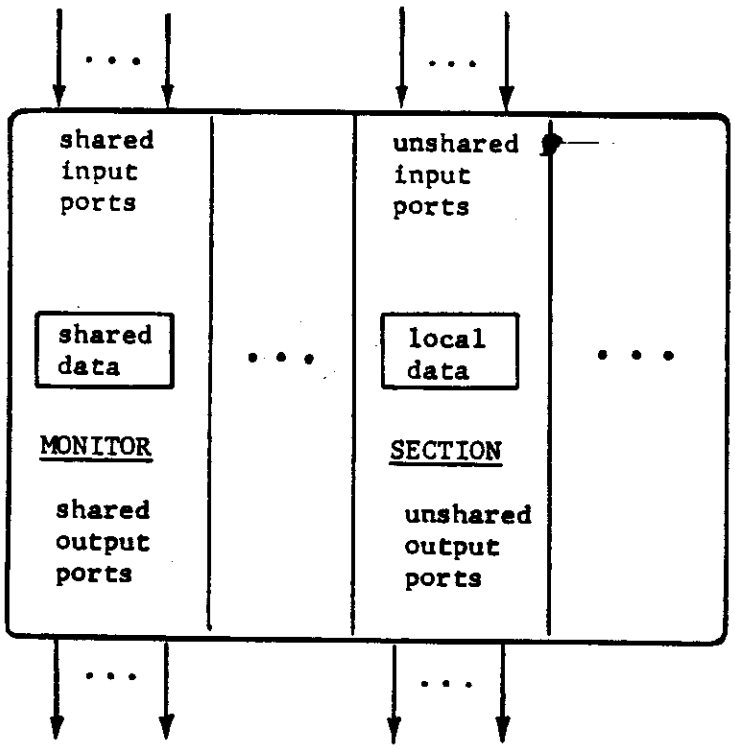


Figure II. Organization of a behavior description

5. Multi-level System Description

An architecture design is usually expressed as an interconnection of major subsystems: fixed-point arithmetic processors, floating-point arithmetic processors, memory section, interconnection networks. We have already noted that a PCS can be decomposed into an interconnection of modules each of which is in turn a PCS. We propose to support structural decomposition directly in ADL, and to use it as a basic step in design refinement. Structural decomposition is expressed in ADL by a **structure module type**:

type m = **module**

inlet i_1: ptype-i_1, ..., i_m: ptype-i_m;

outlet o_1: ptype-o_1, ..., o_n: ptype-o_n;

submodule S_1: mtype-S_1; ..., S_p: mtype-S_p;

connection

$S_{r1}.outputport_{j1} \rightarrow S_{q1}.inputport_{k1}$;

...

$S_{rx}.outputport_{jz} \rightarrow S_{qz}.inputport_{kx}$

end m

A structure module type defines:

(i) a list of input ports and a list of output ports, along with the type of packets transmitted through that port.

(ii) a list of submodules.

(iii) a list of connections. Each connection connects an input port of a submodule to an output port of another submodule. An input port (output port) of the defined module is also identified with an input port (output port) of a submodule by a connection.

An example defining the structural decomposition of an arithmetic-logic processor into a controller and a number of arithmetic-logic units is given in Figure 12. The **foreach** construct is a macro facility. Macro parameters in the text are marks by "*", e.g. *i in Figure 12.

Using the notions of behavioral description and structural decomposition, we can design PCS's in a more systematic manner. As a first step we can start by giving the input/output behavior of a PCS P to be constructed. This initial design can be further refined by giving a structural decomposition of P into an interconnection of PCS modules P_1, ..., P_n, together with a

```
type ALP = module
              inlet opn-in : opn-pkt;
              outlet res-out : result-pkt;
              param n-of-alu : integer;

    submod
      k : controller(n-of-alu)
        inlet opn-in : opn-pkt;
              alu-in[1..n-of-alu] : alu-res;
        outlet res-out : result-pkt;
              alu-out[1..n-of-alu] : alu-opn;
        end;

    foreach i : integer in [1..n-of-alu]
    { a*i : alu
          inlet ain :alu-opn;
          outlet aout : alu-res;
          end  };

    connections
      opn-in → k.opn-in;
      k.res-out → res-out;
    foreach i : integer in [1..n-of-alu]
    {  k.alu-out*i → a*i.ain;
       a*i.aout → k.alu-in*i;  }

end ALP;
```

Figure 12. A structural decomposition of ALP

behavior description of each of the P_i's. This refinement step can be reiterated for each of the P_i's, leading to a multi-level design procedure. A complete specification at each level consists of a set of structure module types and behavior module types, which can be ordered hierarchically by the relation "is a submodule of," with the behavior module types occupying the leaf nodes of the hierarchy graph.

6. An Example: An Arithmetic-Logic Processor Design

In this section we illustrate the concepts introduced in previous sections by giving a complete description of the arithmetic-logic processor introduced in Section 2. In doing these examples, we find it very convenient to use a macro facility and to parametrize module type definitions. The addition of these two features enhances the expressiveness of ADL, without introducing semantic complexity. Before giving the examples, we briefly explain these two features.

Module Parameters

In designing modular systems it is often convenient to specify a family of architectures by introducing design parameters in the specification. In a multi-level design process *independent* parameters can be introduced at each level. In a complete specification the set of design parameters *completely* characterizes the degree of variability in an ADL module. The possible parameters are:

- number of submodules of a given type

- number of input and output ports of a module

- number of state variables (depending on the number of input ports, for example)

Macro Facilities

The macro facility proposed for ADL is a **foreach** construct. It has the general syntactic form:

foreach i:**integer in** [1..<arithexp>] {<syntactic construct>}

and stands for the expanded text:

subst [1, *i, <syntactic construct>] <delimiter>

...

subst [**valof** <arithexp>, *i, <syntactic construct>]

where **subst** is the substitution function, and *i is the macro parameter appearing in <syntactic construct>.

For input port declarations and output port declarations, we also use:

inlet a[1..<arithexp>] : ptype-a

as an abbreviation for

foreach i: **integer in** [1..<arithexp>]

{**inlet** a*i : ptype-a}

The complete specification of the arithmetic-logic processor consists of a structural description for *ALP*, behavioral descriptions for its submodules *controller* and *ALU*, and definition of the various packets. To simplify this example we impose no constraint on the order in which input packets are processed and output packets are generated.

The packet definitions in Figure 13 should be familiar to PASCAL users. The **oneof** data type is explained in Appendix C. The structural description of ALP is given in Figure 12. Note the introduction of a module parameter *n-of-alu* and its use as a parameter in the various macro definitions. A behavioral description of *controller* is given in Figure 14. Its input stream *opn-in* is shared via monitor *in*, allowing the requests to be serviced by any available arithmetic-logic unit. Packets returned by the arithmetic-logic units are merged together to form output stream *res-out* via the monitor *out*. *Controller* may be viewed as consisting of many sections, one for driving each arithmetic-logic unit. The data flow graph for one such section is shown in Figure 15. Each *controller* section in turn consists of four subsections, one for each

```
    /* Packet Definitions */
type opcode = (add, sub, mul, div, logand, logor);

type destination-address = -1 .. 2**15 -1
    /* unused destination addresses are set to -1 */

type intbool = oneof[ int: integer, bool: boolean];

type operation-pkt = record op: opcode,
                            opd: array[1..2] of intbool;
                       destaddr: array[1..4] of destination-address
                       end;

type result-pkt =
    record result: intbool,
           destaddr: destination-address
    end

type alu-opn =
    record op: opcode,
      opcode: array[1..2] of intbool
    end

type alu-res = record result: intbool end;
```

Figure 13. Packet type definitions for ALP

destination address carried in the operation packet. A copy of the arithmetic-logic result is forwarded to each destination address in use by calling the monitor procedure *sendrespkt*. A behavioral description of *ALU* is given in Figure 16. Note the use of **one-of** types, the **acons** operator and **block** expressions.

```
type controller = module(n-of-alu : integer)
      inlet opn-in : opn-pkt;
            alu-in[1..n-of-alu] : alu-res;
      outlet res-out : result-pkt;
            alu-out[1..n-of-alu] : alu-opn;


/* monitor guarding input port opn-in */
   in : monitor share opn-in;
   procedure getoppkt() result(opn-pkt);
         x : opn-pkt = opn-in;
         return(x)
   end getoppkt();
   end in;


/* monitor guarding output port res-out */
    out : monitor share res-out;
    procedure sendrespkt(x : result-pkt);
         res-out ← x;
    end sendrespkt;
    end out;
```

Figure 14. Behavioral description of *controller*

```
    /* alu states */
foreach i : integer in [1..n-of-alu]
  { state free*i : signal := sig };


/* Each section is a driver for an alu. */
foreach i : integer in [1..n-of-alu]
{ /* get opn-pkt for free alu */
  opt*i : opn-pkt = on free*i return in.getoppkt();

      /* start alu-i */
  alu-out*i ← rcons[op : opt*i.opcode, opd : opt*i.opd];

      /* alu-i returns result */
  res*i : alu-res = alu-in*i;

      /* send out tagged result packets */
  foreach j : integer in [1..4]
     { x*i*j : dest-addr = opt*i.destaddr[*j];
       if x*i*j noteq -1
            /* destination address in use */
       then out.sendrespkt(
                          rcons[result : res*i.result
                               destaddr : x*i*j] )
     } /* end foreach j */

      /* state update, regenerate free*i */
  update free*i := await res*i;

} /* end foreach i */

end controller;
```

Figure 14. (continued) A behavioral description of the *controller*

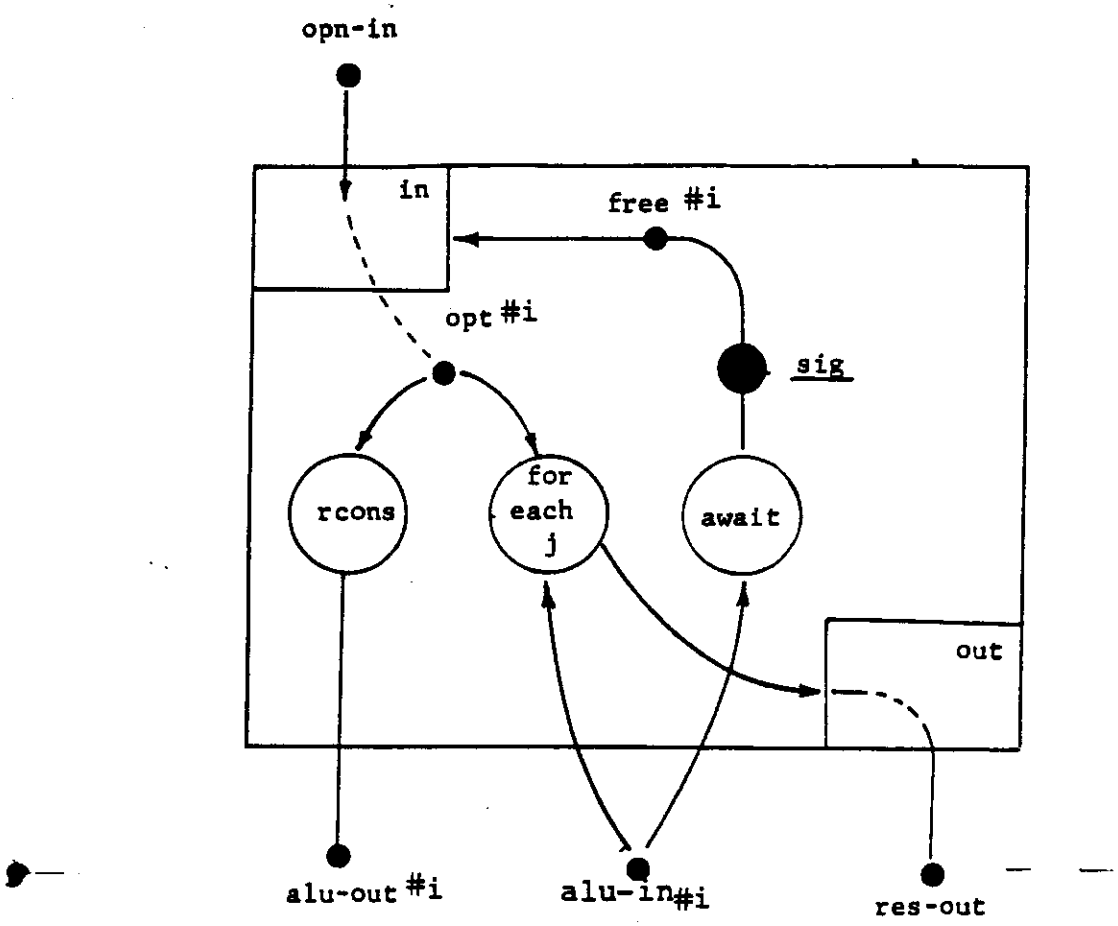


Figure 15. A section of *controller*

```
/* arithmetic logic unit */

type ALU =
    module
    inlet ain: alu-opn;
    outlet aout: alu-res;

    /* input */
    x: alu-opn = ain;

    /* process input */
    aout ←
       case x.opcode of
          add: { begin uses x;
                       opd: array[1..2] of integer =
                              acons[ foreach i: integer in [1..2]
                                       { tagcase x.opd[*i]
                                         tag int( opnd: integer ): opnd;
                                         tag bool(): error } ];
                       resultis intbool$make.int( opd[1] + opd[2] )
                 end }
                                  . . .
          logor: { begin uses x;
                       opd: array[1..2] of boolean =
                              acons[ foreach i: integer in [1..2]
                                       { tagcase x.opd[*i]
                                         tag int(): error;
                                         tag bool( opnd: boolean ): opnd } ]
                       resultis intbool$make.bool ( opd[1] or opd[2] )
                 end }
end ALU;
```

Figure 16. Behavior Description of ALU

7. Concluding Remarks

In this paper we have presented some features of ADL, an architecture description language for packet communication systems. An architecture description in ADL consists of a collection of hierarchically organized ADL module type definitions. A structural type definition formalizes the diagrammatic representation of the internal structure of a module. Submodules, module input and output ports and internal channels are listed and typed. The behavior of a module is synthesized in ADL by composing expressions. The semantics of expression evaluation is based on the principle of data-flow. Each evaluation of an expression is initiated as soon as a new set of operands is available and the results generated by its previous evaluation are no longer needed. Many expressions can thus be viewed as functional modules with well-defined input and output interfaces. The functional capability of these elementary expressions is expanded in two steps. The concept of a module state which can be updated is incorporated to allow definition of functions on data streams. Next a simplified version of Hoare's monitors is added, introducing non-determinism via shared state variables, input and output ports. These extensions are structured so that expression evaluation can still be governed by the flow of data.

A formal architecture description is just a starting point in constructing a packet system. The challenge is to devise a practical machine that is provably equivalent to the one specified in an architecture description. An attractive approach to meet this challenge is to successively refine descriptions in ADL to the point where a description can be systematically translated into hardware. The behavior description in this paper are given mostly using operations defined on records while binary operations on bit arrays are more suitable for hardware implementation. Data abstraction concepts such as those embodied in CLU [7] are directly applicable to bridge this gap, and their incorporation into ADL should be straightforward.

Upon translating an ADL description consisting entirely of binary operations on bit arrays into a data flow graph, it is also relatively straightforward to translate the data flow graph into an interconnection of self-timed hardware modules. A set of hardware modules designed specifically for this purpose and the corresponding translation techniques are described in [6]. These modules communicate via asynchronous packet communication protocols and operate under an unbounded gate delay assumption [1]. Refinement and automated implementation techniques are both topics for further research.

Properties of an ADL description can be verified mathematically, or by interpretation. An ADL interpreter can be extended to be a packet communication system simulator and used to both verify and evaluate the performance of data flow computer architectures.

8. References

[1] Armstrong, D.B. et al. Design of Asynchronous Circuits Assuming Unbounded Gate Delays. IEEE Trans. on Computers, Vol C-18, 12, December 1969.

[2] Dennis, J.B. First Version of a Data Flow Language Lecture Notes in Computer Science 19, Springer-Verlag, New York 1974

[3] Dennis, J.B. Packet Communication Architecture Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing, IEEE, New York, 1975

[4] Dennis, J.B., C.K.C.Leung & D.P.Misunas, A Highly Parallel Processor Using a Data Flow Machine Language Computation Structures Group Memo 134-1, LCS, MIT, January 1977

[5] Hoare, C.A.R. Monitors: An Operating System Structuring Concept CACM 17, 10, October 1974

[6] Leung, C.K.C. A Top-Down Design of a 2 x 2 Router CSG Group Memo Under Preparation, Laboratory for Computer Science, MIT

[7] Liskov, B.H. et al. *Abstraction Mechanisms in CLU* CACM, 20, 8, August 1977

[8] Liskov, B.H. and Valdis Berzins. An Appraisal of Program Specifications, in Research Directions in Software Technology, Peter Wegner (Ed), MIT press, 1979

Appendix A. Iteration Expression

The iteration expression has evolved through a continuing effort of the Computation Structures Group. The construct discussed in this paper is designed by J. B. Dennis:

for v_1: $type_1$ = exp_1, ..., v_m: $type_m$ = exp_m
if $cond_1$ **then iteron** exp_{11}, ..., exp_{1m};
else
if $cond_2$ **then iteron** exp_{21}, ..., exp_{2m};
else
if $cond_n$ **then iteron** exp_{n1}, ..., exp_{nm};
else
resultis exp_α, ..., exp_β;

This iteration expression can be defined functionally by

$f(exp_1, ..., exp_m)$
where $f(v_1, v_2, ..., v_m)$ =
if $cond_1$ **then** $f(exp_{11}, ..., exp_{1m})$
else
if $cond_2$ **then** $f(exp_{21}, ..., exp_{2m})$
...
if $cond_n$ **then** $f(exp_{n1}, ..., exp_{nm})$
else
return exp_α, ..., exp_β.

i.e. as applying a recursively defined function f to arguements exp_1, ..., exp_m.

Its data flow semantics is given in Figure A.I.

The data flow graph of Figure A.I consists of several stages. Given values $v_1, ..., v_m$ for $v_1, ..., v_m$, stage i is entered iff none of the conditions $cond_1, ..., cond_{i-1}$ is satisfied by $v_1, ..., v_m$. If these values satisfy $cond_i$, iteration continues with the results of evaluating $e_{i1}, ..., e_{im}$ as new values for $v_1, ..., v_m$. If none of the conditions is satisfied the expressions $exp_\alpha, ..., exp_\beta$ are evaluated with $v_1, ..., v_m$ and their results become the results of this evaluation of the iteration expression. Note the two columns of non-determinate merge gates for relaying values between stages. A non-determinate merge gate merges two input streams into one output stream under fair arbitration. For each iteration cycle, however, only one input will be delivered to each non-determinate merge gate in Figure A.I.

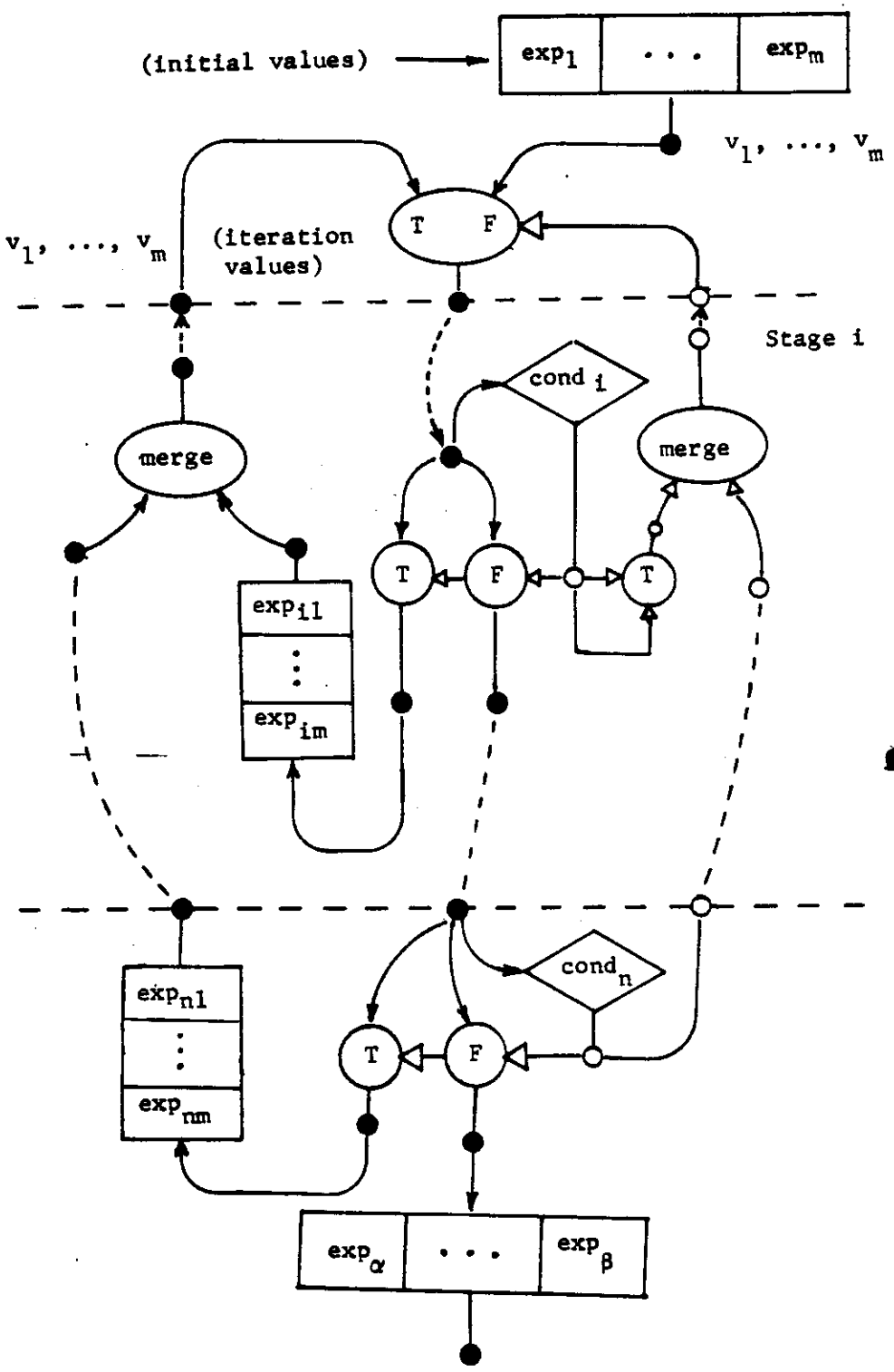


Figure A.1 Data Flow Semantics for an Iteration Expression

Appendix B. Monitors

Monitors are provided for resource sharing. The data flow graph of a monitor is shown in Figure B.1. The monitor depicted has the following parts:

(i) n monitor procedures. They access a shared state variable and a shared input port, but each guards its own output port.

(ii) P_i invocation sites for the i^{th} monitor procedure.

Arbitration among concurrent invocations to monitor procedures are resolved at the non-deterministic merge operator which merges several input streams into one output stream under fair arbitration. Tagging is used to distinguish among calls to different monitor procedures and among calls from different invocation sites on the same monitor procedure. To simplify the diagramming, we have introduced a SCASE operator and a MCASE operator (Figure B.1). The former passes its input to one of its output links, while the latter passes one of its inputs to its output link, under the control of a tag.

When the resource sharing capabilities of a monitor is restricted, its semantics can be expressed by simpler data flow graphs. Two such examples are shown in Figure B.2 and B.3. The data flow semantics of the monitor in Figure 4.7 is given in Figure B.4.

The organization shown in Figure B.1 is also sufficiently general to allow sharing several input ports, several state variables and sharing different sets of resources among different groups of monitor procedures.

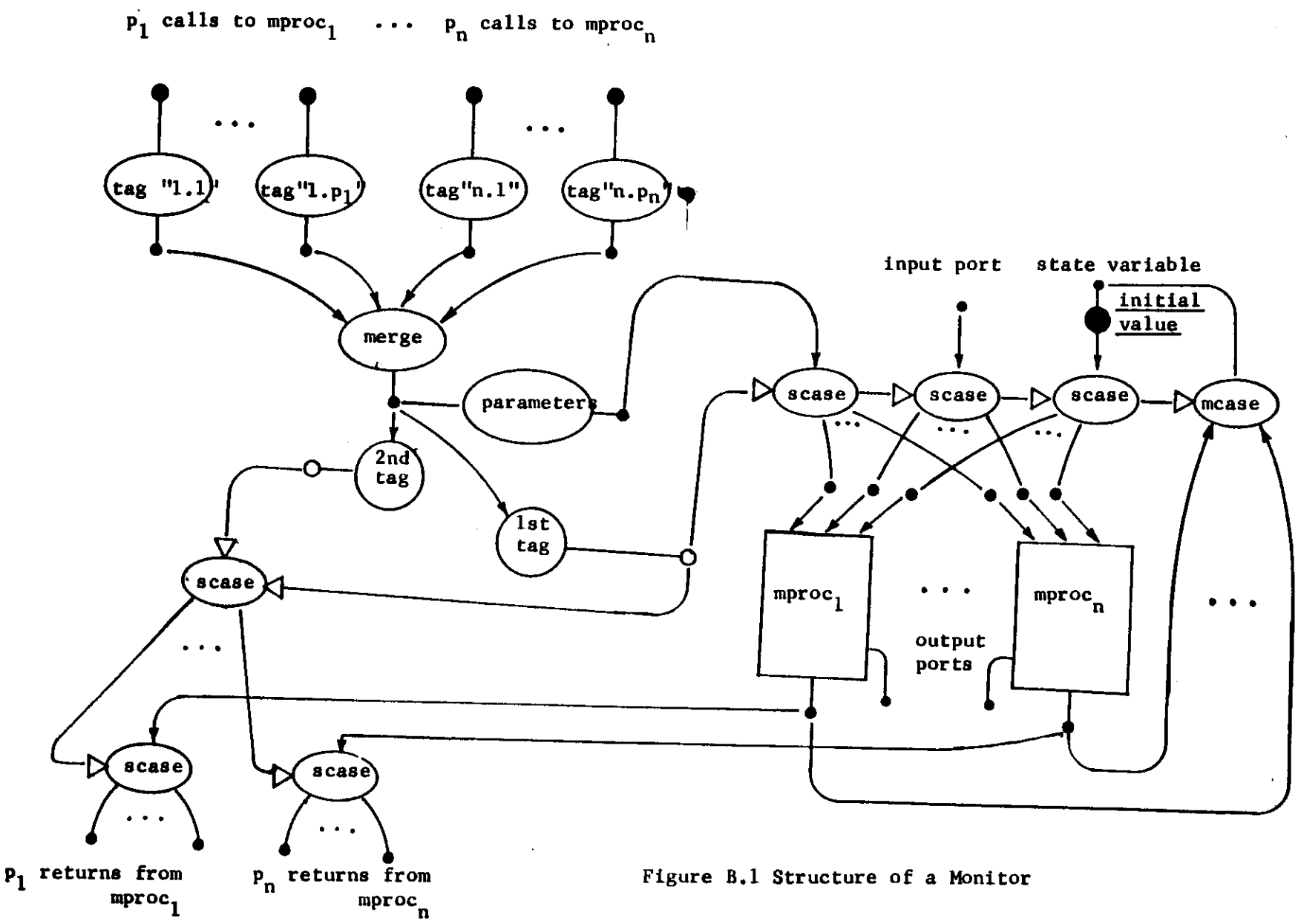


Figure B.1 Structure of a Monitor

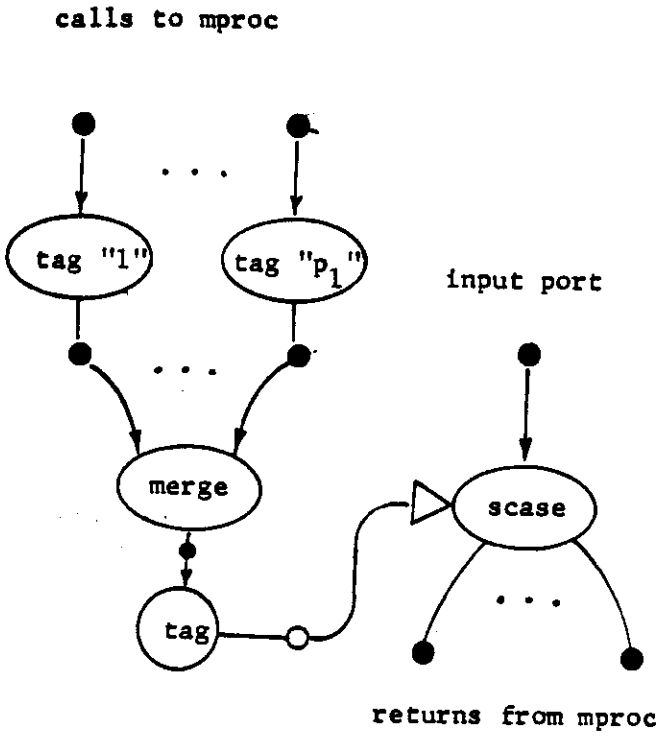


Figure B.2 A monitor for input port sharing

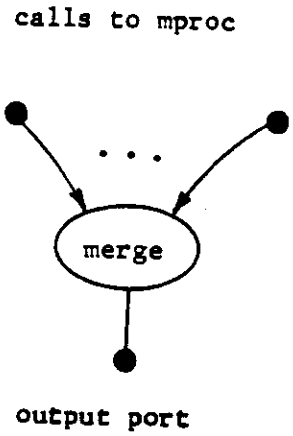


Figure B.3 A monitor for output port sharing

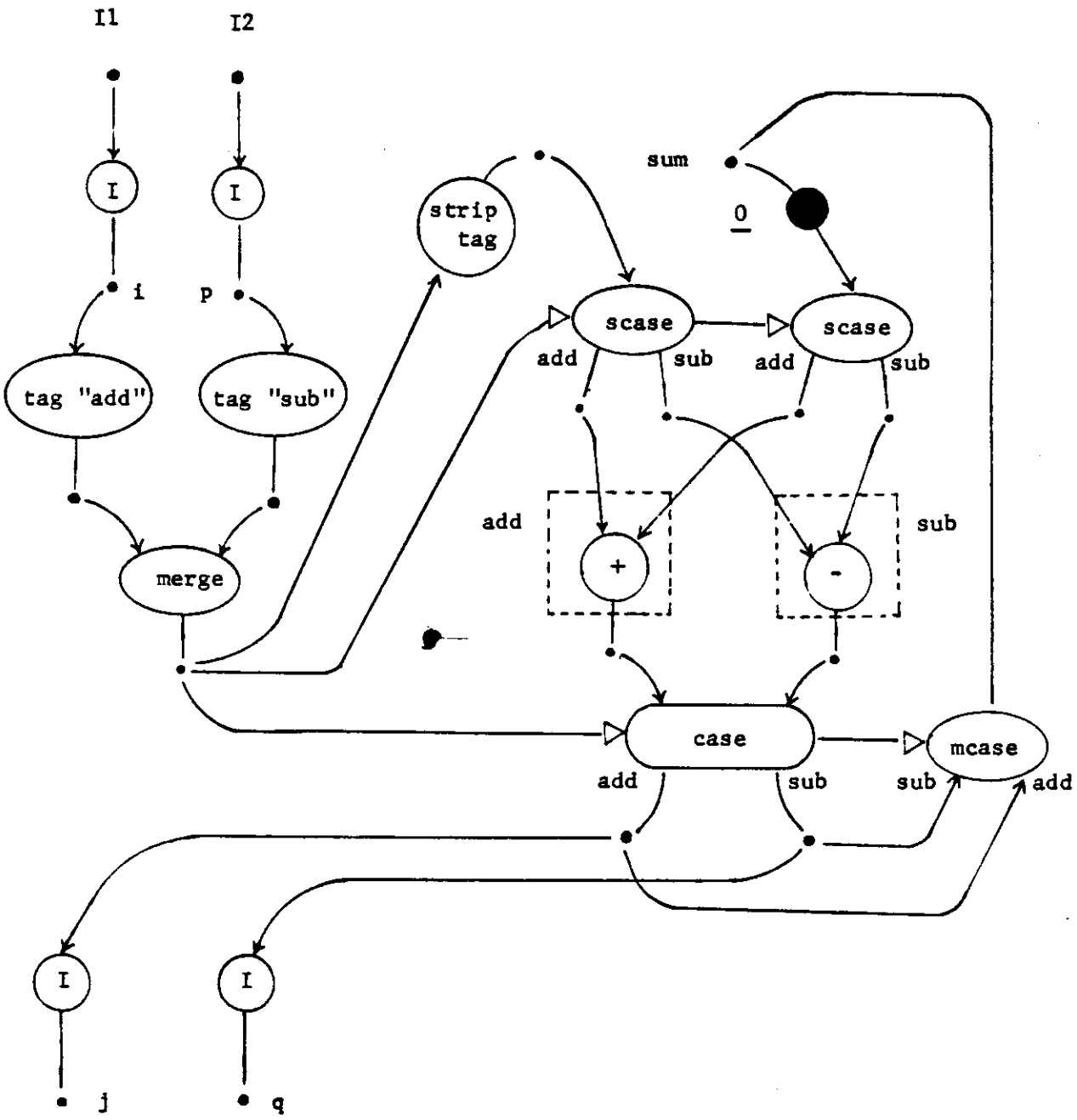


Figure B.4 Data flow semantics for monitor "m" of Figure 10

Appendix C. One-of Type

The **oneof** type of ADL is copied from CLU. The **oneof** type is a mechanism to form an object whose type is "one of" a set of alternatives.

An object of type **oneof** [S_1:T_1, ..., S_n:T_n] can be thought of as a pair. The "tag" component is an identifier from the set (S_1,..., S_n). The "value" component is an object of the type corresponding to the tag. That is, if the tag component is S_i then the value is some object of type T_i.

A special form of constructor is used to construct an object of type

type T = **oneof** [S_1 : T_1, ..., S_n : T_n]

as follows:

T$make.$S_i$ (<simple expression>)

The simple expression should evaluate to an object of type T_i. This object is then tagged by the identifier S_i.

Objects of a **oneof** type are decomposed and manipulated with a special type of case expression called *tagcase* expression.

<tagcase expression> ::= tagcase <simple expression>

tag S_1 <id*1*: T_1>: <expression list*1*>;

. . .

tag S_n <id*n*: T_n>: <expression list*n*>;

end;

The simple expression should evaluate to an object (S, T) of type **oneof**[S_1: T_1, ..., S_n: T_n]. If the tag is Si, T is make available for evaluating expression list i through the link id$i$. The scope of the declaration of id$i$ is expression list i.