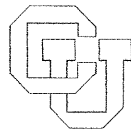
Abstract Process Types *

William E. Riddle

CU-CS-121-77



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

* This work was supported by a grant from Sycor, Inc.

ANY OPINIONS, FINDINGS, AND CONCLUSIONS OR RECOMMENDATIONS EXPRESSED IN THIS PUBLICATION ARE THOSE OF THE AUTHOR(S) AND DO NOT NECESSARILY REFLECT THE VIEWS OF THE AGENCIES NAMED IN THE ACKNOWLEDGMENTS SECTION.

RSSM/42

ABSTRACT PROCESS TYPES

William E. Riddle
Department of Computer Science
University of Colorado at Boulder
Boulder, Colorado 80309

CU-CS-121-77 December, 1977

This work was supported by a grant from Sycor, Inc.

Introduction

The design and implementation of large-scale software systems requires description schemes which allow the abstract, high-level specification of the system. Without the ability to capture the essence of the various processing and data structuring components, a system's description quickly becomes complicated and incomprehensible. It is only through a description of the system as a collection of less-complex parts which interact in well-defined ways that anyone trying to understand the system can succeed within a reasonable period of time.

Abstract data types have recently emerged as an important facility for abstract description in the domain of sequential programs. The basic concept originally appeared in the simulation language SIMULA [7] and was introduced into programming languages as a facility to support structured programming [14]. Their recent refinement has taken three directions. First, they have been developed to support top-down design methods [17]. Second, they have been extended so as to provide a basis for formal verification [40]. Finally, they have been used as the basis for a tool to aid program development [12] which admits rigorous, but perhaps incomplete, analysis [11]. The variety of these uses indicates the breadth of the benefits which accrue from facilities for high-level, abstract description.

Abstract data types are ideally suited to the high-level description of a software system's data storage components. But they are not adequate for the succinct description of those system components which are intended more for the processing of data than for the storage of data. This is particularly true when the components operate concurrently[1]. The major problem is that abstract data types are oriented toward describing components as structures of data which are operated upon via procedure calls. Many components -- e.g. a file system in a multiprocessor computing facility -- are not naturally described as collections of data objects which are accessed in a basically sequential manner.

---

[1]By concurrent we mean parallelism which may actually be achieved by running the system in a multiprocessing environment or which may only be apparent at abstract levels of description of the system.

Extensions and modifications to abstract data types could be developed to increase their effectiveness in describing processing components in software systems -- this has been done in the Gypsy system [1] and the Modula programming language [39]. In this paper, however, we develop a description scheme that retains many of the concepts of abstract data types but is based upon the concept of a sequential process [16]. This leads to a means of abstractly describing processing structures which focuses upon describing a system as collection of interacting, concurrent subsystems.

In the next section, the view of a system as a collection of interacting subsystems is made more concrete as we describe the various ways in which the subsystems may interact. Following that, the various aspects of abstract process descriptions are presented. First, the external description of a subsystem in terms of how it interacts with other subsystems is discussed. Then, the discussion turns to the manner in which an internal description of a subsystem may be given in terms of coordinated interactions among its (sub-)subsystems. In the final section, we suggest that the scheme admits approaches to verification which are similar to those developed in conjunction with abstract data types.

The constructs discussed here are part of the design language developed for the Design Realization Evaluation and Modelling (DREAM) system ([19], [24]-[26]). DREAM is a tool for the design of large-scale software systems, providing the designer with bookkeeping and analysis aid. A description in the DREAM Design Notation (DDN) consists of a collection of (nested) description fragments, called <u>textual units</u>. DREAM provides facilities which allow the user (or users, in the case of a design being carried out by a design team) to modify the information in a data base on a textual unit basis.

The focus of this paper is upon the DDN constructs for describing abstract process types. Other aspects of DDN are discussed in [22], [28], [32], and [38]. Also the focus is upon the use of the constructs -- their syntax is covered in [27]. Some justifications for the constructs are given; others lie within the DREAM system's general philosophy which is discussed in [26] and [30].

An Abstract View of Systems

A convenient view of software systems is that they are composed of parts, subsystems, which operate concurrently and asynchronously. This may be made concrete by viewing the subsystems as interacting solely through the sending and receiving of messages and the sharing of data objects. This is a bit redundant since the shared data objects could be viewed as subsystems which receive read and write commands as messages and send messages containing the requested value in response to a read command. But recognizing shared data objects as distinct from subsystems allows message exchange to be used to focus upon control interactions among subsystems. Also, data objects are needed to describe the structure of the messages transmitted among the subsystems.

This paper focuses upon the description of control interactions among asynchronously operating subsystems. We assume, therefore, the existence of a scheme for describing data objects that may be shared by a community of concurrent subsystems. (Several such schemes have been developed, e.g., [2], [15], [39]. A scheme that was developed in conjunction with the work discussed here is described in [28].) Only two aspects of the scheme are important here. First, a definition of a data object specifies all the procedures which may be invoked upon it. Second, data objects have built-in synchronization mechanisms which can be used to preclude interference among the operations performed upon the object.

Communication among asynchronous message senders and receivers requires a transmission controller that is able to store both messages that have been sent but not yet received and requests for messages which have been lodged but not yet satisfied. In DDN, an idealized controller, called a link, is provided. Links hold messages and requests in (unbounded) bag data structures. Thus they do not necessarily pass messages on in the order the messages were sent, nor do they necessarily service requests for messages in the order the requests were lodged.

As an example of this view of software systems, consider the following description of HEARSAY [9], a multiprocessor speech processing system developed at Carnegie-Mellon University. HEARSAY may be decomposed into two major parts. The first is a data base, called the blackboard, which contains all the information about the utterance being processed and the hypotheses which have been made as to its linguistic structure. The second part is a collection of processing subsystems, each called a knowledge source. A knowledge source inspects the information in the data base and augments or modifies it according to rules which it is programmed to enforce. The subsystems in this description of HEARSAY are therefore the blackboard and the knowledge sources. Note that the blackboard could alternatively be described as a shared data object since its primary function is the storage of data and the operations performed upon it by the knowledge sources must be synchronized in some way. We ignore this design choice and focus upon the knowledge sources.

The interactions among the HEARSAY subsystems arise as follows. It would be wasteful to have each knowledge source constantly inspect the data base to see if it should perform any processing upon it. Therefore, the data base is programmed to know which data base entries are of interest to each knowledge source and to send a signal to a knowledge source when one of the data base entries of interest to it changes value. When a knowledge source is awakened in this way, it inspects the data base and makes any modifications deemed necessary.

Multiprocessing systems, such as HEARSAY, may naturally be described in the manner advocated here since these systems actually have subsystems which interact by message exchange. But this view is also appropriate, as evidenced by two recent texts ([3], [10]), when it is only a logical one and the system actually runs in a uniprocessor environment. In this case, the view facilitates the decomposition of the system and hence the mastery of its complexity [33]. The transmission of messages may never really take place, and message interchange may be used only as a _model_ of the actual interactions. This is the primary reason that a more sophisticated transmission control mechanism is not defined within DDN.

In the following sections, a DDN description of the knowledge source subsystems of HEARSAY[2] will be developed. All of the knowledge source subsystems are similar with respect to their interactions with the data base. Therefore, the description is of the class of knowledge source subsystems. To reflect that different instances of this class vary with respect to their details, such as the number of entries in the data base that are of interest to the knowledge source, we define a parameterized class. This means that part of the class definition is a specification of qualifiers which may be assigned values when an instance of the class is created. Class definitions and qualifiers are discussed in [22].

External Description of a Subsystem

An external description of a subsystem defines the ways in which the subsystem interacts with other subsystems in its environment. In keeping with the principle of information hiding [18], the external description indicates only the effect of the combined operation of the subsystem's internal components and defines nothing as to the manner in which the internal components are organized or actually interact.

The external description also serves to guide the implementation of the subsystem. Being, by definition, a complete specification of the subsystem, it provides the designer with a definition of the (minimal) behavior which must be implemented. It may also guide the designer's decision making, helping in the determination of the set of decisions to be made and the inter-relationships among the decisions.

The external description must define three aspects of the subsystem. First, it must specify the interfaces to the subsystem not only in terms of the format of the information flowing through each interface but also in terms of what information may legally flow through each interface. Second, it must specify the correlation between the information flowing through one interface and the information flowing through another interface. Finally, it must relate the system's operation at one point in time to its operation at a previous point in time -- that is, it must specify the more global aspects of the subsystem's operation.

---

2 The description is an approximation of the structure and operation of knowledge sources actually in HEARSAY. It reflects our understanding of the description which appears in [9], but has also been tailored so as to provide examples of the facilities in DDN.

Subsystem Interfaces

An interface to a subsystem is a port through which messages may flow. Conceptually, a port is a named communication line along which messages flow, one at a time, and which does not have any message storage capabilities. Ports correspond to one-way communication lines and therefore have a direction in or out.

The messages which flow through a port are sets of data objects. Each port therefore has associated with it a set of buffers, each able to store one data object. The set of buffers is ordered, in the same sense and with the same implications as the ordering among a set of parameters to a procedure. The set of buffers indicates the types of data objects which comprise a message and the order in which the data objects are composed to form a message.

The messages that may legally flow through a port are specified by giving buffer conditions for the port. A buffer condition is a predicate over the buffer data objects, indicating the set of legal values for the buffer data objects as well as the legal correlations among the values. OR'ed together, the buffer conditions associated with an in-port (out-port) are analogous to a pre-condition (post-condition) [13].

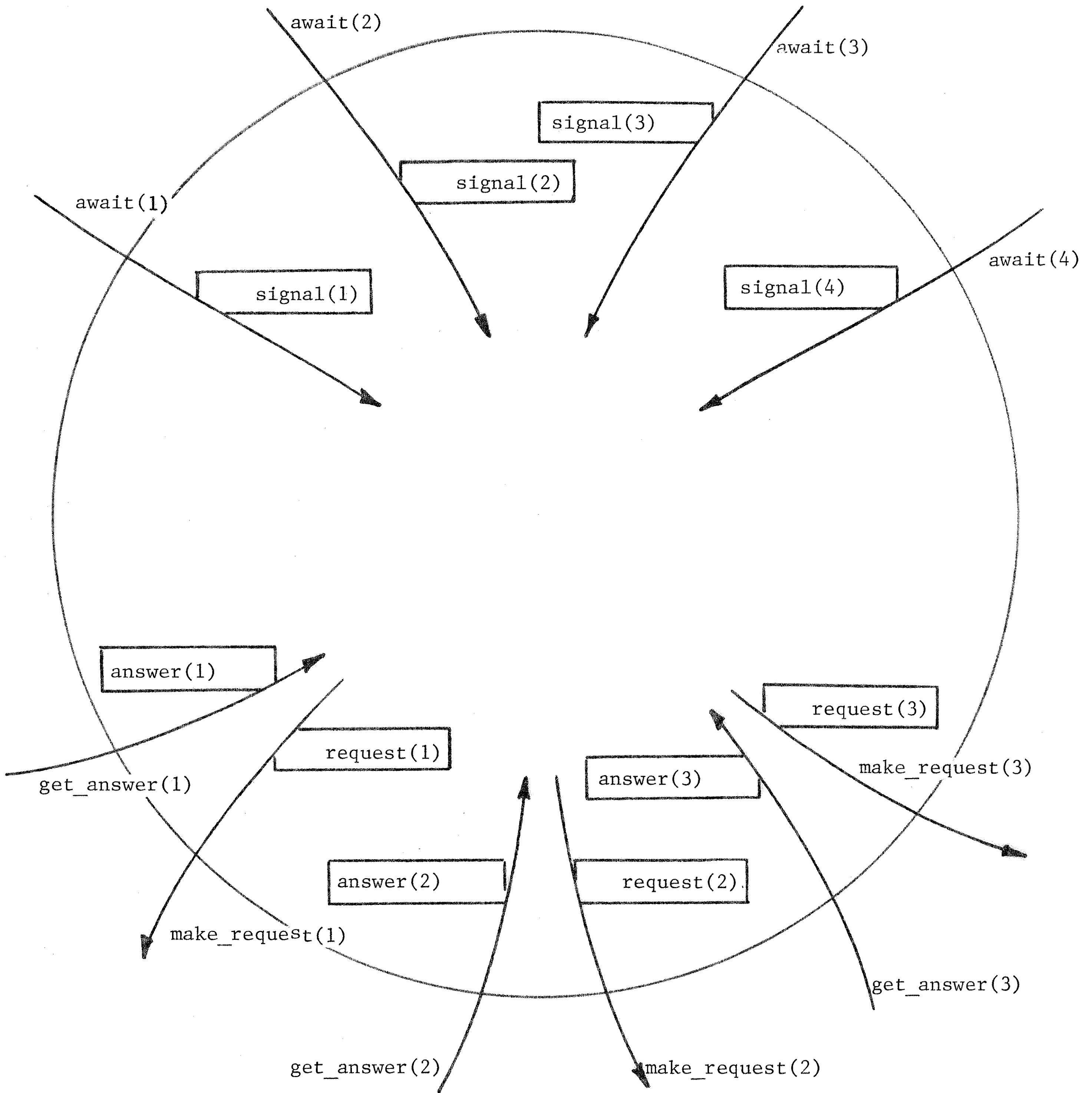
In DDN, a port is defined by giving a textual unit which specifies the port's name and direction and which has nested textual units which specify the buffers and the buffer conditions associated with the port. A set of ports is defined in figure 1 for objects of class[3] [knowledge_source]. If an object of this class were created with *#_values* having the value 4 and *#_servers* having the value 3, then the object would appear as pictured in figure 2. There are ten ports, grouped into three arrays and each port has a singleton set of buffers associated with it. Notice that defining an array of ports implies that there is an array of sets of buffers, one set for each port. The buffer conditions associated with the *make_request* ports indicate that messages flowing out through these ports will have only the values *inspect* or *modify*. The absence of buffer conditions associated with the other

---

[3] It is convention in DDN to enclose an identifier in square brackets when it is used to name a class.

```
[knowledge_source]:  SUBSYSTEM CLASS;

   QUALIFIERS;
      DOCUMENTATION;
         #_values is the number of values monitored
         for this knowledge source; #_servers is
         the number of parallel servers in this
         knowledge source
         END DOCUMENTATION;
      #_values, #_servers
      END QUALIFIERS;

   await:  ARRAY [1::#_values] OF IN PORT;
      BUFFER SUBCOMPONENTS;
         signal OF [on_off_switch]
         END BUFFER SUBCOMPONENTS;
      END IN PORT;

   make_request:  ARRAY [1::#_servers] OF OUT PORT;
      BUFFER SUBCOMPONENTS;
         request OF [data_base_operation]
         END BUFFER SUBCOMPONENTS;
      BUFFER CONDITIONS;
         request=inspect,
         request=modify
         END BUFFER CONDITIONS;
      END OUT PORT;

   get_answer:  ARRAY [1::#_servers] OF IN PORT;
      BUFFER SUBCOMPONENTS;
         answer OF [data_base_response]
         END BUFFER SUBCOMPONENTS;
      END IN PORT;

   END SUBSYSTEM CLASS;
```

Figure 1



ports are represented by labelled arcs

buffers are represented by boxes attached to the arcs

Figure 2

ports indicates that there are no restrictions on the values of the data objects which compose messages flowing through these ports - this freedom could be reduced as the subsystem is further designed.

A port definition is comparable to a heading for a procedure stated in a programming language. It is similar in that it specifies a name by which the port (procedure) may be referenced and the number, type and order of the data objects (parameters) in messages (parameter lists) processed by the internal components (procedure body). It differs because ports allow one-way, asynchronous communication whereas procedures provide two-way, synchronous communication.

The buffer conditions of DDN are analogous to entry and exit specifications of Alphard [40] with the additional aspect that they are required to be valid whenever a message flows through the port. Buffer conditions may be viewed as assumptions which may be made while formally verifying either a design of the subsystem or a design of another subsystem which uses the subsystem. Or they may be viewed as indicating checks to be made during the execution or simulation of the subsystem. They have been included in DDN with the intent of using them in both these ways.

Message Flow Through a Subsystem

The role that a subsystem plays within a community of subsystems is specified by a definition of the correlations among the messages flowing into and out of the subsystem. This can serve to define either the facilities provided by the subsystem or its utilization of the facilities provided by the other subsystems. Note that this means there will be a redundant specification of the interaction among subsystems, with a definition of the interaction within both the users and the provider of a facility. This is useful redundancy, affording the opportunity to perform verification as will be discussed later.

Succinct definition of the subsystem's control of message flow is frequently procedural in nature -- this is especially true when programmers are the intended audience, since such descriptions are quite natural to them. This leads to an ambiguity as to whether the description is actually an external one or whether it is not also

describing the system's internal operation. While a procedural definition may correspond very closely to the subsystem's internal operation, it may also differ radically. Thus, it is serving as a convenient description of the effect of the subsystem's internal operation and, in and of itself, indicates nothing that may be relied upon regarding the way that effect is achieved.

Some of the message flow characteristics of a subsystem may be defined in terms of sequences of message transmissions through the subsystem's ports. This is analogous to defining a procedure in terms of a set of parameter/result pairs. More complex characteristics, called _global characteristics_, must be defined in terms of the correlations between transmissions occurring in different message transmission sequences. This is analogous to specifying, for a collection of stack manipulation procedures, that only certain combinations of procedure executions are legal -- for example, it should be specified that each pop operation must be preceded, at some point in time, by a corresponding push operation. In this section, the concern is with the more easily specified sequential message transmission characteristics. The specification of global message flow characteristics is treated in the next section.

Sequential message transmission characteristics are specified by a set of "programs", each of which is an abstract model of a sequential process. Although these _control process_ models are in the form of programs for an abstract machine, their purpose is not to provide (or even suggest) a definition of the operational detail of the subsystem. Rather, they provide a pseudo-procedural definition of the _effect_ of the subsystem's operation by means of an algorithmic definition of the message flow through the subsystem's ports.

Each control process model may be viewed as a sequential process which controls communication between a subsystem and other subsystems in its environment. This communication may generally be partitioned into several message transmission streams which are arbitrarily interleaved. A subsystem is therefore generally described by several control process models with each describing a single message transmission stream.

In a control process model, messages flow in and out through ports as a result of operations called <u>receive</u> and <u>send</u>. When a send operation is performed, a message is first composed using the values of the buffers associated with the port that is named in the send operation. Then, the message is placed in the link to which the port is attached and thereby made available for reception by some subsystem. (The link is an object which is not a part of any of the subsystems which utilize it. The process of attaching ports to links will be covered in a subsequent section.) Control passes to the operation following the send operation once the message is constructed and placed in the link. Since links have infinite storage capacity, this delay is relatively short and the send operation can therefore be considered to be a non-blocking operation.

When a receive operation is performed, flow of control is suspended until a message is retrieved, decomposed and distributed among the buffers associated with the port specified in the receive operation. Since it is possible that a request for a message may be lodged when none is currently available, the receive operation can cause relatively long delays.

Prior to a send operation, the values of the data objects which compose the message must be placed in the buffers. The computation which is actually carried out to accomplish this may be lengthy and complicated, but neither its time consumption nor its detail are of interest in defining sequential message transmission characteristics. The only aspects of interest are the result of the computation and the relationship of the result to any messages previously received. Thus, in a control process model, computational detail is suppressed by modelling it with a <u>set-to</u> operation which may be applied to a buffer data object and which results in the buffer assuming a value prescribed in the set-to operation. The dependency of the value upon previously received or computed values is modelled by conditioning the flow of control upon the values of the buffer data objects and hence upon previously received or "computed" messages.

To describe the sequential message transmission characteristics of [knowledge_source] subsystems, two sets of control processes are

needed. The first, specified[4] in figure 3, models the consumption of messages which arrive at the *await* port. This models the subsystem's operation with respect to signals sent to indicate a change of some data base entry of interest to the knowledge source.

```
▼[knowledge_source]: SUBSYSTEM CLASS▼

  listener:  ARRAY [1::#_values] OF CONTROL PROCESS;
     MODEL:  ITERATE
                RECEIVE await(MY_INDEX);
                END ITERATE;
             END MODEL;
     END CONTROL PROCESS;
```

Figure 3

The second set of control processes is specified in figure 4. These latter control processes model the operation of the servicers of the incoming signals, indicating that they present a request to the data base through one of the *make_request* ports and receive answers to the request through one of the *get_answer* ports. The entity *MY_INDEX* is a variable which has a value in the range declared as the bounds of the array of control processes and is used to make each control process model distinct with respect to the buffers and ports to which it refers.

```
▼[knowledge_source]: SUBSYSTEM CLASS▼
  requestor:  ARRAY [1::#_servers] OF CONTROL PROCESS;
     MODEL; ITERATE
               request(MY_INDEX) SET TO modify OR inspect;
               SEND make_request(MY_INDEX);
               RECEIVE get_answer(MY_INDEX);
               END ITERATE;
            END MODEL;
     END CONTROL PROCESS;
```

Figure 4

---

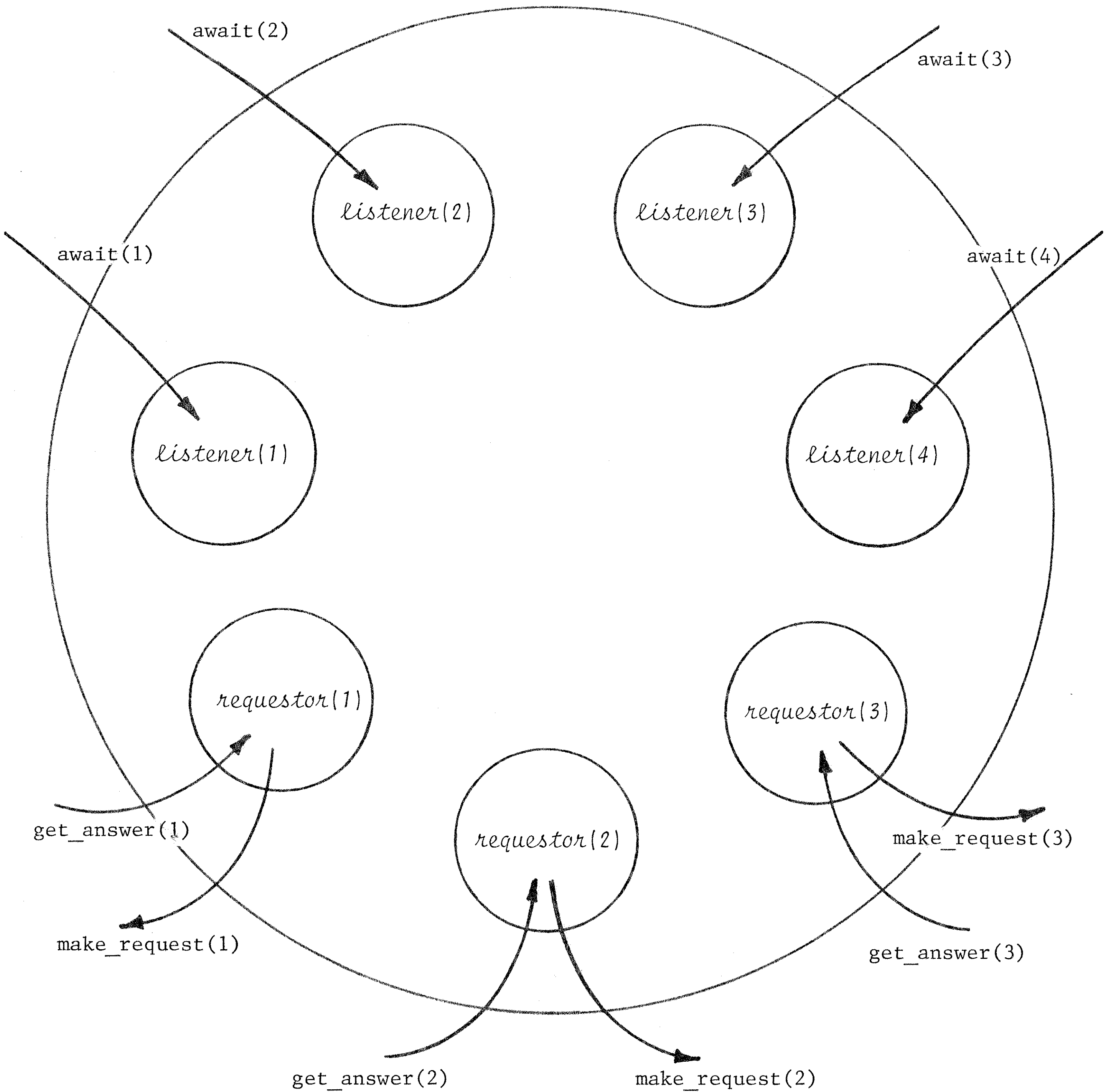
[4] The quoted prefix in the textual unit of figure 3 indicates that this textual unit gives further information about the class [knowledge_source].

The control process structure of a [knowledge_source] subsystem in which *#_servers* is 3 and *#_values* is 4 may be pictorially represented as in figure 5.

Each model is a control program over send and receive operations on ports and set-to operations upon buffers. When necessary for the accurate modelling of a subsystem's sequential message transmission behavior, operations defined for the class of objects of which a buffer is an instance can also be invoked upon that buffer. The control constructs used in control process models are Algol-like. There are constructs for definite iteration: an "ITERATE n TIMES" contruct and a "FOR ALL i IN set_of_values" construct. WHILE and UNTIL constructs are available for indefinite iteration. Since many subsystems are designed to never terminate, there is also an "ITERATE" construct for infinite iteration. Conditional control may be specified by the usual forms of the IF construct. There is also a generalized CASE construct which allows "labelling" of the cases with logical expressions.

All of the control constructs have a nondeterministic version. For example, the construct "ITERATE n OR MORE TIMES" indicates that the number of times, while known before iteration begins, can be any number greater than or equal to n. A nondeterministic variant of the WHILE construct (see figure 6) is obtained in a way in which many of the nondeterministic versions are obtained, by using the nondeterministic boolean expression PERHAPS. Nondeterminism may also be specified in the set-to operation (see Figure 4) by giving a logical expression which specifies the set of values which could possibly be assigned to the data object.

Control process models provide for the nondeterministic, pseudo-procedural modelling of a subsystem. Nondeterminism is used because it contributes to the clarity of the model, allowing succinct definition of the subsystem. A pseudo-procedural description scheme is also used to enhance the clarity of the description. The resulting procedure-oriented modelling scheme is quite natural for describing the effect of a subsystem's operation in terms of its sequential message transmission characteristics. (As will be seen in the next section, DDN



control processes are represented by circles containing the control process name in script

Figure 5

relies heavily upon non-procedural specifications for defining global message transmission characteristics.)

A multiprocessor system programming language (for example, [1] and [8]) typically provides a variety of operations for the synchronization of message transmission. In the DDN modelling language, however, we have provided only two relatively simple,but sufficient, synchronization operations, send and receive. While this makes it necessary for designers to develop DDN descriptions of other synchronization operations, we feel that this has the beneficial effect of forcing the designers to develop the details of the operations with consideration given to the ways in which they will be used to effect synchronization.

Global Behavior

The facilities provided by a subsystem cannot generally be used in a totally arbitrary order -- e.g., the facility which a file system provides for opening a file must be used prior to the facilities for reading and writing the file. Thus there are correlations between what happens in one sequential message transmission sequence and what may happen in another sequential message transmission sequence. (Note that these sequences may pertain to the subsystem's use of its environment as well as to the use, by others, of the facilities provided by the subsystem.) A specification of these correlations is a specification of the subsystem's global behavior since it concerns more than just the activity of a single part of the subsystem for a relatively short period of time.

Global behavior must be specified non-procedurally since it can rarely be expressed succinctly as some controlled sequence of operations invoked by the subsystem. Necessarily, it concerns the operation of other subsystems which are in the subsystem's environment. It also pertains to operations that are distributed over time, stemming from different sequential algorithms. Procedural descriptions would therefore be both misleading and complicated.

Global behavior is specified in DDN in terms of events, activities that may be observed external to the subsystem. Usually an event is

the transmission of a message through a port, and thus may be identified with the execution of a send or receive operation in one of the control process models. In general, an event may be associated with the execution of any one of the instructions in a control process model. (Recall that since control process models are used as part of the external description of a subsystem, they are known to an external observer.)

Global behavior is specified by defining a set of sequences of events. Note that this is similar to what was accomplished procedurally via the control process models -- each model defines a set of sequences of message transmissions where each sequence corresponds to an execution of the model. Formal language theory provides a base for the non-procedural specification of behavior, since the set of events may be considered to be an alphabet and a behavior is then a language over this alphabet [19].

Two aspects of global behavior -- one mandatory for correct operation and the other stemming from a design decision -- need to be specified for [knowledge_source] subsystems. First, interactions with the data base occur subsequent to the reception of an activation signal indicating that an entry in the data base has been changed. Second, the interactions with the data base should be ordered such that no request is lodged while there is an outstanding request which has not been answered -- that is, interactions with the data base are to be ordered so that its facilities are utilized in a subroutine fashion.

To describe this behavior, we first define some events as indicated in figure 6. The statement labels define names for events which correspond to the execution of the labelled statement. Note the use of the NULL instruction to allow denotation of the event "a sequence of interactions with the data base is begun".

```
▼[knowledge_source]: SUBSYSTEM CLASS▼

  listener:  ARRAY [1::#_values] OF CONTROL PROCESS;
     MODEL;  ITERATE
          hear: RECEIVE await(MY_INDEX);
                END ITERATE;
             END MODEL;
     END CONTROL PROCESS;

▼[knowledge_source]: SUBSYSTEM CLASS▼

  requestor:  ARRAY [1::#_servers] OF CONTROL PROCESS
     MODEL; ITERATE
        start: NULL;
               ITERATE WHILE PERHAPS
                  request(MY_INDEX) SET TO modify OR inspect;
          ask:    SEND make_request(MY_INDEX);
          get:    RECEIVE get_answer(MY_INDEX);
                  END ITERATE;
               END MODEL;
     END CONTROL PROCESS;
```

Figure 6

More macroscopic events are needed to conveniently define the global behavior. These are defined by the textual unit shown in figure 7. The SEQUENCE operator denotes that the events which are its arguments are sequenced in the specified order. The REPEAT operator denotes that the *consult* event may be repeated zero or more times within each *hear_and_do_something* sequence. Thus, the *hear_and_do_something* event corresponds to a signal arriving from the data base being followed by the interactions with the data base.

```
▼[knowledge_source]:  SUBSYSTEM CLASS▼

   EVENT DEFINITIONS;
      consult:  SEQUENCE(ask, get),
      hear_and_do_something:
         SEQUENCE(hear, start, REPEAT(consult))
      END EVENT DEFINITIONS;
```

Figure 7

With these events defined, the global behavior may be specified as in figure 8. The first part of the specification indicates that, for any particular instance, up to *#_servers hear_and_do_something* events may be proceeding at any point in time, i.e., that servicing may go on in parallel up to the limit imposed by having only *#_servers* servicers. The second part indicates that a *consult* event for any particular instance must be exclusive of any other *consult* event for any instance of class [knowledge_source], i.e., that the interactions with the data base are ordered in time.

```
▼[knowledge_source]:  SUBSYSTEM CLASS▼

   DESIRED BEHAVIOR;
      POSSIBLY #_servers CONCURRENT
          (hear_and_do_something, hear_and_do_something),
      MUTUALLY EXCLUSIVE (consult, [knowledge_source]|consult)
      END DESIRED BEHAVIOR;
```

Figure 8

This example has used only a portion of the facilities available in DDN for behavior specification. Events may be associated with other aspects of a subsystem's operation, more complex relationships among events may be defined, relationships between events defined for different subsystems may be established, and events which are not associated with any computational part of a system may be described. A complete description of DDN facilities for non-procedural, event-based behavior specification is given in [38].

To review, a subsystem's global behavior is specified by describing the sequencing among observable events that relate to the operation of the subsystem. Global behavior may concern either the operation of the subsystem over a period of time or the interactions among different subsystems, or both. In any case, the sequencing is specified non-procedurally by defining a language over an alphabet of events.

The techniques for behavior specification used in DDN are a modification of event expressions as defined in [19]. Event ex-

pressions provide a means of defining a behavior by algebraically defining a language over a set of event names. Essentially all of the descriptional capabilities of event expressions are available in DDN, but they have been put in a form which makes them more easily used.

The DDN behavior specification constructs have also borrowed some concepts from path expressions [4]. In particular, the specification of behavior as a set of partial specifications is taken from path expressions. However, the specification of behavior in DDN is radically different in intent from path expression behavior description. Path expressions were originally developed as a programming language construct for imposing scheduling and synchronization constraints upon a community of concurrent processes. Behavior specification in a DDN description is not prescriptive in this sense -- the specification's intent is to report the desired behavior rather than to lead to the automatic generation of synchronization code. Thus, the behavior specification prescribes necessary scheduling and synchronization only in the sense that it indicates the subsystem designer's intentions as to what behavior is legal.

Internal Description of a Subsystem

A subsystem's external description provides a definition of how the subsystem may be used and how it will interact with other subsystems in its environment. It provides all the information about the subsystem that one is allowed to know when developing other subsystems which execute concurrently with the defined subsystem.

An internal description, however, gives details about the internal componentry of the subsystem and the manner in which the components interact to create the effects defined in the subsystem's external description. The internal description is what would typically be produced by a series of steps in a top-down design method.

A subsystem's internal description must define three aspects of the subsystem's internal operation. First, it must define the components which comprise the subsystem. These will be both subsystem

components and shared data objects. In this paper, we focus exclusively upon subsystem components[5]. The second aspect which must be defined is the communication pathways among the subcomponents. This involves the definition of links and the attachment of ports to links so that messages may flow among the subcomponents. Finally, the message flow into and out of the subsystem must be related to the message flow into and out of the collection of subcomponents. This is done by establishing controllers which distribute incoming messages among the subcomponents and collect messages from the subcomponents and pass them out through the subsystem's out ports.

Subcomponent Declaration

Subcomponents will themselves be instances of some other class of subsystems. Thus a definition of the subcomponents consists of a set of declarations, each specifying a name for a subcomponent and indicating the class of which it is an instance.

The declaration of one possible set of subcomponents for [knowledge_source] subsystems appears in figure 9. The *activator* subcomponent receives the signals sent by the data base and evaluates a logical condition to determine whether or not one of the *manipulator* objects is to be activated. The *manipulator* objects await activation by the *activator* and then interact with the data base to effect any necessary changes.

```
▼[knowledge_source]:  SUBSYSTEM CLASS▼

   SUBCOMPONENTS;
      activator OF [pre_condition (#_servers)],
      manipulator ARRAY [1::#_servers]
                        OF [data_base_modifier]
      END SUBCOMPONENTS;
```

Figure 9

[5] To avoid having to use the awkward term sub-subsystem, the term subcomponent is used for the remainder of this paper to mean a subsystem which is a component of the subsystem being described.

The subcomponentry of this example is relatively simple. More extensive facilities are available in DDN for describing subcomponents and these are discussed in [22].

Subcomponent Connections

Communication pathways must be established among the subcomponents so that they may interact. Since links are used to control message transmission, it is necessary to establish the requisite links and specify which ports are connected to which links.

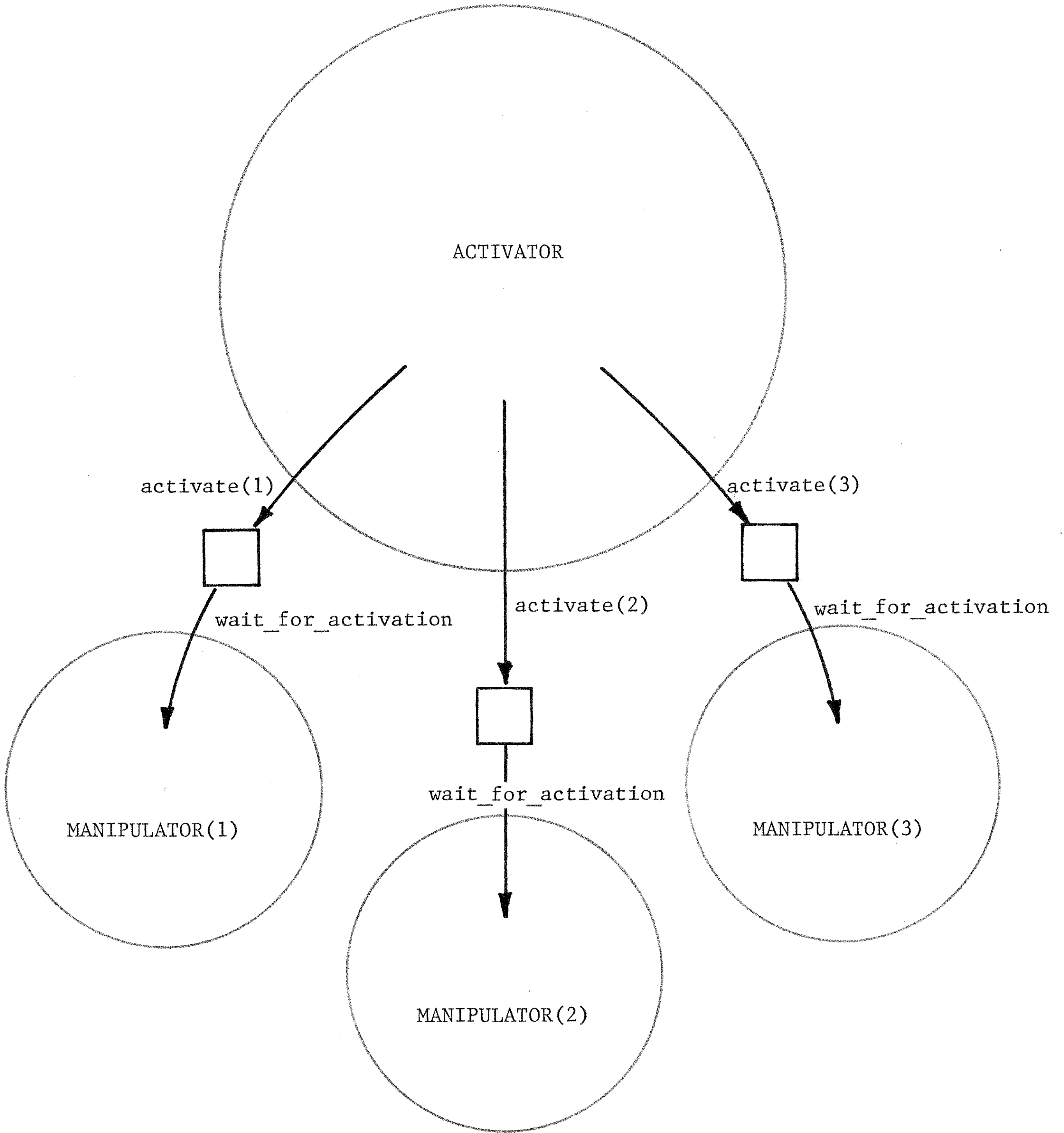
In DDN, this is accomplished by "plugging" ports together. For every set of ports that are plugged together, a link is established and the ports are attached to the link. In the example, communication pathways may be described by the textual unit appearing in figure 10. This establishes a set of links, each of which has an *activate* port from one of the *activator* objects and a *wait_for_activation* port from one of the *manipulator* objects attached to it. This leads, when there are three servers, to the communication pathway structure depicted in figure 11.

```
▼[knowledge_source]:  SUBSYSTEM CLASS▼

   CONNECTIONS;
      FOR ALL i IN [1::#_servers]
         PLUG (activator | activate (i), manipulator (i) |
                                           wait_for_activation) ;
         END FOR;
      END CONNECTIONS;
```

Figure 10

The control constructs provided for writing programs in a connection textual unit are essentially those provided for writing control process models except that non-deterministic constructs are not allowed. Also, the variables that are referenced in a plug program must be able to be evaluated prior to subsystem "execution" since the structure of a subsystem is static and may not change during the running of the system.[6]

[6]We have investigated extensions to the DDN modelling language which would allow the description of dynamic system structure [37], but have not incorporated these extensions in the current version of DREAM.



subcomponents are represented by circles containing the subcomponent name in capital letters

links are represented by boxes

Figure 11

The facilities provided by DDN for establishing communication pathway structures permit any (static) pathway structure to be defined. Note that it is only through the control of the pathway structure that message transmission may be directed from one subcomponent to another -- the send operation cannot specify which subcomponent is to ultimately receive a message. Thus the pathway structure itself defines all of the possible communication interactions. This is beneficial for the purpose of defining a system's structure. Operations, such as defined in [8], which permit constraint of the ultimate receiver could lead to more efficient system implementations but would obscure the definition of the interactions. Thus, DDN uses relatively primitive constructs which force the explicit representation of interactions.

Message Distribution

The messages which flow into the subsystem are received by the subcomponents and the messages flowing out of the subsystem are produced by the subcomponents. Thus there are some relationships among the ports of the subcomponents and the ports of the subsystem. In many cases, this is a simple identification of a port in a subsystem with a port in one of the subcomponents so that, for instance, when the subcomponent sends a message through its port, the message flows out through the subsystem's port. This simple identification occurs when there is no intermediate processing required as the message passes out of the subsystem. More complex cases arise when intermediate processing is required -- for example, an incoming message may need to be broken up into parts which are distributed among the subcomponents.

For the simple cases in which a subsystem port is identified with a port in a subcomponent, DDN allows ports to be overlaid as shown in figure 12. The OVERLAY statements establish an equivalence between one of the ports of the subsystem and one of the ports within a subcomponent. When the ports are overlaid, no overhead processing is incurred in order to get messages into and out of

the subcomponents since they have direct paths to the subsystem's ports and hence to the environment in which the subsystem is operating.

```
▼[knowledge_source]:  SUBSYSTEM CLASS ▼

   CONNECTIONS;
      FOR ALL i IN [1::#_servers]
         OVERLAY (manipulator (i) | request_out,
                                     make_request(i) );
         OVERLAY (manipulator (i) | answer_in, get_answer(i));
         END FOR;
      END CONNECTIONS;
```

Figure 12

More complex message distribution among the subcomponents is described in DDN by establishing special subcomponents which control the flow of messages between the subsystem's ports and the ports of the subcomponents. Since the purpose of control processes is to control the flow of messages through the subsystem's ports, these special subcomponents are described as bodies for the control processes. For our example, this is shown in figure 13. The body of a control process is a control program over 1) send and receive operations upon the subsystem's ports and the control process' ports and 2) operations upon the buffer and data object components of both the subsystem and the control process. Note that the operation <u>assign</u> has been invoked upon the *db_signal* buffer, passing as a parameter the value of the *signal* buffer -- the operation assign must be among the operations defined for the class [on_off_signal]. The control constructs that are available for stating control process bodies are the same as those for specifying control process models except that the non-deterministic constructs may not be used.

The textual units given in this section give rise to a set of communication paths among the subcomponents. For #_*servers* being 3 and #_*values* being 4, the paths would appear as pictured

```
▼[knowledge_source]:  SUBSYSTEM CLASS▼

  listener:  ARRAY [1::#_values] OF CONTROL PROCESS;
     pass_on: LOCAL OUT PORT;
        BUFFER SUBCOMPONENTS;
           db_signal OF [on_off_signal]
           END BUFFER SUBCOMPONENTS;
        END OUT PORT;
     MODEL;  ITERATE
          hear: RECEIVE await(MY_INDEX);
                END ITERATE;
              END MODEL;
     BODY;  ITERATE
              RECEIVE await(MY_INDEX);
              db_signal.assign(signal(MY_INDEX));
              SEND pass_on;
              END ITERATE;
          END BODY;
     END CONTROL PROCESS;

▼[knowledge_source]:  SUBSYSTEM CLASS▼

  CONNECTIONS;
     FOR ALL i IN [1::#_values]
        PLUG (listener(i)|pass_on, activator|get_signal);
        END FOR;
     END CONNECTIONS;
```

Figure 13

in figure 14. In comparing figure 6 and figure 13, notice that the *listener* control processes define, in this implementation, actual subcomponents whereas the *requester* control processes do not. This is because each *listener* control process models the operation of a single subcomponent and this subcomponent can most naturally be described as the body of the control process. Thus control process definitions serve two purposes. First, they may define, through a model textual unit, the message transfer effect of the operation of the subcomponents. Second, they may specify, through a body textual unit, the operation of an actual subcomponent.

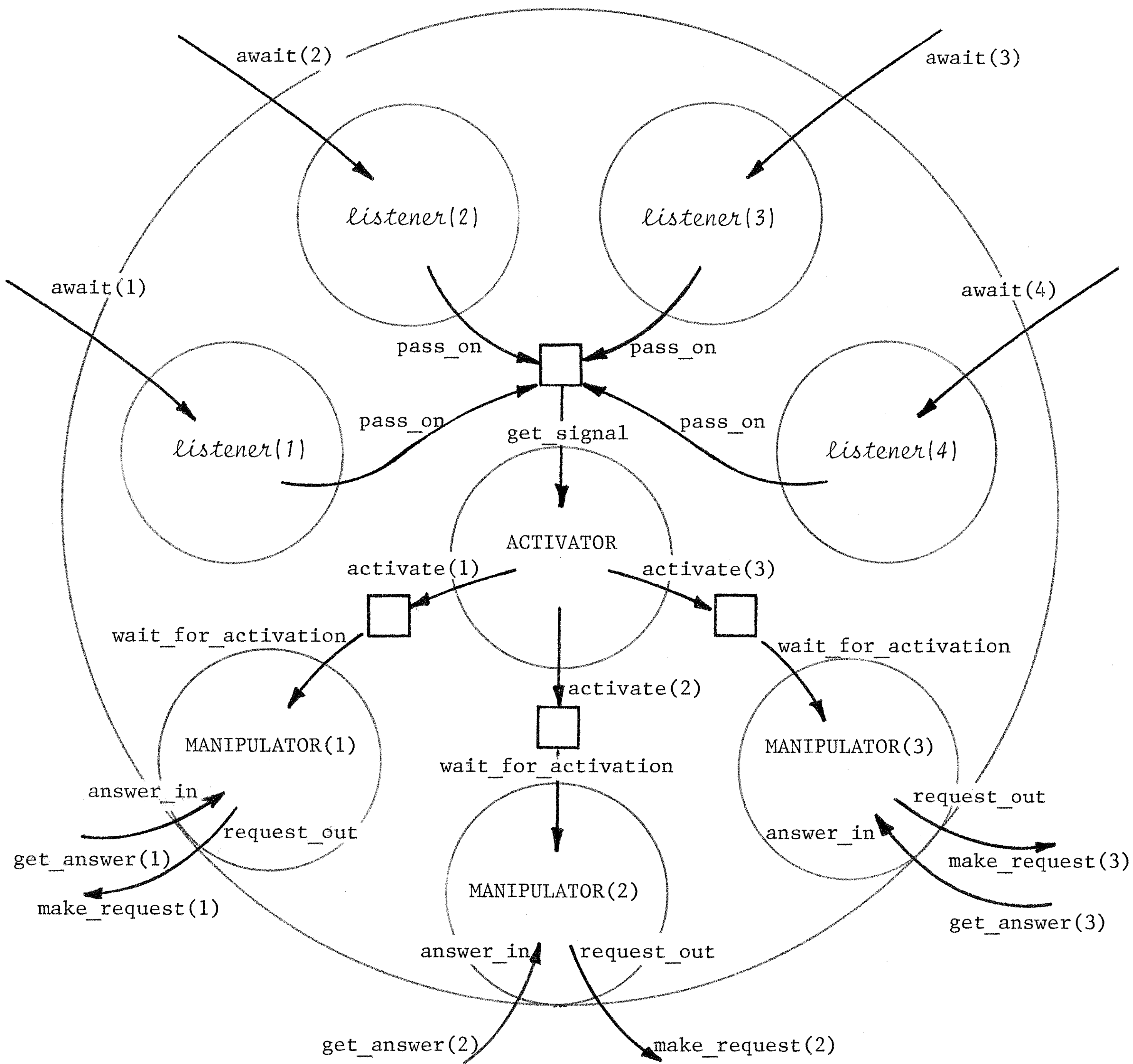


Figure 14

Analysis of Subsystems

The scheme that we have developed for describing collections of sequential processes affords many opportunities for rigorous analysis.[7] The simplest forms of analysis involve checks that can be made without interpretation of the text describing a subsystem's operation. Since such analysis would not account for the dynamic, run-time characteristics of the subsystem, it can be called static analysis. One example is the derivation of graphs such as in figure 14. Such paraphrasing of the subsystem's description in a form different from that used by the designer is frequently valuable in uncovering inconsistencies in the organization of a collection of subcomponents.

Another static analysis that could be performed is a check that referenced entities, such as ports or classes, have actually been defined. Such an analysis is a check of the completeness of the subsystem's design, rather than a correctness check, since the designer may not have yet designed the referenced entity. This indicates that much of the analysis performed during design is feedback analysis [20] in which it is left to the designer to conclude whether correctness or incorrectness of the design can be determined, based upon the derived information.

Of much more value to a designer is dynamic analysis in which information is derived about the run-time characteristics of the system under design. While dynamic analysis could be delayed until the system is completely designed, it is of more value when performed early in the design process, at which point it serves to predict the system's eventual run-time characteristics and offers the opportunity to uncover errors when they may more easily be corrected. The discussion of dynamic analysis will be with respect to this context -- in particular, implicit in the discussion is that dynamic analysis will be used as an integral part of a top-down design method.

---

[7] A general taxonomy of different types of analysis that would be useful is given in [35].

One form of dynamic analysis consists of checking the consistency among the various parts of a subsystem's external description. While the desired behavior construct of DDN is intended for a different purpose than the control process model construct, one way to achieve the redundancy necessary for consistency checking is to give a desired behavior specification of the behavior specified by the control process models. Such redundancy may also arise from desired behavior specifications appearing elsewhere in the DDN description of the system which have implications for the behavior that can legally be specified by the subsystem's control process models.

A check of the consistency between the desired behavior specification and the control process models would proceed as follows. Using an algorithm similar to that defined in [19], a description of the sequences of events defined by a control process model would first be derived. The desired behavior specification and the control process model would then be consistent if, for those events common to the two descriptions, all sequences defined by the control process model are also defined by the desired behavior specification[8].

Another dynamic analysis that could be performed is a check of the consistency of the interfaces among a collection of subsystems. A subsystem's external description may be used to indicate both how it responds to messages from other subsystems and what responses it expects from other subsystems. Therefore, once a configuration of subcomponents has been established, the consistency of the subcomponents' external descriptions can be checked. This analysis could be performed in the same manner as described above. The event sequences resulting from the operation of the control process models could be determined and then these sequences can be checked for inclusion in the sets of sequences specified by the desired behavior descriptions.

---

[8] The requirement that sequences defined by control process models be included in the set of sequences defined by desired behavior descriptions follows from the constraint nature of desired behavior descriptions.

A final form of dynamic analysis is the checking of the consistency between the external description of a subsystem's behavior and the behavior actually created by the subsystem's implementation. This would involve using the bodies of the control processes and the external descriptions of the subcomponents to determine the actual sequences of message flow through the ports. This would be consistent with the control process model definition of the subsystem's behavior as long as this set of message flow sequences was a subset of the set defined by the control process models.[9]

Alternatively, the event sequences caused by the operation of the internal components could be determined and compared to the event sequences specified as desired behavior. This can be done as long as event definitions exist which relate the events defined for the subsystem and the events defined for the subcomponents. With such definitions, the set of event sequences for the subcomponents could be mapped onto a set of event sequences for the subsystem. This set could then be compared, for containment, with the set of event sequences specified by the desired behavior description. (This type of analysis is used in the TOPD program development system [11].)

These dynamic analysis techniques all require some means of comparing two sets of sequences. In general, this comparison cannot be performed algorithmically [19]. Therefore, either algorithms for special situations would have to be used and cases which did not fall into these situations would have to go unanalyzed, or the comparison would have to be performed by the designer. The latter approach may not be unreasonable, as the designer may have the insight and intuition necessary to show equality or inclusion or find counter-examples.

---

[9] Alternatively, it could be required that the two sets be equal, reflecting a more strict condition that all, as well as only, the specified behavior is actually achieved.

The inability to algorithmically perform the entire analysis is not necessarily a serious limitation. First, in the absence of rigid interpretations imposed by specific analysis techniques, a designer is free to impose whatever relationships between the sets of sequences are deemed appropriate.[10] Thus designers may adjust the analysis to either their own style of design or the nature of the behavior being described. For example, a designer might use the desired behavior descriptions to indicate bounds upon the actual behavior such that part of the specified behavior must be achieved and part of it would be legal but is not required.

Second, simulation-based analysis may be used when formal (or informal) analytic techniques do not exist or are computationally unattractive. Simulation is possible since the procedural descriptions of a subsystem (i.e., the control process bodies) provide rigorous definitions of its operation. Several constructs have been included in DDN in order to increase the effectiveness of simulation. First, the buffer conditions can be checked, during simulation, whenever a message passes through the port associated with the buffer condition. Second, a history of the events occurring during the simulated execution could be checked against the set of desired sequences. In addition, the DDN notation could be extended to allow its use in the prediction of a subsystem's performance characteristics, following an approach defined in [29]. The description scheme would be augmented with constructs for associating probabilities with nondeterministic operations and time distributions with modelled operations. Once this was done, the simulator could accumulate statistics concerning characteristics such as running time, resource utilization and queue sizes.

---

[10]It is our belief that any fixed set of predetermined interpretations would omit some relationships significant to some designers, leaving those relationships to be checked by the designers themselves, without assistance.

Conclusions

We have presented a scheme for describing collections of asynchronous sequential processes. The scheme provides for the description of the communication pathways among the processes in the collection and hence the interactions among the processes. This allows the description of the internal, operational details of the collection. The scheme also provides for the description of the behavior of the collection as a whole in terms of the message flow into and out of the collection. This allows the description of the ways in which the collection may be used without a description of how this behavior is achieved.

We have used this scheme in formulating descriptions for a variety of software systems ([5],[6],[23],[31],[36],[37]), and have also used it in one exercise [34] which attempted to simulate a design effort. We feel that the scheme demarcates an important set of facilities for the description of software systems and that it provides a valuable basis for a variety of tools to aid designers of large-scale software systems.

We feel that effective design aid tools based upon this modelling scheme should allow the derivation of information helpful in determining whether or not the design being developed is correct. Some of this information may be a paraphrase of the designer's description in a form that allows the designer to more easily see inconsistencies which may exist. More helpful information is that which indicates the dynamic, run-time behavior of the system in a form which allows algorithmic or heuristic comparison with the behavior which the designer intends to have happen.

In and of itself, the description scheme introduces a rigor into the design of large-scale software systems which allows the designer to more carefully develop and expand the design. In this regard, the scheme is particularly useful in conjunction with a top-down design method. The ability to analyze the design and derive information that is not explicitly represented provides

the designer with a means of checking the appropriateness of the design at each design step. This incremental analysis of the design affords the opportunity to check design decisions as they are made rather than at the end of the design process.

Acknowledgements

The development of abstract process types was greatly facilitated by the contributions and constructive criticism of Jack Wileden, Alan Segal, Allan Stavely, John Sayler, Dirk Kabcenell and Victor Lesser.

REFERENCES

1. A.L. Ambler, et al. GYPSY: A Language for Specification and Implementation of Verifiable Program. ICSCA-CMP-2, Certifiable Minicomputer Project, Univ. of Texas, Austin, January 1977.

2. P. Brinch Hansen. The Programming Language Concurrent Pascal. IEEE Trans. on Software Engineering, 1, 2 (June 1975), 199-207.

3. P. Brinch Hansen. The Architecture of Concurrent Programs. Prentice-Hall, Englewood Cliffs, N. J., 1977.

4. R.A. Campbell and A.N. Habermann. The Specification of Process Synchronization by Path Expressions. Lecture Notes in Computer Science, 16, Springer Verlag, Heidelberg, 1974.

5. J.Cuny. A DREAM Model of the RC4000 Multiprogramming System. RSSM/48, Dept. of Computer and Comm. Sciences, Univ. of Mich., Ann Arbor, July 1977.

6. J.Cuny. The GM Terminal System. RSSM/63, Dept. of Computer and Comm. Sciences, Univ. of Mich., Ann Arbor, August 1977.

7. O. Dahl and K. Nygaard. SIMULA -- an ALGOL-Based Simulation Language. Comm. ACM, 9, 9 (September 1966), 671-678.

8. J.A. Feldman. A Programming Methodology for Distributed Computing (among other things). TR9, Dept. of Computer Science, Univ. of Rochester.

9. R. Fennel and V. Lesser. Parallelism in Artificial Intelligence Problem Solving: A Case Study of Hearsay II. IEEE Trans. on Computers, C-26, 2 (February 1977).

10. A.N. Habermann. Introduction to Operating System Design. SRA, Chicago, 1976.

11. P. Henderson. Finite State Modelling in Program Development. Proc. 1975 International Conf. on Reliable Software, Los Angeles, April 1975.

12. P. Henderson, et al. The TOPD System. Tech. Report 77, Computing Laboratory, University of Newcastle upon Tyne, England, September 1975.

13. C.A.R. Hoare. An Axiomatic Basis for Computer Programming. Comm. ACM, 12, 10 (October 1969), 576-580, 583.

14. C.A.R. Hoare. Notes on Data Structuring. In Dahl, Dijkstra and Hoare, Structured Programming, Academic Press, New York, 1973.

15. C.A.R. Hoare. Monitors: An Operating System Structuring Concept. Comm. ACM, 17, 10 (October 1974), 549-557.

16. J.J. Horning and B. Randell. Process Structuring. Computing Surveys, 5, 1 (March 1973), 5-30.

17. B.H. Liskov and S.N. Zilles. Specification Techniques for Data Abstractions. IEEE Trans. on Software Engineering, SE1, 1 (March 1975), 7-19.

18. D.L. Parnas. Information Distribution Aspects of Design Methodology. Proc. IFIP Congress 71, Ljubljana, August 1971, pp. TA-3-26-TA-3-30.

19. W.E. Riddle. An Approach to Software System Modelling, Behavior Specification and Analysis. RSSM/25, Dept. of Computer and Comm. Sciences, Univ. of Michigan, Ann Arbor, July 1976 (revised November 1977).

20. W.E. Riddle. A Formalism for the Comparison of Software Analysis Techniques. RSSM/29, Dept. of Computer and Comm. Sciences, Univ. of Michigan, Ann Arbor, July 1977.

21. W. Riddle, J. Sayler, A. Segal and J. Wileden. An Introduction to the DREAM Software Design System. Software Engineering Notes, 2, 4 (July 1977).

22. W. E. Riddle. Hierarchical Description of Software System Structure. RSSM/40, Dept. of Computer Science, Univ. of Colorado at Boulder, November 1977.

23. W.Riddle. DREAM Design Notation Example: The T.H.E. Operating System. RSSM/50, Dept. of Computer Science, Univ.of Colorado, Boulder, April 1978.

24. W.Riddle, J. Sayler, A. Segal, A. Stavely and J.Wileden. A Description Scheme to Aid the Design of Collections of Concurrent Processes. Proc. National Computer Conf., Anaheim, June 1978.

25. W.Riddle, J.Wileden, J.Sayler, A.Segal and A.Stavely. Behavior Modelling During Software Design. IEEE Trans. on Software Engineering, SE-4, 4 (July 1978).

26. W.Riddle, J.Sayler, A.Segal, A.Stavely and J. Wileden. DREAM: A Software Design Tool. Proc. 3rd Jerusalem Conf. on Information Technology, Jerusalem, August 1978.

27. W. Riddle. DDN User's Guide. RSSM/37, Dept. of Computer Science, Univ. of Colorado at Boulder, in preparation.

28. W. Riddle. Abstract Monitor Types. RSSM/41, Dept. of Computer Science, Univ. of Colorado at Boulder, in preparation.

29. J. Sanguinetti. Performance Prediction in an Operating System Design Methodology. RSSM/32 (Thesis), Dept. of Computer and Comm. Sciences, Univ. of Michigan, Ann Arbor, May 1977.

30. J. Sayler. Philosophy of the DREAM System. RSSM/39, Dept. of Computer and Comm. Sciences, Univ. of Michigan, January 1977.

31. A.Segal. DREAM Design Notation Example: A Multiprocessor Supervisor. RSSM/53, Dept. of Computer and Comm. Sciences, Univ. of Michigan, Ann Arbor, August 1977.

32. A. Segal. Design Description Management. RSSM/45, Dept. of Computer and Comm. Sciences, Univ. of Michigan, in preparation.

33. H.A. Simon. The Architecture of Complexity. Proc. Am. Phil. Soc., 106, (December 1962), 467-482. Also in Simon, Sciences of the Artificial, MIT Press, Cambridge, 1969.

34. A.M.Stavely. DREAM Design Notation Example: An Aircraft Engine Monitoring System. RSSM/49, Dept. of Computer and Comm. Sciences, Univ. of Michigan, Ann Arbor, July 1977.

35. A.M. Stavely. Feedback Aids in Software Design Aid Systems. RSSM/60, Dept. of Computer and Comm. Sciences, Univ. of Michigan, Ann Arbor, November 1977.

36. J.Wileden. DREAM Design Notation Example: Scheduler for a Multiprocessor System. RSSM/51, Dept. of Computer and Comm. Sciences, Univ. of Michigan, Ann Arbor, October 1977.

37. J.Wileden. Modelling Parallel Systems with Dynamic Structure. RSSM/71 (Thesis), Dept. of Computer and Comm. Sciences, Univ. of Michigan, Ann Arbor, January 1978.

38. J. Wileden. Behavior Specification in a Software Design Aid System. RSSM/43, Dept. of Computer and Information Science, Univ. of Massachusetts, August 1978.

39. N. Wirth. Modula: a Language for Modular Multiprogramming. Software - Practice and Experience, 7, (1977), 3-35.

40. W.A. Wulf, R.A. London and M. Shaw. Abstraction and Verification in ALPHARD: Introduction to Language and Methodology. Report 76-46, Information Sciences Inst., Univ. of Southern California, 1976.

Abstract

Abstract process types are introduced as a means of giving high-level descriptions of the components in large-scale software systems. Abstract process types provide facilities for the description of a system's processing components that are analogous to the facilities which abstract data types provide for describing a system's data storage components. In particular, abstract process types provide a basis for approaches to verification that parallel the approaches developed in conjunction with abstract data types.

Key Words and Phrases

abstract process types, abstract data types, hierarchical system description, non-procedural specification, verification, DREAM

Introduction

The design and implementation of large-scale software systems requires description schemes which allow the abstract, high-level specification of the system. Without the ability to capture the essence of the various processing and data structuring components, a system's description quickly becomes complicated and incomprehensible. It is only through a description of the system as a collection of less-complex parts which interact in well-defined ways that anyone trying to understand the system can succeed within a reasonable period of time.

Abstract data types have recently emerged as an important facility for abstract description in the domain of sequential programs. The basic concept originally appeared in the simulation language SIMULA [7] and was introduced into programming languages as a facility to support structured programming [14]. Their recent refinement has taken three directions. First, they have been developed to support top-down design methods [17]. Second, they have been extended so as to provide a basis for formal verification [40]. Finally, they have been used as the basis for a tool to aid program development [12] which admits rigorous, but perhaps incomplete, analysis [11]. The variety of these uses indicates the breadth of the benefits which accrue from facilities for high-level, abstract description.

Abstract data types are ideally suited to the high-level description of a software system's data storage components. But they are not adequate for the succinct description of those system components which are intended more for the processing of data than for the storage of data. This is particularly true when the components operate concurrently[1]. The major problem is that abstract data types are oriented toward describing components as structures of data which are operated upon via procedure calls. Many components -- e.g. a file system in a multiprocessor computing facility -- are not naturally described as collections of data objects which are accessed in a basically sequential manner.

---

[1]By concurrent we mean parallelism which may actually be achieved by running the system in a multiprocessing environment or which may only be apparent at abstract levels of description of the system.

Extensions and modifications to abstract data types could be developed to increase their effectiveness in describing processing components in software systems -- this has been done in the Gypsy system [1] and the Modula programming language [39]. In this paper, however, we develop a description scheme that retains many of the concepts of abstract data types but is based upon the concept of a sequential process [16]. This leads to a means of abstractly describing processing structures which focuses upon describing a system as collection of interacting, concurrent subsystems.

In the next section, the view of a system as a collection of interacting subsystems is made more concrete as we describe the various ways in which the subsystems may interact. Following that, the various aspects of abstract process descriptions are presented. First, the external description of a subsystem in terms of how it interacts with other subsystems is discussed. Then, the discussion turns to the manner in which an internal description of a subsystem may be given in terms of coordinated interactions among its (sub-)subsystems. In the final section, we suggest that the scheme admits approaches to verification which are similar to those developed in conjunction with abstract data types.

The constructs discussed here are part of the design language developed for the Design Realization Evaluation and Modelling (DREAM) system ([19], [24]-[26]). DREAM is a tool for the design of large-scale software systems, providing the designer with bookkeeping and analysis aid. A description in the DREAM Design Notation (DDN) consists of a collection of (nested) description fragments, called <u>textual units</u>. DREAM provides facilities which allow the user (or users, in the case of a design being carried out by a design team) to modify the information in a data base on a textual unit basis.

The focus of this paper is upon the DDN constructs for describing abstract process types. Other aspects of DDN are discussed in [22], [28], [32], and [38]. Also the focus is upon the use of the constructs -- their syntax is covered in [27]. Some justifications for the constructs are given; others lie within the DREAM system's general philosophy which is discussed in [26] and [30].

An Abstract View of Systems

A convenient view of software systems is that they are composed of parts, subsystems, which operate concurrently and asynchronously. This may be made concrete by viewing the subsystems as interacting solely through the sending and receiving of messages and the sharing of data objects. This is a bit redundant since the shared data objects could be viewed as subsystems which receive read and write commands as messages and send messages containing the requested value in response to a read command. But recognizing shared data objects as distinct from subsystems allows message exchange to be used to focus upon control interactions among subsystems. Also, data objects are needed to describe the structure of the messages transmitted among the subsystems.

This paper focuses upon the description of control interactions among asynchronously operating subsystems. We assume, therefore, the existence of a scheme for describing data objects that may be shared by a community of concurrent subsystems. (Several such schemes have been developed, e.g., [2], [15], [39]. A scheme that was developed in conjunction with the work discussed here is described in [28].) Only two aspects of the scheme are important here. First, a definition of a data object specifies all the procedures which may be invoked upon it. Second, data objects have built-in synchronization mechanisms which can be used to preclude interference among the operations performed upon the object.

Communication among asynchronous message senders and receivers requires a transmission controller that is able to store both messages that have been sent but not yet received and requests for messages which have been lodged but not yet satisfied. In DDN, an idealized controller, called a link, is provided. Links hold messages and requests in (unbounded) bag data structures. Thus they do not necessarily pass messages on in the order the messages were sent, nor do they necessarily service requests for messages in the order the requests were lodged.

As an example of this view of software systems, consider the following description of HEARSAY [9], a multiprocessor speech processing system developed at Carnegie-Mellon University. HEARSAY may be decomposed into two major parts. The first is a data base, called the blackboard, which contains all the information about the utterance being processed and the hypotheses which have been made as to its linguistic structure. The second part is a collection of processing subsystems, each called a knowledge source. A knowledge source inspects the information in the data base and augments or modifies it according to rules which it is programmed to enforce. The subsystems in this description of HEARSAY are therefore the blackboard and the knowledge sources. Note that the blackboard could alternatively be described as a shared data object since its primary function is the storage of data and the operations performed upon it by the knowledge sources must be synchronized in some way. We ignore this design choice and focus upon the knowledge sources.

The interactions among the HEARSAY subsystems arise as follows. It would be wasteful to have each knowledge source constantly inspect the data base to see if it should perform any processing upon it. Therefore, the data base is programmed to know which data base entries are of interest to each knowledge source and to send a signal to a knowledge source when one of the data base entries of interest to it changes value. When a knowledge source is awakened in this way, it inspects the data base and makes any modifications deemed necessary.

Multiprocessing systems, such as HEARSAY, may naturally be described in the manner advocated here since these systems actually have subsystems which interact by message exchange. But this view is also appropriate, as evidenced by two recent texts ([3], [10]), when it is only a logical one and the system actually runs in a uniprocessor environment. In this case, the view facilitates the decomposition of the system and hence the mastery of its complexity [33]. The transmission of messages may never really take place, and message interchange may be used only as a <u>model</u> of the actual interactions. This is the primary reason that a more sophisticated transmission control mechanism is not defined within DDN.

In the following sections, a DDN description of the knowledge source subsystems of HEARSAY[2] will be developed. All of the knowledge source subsystems are similar with respect to their interactions with the data base. Therefore, the description is of the class of knowledge source subsystems. To reflect that different instances of this class vary with respect to their details, such as the number of entries in the data base that are of interest to the knowledge source, we define a parameterized class. This means that part of the class definition is a specification of qualifiers which may be assigned values when an instance of the class is created. Class definitions and qualifiers are discussed in [22].

External Description of a Subsystem

An external description of a subsystem defines the ways in which the subsystem interacts with other subsystems in its environment. In keeping with the principle of information hiding [18], the external description indicates only the effect of the combined operation of the subsystem's internal components and defines nothing as to the manner in which the internal components are organized or actually interact.

The external description also serves to guide the implementation of the subsystem. Being, by definition, a complete specification of the subsystem, it provides the designer with a definition of the (minimal) behavior which must be implemented. It may also guide the designer's decision making, helping in the determination of the set of decisions to be made and the inter-relationships among the decisions.

The external description must define three aspects of the subsystem. First, it must specify the interfaces to the subsystem not only in terms of the format of the information flowing through each interface but also in terms of what information may legally flow through each interface. Second, it must specify the correlation between the information flowing through one interface and the information flowing through another interface. Finally, it must relate the system's operation at one point in time to its operation at a previous point in time -- that is, it must specify the more global aspects of the subsystem's operation.

---

2 The description is an approximation of the structure and operation of knowledge sources actually in HEARSAY. It reflects our understanding of the description which appears in [9], but has also been tailored so as to provide examples of the facilities in DDN.

Subsystem Interfaces

An interface to a subsystem is a port through which messages may flow. Conceptually, a port is a named communication line along which messages flow, one at a time, and which does not have any message storage capabilities. Ports correspond to one-way communication lines and therefore have a direction in or out.

The messages which flow through a port are sets of data objects. Each port therefore has associated with it a set of buffers, each able to store one data object. The set of buffers is ordered, in the same sense and with the same implications as the ordering among a set of parameters to a procedure. The set of buffers indicates the types of data objects which comprise a message and the order in which the data objects are composed to form a message.

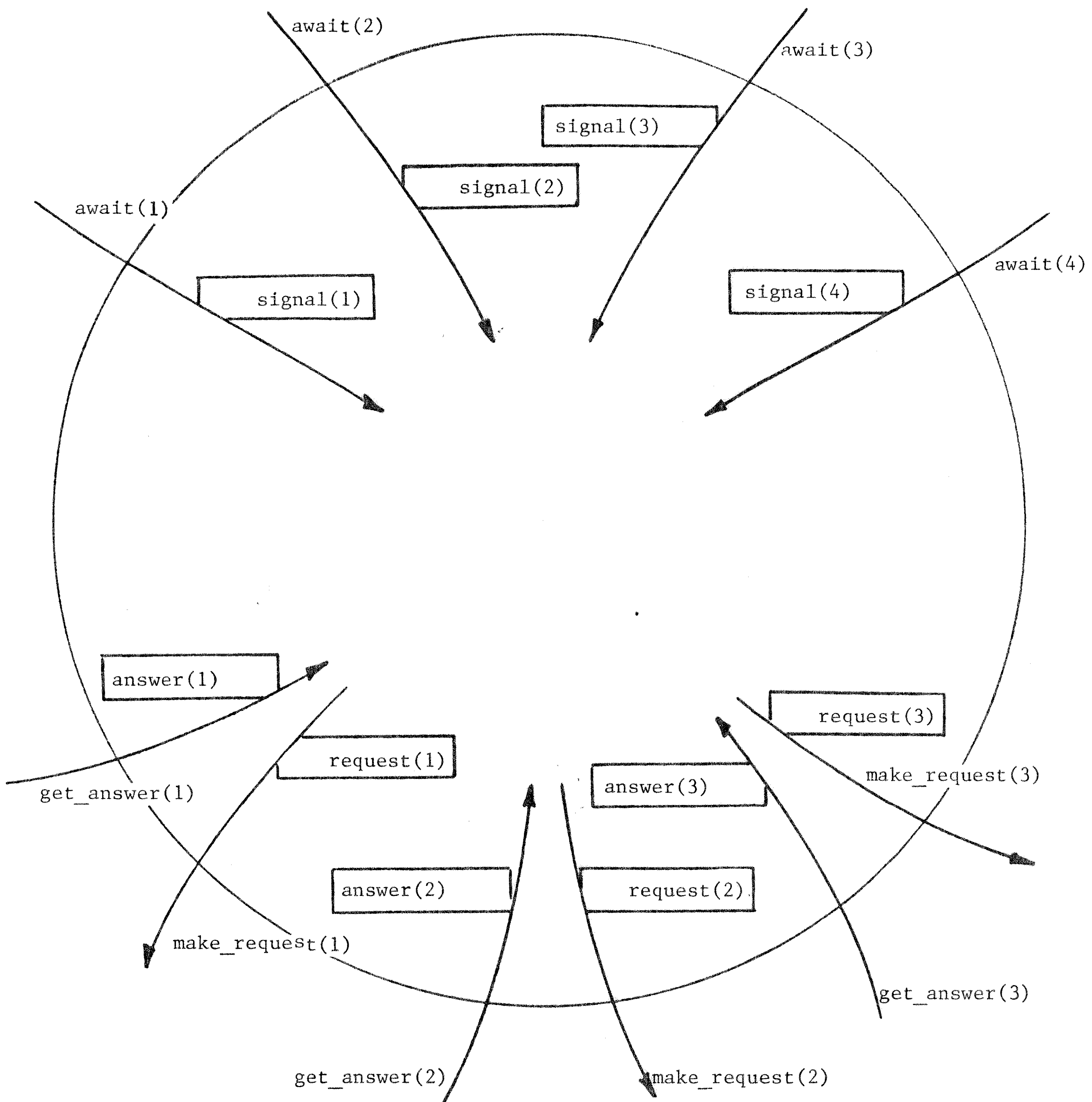
The messages that may legally flow through a port are specified by giving buffer conditions for the port. A buffer condition is a predicate over the buffer data objects, indicating the set of legal values for the buffer data objects as well as the legal correlations among the values. OR'ed together, the buffer conditions associated with an in-port (out-port) are analogous to a pre-condition (post-condition) [13].

In DDN, a port is defined by giving a textual unit which specifies the port's name and direction and which has nested textual units which specify the buffers and the buffer conditions associated with the port. A set of ports is defined in figure 1 for objects of class[3] [knowledge_source]. If an object of this class were created with *#_values* having the value 4 and *#_servers* having the value 3, then the object would appear as pictured in figure 2. There are ten ports, grouped into three arrays and each port has a singleton set of buffers associated with it. Notice that defining an array of ports implies that there is an array of sets of buffers, one set for each port. The buffer conditions associated with the *make_request* ports indicate that messages flowing out through these ports will have only the values *inspect* or *modify*. The absence of buffer conditions associated with the other

[3] It is convention in DDN to enclose an identifier in square brackets when it is used to name a class.

```
[knowledge_source]:  SUBSYSTEM CLASS;

   QUALIFIERS;
      DOCUMENTATION;
         #_values is the number of values monitored
         for this knowledge source; #_servers is
         the number of parallel servers in this
         knowledge source
         END DOCUMENTATION;
      #_values, #_servers
      END QUALIFIERS;

   await:  ARRAY [1::#_values] OF IN PORT;
      BUFFER SUBCOMPONENTS;
         signal OF [on_off_switch]
         END BUFFER SUBCOMPONENTS;
      END IN PORT;

   make_request:  ARRAY [1::#_servers] OF OUT PORT;
      BUFFER SUBCOMPONENTS;
         request OF [data_base_operation]
         END BUFFER SUBCOMPONENTS;
      BUFFER CONDITIONS;
         request=inspect,
         request=modify
         END BUFFER CONDITIONS;
      END OUT PORT;

   get_answer:  ARRAY [1::#_servers] OF IN PORT;
      BUFFER SUBCOMPONENTS;
         answer OF [data_base_response]
         END BUFFER SUBCOMPONENTS;
      END IN PORT;

   END SUBSYSTEM CLASS;
```

Figure 1



ports are represented by labelled arcs
buffers are represented by boxes attached to the arcs

Figure 2

ports indicates that there are no restrictions on the values of the data objects which compose messages flowing through these ports - this freedom could be reduced as the subsystem is further designed.

A port definition is comparable to a heading for a procedure stated in a programming language. It is similar in that it specifies a name by which the port (procedure) may be referenced and the number, type and order of the data objects (parameters) in messages (parameter lists) processed by the internal components (procedure body). It differs because ports allow one-way, asynchronous communication whereas procedures provide two-way, synchronous communication.

The buffer conditions of DDN are analogous to entry and exit specifications of Alphard [40] with the additional aspect that they are required to be valid whenever a message flows through the port. Buffer conditions may be viewed as assumptions which may be made while formally verifying either a design of the subsystem or a design of another subsystem which uses the subsystem. Or they may be viewed as indicating checks to be made during the execution or simulation of the subsystem. They have been included in DDN with the intent of using them in both these ways.

Message Flow Through a Subsystem

The role that a subsystem plays within a community of subsystems is specified by a definition of the correlations among the messages flowing into and out of the subsystem. This can serve to define either the facilities provided by the subsystem or its utilization of the facilities provided by the other subsystems. Note that this means there will be a redundant specification of the interaction among subsystems, with a definition of the interaction within both the users and the provider of a facility. This is useful redundancy, affording the opportunity to perform verification as will be discussed later.

Succinct definition of the subsystem's control of message flow is frequently procedural in nature -- this is especially true when programmers are the intended audience, since such descriptions are quite natural to them. This leads to an ambiguity as to whether the description is actually an external one or whether it is not also

describing the system's internal operation. While a procedural definition may correspond very closely to the subsystem's internal operation, it may also differ radically. Thus, it is serving as a convenient description of the effect of the subsystem's internal operation and, in and of itself, indicates nothing that may be relied upon regarding the way that effect is achieved.

Some of the message flow characteristics of a subsystem may be defined in terms of sequences of message transmissions through the subsystem's ports. This is analogous to defining a procedure in terms of a set of parameter/result pairs. More complex characteristics, called global characteristics, must be defined in terms of the correlations between transmissions occurring in different message transmission sequences. This is analogous to specifying, for a collection of stack manipulation procedures, that only certain combinations of procedure executions are legal -- for example, it should be specified that each pop operation must be preceded, at some point in time, by a corresponding push operation. In this section, the concern is with the more easily specified sequential message transmission characteristics. The specification of global message flow characteristics is treated in the next section.

Sequential message transmission characteristics are specified by a set of "programs", each of which is an abstract model of a sequential process. Although these control process models are in the form of programs for an abstract machine, their purpose is not to provide (or even suggest) a definition of the operational detail of the subsystem. Rather, they provide a pseudo-procedural definition of the effect of the subsystem's operation by means of an algorithmic definition of the message flow through the subsystem's ports.

Each control process model may be viewed as a sequential process which controls communication between a subsystem and other subsystems in its environment. This communication may generally be partitioned into several message transmission streams which are arbitrarily interleaved. A subsystem is therefore generally described by several control process models with each describing a single message transmission stream.

In a control process model, messages flow in and out through ports as a result of operations called receive and send. When a send operation is performed, a message is first composed using the values of the buffers associated with the port that is named in the send operation. Then, the message is placed in the link to which the port is attached and thereby made available for reception by some subsystem. (The link is an object which is not a part of any of the subsystems which utilize it. The process of attaching ports to links will be covered in a subsequent section.) Control passes to the operation following the send operation once the message is constructed and placed in the link. Since links have infinite storage capacity, this delay is relatively short and the send operation can therefore be considered to be a non-blocking operation.

When a receive operation is performed, flow of control is suspended until a message is retrieved, decomposed and distributed among the buffers associated with the port specified in the receive operation. Since it is possible that a request for a message may be lodged when none is currently available, the receive operation can cause relatively long delays.

Prior to a send operation, the values of the data objects which compose the message must be placed in the buffers. The computation which is actually carried out to accomplish this may be lengthy and complicated, but neither its time consumption nor its detail are of interest in defining sequential message transmission characteristics. The only aspects of interest are the result of the computation and the relationship of the result to any messages previously received. Thus, in a control process model, computational detail is suppressed by modelling it with a set-to operation which may be applied to a buffer data object and which results in the buffer assuming a value prescribed in the set-to operation. The dependency of the value upon previously received or computed values is modelled by conditioning the flow of control upon the values of the buffer data objects and hence upon previously received or "computed" messages.

To describe the sequential message transmission characteristics of [knowledge_source] subsystems, two sets of control processes are

needed. The first, specified[4] in figure 3, models the consumption of messages which arrive at the *await* port. This models the subsystem's operation with respect to signals sent to indicate a change of some data base entry of interest to the knowledge source.

```
▼[knowledge_source]: SUBSYSTEM CLASS▼

  listener:  ARRAY [1::#_values] OF CONTROL PROCESS;
     MODEL:  ITERATE
                RECEIVE await(MY_INDEX);
                END ITERATE;
             END MODEL;
     END CONTROL PROCESS;
```

Figure 3

The second set of control processes is specified in figure 4. These latter control processes model the operation of the servicers of the incoming signals, indicating that they present a request to the data base through one of the *make_request* ports and receive answers to the request through one of the *get_answer* ports. The entity *MY_INDEX* is a variable which has a value in the range declared as the bounds of the array of control processes and is used to make each control process model distinct with respect to the buffers and ports to which it refers.

```
▼[knowledge_source]: SUBSYSTEM CLASS▼
  requestor:  ARRAY [1::#_servers] OF CONTROL PROCESS;
     MODEL; ITERATE
               request(MY_INDEX) SET TO modify OR inspect;
               SEND make_request(MY_INDEX);
               RECEIVE get_answer(MY_INDEX);
               END ITERATE;
            END MODEL;
     END CONTROL PROCESS;
```

Figure 4

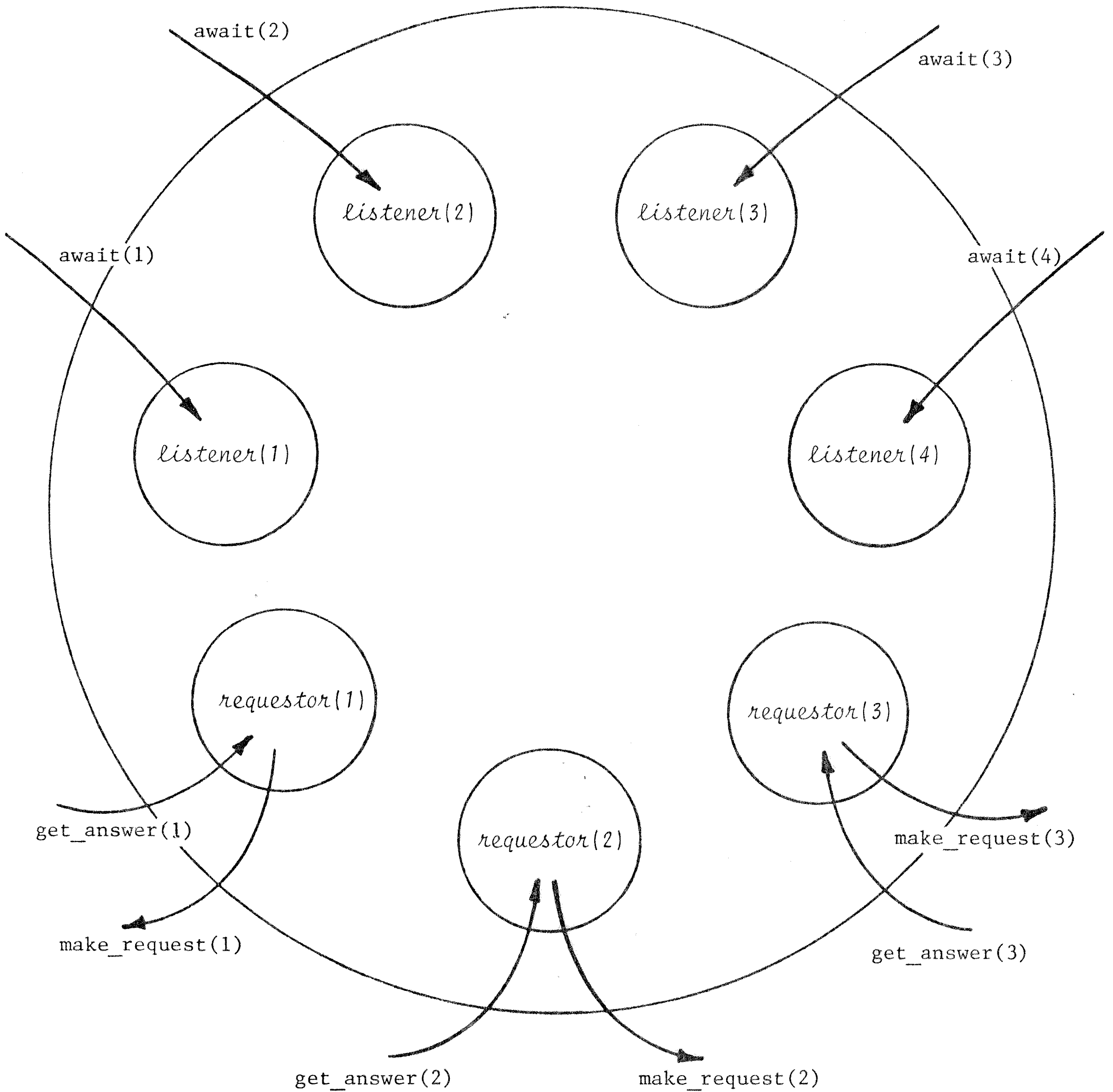
[4] The quoted prefix in the textual unit of figure 3 indicates that this textual unit gives further information about the class [knowledge_source].

The control process structure of a [knowledge_source] subsystem in which *#_servers* is 3 and *#_values* is 4 may be pictorially represented as in figure 5.

Each model is a control program over send and receive operations on ports and set-to operations upon buffers. When necessary for the accurate modelling of a subsystem's sequential message transmission behavior, operations defined for the class of objects of which a buffer is an instance can also be invoked upon that buffer. The control constructs used in control process models are Algol-like. There are constructs for definite iteration: an "ITERATE n TIMES" contruct and a "FOR ALL i IN set_of_values" construct. WHILE and UNTIL constructs are available for indefinite iteration. Since many subsystems are designed to never terminate, there is also an "ITERATE" construct for infinite iteration. Conditional control may be specified by the usual forms of the IF construct. There is also a generalized CASE construct which allows "labelling" of the cases with logical expressions.

All of the control constructs have a nondeterministic version. For example, the construct "ITERATE n OR MORE TIMES" indicates that the number of times, while known before iteration begins, can be any number greater than or equal to n. A nondeterministic variant of the WHILE construct (see figure 6) is obtained in a way in which many of the nondeterministic versions are obtained, by using the nondeterministic boolean expression PERHAPS. Nondeterminism may also be specified in the set-to operation (see Figure 4) by giving a logical expression which specifies the set of values which could possibly be assigned to the data object.

Control process models provide for the nondeterministic, pseudo-procedural modelling of a subsystem. Nondeterminism is used because it contributes to the clarity of the model, allowing succinct definition of the subsystem. A pseudo-procedural description scheme is also used to enhance the clarity of the description. The resulting procedure-oriented modelling scheme is quite natural for describing the effect of a subsystem's operation in terms of its sequential message transmission characteristics. (As will be seen in the next section, DDN



control processes are represented by circles containing the control process name in script

Figure 5

relies heavily upon non-procedural specifications for defining global message transmission characteristics.)

A multiprocessor system programming language (for example, [1] and [8]) typically provides a variety of operations for the synchronization of message transmission. In the DDN modelling language, however, we have provided only two relatively simple,but sufficient, synchronization operations, send and receive. While this makes it necessary for designers to develop DDN descriptions of other synchronization operations, we feel that this has the beneficial effect of forcing the designers to develop the details of the operations with consideration given to the ways in which they will be used to effect synchronization.

Global Behavior

The facilities provided by a subsystem cannot generally be used in a totally arbitrary order -- e.g., the facility which a file system provides for opening a file must be used prior to the facilities for reading and writing the file. Thus there are correlations between what happens in one sequential message transmission sequence and what may happen in another sequential message transmission sequence. (Note that these sequences may pertain to the subsystem's use of its environment as well as to the use, by others, of the facilities provided by the subsystem.) A specification of these correlations is a specification of the subsystem's _global behavior_ since it concerns more than just the activity of a single part of the subsystem for a relatively short period of time.

Global behavior must be specified non-procedurally since it can rarely be expressed succinctly as some controlled sequence of operations invoked by the subsystem. Necessarily, it concerns the operation of other subsystems which are in the subsystem's environment. It also pertains to operations that are distributed over time, stemming from different sequential algorithms. Procedural descriptions would therefore be both misleading and complicated.

Global behavior is specified in DDN in terms of _events_, activities that may be observed external to the subsystem. Usually an event is

the transmission of a message through a port, and thus may be identified with the execution of a send or receive operation in one of the control process models. In general, an event may be associated with the execution of any one of the instructions in a control process model. (Recall that since control process models are used as part of the external description of a subsystem, they are known to an external observer.)

Global behavior is specified by defining a set of sequences of events. Note that this is similar to what was accomplished procedurally via the control process models -- each model defines a set of sequences of message transmissions where each sequence corresponds to an execution of the model. Formal language theory provides a base for the non-procedural specification of behavior, since the set of events may be considered to be an alphabet and a behavior is then a language over this alphabet [19].

Two aspects of global behavior -- one mandatory for correct operation and the other stemming from a design decision -- need to be specified for [knowledge_source] subsystems. First, interactions with the data base occur subsequent to the reception of an activation signal indicating that an entry in the data base has been changed. Second, the interactions with the data base should be ordered such that no request is lodged while there is an outstanding request which has not been answered -- that is, interactions with the data base are to be ordered so that its facilities are utilized in a subroutine fashion.

To describe this behavior, we first define some events as indicated in figure 6. The statement labels define names for events which correspond to the execution of the labelled statement. Note the use of the NULL instruction to allow denotation of the event "a sequence of interactions with the data base is begun".

```
▼[knowledge_source]: SUBSYSTEM CLASS▼

  listener:  ARRAY [1::#_values] OF CONTROL PROCESS;
     MODEL;  ITERATE
          hear: RECEIVE await(MY_INDEX);
                END ITERATE;
             END MODEL;
     END CONTROL PROCESS;

▼[knowledge_source]: SUBSYSTEM CLASS▼

  requestor:  ARRAY [1::#_servers] OF CONTROL PROCESS
     MODEL; ITERATE
        start: NULL;
               ITERATE WHILE PERHAPS
                  request(MY_INDEX) SET TO modify OR inspect;
          ask:    SEND make_request(MY_INDEX);
          get:    RECEIVE get_answer(MY_INDEX);
                  END ITERATE;
               END MODEL;
     END CONTROL PROCESS;
```

Figure 6

More macroscopic events are needed to conveniently define the global behavior. These are defined by the textual unit shown in figure 7. The SEQUENCE operator denotes that the events which are its arguments are sequenced in the specified order. The REPEAT operator denotes that the *consult* event may be repeated zero or more times within each *hear_and_do_something* sequence. Thus, the *hear_and_do_something* event corresponds to a signal arriving from the data base being followed by the interactions with the data base.

```
▼[knowledge_source]:  SUBSYSTEM CLASS▼

   EVENT DEFINITIONS;
      consult:  SEQUENCE(ask, get),
      hear_and_do_something:
         SEQUENCE(hear, start, REPEAT(consult))
      END EVENT DEFINITIONS;
```

Figure 7

With these events defined, the global behavior may be specified as in figure 8. The first part of the specification indicates that, for any particular instance, up to *#_servers hear_and_do_something* events may be proceeding at any point in time, i.e., that servicing may go on in parallel up to the limit imposed by having only *#_servers* servicers. The second part indicates that a *consult* event for any particular instance must be exclusive of any other *consult* event for any instance of class [knowledge_source], i.e., that the interactions with the data base are ordered in time.

```
▼[knowledge_source]:  SUBSYSTEM CLASS▼

   DESIRED BEHAVIOR;
      POSSIBLY #_servers CONCURRENT
          (hear_and_do_something, hear_and_do_something),
      MUTUALLY EXCLUSIVE (consult, [knowledge-source]|consult)
      END DESIRED BEHAVIOR;
```

Figure 8

This example has used only a portion of the facilities available in DDN for behavior specification. Events may be associated with other aspects of a subsystem's operation, more complex relationships among events may be defined, relationships between events defined for different subsystems may be established, and events which are not associated with any computational part of a system may be described. A complete description of DDN facilities for non-procedural, event-based behavior specification is given in [38].

To review, a subsystem's global behavior is specified by describing the sequencing among observable events that relate to the operation of the subsystem. Global behavior may concern either the operation of the subsystem over a period of time or the interactions among different subsystems, or both. In any case, the sequencing is specified non-procedurally by defining a language over an alphabet of events.

The techniques for behavior specification used in DDN are a modification of event expressions as defined in [19]. Event ex-

pressions provide a means of defining a behavior by algebraically defining a language over a set of event names. Essentially all of the descriptional capabilities of event expressions are available in DDN, but they have been put in a form which makes them more easily used.

The DDN behavior specification constructs have also borrowed some concepts from path expressions [4]. In particular, the specification of behavior as a set of partial specifications is taken from path expressions. However, the specification of behavior in DDN is radically different in intent from path expression behavior description. Path expressions were originally developed as a programming language construct for imposing scheduling and synchronization constraints upon a community of concurrent processes. Behavior specification in a DDN description is not prescriptive in this sense -- the specification's intent is to report the desired behavior rather than to lead to the automatic generation of synchronization code. Thus, the behavior specification prescribes necessary scheduling and synchronization only in the sense that it indicates the subsystem designer's intentions as to what behavior is legal.

Internal Description of a Subsystem

A subsystem's external description provides a definition of how the subsystem may be used and how it will interact with other subsystems in its environment. It provides all the information about the subsystem that one is allowed to know when developing other subsystems which execute concurrently with the defined subsystem.

An internal description, however, gives details about the internal componentry of the subsystem and the manner in which the components interact to create the effects defined in the subsystem's external description. The internal description is what would typically be produced by a series of steps in a top-down design method.

A subsystem's internal description must define three aspects of the subsystem's internal operation. First, it must define the components which comprise the subsystem. These will be both subsystem

components and shared data objects. In this paper, we focus exclusively upon subsystem components[5]. The second aspect which must be defined is the communication pathways among the subcomponents. This involves the definition of links and the attachment of ports to links so that messages may flow among the subcomponents. Finally, the message flow into and out of the subsystem must be related to the message flow into and out of the collection of subcomponents. This is done by establishing controllers which distribute incoming messages among the subcomponents and collect messages from the subcomponents and pass them out through the subsystem's out ports.

Subcomponent Declaration

Subcomponents will themselves be instances of some other class of subsystems. Thus a definition of the subcomponents consists of a set of declarations, each specifying a name for a subcomponent and indicating the class of which it is an instance.

The declaration of one possible set of subcomponents for [knowledge_source] subsystems appears in figure 9. The *activator* subcomponent receives the signals sent by the data base and evaluates a logical condition to determine whether or not one of the *manipulator* objects is to be activated. The *manipulator* objects await activation by the *activator* and then interact with the data base to effect any necessary changes.

```
▼[knowledge_source]:  SUBSYSTEM CLASS▼

   SUBCOMPONENTS;
      activator OF [pre_condition (#_servers)],
      manipulator ARRAY [1::#_servers]
                          OF [data_base_modifier]
      END SUBCOMPONENTS;
```

Figure 9

---

[5] To avoid having to use the awkward term sub-subsystem, the term subcomponent is used for the remainder of this paper to mean a subsystem which is a component of the subsystem being described.

The subcomponentry of this example is relatively simple. More extensive facilities are available in DDN for describing subcomponents and these are discussed in [22].

Subcomponent Connections

Communication pathways must be established among the subcomponents so that they may interact. Since links are used to control message transmission, it is necessary to establish the requisite links and specify which ports are connected to which links.

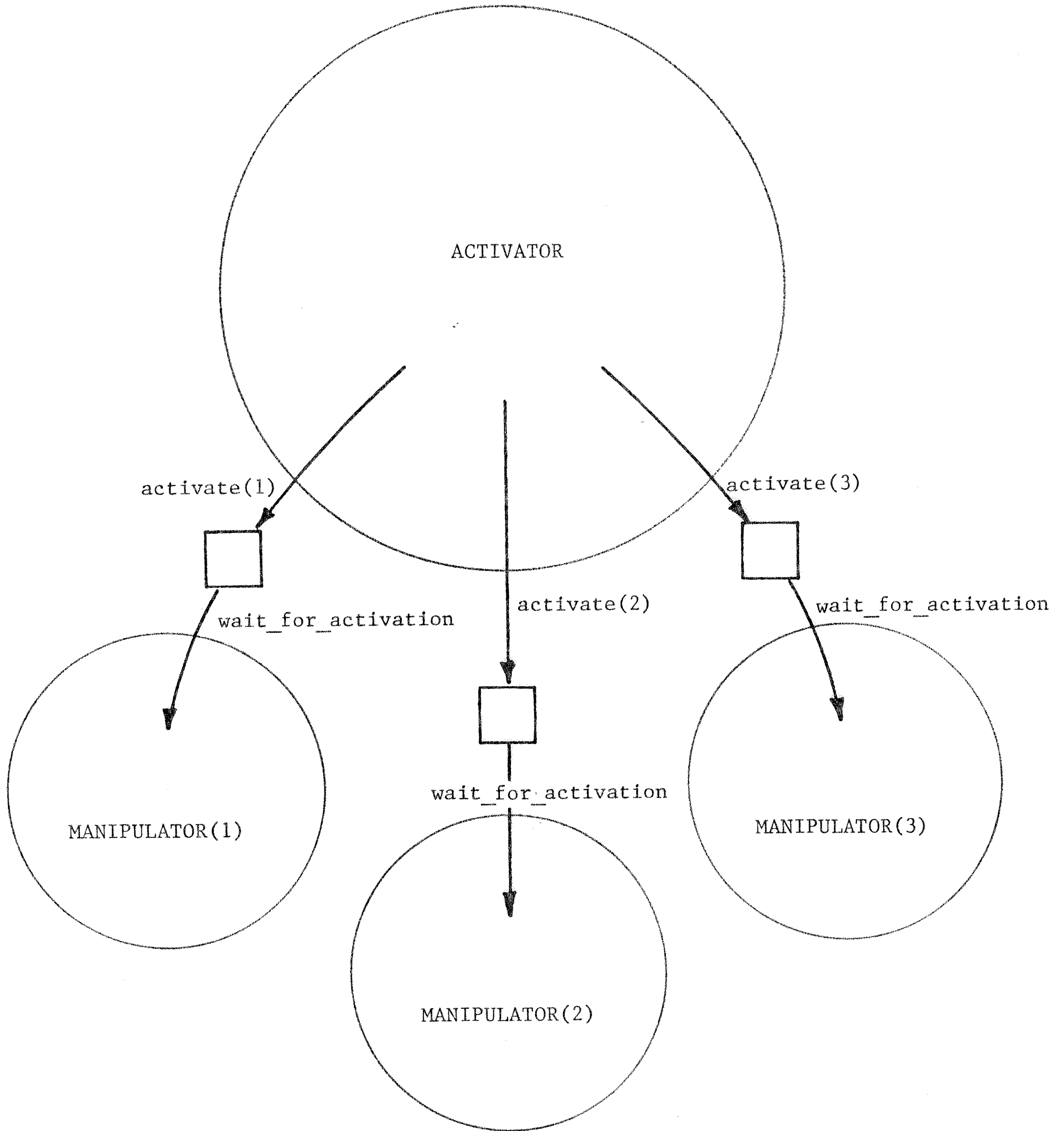
In DDN, this is accomplished by "plugging" ports together. For every set of ports that are plugged together, a link is established and the ports are attached to the link. In the example, communication pathways may be described by the textual unit appearing in figure 10. This establishes a set of links, each of which has an *activate* port from one of the *activator* objects and a *wait_for_activation* port from one of the *manipulator* objects attached to it. This leads, when there are three servers, to the communication pathway structure depicted in figure 11.

```
▼[knowledge_source]:  SUBSYSTEM CLASS▼

   CONNECTIONS;
      FOR ALL i IN [1::#_servers]
         PLUG (activator | activate (i), manipulator (i)|
         END FOR;                         wait_for_activation);
      END CONNECTIONS;
```

Figure 10

The control constructs provided for writing programs in a connection textual unit are essentially those provided for writing control process models except that non-deterministic constructs are not allowed. Also, the variables that are referenced in a plug program must be able to be evaluated prior to subsystem "execution" since the structure of a subsystem is static and may not change during the running of the system.[6]

[6]We have investigated extensions to the DDN modelling language which would allow the description of dynamic system structure [37], but have not incorporated these extensions in the current version of DREAM.



<u>subcomponents</u> are represented by circles containing the subcomponent name in capital letters

<u>links</u> are represented by boxes

Figure 11

The facilities provided by DDN for establishing communication pathway structures permit any (static) pathway structure to be defined. Note that it is only through the control of the pathway structure that message transmission may be directed from one subcomponent to another -- the send operation cannot specify which subcomponent is to ultimately receive a message. Thus the pathway structure itself defines all of the possible communication interactions. This is beneficial for the purpose of defining a system's structure. Operations, such as defined in [8], which permit constraint of the ultimate receiver could lead to more efficient system implementations but would obscure the definition of the interactions. Thus, DDN uses relatively primitive constructs which force the explicit representation of interactions.

Message Distribution

The messages which flow into the subsystem are received by the subcomponents and the messages flowing out of the subsystem are produced by the subcomponents. Thus there are some relationships among the ports of the subcomponents and the ports of the subsystem. In many cases, this is a simple identification of a port in a subsystem with a port in one of the subcomponents so that, for instance, when the subcomponent sends a message through its port, the message flows out through the subsystem's port. This simple identification occurs when there is no intermediate processing required as the message passes out of the subsystem. More complex cases arise when intermediate processing is required -- for example, an incoming message may need to be broken up into parts which are distributed among the subcomponents.

For the simple cases in which a subsystem port is identified with a port in a subcomponent, DDN allows ports to be overlaid as shown in figure 12. The OVERLAY statements establish an equivalence between one of the ports of the subsystem and one of the ports within a subcomponent. When the ports are overlaid, no overhead processing is incurred in order to get messages into and out of

the subcomponents since they have direct paths to the subsystem's ports and hence to the environment in which the subsystem is operating.

```
▼[knowledge_source]:  SUBSYSTEM CLASS ▼

   CONNECTIONS;
      FOR ALL i IN [1::#_servers]
         OVERLAY (manipulator (i) | request_out,
                                     make_request(i) );
         OVERLAY (manipulator (i) | answer_in, get_answer(i));
         END FOR;
      END CONNECTIONS;
```

Figure 12

More complex message distribution among the subcomponents is described in DDN by establishing special subcomponents which control the flow of messages between the subsystem's ports and the ports of the subcomponents. Since the purpose of control processes is to control the flow of messages through the subsystem's ports, these special subcomponents are described as bodies for the control processes. For our example, this is shown in figure 13. The body of a control process is a control program over 1) send and receive operations upon the subsystem's ports and the control process' ports and 2) operations upon the buffer and data object components of both the subsystem and the control process. Note that the operation assign has been invoked upon the *db_signal* buffer, passing as a parameter the value of the *signal* buffer -- the operation assign must be among the operations defined for the class [on_off_signal]. The control constructs that are available for stating control process bodies are the same as those for specifying control process models except that the non-deterministic constructs may not be used.

The textual units given in this section give rise to a set of communication paths among the subcomponents. For *#_servers* being 3 and *#_values* being 4, the paths would appear as pictured

```
▼[knowledge_source]:  SUBSYSTEM CLASS▼

  listener:  ARRAY [1::#_values] OF CONTROL PROCESS;
     pass_on: LOCAL OUT PORT;
        BUFFER SUBCOMPONENTS;
           db_signal OF [on_off_signal]
           END BUFFER SUBCOMPONENTS;
        END OUT PORT;
     MODEL;  ITERATE
          hear: RECEIVE await(MY_INDEX);
                END ITERATE;
              END MODEL;
     BODY;  ITERATE
               RECEIVE await(MY_INDEX);
               db_signal.assign(signal(MY_INDEX));
               SEND pass_on;
               END ITERATE;
            END BODY;
     END CONTROL PROCESS;

▼[knowledge_source]:  SUBSYSTEM CLASS▼

   CONNECTIONS;
      FOR ALL i IN [1::#_values]
         PLUG (listener(i)|pass_on, activator|get_signal);
         END FOR;
      END CONNECTIONS;
```

Figure 13

in figure 14. In comparing figure 6 and figure 13, notice that the *listener* control processes define, in this implementation, actual subcomponents whereas the *requester* control processes do not. This is because each *listener* control process models the operation of a single subcomponent and this subcomponent can most naturally be described as the body of the control process. Thus control process definitions serve two purposes. First, they may define, through a model textual unit, the message transfer effect of the operation of the subcomponents. Second, they may specify, through a body textual unit, the operation of an actual subcomponent.

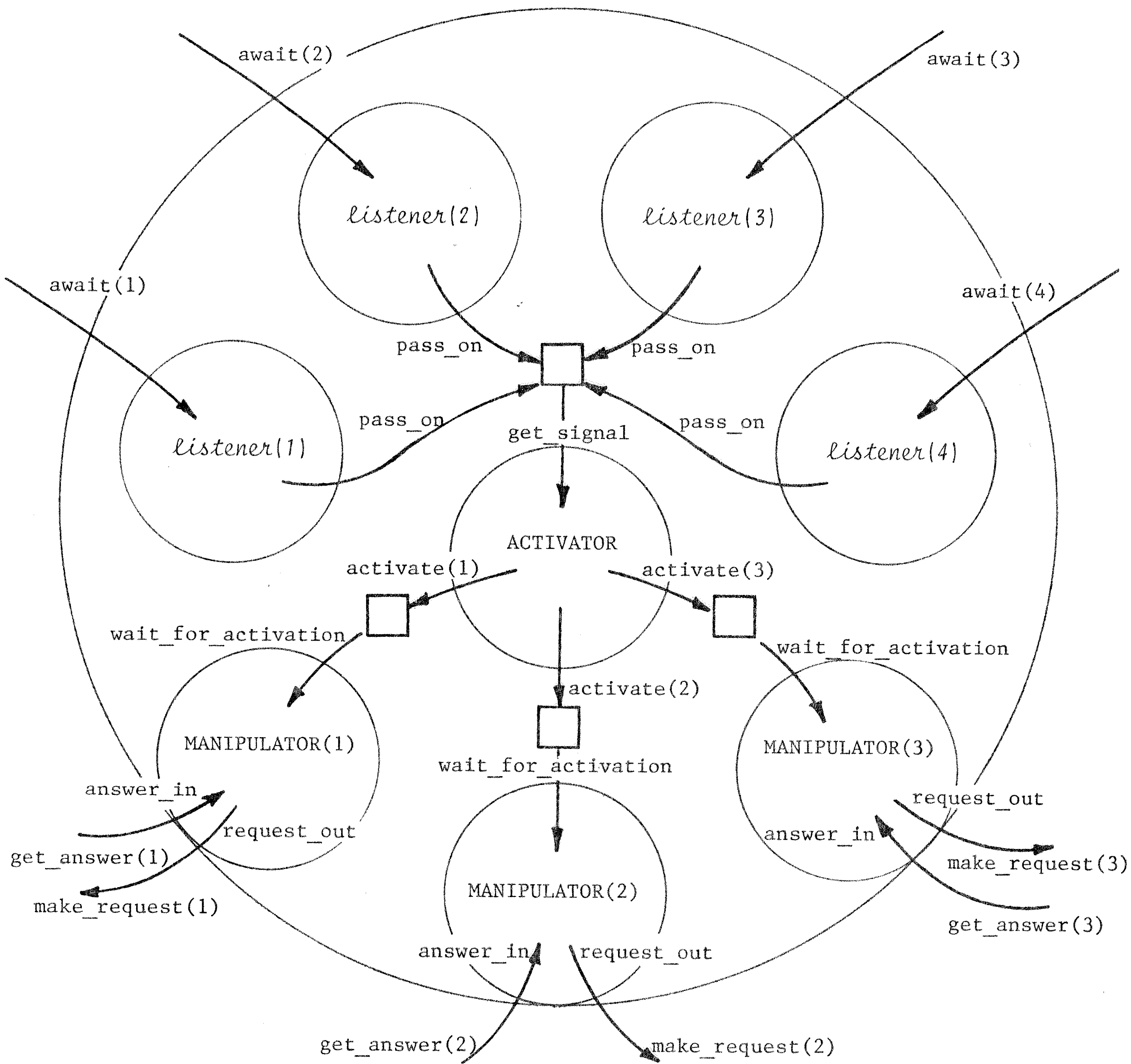


Figure 14

Analysis of Subsystems

The scheme that we have developed for describing collections of sequential processes affords many opportunities for rigorous analysis.[7] The simplest forms of analysis involve checks that can be made without interpretation of the text describing a subsystem's operation. Since such analysis would not account for the dynamic, run-time characteristics of the subsystem, it can be called static analysis. One example is the derivation of graphs such as in figure 14. Such paraphrasing of the subsystem's description in a form different from that used by the designer is frequently valuable in uncovering inconsistencies in the organization of a collection of subcomponents.

Another static analysis that could be performed is a check that referenced entities, such as ports or classes, have actually been defined. Such an analysis is a check of the completeness of the subsystem's design, rather than a correctness check, since the designer may not have yet designed the referenced entity. This indicates that much of the analysis performed during design is feedback analysis [20] in which it is left to the designer to conclude whether correctness or incorrectness of the design can be determined, based upon the derived information.

Of much more value to a designer is dynamic analysis in which information is derived about the run-time characteristics of the system under design. While dynamic analysis could be delayed until the system is completely designed, it is of more value when performed early in the design process, at which point it serves to predict the system's eventual run-time characteristics and offers the opportunity to uncover errors when they may more easily be corrected. The discussion of dynamic analysis will be with respect to this context -- in particular, implicit in the discussion is that dynamic analysis will be used as an integral part of a top-down design method.

---

[7] A general taxonomy of different types of analysis that would be useful is given in [35].

One form of dynamic analysis consists of checking the consistency among the various parts of a subsystem's external description. While the desired behavior construct of DDN is intended for a different purpose than the control process model construct, one way to achieve the redundancy necessary for consistency checking is to give a desired behavior specification of the behavior specified by the control process models. Such redundancy may also arise from desired behavior specifications appearing elsewhere in the DDN description of the system which have implications for the behavior that can legally be specified by the subsystem's control process models.

A check of the consistency between the desired behavior specification and the control process models would proceed as follows. Using an algorithm similar to that defined in [19], a description of the sequences of events defined by a control process model would first be derived. The desired behavior specification and the control process model would then be consistent if, for those events common to the two descriptions, all sequences defined by the control process model are also defined by the desired behavior specification[8].

Another dynamic analysis that could be performed is a check of the consistency of the interfaces among a collection of subsystems. A subsystem's external description may be used to indicate both how it responds to messages from other subsystems and what responses it expects from other subsystems. Therefore, once a configuration of subcomponents has been established, the consistency of the subcomponents' external descriptions can be checked. This analysis could be performed in the same manner as described above. The event sequences resulting from the operation of the control process models could be determined and then these sequences can be checked for inclusion in the sets of sequences specified by the desired behavior descriptions.

---

[8] The requirement that sequences defined by control process models be included in the set of sequences defined by desired behavior descriptions follows from the constraint nature of desired behavior descriptions.

A final form of dynamic analysis is the checking of the consistency between the external description of a subsystem's behavior and the behavior actually created by the subsystem's implementation. This would involve using the bodies of the control processes and the external descriptions of the subcomponents to determine the actual sequences of message flow through the ports. This would be consistent with the control process model definition of the subsystem's behavior as long as this set of message flow sequences was a subset of the set defined by the control process models.[9]

Alternatively, the event sequences caused by the operation of the internal components could be determined and compared to the event sequences specified as desired behavior. This can be done as long as event definitions exist which relate the events defined for the subsystem and the events defined for the subcomponents. With such definitions, the set of event sequences for the subcomponents could be mapped onto a set of event sequences for the subsystem. This set could then be compared, for containment, with the set of event sequences specified by the desired behavior description. (This type of analysis is used in the TOPD program development system [11].)

These dynamic analysis techniques all require some means of comparing two sets of sequences. In general, this comparison cannot be performed algorithmically [19]. Therefore, either algorithms for special situations would have to be used and cases which did not fall into these situations would have to go unanalyzed, or the comparison would have to be performed by the designer. The latter approach may not be unreasonable, as the designer may have the insight and intuition necessary to show equality or inclusion or find counter-examples.

---

[9] Alternatively, it could be required that the two sets be equal, reflecting a more strict condition that all, as well as only, the specified behavior is actually achieved.

The inability to algorithmically perform the entire analysis is not necessarily a serious limitation. First, in the absence of rigid interpretations imposed by specific analysis techniques, a designer is free to impose whatever relationships between the sets of sequences are deemed appropriate.[10] Thus designers may adjust the analysis to either their own style of design or the nature of the behavior being described. For example, a designer might use the desired behavior descriptions to indicate bounds upon the actual behavior such that part of the specified behavior must be achieved and part of it would be legal but is not required.

Second, simulation-based analysis may be used when formal (or informal) analytic techniques do not exist or are computationally unattractive. Simulation is possible since the procedural descriptions of a subsystem (i.e., the control process bodies) provide rigorous definitions of its operation. Several constructs have been included in DDN in order to increase the effectiveness of simulation. First, the buffer conditions can be checked, during simulation, whenever a message passes through the port associated with the buffer condition. Second, a history of the events occurring during the simulated execution could be checked against the set of desired sequences. In addition, the DDN notation could be extended to allow its use in the prediction of a subsystem's performance characteristics, following an approach defined in [29]. The description scheme would be augmented with constructs for associating probabilities with nondeterministic operations and time distributions with modelled operations. Once this was done, the simulator could accumulate statistics concerning characteristics such as running time, resource utilization and queue sizes.

---

[10] It is our belief that any fixed set of predetermined interpretations would omit some relationships significant to some designers, leaving those relationships to be checked by the designers themselves, without assistance.

Conclusions

We have presented a scheme for describing collections of asynchronous sequential processes. The scheme provides for the description of the communication pathways among the processes in the collection and hence the interactions among the processes. This allows the description of the internal, operational details of the collection. The scheme also provides for the description of the behavior of the collection as a whole in terms of the message flow into and out of the collection. This allows the description of the ways in which the collection may be used without a description of how this behavior is achieved.

We have used this scheme in formulating descriptions for a variety of software systems ([5],[6],[23],[31],[36],[37]), and have also used it in one exercise [34] which attempted to simulate a design effort. We feel that the scheme demarcates an important set of facilities for the description of software systems and that it provides a valuable basis for a variety of tools to aid designers of large-scale software systems.

We feel that effective design aid tools based upon this modelling scheme should allow the derivation of information helpful in determining whether or not the design being developed is correct. Some of this information may be a paraphrase of the designer's description in a form that allows the designer to more easily see inconsistencies which may exist. More helpful information is that which indicates the dynamic, run-time behavior of the system in a form which allows algorithmic or heuristic comparison with the behavior which the designer intends to have happen.

In and of itself, the description scheme introduces a rigor into the design of large-scale software systems which allows the designer to more carefully develop and expand the design. In this regard, the scheme is particularly useful in conjunction with a top-down design method. The ability to analyze the design and derive information that is not explicitly represented provides

the designer with a means of checking the appropriateness of the design at each design step. This incremental analysis of the design affords the opportunity to check design decisions as they are made rather than at the end of the design process.

Acknowledgements

The development of abstract process types was greatly facilitated by the contributions and constructive criticism of Jack Wileden, Alan Segal, Allan Stavely, John Sayler, Dirk Kabcenell and Victor Lesser.

REFERENCES

1. A.L. Ambler, et al. GYPSY: A Language for Specification and Implementation of Verifiable Program. ICSCA-CMP-2, Certifiable Minicomputer Project, Univ. of Texas, Austin, January 1977.

2. P. Brinch Hansen. The Programming Language Concurrent Pascal. IEEE Trans. on Software Engineering, 1, 2 (June 1975), 199-207.

3. P. Brinch Hansen. The Architecture of Concurrent Programs. Prentice-Hall, Englewood Cliffs, N. J., 1977.

4. R.A. Campbell and A.N. Habermann. The Specification of Process Synchronization by Path Expressions. Lecture Notes in Computer Science, 16, Springer Verlag, Heidelberg, 1974.

5. J.Cuny. A DREAM Model of the RC4000 Multiprogramming System. RSSM/48, Dept. of Computer and Comm. Sciences, Univ. of Mich., Ann Arbor, July 1977.

6. J.Cuny. The GM Terminal System. RSSM/63, Dept. of Computer and Comm. Sciences, Univ. of Mich., Ann Arbor, August 1977.

7. O. Dahl and K. Nygaard. SIMULA -- an ALGOL-Based Simulation Language. Comm. ACM, 9, 9 (September 1966), 671-678.

8. J.A. Feldman. A Programming Methodology for Distributed Computing (among other things). TR9, Dept. of Computer Science, Univ. of Rochester.

9. R. Fennel and V. Lesser. Parallelism in Artificial Intelligence Problem Solving: A Case Study of Hearsay II. IEEE Trans. on Computers, C-26, 2 (February 1977).

10. A.N. Habermann. Introduction to Operating System Design. SRA, Chicago, 1976.

11. P. Henderson. Finite State Modelling in Program Development. Proc. 1975 International Conf. on Reliable Software, Los Angeles, April 1975.

12. P. Henderson, et al. The TOPD System. Tech. Report 77, Computing Laboratory, University of Newcastle upon Tyne, England, September 1975.

13. C.A.R. Hoare. An Axiomatic Basis for Computer Programming. Comm. ACM, 12, 10 (October 1969), 576-580, 583.

14. C.A.R. Hoare. Notes on Data Structuring. In Dahl, Dijkstra and Hoare, Structured Programming, Academic Press, New York, 1973.

15. C.A.R. Hoare. Monitors: An Operating System Structuring Concept. Comm. ACM, 17, 10 (October 1974), 549-557.

16. J.J. Horning and B. Randell. Process Structuring. Computing Surveys, 5, 1 (March 1973), 5-30.

17. B.H. Liskov and S.N. Zilles. Specification Techniques for Data Abstractions. IEEE Trans. on Software Engineering, SE1, 1 (March 1975), 7-19.

18. D.L. Parnas. Information Distribution Aspects of Design Methodology. Proc. IFIP Congress 71, Ljubljana, August 1971, pp. TA-3-26-TA-3-30.

19. W.E. Riddle. An Approach to Software System Modelling, Behavior Specification and Analysis. RSSM/25, Dept. of Computer and Comm. Sciences, Univ. of Michigan, Ann Arbor, July 1976 (revised November 1977).

20. W.E. Riddle. A Formalism for the Comparison of Software Analysis Techniques. RSSM/29, Dept. of Computer and Comm. Sciences, Univ. of Michigan, Ann Arbor, July 1977.

21. W. Riddle, J. Sayler, A. Segal and J. Wileden. An Introduction to the DREAM Software Design System. Software Engineering Notes, 2, 4 (July 1977).

22. W. E. Riddle. Hierarchical Description of Software System Structure. RSSM/40, Dept. of Computer Science, Univ. of Colorado at Boulder, November 1977.

23. W.Riddle. DREAM Design Notation Example: The T.H.E. Operating System. RSSM/50, Dept. of Computer Science, Univ.of Colorado, Boulder, April 1978.

24. W.Riddle, J. Sayler, A. Segal, A. Stavely and J.Wileden. A Description Scheme to Aid the Design of Collections of Concurrent Processes. Proc. National Computer Conf., Anaheim, June 1978.

25. W.Riddle, J.Wileden, J.Sayler, A.Segal and A.Stavely. Behavior Modelling During Software Design. IEEE Trans. on Software Engineering, SE-4, 4 (July 1978).

26. W.Riddle, J.Sayler, A.Segal, A.Stavely and J. Wileden. DREAM: A Software Design Tool. Proc. 3rd Jerusalem Conf. on Information Technology, Jerusalem, August 1978.

27. W. Riddle. DDN User's Guide. RSSM/37, Dept. of Computer Science, Univ. of Colorado at Boulder, in preparation.

28. W. Riddle. Abstract Monitor Types. RSSM/41, Dept. of Computer Science, Univ. of Colorado at Boulder, in preparation.

29. J. Sanguinetti. Performance Prediction in an Operating System Design Methodology. RSSM/32 (Thesis), Dept. of Computer and Comm. Sciences, Univ. of Michigan, Ann Arbor, May 1977.

30. J. Sayler. Philosophy of the DREAM System. RSSM/39, Dept. of Computer and Comm. Sciences, Univ. of Michigan, January 1977.

31. A.Segal. DREAM Design Notation Example: A Multiprocessor Supervisor. RSSM/53, Dept. of Computer and Comm. Sciences, Univ. of Michigan, Ann Arbor, August 1977.

32. A. Segal. Design Description Management. RSSM/45, Dept. of Computer and Comm. Sciences, Univ. of Michigan,in preparation.

33. H.A. Simon. The Architecture of Complexity. Proc. Am. Phil. Soc., 106, (December 1962), 467-482. Also in Simon, Sciences of the Artificial, MIT Press, Cambridge, 1969.

34. A.M.Stavely. DREAM Design Notation Example: An Aircraft Engine Monitoring System. RSSM/49, Dept. of Computer and Comm. Sciences, Univ. of Michigan, Ann Arbor, July 1977.

35. A.M. Stavely. Feedback Aids in Software Design Aid Systems. RSSM/60, Dept. of Computer and Comm. Sciences, Univ. of Michigan, Ann Arbor, November 1977.

36. J.Wileden. DREAM Design Notation Example: Scheduler for a Multiprocessor System. RSSM/51, Dept. of Computer and Comm. Sciences, Univ. of Michigan, Ann Arbor, October 1977.

37. J.Wileden. Modelling Parallel Systems with Dynamic Structure. RSSM/71 (Thesis), Dept. of Computer and Comm. Sciences, Univ. of Michigan, Ann Arbor, January 1978.

38. J. Wileden. Behavior Specification in a Software Design Aid System. RSSM/43, Dept. of Computer and Information Science, Univ. of Massachusetts, August 1978.

39. N. Wirth. Modula: a Language for Modular Multiprogramming. Software - Practice and Experience, 7, (1977), 3-35.

40. W.A. Wulf, R.A. London and M. Shaw. Abstraction and Verification in ALPHARD: Introduction to Language and Methodology. Report 76-46, Information Sciences Inst., Univ. of Southern California, 1976.

ABSTRACT PROCESS TYPES

William E. Riddle
Department of Computer Science
University of Colorado at Boulder
Boulder, Colorado 80309

CU-CS-121-77 (Revised July 78) December, 1977

This work was supported by a grant from Sycor, Inc.

Abstract

Abstract process types are introduced as a means of giving high-level descriptions of the components in large-scale software systems. Abstract process types provide facilities for the description of a system's processing components that are analogous to the facilities which abstract data types provide for describing a system's data storage components. In particular, abstract process types provide a basis for approaches to verification that parallel the approaches developed in conjunction with abstract data types.

Key Words and Phrases

abstract process types, abstract data types, hierarchical system description, non-procedural specification, verification, DREAM