BEHAVIOR MODELLING DURING SOFTWARE DESIGN

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

Jack C. Wileden
Computer and Communication Sciences Dept.
University of Michigan at Ann Arbor
Ann Arbor, Michigan 48104

CS-CU-119-77 November, 1977

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

Introduction

At an intermediate point in the design of a software system, some of the system's components will be completely designed whereas other components will be only partially designed and the design of some components will not have been begun. At this point, the designer could proceed with the next design step, further detailing the design of one of the incompletely designed components. More effective, however, would be for the designer to first gain, through formal or informal arguments, confidence that the design is appropriate and correct. But this is generally precluded because of the absence of a rigorous specification of the incompletely designed components. In this paper, we develop a description scheme that allows incompletely designed software system components to be rigorously specified so that designers may incrementally gain confidence in a design as it is being developed.

Rigorous specification of an incompletely designed component requires the ability to model the component's behavior. That is, it requires the ability to describe what the component will do -- its behavior -- in terms of an abstraction of the component's implementation which focuses upon effect rather than cause. One means of abstraction is simple elimination of detail. Although not always possible, this can be done when only certain characteristics of the component need to be preserved in the model. For example, the model may describe the component's processing with respect to only a portion of its input. Abstraction may also be accomplished by using a description scheme in which the component's interesting characteristics may be succinctly specified. The vocabulary of the new description scheme should be chosen to allow the direct statement of characteristics that are only implicitly specified in a detailed implementation description. A function procedure, for example, could be succinctly modelled by its mathematical definition.

The modelling scheme presented in this paper allows both these approaches to abstraction to be used during the design of large-scale software systems. In the scheme, software system components are described as collections of concurrent[1] processes or as data objects shared among these collections, and the description of the interactions among the components is emphasized. As a means of rigorously specifying undesigned system components, the scheme is therefore of primary use during the early phases of large-scale software system design, when the system's modules are being delineated and their interactions designed.

---

[1]We use the term concurrent to connote parallelism which may or may not be actually realized when the system is executed.

The scheme presented here was developed for use in an interactive design tool called the Design Realization, Evaluation And Modelling (DREAM) system [1]. DREAM is based upon a design language, the DREAM Design Notation (DDN), and the scheme discussed here is a major part of that language. DDN allows a design to be developed incrementally, in fragments called textual units, and DREAM contains a data base management facility which allows a design description to be augmented or modified on a textual unit basis. DREAM has also been developed in order to provide a variety of analysis aids to designers of large-scale software systems; some of these aids are discussed at the end of this paper.

In the next two sections, we outline the DDN approach to system description and discuss several desirable characteristics of description schemes intended for the modelling of software systems. Next, we introduce the DDN modelling constructs, first those for modelling shared data objects and then those for the modelling of collections of concurrent processes. We then discuss some additional constructs which permit the non-procedural specification of behavior. In the concluding sections, we indicate that the modelling scheme provides a basis for several approaches to design analysis and lends beneficial support to the designers of large-scale software design.

DDN Descriptions

In DDN descriptions, a software system is decomposed into subcomponents of two types. Subsystems are those subcomponents which control and guide the performance of the system's processing, which operate (conceptually at least) concurrently and asynchronously with respect to other subcomponents, and which are individually capable of performing several activities at once.[1] Monitors are those subcomponents which also operate concurrently and asynchronously with respect to other components but which serve primarily as repositories of shared information and are individually capable of performing only a single activity at any point in time.[2] This decomposition may be hierarchical[3], since

---

[1]Those system components which execute concurrently and manipulate shared data objects are usually considered to be sequential processes, as defined in [2]. A subsystem is a more general object, being essentially a collection of sequential processes.

[2]The monitors of DDN are essentially those defined by Hoare [3]. To the usual definition of monitors, we have added constructs for behavior specification, patterned after constructs developed for the TOPD system [4].

[3]We assume, for the purposes of this paper, that this hierarchical decomposition is tree-like. Non-tree-like decomposition is discussed in [5].

a subsystem may be decomposed into (sub-)subsystems and (sub-)monitors and a monitor may be decomposed into (sub-)monitors.

Hierarchical decomposition may proceed to any one of a number of levels. For instance, the primitive (i.e., undecomposed) subcomponents could correspond to the processing units and data objects provided by the system's execution environment. Or they could correspond to the undesigned components existing at some point during the system's design. Whatever the extent of the decomposition, the system's overall operation is the result of the activity of the primitive subsystems as coordinated through their shared usage of the primitive monitors. A particularly important means of coordination, since it corresponds to direct subsystem interaction, is the transfer of messages. This mode of interaction is therefore distinguished in DDN, making DDN a message transfer modelling scheme.

We illustrate this approach to software description by applying it to the HEARSAY speech recognition system [6] developed at Carnegie-Mellon University. In HEARSAY, all information about the utterance being processed and all hypotheses as to its linguistic structure are stored in a central data base called a blackboard. The information in the blackboard is augmented and modified by knowledge sources, each of which enforces a set of speech recognition rules. The obvious subsystems in an initial decomposition of HEARSAY would therefore be the blackboard and the knowledge sources. The message transfer interactions would consist of the request messages sent by the knowledge sources and the responses returned by the blackboard. In addition, a message would be sent by the blackboard to activate a knowledge source when an entry of interest to the knowledge source changes value.

A possible next decomposition step might be to demarcate those subsystems (one for each knowledge source) within the blackboard which exchange messages with the knowledge sources and manage the modifications each knowledge source makes to the region of the blackboard of interest to the knowledge source. The blackboard regions themselves would be monitors since they represent information repositories and since the synchronization primitives provided by monitors allow the succinct description of the coordination among the possibly conflicting reading and writing of the areas falling within more than one region. Also delineated at this level of decomposition would be those subsystems within the blackboard which notify a knowledge source when an entry of interest to it has changed value.

Attributes of Modelling Schemes

The description of systems viewed as proposed in the previous section requires a hierarchical modelling scheme. Such a scheme must be able to describe a subcomponent's external attributes, those characteristics which pertain to its interactions with other subcomponents at the same level of decomposition. It must also be able to describe a subcomponent's internal attributes, those aspects which pertain to the manner in which a subcomponent is composed of other subcomponents and the ways in which these subcomponents are to interact so as to create the intended operation of the subcomponent which they comprise.

With respect to describing the external attributes of subcomponents, and focusing upon the description needs of software system designers, several desirable characteristics of modelling schemes may be delineated. First, facilities must be provided for both outward-directed descriptions, which describe those aspects of a subcomponent's behavior which are relevant to its interactions with other subcomponents, and inward-directed descriptions, which describe those aspects of behavior which are significant in developing the subcomponent's implementation. Second, the scheme should support projection, providing the ability to focus upon and highlight interesting behavior (for a variety of definitions of "interesting") and suppress irrelevant details. Third, a means must be provided for non-procedural specification, allowing the definition of behavior without the specification of an algorithm for achieving the behavior. Fourth, the scheme should admit descriptions that are non-prescriptive in that a wide variety of strategies, mechanisms and algorithms can be used to implement the described behavior. Fifth, redundant specifications must be possible, allowing the same behavior to be specified from different points of view or with respect to different sets of concerns. Sixth, it must be possible to give a description that is orthogonal to the subcomponent's internal description in the sense that it may establish associations among activities which occur within physically different parts of the subcomponent. Seventh, the scheme must admit modular descriptions so that different properties may be independently specified and inter-relationships among these properties may be specified separately from the specification of the properties themselves. Finally, the scheme should lead to analysis-oriented descriptions which can serve as the basis for formal or informal arguments through which the designer may gain increased confidence in the accuracy of the design.

The DDN constructs introduced in the next three sections provide software system designers with a rigorous, formally-defined technique for the modular, non-prescriptive specification of the external attributes of both monitor and subsystem subcomponents. The constructs allow analysis-oriented descriptions which are redundant, orthogonal, projective, non-procedural (and procedural), and both inward-directed and outward-directed. The constructs are illustrated by a series of examples which, taken together, provide an abstract description of the blackboard subsystem within the HEARSAY system to the level of decomposition developed above.[1] In the examples, we focus upon describing the blackboard's organization and behavior and upon specifying the policies concerning the concurrent operation of the region managers. We do not attempt to specify the mechanisms, or even the strategies, by which conflicts are prevented -- this is deliberately done so as to highlight DDN's use as a modelling rather than a programming language.

The Description of Monitors

To avoid having to describe each subcomponent, either monitor or subsystem, explicitly, DDN descriptions are of classes of subcomponents. The class concept was introduced in SIMULA [7] and has subsequently been widely used in computer-oriented description schemes -- it underlies schemes such as abstract data types [8], TOPD classes ([4],[9]), Parnas modules [10], Alphard forms [11], CLU clusters [12] and Pascal types [13]. In DDN, class definitions define the external and internal attributes of each instance of the class -- here, we focus primarily upon those parts of class definitions which are for the specification of external attributes.

For a monitor class, external attributes pertain to the procedures which may be invoked upon instances of the class. For each procedure, its name and parameters must be specified along with a definition of its behavior. A procedure's behavior may be specified non-procedurally via a formal definition of the function it computes. In DDN, this is accomplished by defining the changes the procedure makes in the states of the objects it operates upon.

One aspect of the specification of the external attributes of a class of monitor objects is therefore the definition of a set of observable states.

---

[1]The description reflects our understanding of the HEARSAY system and is oriented toward providing examples of the DDN constructs. We feel that the description is reasonably accurate, but do not claim that it fully corresponds to the actual HEARSAY system.

The concept of observable state is more general than the concept of "value", since a state may encode an instance's past history as well as reflect the instance's current "value". For example, a region within the blackboard could be in the state *wrting_shared* indicating that a write operation upon a portion of the region shared with another region is in progress.

Simple examples of DDN monitor class definitions are given in figure 1. These monitor objects are "variables" which are needed in later class definitions -- objects of class[1] [region_id] are integers which fall within some range and which are used to identify the regions of the blackboard; objects of class [entry_id] are values which are used to address the entries in the blackboard; class [datum] objects are the values stored as entries in the blackboard.

---

[1]In DDN, identifiers used to name classes are always enclosed in square brackets.

```
[region_id]: MONITOR CLASS;
  QUALIFIERS;  range_limit  END QUALIFIERS;
  STATE SUBSETS;  1, in_range, range_limit  END STATE SUBSETS;
  STATE ORDERING;  1 <= in_range <= range_limit
    END STATE ORDERING;
  determine: PROCEDURE;
    PARAMETERS;  id VALUE OF [entry_id]  END PARAMETERS;
    TRANSITIONS;  id=defined --> in_range
      END TRANSITIONS;
    END PROCEDURE;
  END MONITOR CLASS;
[entry_id]:  MONITOR CLASS;
  STATE SUBSETS;  defined, undefined  END STATE SUBSETS;
  END MONITOR CLASS;
[datum]:  MONITOR CLASS;
  STATE SUBSETS;  defined, error-flag   END STATE SUBSETS;
  END MONITOR CLASS;
```

Figure 1

The definition of [region_id] in figure 1 indicates that "parameterized" class definitions are allowed in DDN. The qualifiers textual unit specifies that the limit of the range may vary among instances and may be specified when each instance is declared. Qualifiers, which are discussed more fully in [5], may be used to specify a value for any single lexicographic unit within a class definition.

In the class definitions of figure 1, states are not specified explicitly; rather, (not necessarily disjoint) subsets of the state space are defined. States, themselves, allow the potentially infinite domain of "values" for a monitor object to be modelled by grouping them into a finite number of disjoint sets. State subsets extend this grouping capability, allowing the description to be focused upon interesting characteristics of monitor activity. Note that an ordering relationship may be established among states so that instances may be compared through the use of the usual set of relational operators.

An instance of a monitor class may be inspected, at any time, to determine its state (or state subset) -- it is in this sense that states are observable. This is quite valuable for the succinct, abstract specification of behavior as will be illustrated in later examples. When used to specify the algorithmic detail of a component's internal operation, however, state inspection may need to be coordinated with other operations upon the monitor. When such coordination is necessary, it may easily be effected by having the state inspection performed by a procedure defined for the monitor class.

The procedure textual unit of figure 1, and the parameter textual unit nested within it, specify that the *determine* operation is available for manipulating instances of the class [region_id] and that an instance of the class [entry_id] must be passed as a value parameter. The intended purpose of this procedure is to determine in which region an identified entry lies. This is reflected by the transitions textual unit which specifies that a pre-condition for the invocation of the procedure is that the parameter be in a state in its *defined* state subset and that the effect of the procedure is to leave the state of the parameter unchanged and to leave the object upon which the procedure is invoked in a state in its *in_range* state subset.

The regions of the blackboard are described in figure 2. The state variables textual unit indicates that a coordinatization of the state space may be used to specify the states. For example, a [region] class object could be in the state <<selected=yes, doing=read>> which is intended to denote that

the region has the right to access information which it shares with another region and is in the process of reading that shared information. With the specification of state variables, state subsets may be defined as indicated

```
[region]:  MONITOR CLASS;
  QUALIFIERS;  my_id  END QUALIFIERS;
  STATE VARIABLES;  selected: VALUES(yes, no),
                    doing:  VALUES(read, write, neither)
    END STATE VARIABLES;
  STATE SUBSETS;
    rding_private:   <<--, doing=read>>,
    wrting_private:  <<--, doing=write>>,
    rding_shared:    <<selected=yes, doing=read>>,
    wrting_shared:   <<selected=yes, doing=write>>,
    stalled:         <<selected=no, doing=read OR doing=write>>,
    unoccupied:      <<--, doing=neither>>
    END STATE SUBSETS;
  read:  PROCEDURE;
    PARAMETERS;  value_read RESULT OF [datum]  END PARAMETERS;
    TRANSITIONS;
      prt_read:  unoccupied
                   ||rding_private||
                     unoccupied AND (value_read=defined OR value_read=error_flag
      unoccupied
        ||OR(SEQUENCE(stalled,rding_shared),rding_shared)||
          unoccupied AND (value_read=defined OR value_read=error_flag)
      END TRANSITIONS;
    END PROCEDURE;
  write:  PROCEDURE;
    PARAMETERS; value_to_write VALUE OF [datum]
      END PARAMETERS;
    TRANSITIONS;
      prt_wrt:  value_to_write=defined AND unoccupied
                  ||wrting_private||
                    unoccupied,
      value_to_write=defined AND unoccupied
        ||OR(SEQUENCE(stalled,wrting_shared),writing_shared)||
          unoccupied
      END TRANSITIONS;
    END PROCEDURE;
  END MONITOR CLASS;
```

Figure 2

in the state subsets textual unit appearing in figure 2. The notation "--" specifies that the corresponding state variable may have any one of its possible values.

The example of figure 2 also illustrates the general form for the specification of transitions. The "-->" notation used previously specifies that the state change occurs without the objects that are being manipulated being in any observable intermediate states. The transitions of figure 2, however, specify that intermediate states are observable. The second transition for the *read* procedure, for example, indicates that during the procedure's execution, the class [region] instance being manipulated passes through the sequence of states *stalled, rding_shared* or the sequence *rding_shared*.[1]

The Description of Subsystems

Since direct interaction between subsystems takes place via message transfer, a subsystem's external attributes may be described by specifying the message flow into and out of the subsystem. Part of this specification is a definition of the communication paths which cross the subsystem's boundary and the demarcation of any restrictions as to what messages may legally flow across the boundary -- this is a specification of the subsystem's interface. A second part of the specification describes correlations of message flow into and out of the subsystem -- this is a description of the subsystem's behavior over time.

In DDN, communication channels are represented by specialized monitors, called links, which can store and forward messages and to which subsystems may be attached. The point of attachment of a subsystem to a link is called a port. Each port is therefore a "hole" through which messages may flow, having a directional attribute, either in or out. The DDN constructs for port definition are illustrated in figure 3, a subsystem class definition describing those subcomponents of a blackboard which notify the knowledge sources of changes to entries in the stored information.

The messages which flow through a port are specified by a set of buffers associated with the port definition. In the example, each *note(i)* port has a single buffer, *notice(i)* associated with it. If the port is an out-port, then a message sent out through the port is the (ordered) composition of the

[1]The constructs for describing sequences of states are discussed in [14].

contents of the buffers at the time that the send operation causing the message flow is performed. For an in-port, when a message passes in through the port (as a consequence of a receive[1] operation) the message is decomposed and used to determine new contents for the buffers associated with the port.

The buffer conditions textual unit of figure 3 indicates that the state of the *notice* portion of the message (in this case, the notice portion is the entire message) will be in its *change* state subset. The definition of class [datum_change], given in figure 4, indicates that this means the *notice* portion will never be in a state that is in its *out_of_range* state subset.

[1]Receive is a potentially-blocking operation whereas send is a non-blocking operation. The semantics of these operations, and the operation of links, is explained more fully in [15] and [16].

```
[noticer]:  SUBSYSTEM CLASS;
  QUALIFIERS;  #_under_surveillance  END QUALIFIERS;
  note:  ARRAY[1::#_under_surveillance]  OF OUT PORT;
    BUFFER SUBCOMPONENTS;   notice OF [datum_change]
     END BUFFER SUBCOMPONENTS;
    BUFFER CONDITIONS;  notice=change  END BUFFER CONDITIONS:
    END OUT PORT;
  observer:  ARRAY[1::#_under_surveillance] OF CONTROL PROCESS;
    MODEL;  ITERATE
     see_it: SET  notice(MY_INDEX) TO change;
             SEND note (MY_INDEX);
             END ITERATE;
     END MODEL;
    END CONTROL PROCESS;
  END SUBSYSTEM CLASS;
```

Figure 3

```
[datum_change]:  MONITOR CLASS;
  STATE SUBSETS;  change, out_of_range  END STATE SUBSETS;
  END MONITOR CLASS;
```

Figure 4

Buffer conditions are both outward-directed and inward-directed specifications. With respect to interactions with other subsystems, they specify which messages will be sent out (for out-ports) or are expected to be received (for in-ports). With respect to the eventual design of the subsystem, they inform the designer of the limitations concerning which messages may be sent out (for out-ports) and which incoming messages should be expected (for in-ports).

The control process portion of the example in figure 3 provides a procedural specification of the sequential portions of the subsystem's behavior. In general, control process models specify sequences of message flow across the subsystem's boundary and therefore define correlations among messages flowing at different times through one or more of the ports. In the example, the control process model indicates that there is a constant stream of *change* messages flowing out through each port. This is the appropriate behavior, at this level of decomposition, for that part of the HEARSAY data base which notifies the knowledge sources of changes in the entries in the data base.

Control processes serve to abstractly model the actual operation of the subsystem. In the example of figure 3, abstraction is partially achieved through the elimination of detail afforded by the set-to operation. This statement models the possibly complex and lengthy processing needed to cause the *notice(MY_INDEX)* buffer[1] to be in a state within its *change* state subset. Abstraction is also achieved by allowing operations to be performed only upon the ports and their buffers -- inside a control process model, reference may not be made to the subsystem's internal componentry and hence the model may not specify anything about the algorithmic detail of the subsystem's internal operation. In the example, this results in a desirable hiding of information as to what internal activity actually causes messages to be sent out. Though always non-detailed with respect to the subsystem's internal operation, models may be very elaborate (when the modelled behavior is itself elaborate); this and other aspects of control processes are described in[16].

The DDN constructs for describing a subsystem's external attributes allow the focusing of the description upon the interactions in which the subsystem is able to participate. This is accomplished by requiring the explicit

---

[1]Within arrays of control processes, *MY_INDEX* is a variable which has a different value for each of the models and may be used, as here, to make each model specific to a different set of ports and buffers.

specification of subsystem interaction in terms of message transfer. Further, because of the ability to construct abstract models of the subsystem's behavior, the description may be rigorous, projective, outward-directed and modular. Finally, the description is also inward-directed, since it specifies the behavior which the implementation must achieve; but it is orthogonal, since the organization of the subsystem's internal componentry need not bear any direct resemblance to the organization of the control processes.

Event Definition

Procedure transitions and control process models are the basic DDN constructs for abstractly describing the simple, sequential behavior of software system components. More complex behavior, which is not sequential in nature or which pertains to inter-relationships between the behavior of several components, may be described in DDN by the definition of events (significant occurrences during system operation) and the non-procedural specification of sequences of events. Event definition is discussed in this section and event sequence specification is covered in the next section.

We distinguish two broad types of events, endogenous and exogenous, in DDN. Endogenous events are those occurrences which arise from some activity within the currently DDN-described portions of the software system. Exogenous events are those occurrences which are relevant to or impinge upon the system's behavior but arise from some activity outside the currently described portions of the software system. Whether an event is endogeous or exogenous is therefore relative to the extent of the system's description and may change over time -- for example, an exogenous event may become an endogenous event as elaboration of the design leads to the description of the component whose activity gives rise to the event. Some events, however, are inherently exogenous since they pertain to the system's operation but do not stem from the software portion of the system being designed -- examples of such events are activities within some other software system which interacts with the system being designed or operations performed by some physical device controlled by the software system.

The most elementary method for defining endogenous events is to simply attach a label, called an event identifier, to some portion of the DDN description of a procedure or a control process. For example, the [region] monitor class description of figure 2 defines the events *read*, *write*, *pvt_rd* and

pvt_wrt, with the first two corresponding to executions of the respective procedures and the latter two corresponding to occurrences of (some of the) transitions defined for those procedures. Thus, an execution of the *read* procedure of some instance of class [region] would be an instance of a *read* event, while an occurrence of the first transition defined for the *read* procedure would be an instance of the event *pvt_rd*. Similarly, an occurrence of the buffer modification described by the set-to statement within the *observer* control process model in the [noticer] subsystem (figure 3) would correspond to a *see_it* event. Note that events defined within a DREAM design description may occur simultaneously and that one event may occur as part of another, as in the case of *pvt_read* and *read*.

Unlike endogenous events, definitions of which may be embedded within the monitor or subsystem classes whose activities give rise to them, exogenous events are not naturally associated with any monitor or subsystem class definition. Therefore, a third class type, the event class, is available in DDN for the definition of exogenous events. This is illustrated in figure 5 in which we describe part of the activity of knowledge sources as it relates to the operation of the blackboard.

In addition to its use in the definition of exogenous events as illustrated in figure 5, the event definition textual unit may be used within a monitor, subsystem or event class definition for the specification of more complex events. These events may be specified in terms of other events, sequences of states of a monitor class, or sequences of statements in a control process model. In each case the specification is labelled with an event identifier naming the specified event. Details and examples of DDN

```
[ks]:  EVENT CLASS;
   EVENT DEFINITION;
     request_write:  DESCRIPTION;
       An event which occurs when a knowledge source requests a
       write operation upon a region within the blackboard.
       END DESCRIPTION;
     END EVENT DEFINITION;
   END EVENT CLASS;
```

Figure 5

---

event definition facilities may be found in [17]; here we provide only a simple illustration (figure 6[1]) in which the events *shared_rd* and *shared_wrt* are defined as state sequences (of length one). These definitions indicate that the *shared_rd* and *shared_wrt* events correspond to a [region] monitor instance's being in the state subsets *rding_shared* and *wrting_shared*, respectively.

The event definition mechanisms of DDN provide a flexible and powerful tool for defining arbitrarily complicated events in a software system design. This event definition capability is the foundation of the DREAM behavior specification technique. Moreover, its flexibility and generality are largely responsible for the technique's projection properties, since various interesting aspects of system behavior may be highlighted by appropriately defining events related to those aspects.

Desired Behavior Specification in DREAM

Having defined a set of events by means of the DDN mechanisms described above, a software system designer may specify intended behavior for the system and its components by describing the possible sequencing and simultaneity of event occurrences which would be considered acceptable during the system's operation. This is accomplished in DDN by using event sequence expressions and concurrency expressions within desired behavior textual units.

As an example of desired behavior specification, consider the textual units shown in figure 7. The concurrency expressions in this figure use the

[1]The prefix '[region]: MONITOR CLASS' attached to the event definition textual unit in figure 6 indicates that this textual unit is intended to be an additional part of the definition of the [region] monitor class.

```
'[region]:  MONITOR CLASS' EVENT DEFINITIONS;
    shared_rd:  STATE SEQUENCE(rding_shared),
    shared_wrt:  STATE SEQUENCE(wrting_shared)
    END EVENT DEFINITIONS;
```

Figure 6

function operators MUTUALLY EXCLUSIVE and POSSIBLY CONCURRENT[1] to describe the set of behaviors for instances of the [region] monitor class which would be acceptable to the designer of the blackboard system.

The function operators for concurrency expressions are binary, their operands being sets of events. When appearing in an operand of a concurrency expression, event identifiers not qualified[2] by a class or instance identifier refer to event occurrences specific to any single instance of the class for which they are defined, while those qualified by a class identifier refer to event occurrences arising from any instance of the class and those qualified by an instance name refer to event occurrences specific to the named instance. MUTUALLY EXCLUSIVE(x,y) represents the constraining behavioral specification that no occurrence of an event from the set of events x may overlap any occurrence of an event from the set of events y, except that an event which is an element of both x and y is not precluded from occurring. Thus the first concurrency expression of the figure 7 example represents the restriction that while some [region] monitor class instance is performing a shared write, i.e. its event *shared_wrt* is occurring, no other [region] monitor class instance may be performing a shared write and no [region] monitor class instance may be performing a shared read. Similarly, the second concurrency expression expresses the restriction that while some [region] monitor class instance is

---

[1]POSSIBLY n CONCURRENT(x,y), where n is an integer expression, is a third DDN concurrency expression function operator which may be used for describing bounded concurrency situations.

[2]x|y specifies the identifier y which is defined within the definition of the identifier x.

```
'[region]:  MONITOR CLASS' DESIRED BEHAVIOR;
    MUTUALLY EXCLUSIVE(shared_wrt,OR([region]|shared_wrt,
                                     [region]|shared_rd) ),
    MUTUALLY EXCLUSIVE(shared_rd,[region]|shared_wrt),
    MUTUALLY EXCLUSIVE(OR(pvt_wrt,pvt_rd,shared_wrt,shared_rd),
                       OR(pvt_wrt,pvt_rd,shared_wrt,shared_rd) ),
    POSSIBLY CONCURRENT(shared_rd,[region]|shared_rd),
    POSSIBLY CONCURRENT(OR(pvt_wrt,pvt_rd),
                        OR([region]|pvt_wrt,[region]|pvt_rd,
                           [region]|shared_wrt,
                           [region]|shared_rd) )
    END DESIRED BEHAVIOR;
```

Figure 7

---

performing a shared read, i.e. its *shared_rd* event is occurring, no [region] monitor class instance may be performing a shared write. (Notice that neither of these concurrency expressions precludes the possibility of shared reads being performed by several [region] monitor class instances simultaneously.) The third concurrency expression indicates the designer's intention that at most one of the *pvt_wrt*, *pvt_rd*, *shared_wrt* or *shared_rd* events will be occurring at any time within any given instance of the [region] monitor class.

The concurrency expression POSSIBLY CONCURRENT(x,y) represents the permissive behavioral specification that any occurrence of an event from the set of events x may overlap any occurrence of an event from the set of events y. Thus the fourth concurrency expression of the figure 7 example indicates the designer's intention to allow multiple instances of the [region] monitor class to be performing shared reads simultaneously, i.e. an instance's *shared_rd* event may overlap the *shared_rd* event of any instance. Similarly, the final concurrency expression of the example expresses the designer's willingness to allow *pvt_wrt* or *pvt_rd* events of one instance of the [region] monitor class to overlap any of the events *pvt_wrt*, *pvt_rd*, *shared_wrt* or *shared_rd* of any [region] monitor class instance.

Taken together, the concurrency expressions of the desired behavior textual unit in the figure 7 example represent precisely the behavioral specifications which a software designer might wish to indicate regarding the operation of and interactions among the regions of the blackboard. That is, they specify that the writing of a shared subregion must not be concurrent with any other manipulations of the shared subregion (first concurrency expression), while the reading of a shared subregion may be concurrent with other reading but not writing in that subregion (second and fourth concurrency expressions). Further, this desired behavior textual unit indicates that reading or writing of private subregions may be concurrent with reading or writing, shared or private, by other instances of the [region] monitor class (fifth concurrency expression), but that at most one of the four operation types may be occurring within any given instance of the [region] monitor class at any given time (third concurrency expression).

DDN constructs for describing desired behavior are discussed in greater detail in [17]. The example of this section is sufficient, however, to indicate that the constructs provide a very general facility for describing a designer's intentions regarding acceptable behavior for the system under

design. The example also indicates the ways in which DDN facilitates the non-procedural, modular, perhaps redundant specification of behavior that spans more than one component or relates to a component's behavior over time.

Hierarchical System Description

In this paper, we are primarily interested in the description of the external attributes of software system components. However, we finish our series of examples with that of figure 8 which illustrates several DDN constructs for the description of internal attributes. We include this example because it completes our description of the HEARSAY data base and indicates that hierarchical descriptions may be constructed by using the external attributes of a set of subcomponents to define the internal attributes of another subcomponent.

Before commenting on the constructs for describing internal attributes, two comments may be made about the textual units in figure 8 which are concerned with the definition of external attributes. First, in the model for the *manager* array of control processes, note the use of nondeterminism in the set-to statement. DDN provides a variety of nondeterministic constructs since such constructs provide a convenient vocabulary for the task of abstraction. Second, in the event definition textual unit, note the ability to reference events outside of the class definition in order to give an outward-directed specification of behavior as well as the ability to reference internal events in order to give an inward-directed behavior specification.

With respect to the specification of internal attributes, in figure 8 there are several major subcomponents declared for each instance of the class. Three are declared in a straightforward way within the subcomponents textual unit -- the declarations specify the names of the subcomponents and their types. (The use of *MY_INDEX* within the declaration of the *reg* array of subcomponents indicates that *reg(1)* is of class [region(1)] and that *reg(2)* is of class [region(2)], where the arguments in the class reference specify values for the qualifiers in the class' definition.) The other major subcomponents are declared, without reference to a previously defined class, by giving a body textual unit for the *manager* array of control processes. This textual unit specifies the algorithm that is to be performed by the control process during subsystem operation. While it is not always the case, in this example the control processes map one-to-one onto subcomponents which control

```
[db_op]:  MONITOR CLASS;
  STATE SUBSETS; read, write, initialize END STATE SUBSETS;
  END MONITOR CLASS;
[blackboard]:  SUBSYSTEM CLASS;
  QUALIFIERS;  #_ks, #_under_surveillance  END QUALIFIERS;
  SUBCOMPONENTS;  reg ARRAY[1::2] OF [region(MY_INDEX)],
                  spy OF [noticer(#_under_surveillance)]
    END SUBCOMPONENTS;
  request:  ARRAY[1::#_ks] OF IN PORT;
    BUFFER SUBCOMPONENTS;   op OF [db_op],
                            id OF [entry_id],
                            val_to_write OF [datum]
      END BUFFER SUBCOMPONENTS;
    BUFFER CONDITIONS;
      op=read OR op=write
      END BUFFER CONDITIONS;
    END IN PORT;
  rd_response:  ARRAY[1::#_ks] OF OUT PORT;
    BUFFER SUBCOMPONENTS; val OF [datum]  END BUFFER SUBCOMPONENTS;
    END OUT PORT;
  note:  ARRAY[1::#_under_surveillance]  OF OUT PORT;
    BUFFER SUBCOMPONENTS;  notice OF [datum_change]
      END BUFFER SUBCOMPONENTS;
    BUFFER CONDITION;  notice=change  END BUFFER CONDITIONS;
    END OUT PORT;
  manager:  ARRAY[1::#_ks]  OF CONTROL PROCESS;
    MODEL;  ITERATE   RECEIVE request(MY_INDEX);
                      IF op(MY_INDEX)=read THEN
          gotread:              SET val(MY_INDEX) TO defined OR errorflag;
          answer:               SEND rd_response(MY_INDEX);
                                    ELSE
          gotwrite:             NULL;
                        END IF;
              END ITERATE;
      END MODEL;
    LOCAL SUBCOMPONENTS; region_number OF [region_id(2)]
      END LOCAL SUBCOMPONENTS;
    BODY;   ITERATE  RECEIVE request(MY_INDEX);
                     region_number.determine(id(MY_INDEX));
                     IF op(MY_INDEX)=write
                       THEN reg(region_number).write(val_to_write(MY_INDEX));
                       ELSE reg(region_number).read(val(MY_INDEX));
                            SEND rd_response(MY_INDEX);
                       END IF;
              END ITERATE;
      END BODY;
    END CONTROL PROCESS;
```

Figure 8, part 1

```
EVENT DEFINITIONS;
  notice: DESCRIPTION;passage of a message out through
                      one of the note ports,
                      END DESCRIPTION,
  noticed_write:
    SEQUENCE(OR(reg(1)|write, reg(2)|write), spy|see_it),
  some_request:
    OR(gotread, SEQUENCE(gotwrite, notice)),
  serviced_write:
    SEQUENCE([ks]|request_write, gotwrite, notice)
  END EVENT DEFINITION;

DESIRED BEHAVIOR;
  POSSIBLY CONCURRENT(noticed_write, [blackboard]|noticed_write),
  POSSIBLY CONCURRENT(some_request, [blackboard]|some_request),
  POSSIBLY CONCURRENT(serviced_write, [blackboard]|serviced_write)
  END DESIRED BEHAVIOR;

END SUBSYSTEM CLASS;
```

Figure 8, part 2

---

the message flow through the ports and the activation of the other subcomponents. When this is the case, it is convenient to describe the subcomponents directly as the bodies of the control processes.

The DDN facilities for hierarchical description of internal attributes are discussed more fully in [5]. The major point to be noted here is that the description of the class [blackboard] is redundant, with the description of the componentry within a class of subsystems procedurally indicating how the subsystems operate and the rest of the description providing a basically non-procedural specification of behavior.

The Use of Behavior Specifications in Software Design Analysis

The DDN constructs for describing a software system design present numerous possibilities for analysis of a design at various stages in its evolution. The DDN behavioral specification constructs discussed here are particularly valuable as a basis for simulation and formal or informal consistency checking approaches to design verification.

The transitions and control process models of a DDN software system description provide the basis for simulation of a system at any stage in the design effort. Following the approach described in [18], expected values for timing data and random variables controlling nondeterministic processing can be used to augment the control process model specifications, permitting performance statistics to be derived for the system as it is described at any

stage in its development. Such simulations can uncover potential performance problems at an early point in the design process, rather than allowing their discovery to await a completed system implementation.

The DREAM behavioral specification technique allows for both formal and informal analysis of a software system design based on consistency checking. Informal arguments for correctness of a design may be made by showing the consistency of the various redundant specifications for the designed system. Such arguments may compare internal specifications, such as found in control process bodies, with external specifications, such as given by control process models or stated in the desired behavior textural units. Alternatively, informal arguments may demonstrate the consistency among different behavioral specifications, such as control process models and desired behavior textural units or possibly the desired behavior textual units from several DDN class definitions. Such arguments are clearly not, in and of themselves, sufficient to prove the correctness of a design. However, by helping a designer to consider the proposed design from various viewpoints, they can be very helpful in revealing errors or, when several different descriptions prove consistent, in increasing confidence in the design's correctness.

Formal approaches to the analysis of DDN software system designs can utilize essentially this same type of consistency checking between different descriptions of the system's intended behavior. These formal methods, slated for inclusion in future versions of the DREAM system, are based upon the automatic derivation, using an algorithm similar to that defined in [19], of an event sequence expression describing the behavior represented by each control process model. The behaviors represented by these expressions may then be compared with those specified in desired behavior textual units to determine whether the system as designed would conform to the designer's intentions as expressed in the desired behavior information. In general, this comparison cannot be conducted algorithmically [19], but cases can be found [20] in which algorithmic manipulation is tractable. Alternatively, the derived expressions may be presented to the software system designer, whose knowledge and insight may facilitate the comparison with desired behavior specifications, or who may simply find this alternative representation of the control process' behavior informative. This latter approach is an example of the "feedback analysis" [21] capabilities made possible by the DDN software design language, with information concerning the characteristics of the system under design being presented to the designer who uses human insight to check the correctness of the design.

Conclusions

The behavior specification constructs described in this paper are intended to serve as useful tools for software system designers. They can assist in the rigorous and formal description of a complex software system design and also provide a basis for some analysis of that design at various stages of the design effort.

The DDN design description facilities offer a medium for presenting rigorous, formal and abstract specifications of software system components, describing their behavior without revealing or requiring details of their internal operation. Control process model, procedure transition, event definition and desired behavior textual units can all serve as possibly redundant, abstract behavioral specifications, orthogonal to the operational descriptions of the implementations of the subcomponents. Such specifications are valuable at early stages of design, when they allow for a high-level, non-detailed description of the evolving software system. In later phases of the design effort they are ideally suited for use as external specifications for component behavior, fulfilling an information-hiding function while presenting a complete and formal component description.

Acknowledgements

The authors are indebted to John Sayler and Alan Segal who participated in the development of the DREAM and contributed significantly to the material presented in this paper. We are also indebted to Allan Stavely for his contribution to this material. We would like to thank Derry Kabcenell and Mark Welter for their contributions to the development of the DREAM System and Jan Cuny, Victor Lesser, Carolyn Steinhaus and Pamela Zave for asking the right questions at the right time.

We feel ourselves fortunate to have been able to use the TOPD System ([4], [9]) over the last two years. TOPD was developed as a tool to aid program implementation, but using it for program design tasks greatly facilitated our development of the DDN language and the DREAM System.

References

[1] W. E. Riddle, J. Sayler, A. Segal and J. Wileden. An Introduction to the DREAM Software Design System. Software Engineering Notes, 2, 4 (July 1977), 11-23.

[2] J. J. Horning and B. Randell. Process Structuring. Comp. Surveys, 5, 1 (March 1973), 5-30.

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

[4] P. Henderson, et al. The TOPD System. Tech. Report 77, Computing Laboratory, Univ. of Newcastle upon Tyne, England, September 1975.

[5] W. E. Riddle. Hierarchical Description of Software System Structure. RSSM/40, Department of Computer Science, University of Colorado at Boulder, Boulder, October 1977.

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

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

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

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

[10] D. L. Parnas. A Technique for Software Module Specification with Examples. Comm. ACM, 15, 5 (May 1972), 330-336.

[11] 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 Calif., 1976.

[12] Barbara Liskov, et al. Abstraction Mechanisms in CLU. Comm. ACM, 20, 8 (August 1977), 564-576.

[13] N. Wirth. The Programming Language PASCAL. Acta Informatica, 1, (1971), 35-63.

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

[15] W. E. Riddle. The Hierarchical Modelling of Operating System Structure and Behavior. Proc. ACM 72 National Conf., August 1972.

[16] W. E. Riddle. Abstract Process Types. RSSM/42, Dept. of Computer Science, Univ. of Colorado at Boulder, Boulder, November 1977.

[17] J. Wileden. Behavior Specification in a Software Design System. RSSM/43, Dept. of Computer and Communication Sciences, Univ. of Michigan, Ann Arbor, in preparation.

[18] J. W. Sanguinetti. Performance Prediction in an Operating System Design Methodology. RSSM/32, Dept. of Computer and Communication Sciences, Univ. of Michigan, Ann Arbor, May 1977.

[19] W. E. Riddle. An Approach to Software System Modelling, Behavior Specification and Analysis. RSSM/25, Dept. of Computer and Communication Sciences, University of Michigan, Ann Arbor, July 1976.

[20] M. Welter. Counter Expressions. RSSM/24, Dept. of Computer and Communcation Sciences, University of Michigan, Ann Arbor, October 1976.

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

Abstract: A modelling scheme is presented which provides a medium for the rigorous, formal and abstract specification of large-scale software system components. The scheme allows the description of component behavior without revealing or requiring the description of a component's internal operation. Both collections of sequential processes and the data objects which they share may be described. The scheme is of particular value during the early stages of software system design, when the system's modules are being delineated and their interactions designed, and when rigorous, well-defined specifications of undesigned components allows formal and informal arguments concerning the design's correctness to be formulated.

Key Words and Phrases: software design languages, software system behavior modelling, message transfer models, event-based models, non-procedural specification, desired behavior specification, software design analysis, DREAM

CR Categories: 4.9, 4.29

Proc. 3rd International Conf. on Software Engineering, Atlanta, May 1978.

BEHAVIOR MODELLING DURING SOFTWARE DESIGN[†]

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

Jack C. Wileden
Computer and Information Sciences Department, University of Massachusetts
Amherst, Massachusetts

John H. Sayler, Alan R. Segal
Computer and Communication Sciences Department, University of Michigan
Ann Arbor, Michigan

Allan M. Stavely
Computer Science Department, New Mexico Institute of Mining and Technology
Socorro, New Mexico

Abstract: A modelling scheme is presented which provides a medium for the rigorous, formal and abstract specification of large-scale software system components. The scheme allows the description of component behavior without revealing or requiring the description of a component's internal operation. Both collections of sequential processes and the data objects which they share may be described. The scheme is of particular value during the early stages of software system design, when the system's modules are being delineated and their interactions designed, and when rigorous, well-defined specification of undesigned components allows formal and informal arguments concerning the design's correctness to be formulated.

Key Words and Phrases: software design languages, software system behavior modelling, message transfer models, event-based models, non-procedural specification, desired behavior specification, software design analysis, DREAM

Introduction

At an intermediate point in the design of a software system, some of the system's components will be completely designed whereas other components will be only partially designed and the design of some components will not have been begun. At this point, the designer could proceed with the next design step, further detailing the design of one of the incompletely designed components. More effective, however, would be for the designer to first gain confidence, through formal or informal arguments, that the design is appropriate and correct. But this is generally precluded because of the absence of a rigorous specification of the incompletely designed components. In this paper, we develop a description scheme that allows incompletely designed software system components to be rigorously specified so that designers may incrementally gain confidence in a design as it is being developed.

Rigorous specification of an incompletely designed component requires the ability to model the component's behavior. That is, it requires the ability to describe what the component will do -- its behavior -- in terms of an abstraction of the component's operation which focuses upon effect rather than cause. One means of abstraction is simple elimination of detail. Although not always possible, this can be done when only certain characteristics of the component need to be preserved in the model. For example, the model may describe the component's processing with respect to only a portion of its input. Alternately, abstraction may be accomplished by using a description scheme in which the component's interesting characteristics may be succinctly specified. The vocabulary of the new description scheme can be chosen to allow the direct statement of characteristics that are only implicitly specified in a detailed implementation description. A function procedure, for example, can be succinctly modelled by its mathematical definition.

The modelling scheme presented in this paper allows both these approaches to abstraction to be used during the design of large-scale software systems. In the scheme, software system components are described as collections of concurrent[1] processes or as data objects shared among these collections, and the description of the interactions among the components is emphasized. As a means of rigorously specifying undesigned system components, the scheme is therefore of primary use during the early phases of large-scale software system design, when the system's modules are being delineated and their interactions designed.

The scheme presented here was developed for use in an interactive design tool called the Design Realization, Evaluation And Modelling (DREAM) system [1]. DREAM is based upon a design language, the DREAM Design Notation (DDN), and the scheme discussed here is a major part of that language.

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

[1]We use the term concurrent to connote parallelism which may or may not be actually realized when the system is executed. A multiprogrammed system, for instance, is a concurrent system in which no operations are actually performed in parallel.

DDN allows a design to be developed incrementally, in fragments called textual units, and DREAM contains a data base management facility which allows a design description to be augmented or modified on a textual unit basis. DREAM has also been developed in order to provide a variety of analysis aids to designers of large-scale software systems; some of these aids are discussed at the end of this paper.

It is not the purpose of this paper to give a full description of the DDN language; that is done in the referenced reports on various aspects of the language. Rather, in this paper we discuss and illustrate those features of DDN which support the modelling and analysis of large-scale software systems. In the next two sections, we outline the DDN approach to system description and discuss several desirable characteristics of description schemes intended for the modelling of software systems. Next, we introduce the DDN modelling constructs, first those for modelling shared data objects and then those for the modelling of collections of concurrent processes. We then discuss some additional constructs which permit the non-procedural specification of behavior. In the concluding sections, we indicate that the modelling scheme provides a basis for several approaches to design analysis and lends beneficial support to the designers of large-scale software design.

DDN Descriptions

In DDN descriptions, a software system is decomposed into components of two types. Subsystems are those components which control and guide the performance of the system's processing, which operate (conceptually at least) in parallel and asynchronously with respect to other components, and which are individually capable of performing several activities at once.[1] Monitors are those components which also operate concurrently and asynchronously with respect to other components but which serve primarily as repositories of shared information and are individually capable of performing only a single activity at any point in time.[2] This decomposition may be hierarchical[3], since a subsystem may be decomposed into (sub-)subsystems and (sub-)monitors and a monitor may be decomposed into (sub-)monitors.

Hierarchical decomposition may proceed to any one of a number of levels. For instance, the primitive (i.e., undecomposed) components could correspond to the processing units and data objects provided by the system's execution environment. Or they could correspond to the undesigned components existing at some point during the system's design. Whatever the extent of the decomposition, the system's overall operation is the result of the activity of the primitive subsystems as coordinated through their shared usage of the primitive monitors. A particularly important means of coordination, since it corresponds to direct subsystem interaction, is the transfer of messages. This mode of interaction is therefore distinguished in DDN, making DDN a message transfer modelling scheme.

We illustrate this approach to software description by applying it to the HEARSAY speech recognition system [6] developed at Carnegie-Mellon University. In HEARSAY, all information about the utterance being processed and all hypotheses as to its linguistic structure are stored in a central data base called a blackboard. The information in the blackboard is augmented and modified by knowledge sources, each of which enforces a set of speech recognition rules. The obvious subsystems in an initial decomposition of HEARSAY would therefore be the blackboard and the knowledge sources. The message transfer interactions would consist of the request messages sent by the knowledge sources and the responses returned by the blackboard. In addition, a message would be sent by the blackboard to activate a knowledge source when an entry of interest to the knowledge source changes value.

A possible next decomposition step might be to demarcate several components within the blackboard. There would be subsystems (one for each knowledge source) within the blackboard which exchange messages with the knowledge sources and manage the modifications each knowledge source makes to the region of the blackboard of interest to the knowledge source. The blackboard regions themselves would be monitors since they represent information repositories and since the synchronization primitives provided by monitors allow the succinct description of the coordination among the possibly conflicting reading and writing of the areas falling within more than one region. Also delineated at this level of decomposition would be those subsystems within the blackboard which notify a knowledge source when an entry of interest to it has changed value.

Attributes of Modelling Schemes

The description of systems viewed as proposed in the previous section requires a hierarchical modelling scheme. Such a scheme must be able to describe a component's external attributes, those characteristics which pertain to its interactions with other components at the same level of decomposition. It must also be able to describe a component's internal attributes, those aspects which pertain to the manner in which a component is composed of other components and the ways in which these components are to interact so as to create the intended operation of the component which they comprise.

---

[1] Those system components which execute concurrently and manipulate shared data objects are usually considered to be sequential processes, as defined in [2]. A subsystem is a more general object, being essentially a collection of sequential processes.

[2] The monitors of DDN are essentially those defined by Hoare [3]. To the usual definition of monitors, we have added constructs for behavior specification, patterned after constructs developed for the TOPD system [4].

[3] We assume, for the purposes of this paper, that this hierarchical decomposition is tree-like. Non-tree-like decomposition is discussed in [5].

With respect to describing the external attributes of components, and focusing upon the description needs of software system designers, several desirable characteristics of modelling schemes may be delineated. First, facilities should be provided for both outward-directed descriptions, which describe those aspects of a component's behavior which are relevant to its interactions with other components, and inward-directed descriptions, which describe those aspects of behavior which are significant in developing the component's implementation. Second, the scheme should support projection, providing the ability to focus upon and highlight interesting behavior (for a variety of definitions of "interesting") and suppress irrelevant details. Third, a means should be provided for non-procedural specification, allowing the definition of behavior without the specification of an algorithm for achieving the behavior. Fourth, the scheme should admit descriptions that are non-prescriptive in that a wide variety of strategies, mechanisms and algorithms can be used to implement the described behavior. Fifth, redundant specifications should be possible, to allow the same behavior to be specified from different points of view or with respect to different sets of concerns. Sixth, it should be possible to give a description that is orthogonal to the component's internal description in the sense that it may establish associations among activities which occur within physically different parts of the component. Seventh, the scheme should admit modular descriptions so that different properties may be independently specified and inter-relationships among these properties may be specified separately from the specification of the properties themselves. Finally, the scheme should lead to analysis-oriented descriptions which can serve as the basis for formal or informal arguments through which the designer may gain increased confidence in the accuracy of the design.

The DDN constructs introduced in the next three sections provide software system designers with a rigorous, formally-defined technique for the modular, non-prescriptive specification of the external attributes of both monitor and subsystem subcomponents. The constructs allow analysis-oriented descriptions which are redundant, orthogonal, projective, non-procedural (and procedural), and both inward-directed and outward-directed. The constructs are illustrated by a series of examples which, taken together, provide an abstract description of the blackboard subsystem within the HEARSAY system to the level of decomposition developed above.[1] In the examples, we focus upon describing the blackboard's organization and behavior and upon specifying the policies concerning the concurrent operation of the region managers. We do not attempt to specify the mechanisms, or even the strategies, by which conflicts are prevented -- this is deliberately done so as to highlight DDN's use as a modelling rather than a programming language.

[1]The description reflects our understanding of the HEARSAY system and is oriented toward providing examples of the DDN constructs. We feel that the description is reasonably accurate, but do not claim that it fully corresponds to the actual HEARSAY system.

The Description of Monitors

To avoid describing each component individually, be it monitor or subsystem, DDN descriptions are of classes of components. The class concept was introduced in SIMULA [7] and various aspects of it have subsequently been widely used in computer-oriented description schemes -- abstract data types (as defined in [8]), TOPD classes ([4],[9]), Parnas modules [10], Alphard forms [11], CLU clusters [12] and Pascal types [13]. In DDN, class definitions define the external and internal attributes of each instance of the class -- here, we focus primarily upon those parts of class definitions which specify external attributes.

For a monitor class, external attributes pertain to the procedures which may be invoked upon instances of the class. For each procedure, its name and parameters must be specified along with a definition of its behavior. A procedure's behavior may be specified non-procedurally via a formal definition of the relationship between its input and output. In DDN, this is accomplished by defining the changes the procedure makes in the states of the objects it operates upon.

One aspect of the specification of the external attributes of a class of monitor objects is therefore the definition of a set of observable states. The concept of observable state is more general than the concept of "value", since a state may encode an instance's past history as well as reflect the instance's current "value". For example, a region within the blackboard could be in the state *writing_shared* indicating that a write operation is in progress upon a portion of the region that is shared with another region.

Simple examples of DDN monitor class definitions are given in figure 1. Instances of these

```
[region_id]: MONITOR CLASS;
  QUALIFIERS; range_limit  END QUALIFIERS;
  STATE SUBSETS; 1, in_range, range_limit  END STATE SUBSETS;
  STATE ORDERING; 1 <= in_range <= range_limit
    END STATE ORDERING;
  determine: PROCEDURE;
    PARAMETERS; id VALUE OF [entry_id]  END PARAMETERS;
    TRANSITIONS; id=defined --> in_range
      END TRANSITIONS;
    END PROCEDURE;
  END MONITOR CLASS;
[entry_id]: MONITOR CLASS;
  STATE SUBSETS; defined, undefined  END STATE SUBSETS;
  END MONITOR CLASS;
[datum]: MONITOR CLASS;
  STATE SUBSETS; defined, error-flag  END STATE SUBSETS;
  END MONITOR CLASS;
```

Figure 1

classes are "variables" which are needed in later class definitions -- objects of class[2] [region_id] are integers which fall within some range and which

[2]In DDN, identifiers used to name classes are always enclosed in square brackets.

are used to identify the regions of the blackboard; objects of class [entry_id] are values which are used to address the entries in the blackboard; class [datum] objects are the values stored as entries in the blackboard.

The definition of [region_id] in figure 1 indicates that "parameterized" class definitions are allowed in DDN. The QUALIFIERS textual unit specifies that the limit of the range may vary among instances and may be specified when each instance is declared. Qualifiers, which are discussed more fully in [5], may be used to specify a value for any single lexicographic unit within a class definition.

In the class definitions of figure 1, states are not specified explicitly; rather,(not necessarily disjoint) subsets of the state space are defined. States, themselves, permit the potentially infinite domain of "values" for a monitor object to be modelled by partitioning into a finite number of disjoint sets. State subsets extend this grouping capability, allowing the description to be focused upon interesting characteristics of monitor activity. Note that an ordering relationship may be established among states so that instances may be compared through the use of the usual set of relational operators.

An instance of a monitor class may be inspected, at any time, to determine its state (or state subset) -- it is in this sense that states are observable. This is quite valuable for the succinct, abstract specification of behavior as will be illustrated in later examples. When used to specify the algorithmic detail of a component's internal operation, however, state inspection may need to be coordinated with other operations upon the monitor. When such coordination is necessary, it may easily be effected by having the state inspection performed by a procedure defined for the monitor class.

The PROCEDURE textual unit of figure 1, and the PARAMETER textual unit nested within it, specify that the *determine* operation is available for manipulating instances of the class [region_id] and that an instance of the class [entry_id] must be passed as a value parameter. The intended purpose of this procedure is to determine in which region an identified entry lies. This is reflected by the TRANSITIONS textual unit which specifies that a pre-condition for the invocation of the procedure is that the parameter be in a state in its *defined* state subset and that the effect of the procedure is to leave the state of the parameter unchanged and to leave the object upon which the procedure is invoked in a state in its *in_range* state subset.

The regions of the blackboard are described in figure 2. The STATE VARIABLES textual unit indicates that a coordinatization of the state space may be used to specify the states. For example, a class [region] object could be in the state <<selected=yes, doing=read>> which is intended to denote that the region has the right to access information which it shares with another region and is in the process of reading that shared information. With the specification of state variables, state subsets may be defined as indicated in the STATE SUBSETS textual unit appearing in figure 2. The notation "--" specifies that the corresponding state variable may have any one of its possible values.

The example of figure 2 also illustrates the general form for the specification of transitions.

```
[region]:  MONITOR CLASS;
  QUALIFIERS;  my_id  END QUALIFIERS;
  STATE VARIABLES;  selected: VALUES(yes, no),
                    doing:  VALUES(read, write, neither)
    END STATE VARIABLES;
  STATE SUBSETS;
    reading_private: <<--,              doing=read>>,
    writing_private: <<--,              doing=write>>,
    reading_shared:  <<selected=yes,    doing=read>>,
    writing_shared:  <<selected=yes,    doing=write>>,
    stalled:         <<selected=no,     doing=read OR doing=write>>,
    unoccupied:      <<--,              doing=neither>>
    END STATE SUBSETS;
  read:  PROCEDURE;
    PARAMETERS;  value_read RESULT OF [datum]  END PARAMETERS;
    TRANSITIONS;
      unoccupied
        ||OR(SEQUENCE(stalled,reading_shared),reading_shared)||
          unoccupied AND (value_read=defined OR value_read=error_flag)
      private_read:  unoccupied
        ||reading_private||
        unoccupied AND (value_read=defined OR value_read=error_flag),
      END TRANSITIONS;
    END PROCEDURE;
  write:  PROCEDURE;
    PARAMETERS; value_to_write VALUE OF [datum]
      END PARAMETERS;
    TRANSITIONS;
      private_write:  value_to_write=defined AND unoccupied
                        ||writing_private||
                          unoccupied,
      value_to_write=defined AND unoccupied
        ||OR(SEQUENCE(stalled,writing_shared),writing_shared)||
          unoccupied
      END TRANSITIONS;
    END PROCEDURE;
  END MONITOR CLASS;
```

Figure 2

The "-->" notation used previously specifies that the state change occurs without the objects that are being manipulated being in any observable intermediate states. The transitions of figure 2, however, specify that intermediate states are observable. The first transition for the *read* procedure, for example, indicates that during the procedure's execution, the class [region] instance being manipulated passes through the sequence of states *stalled*, *reading_shared* or the sequence *reading_shared*.[1]

The Description of Subsystems

Since direct interaction between subsystems takes place via message transfer, a subsystem's external attributes may be described by specifying the message flow into and out of the subsystem. Part of this specification is a definition of the

[1]The constructs for describing sequences of states are discussed in [14].

communication paths which cross the subsystem's boundary and the demarcation of any restrictions as to what messages may legally flow across the boundary -- this is a specification of the subsystem's interface. A second part of the specification describes correlations among messages flowing into and out of the subsystem -- this is a description of the subsystem's behavior over time.

In DDN, communication channels are represented by specialized monitors, called links, which can store and forward messages and to which subsystems may be attached. The point of attachment of a subsystem to a link is called a port. Each port is therefore a "hole" through which messages may flow, having a directional attribute, either in or out. The DDN constructs for port definition are illustrated in figure 3, a subsystem class definition describing those components of a blackboard which notify the knowledge sources of changes to entries in the stored information.

The messages which flow through a port are specified by the set of buffers associated with the port definition. In the example, each port *note(i)* has a single buffer, *notice(i)*, associated with it. If the port is an out-port, then a message sent out through the port is the (ordered) composition of the contents of the associated buffers at the time that the send operation causing the message flow is performed. For an in-port, when a message passes in through the port (as a consequence of a receive[1] operation) the message is decomposed and used to determine new contents for the buffers associated with the port.

The BUFFER CONDITIONS textual unit of figure 3 indicates that the state of the *notice* portion of

```
[noticer]: SUBSYSTEM CLASS;
  QUALIFIERS; #_under_surveillance  END QUALIFIERS;
  note: ARRAY[1::#_under_surveillance] OF OUT PORT;
    BUFFER SUBCOMPONENTS; notice OF [datum_change]
      END BUFFER SUBCOMPONENTS;
    BUFFER CONDITIONS; notice=change  END BUFFER CONDITIONS;
    END OUT PORT;
  observer: ARRAY[1::#_under_surveillance] OF CONTROL PROCESS;
    MODEL; ITERATE
      see_it: SET  notice(MY_INDEX) TO change;
              SEND note (MY_INDEX);
              END ITERATE;
    END MODEL;
  END CONTROL PROCESS;
END SUBSYSTEM CLASS;
```

Figure 3

the message (in this case, the *notice* portion is the entire message) will be in its *change* state subset. The definition of class [datum_change], given in figure 4, indicates that this means the

```
[datum_change]: MONITOR CLASS;
  STATE SUBSETS; change, out_of_range  END STATE SUBSETS;
  END MONITOR CLASS;
```

Figure 4

[1]Receive is a potentially-blocking operation whereas send is a non-blocking operation. The semantics of these operations, and the operation of links, is explained more fully in [15] and [16].

notice portion will never be in a state that is in its *out_of_range* state subset.

Buffer conditions are both outward-directed and inward-directed specifications. With respect to interactions with other subsystems, they specify which messages will be sent out (for out-ports) or are expected to be received (for in-ports). With respect to the eventual design of the subsystem, they inform the designer of the limitations concerning which messages may be sent out (for out-ports) and which incoming messages should be expected (for in-ports).

The control process portion of the example in figure 3 provides a procedural specification of the sequential portions of the subsystem's behavior. In general, control process models specify sequences of message flow across the subsystem's boundary and therefore define correlations among messages flowing at different times through one or more of the ports. In the example, the control process model indicates that there is a constant stream of *change* messages flowing out through each port. This is the appropriate behavior, at this level of decomposition, for that part of the HEARSAY blackboard which notifies the knowledge sources of changes in the entries in the data base.

One purpose of control processes is to abstractly model the actual operation of the subsystem. In the example of figure 3, abstraction is partially achieved through the elimination of detail afforded by the set-to operation. This statement models the possibly complex and lengthy processing needed to cause the *notice(MY_INDEX)* buffer[1] to be in a state within its *change* state subset. Abstraction is also achieved by restricting operations to be performed only upon the ports and their buffers -- inside a control process model, reference may not be made to the subsystem's internal componentry and hence the model may not specify anything about the algorithmic detail of the subsystem's internal operation. In the example, this results in a desirable hiding of information as to what internal activity actually causes messages to be sent out. Though always non-detailed with respect to the subsystem's internal operation, models may be very elaborate (when the modelled behavior is itself elaborate); this and other aspects of control processes are described in [16].

The DDN constructs for describing a subsystem's external attributes allow the focusing of the description upon the interactions in which the subsystem is able to participate. This is accomplished by the explicit specification of subsystem interaction in terms of message transfer. Further, because of the ability to construct abstract models of the subsystem's behavior, the description may be rigorous, projective, outward-directed and modular. Finally, the description may be inward-directed, since it specifies the behavior which the implementation must achieve; but

[1]Within arrays of control processes, *MY_INDEX* is a variable which has a different value for each control process in the array and may be used, as here, to make each model specific to a different set of ports and thus buffers.

it may also be orthogonal, since the organization of the subsystem's internal componentry need not bear any direct resemblance to the organization of its control processes.

Event Definition

Procedure transitions and control process models are the basic DDN constructs for abstractly describing the simple, sequential behavior of individual software system components. More complex behavior, which is not sequential in nature or which pertains to inter-relationships between the behavior of several components, may be described in DDN by the definition of events (significant occurrences during system operation) and the non-procedural specification of sequences of events. Event definition is discussed in this section and event sequence specification is covered in the next section.

We distinguish two broad types of events, endogenous and exogenous, in DDN. Endogenous events are those occurrences which arise from some activity within the currently DDN-described portions of the software system. Exogenous events are those occurrences which are relevant to or impinge upon the system's behavior but arise from some activity outside the currently described portions of the software system. Whether an event is endogenous or exogenous is therefore relative to the extent of the system's description and may change over time--for example, an exogenous event may become an endogenous event as elaboration of the design leads to the description of the component whose activity gives rise to the event. Some events, however, are inherently exogenous since they pertain to the system's operation but do not stem from the software portion of the system being designed -- examples of such events are activities within some other software system which interacts with the system being designed or operations performed by some physical device controlled by the software system.

The most elementary method for defining endogenous events is to simply attach a label, called an event identifier, to some portion of the DDN description of a procedure or a control process. For example, the [region] monitor class description of figure 2 defines the events *read* and *write*, corresponding to executions of the respective procedures, and the events *private_read* and *private_write*, corresponding to occurrences of (some of the) transitions defined for those procedures. Thus, an execution of the *read* procedure of some instance of class [region] would be an instance of a *read* event, while an occurrence of the second transition defined for the *read* procedure would be an instance of the event *private_read*. Similarly, an occurrence of the buffer modification described by the set-to statement within the *observer* control process model in the [noticer] subsystem (figure 3) would correspond to a *see_it* event. Note that events defined within a DREAM design description may occur simultaneously and that one event may occur as part of another, as in the case of *private_read* and *read*.

Unlike endogenous events, definitions of which may be embedded within the monitor or subsystem classes whose activities give rise to them, exogenous events cannot be associated with any monitor or subsystem class definition. Therefore, a third class type, the event class, is available in DDN for the definition of exogenous events. This is illustrated in figure 5 in which we describe part of the activity of knowledge sources as it relates to the operation of the blackboard.

```
[ks]:  EVENT CLASS;
   EVENT DEFINITION;
     request_write:  DESCRIPTION;
       An event which occurs when a knowledge source requests a
       write operation upon a region within the blackboard.
       END DESCRIPTION;
     END EVENT DEFINITION;
   END EVENT CLASS;
```

Figure 5

In addition to its use in the definition of exogenous events as illustrated in figure 5, the EVENT DEFINITION textual unit may be used within a monitor, subsystem or event class definition for the specification of more complex events. These events may be specified in terms of other events, sequences of states of a monitor class, or sequences of statements in a control process model. In each case the specification is labelled with an event identifier naming the specified event. Details and examples of DDN event definition facilities may be found in [17]; here we provide only a simple illustration (figure 6[1]) in which the events *shared_read* and *shared_write* are defined as state sequences (of length one). These definitions indicate that the

```
'[region]:  MONITOR CLASS' EVENT DEFINITION;
    shared_read:  STATE SEQUENCE(reading_shared),
    shared_write:  STATE SEQUENCE(writing_shared)
    END EVENT DEFINITION;
```

Figure 6

shared_read and *shared_write* events correspond to an instance of monitor class [region] being in the state subsets *reading_shared* and *writing_shared*, respectively.

The event definition mechanisms of DDN provide a flexible and powerful tool for defining arbitrarily complicated events in a software system design. This event definition capability is the foundation of the DREAM behavior specification technique. Moreover, its flexibility and generality are largely responsible for the technique's projection properties, since various interesting aspects of system behavior may be highlighted by appropriately defining events related to those aspects.

[1]The prefix '[region]: MONITOR CLASS' attached to the EVENT DEFINITION textual unit in figure 6 indicates that this textual unit is intended to be an additional part of the definition of the [region] monitor class.

Desired Behavior Specification in DREAM

Having defined a set of events by means of the DDN mechanisms described above, a software system designer may specify intended behavior for the system and its components by describing the possible sequencing and simultaneity of event occurrences which would be considered acceptable during the system's operation. This is accomplished in DDN by using event sequence expressions and concurrency expressions within DESIRED BEHAVIOR textual units.

As an example of desired behavior specification, consider the textual unit shown in figure 7. The

```
'[region]:  MONITOR CLASS' DESIRED BEHAVIOR;
    MUTUALLY EXCLUSIVE(shared_write, OR([region]|shared_write,
                                        [region]|shared_read)),
    MUTUALLY EXCLUSIVE(shared_read, [region]|shared_write),
    MUTUALLY EXCLUSIVE(OR(private_write,private_read,shared_write,shared_read),
                       OR(private_write,private_read,shared_write,shared_read)),
    POSSIBLY CONCURRENT(shared_read, [region]|shared_read),
    POSSIBLY CONCURRENT(OR(private_write,private_read),
                        OR([region]|private_write,[region]|private_read,
                           [region]|shared_write,
                           [region]|shared_read))
    END DESIRED BEHAVIOR;
```

Figure 7

concurrency expressions in this figure use the operators MUTUALLY EXCLUSIVE and POSSIBLY CONCURRENT[1] to describe the set of behaviors for instances of the [region] monitor class which would be acceptable to the designer of the blackboard system.

The operators for concurrency expressions are binary, their operands being sets of events. When appearing in an operand of a concurrency expression, event identifiers not qualified[2] by a class or instance identifier refer to event occurrences specific to any single instance of the class for which they are defined, while those qualified by a class identifier refer to event occurrences arising from any instance of the class and those qualified by an instance name refer to event occurrences specific to the named instance. MUTUALLY EXCLUSIVE(x,y) represents the constraining behavioral specification that no occurrence of an event from the set of events x may overlap any occurrence of an event from the set of events y, except that an event which is an element of both x and y is not precluded from occurring. Thus the first concurrency expression of the figure 7 example represents the restriction that while some [region] monitor class instance is performing a shared write, i.e. its event *shared_write* is occurring, no other [region] monitor class instance may be performing a shared write and no [region] monitor class instance may be performing a shared read. Similarly, the second concurrency expression expresses the restriction that while some [region] monitor class instance is performing a shared read, i.e. its *shared_read* event is occurring, no [region] monitor class instance may be performing a shared write. (Notice that neither of these concurrency expressions precludes the possibility of shared reads being performed by several [region] monitor class instances simultaneously.) The third concurrency expression indicates the designer's intention that at most one of the *private_write*, *private_read*, *shared_write* or *shared_read* events will be occurring at any time within any given instance of the [region] monitor class.

The concurrency expression POSSIBLY CONCURRENT (x,y) represents the permissive behavioral specification that any occurrence of an event from the set of events x may overlap any occurrence of an event from the set of events y. Thus the fourth concurrency expression of the figure 7 example indicates the designer's intention to allow multiple instances of the [region] monitor class to be performing shared reads simultaneously, i.e. an instance's *shared_read* event may overlap the *shared_read* event of any instance. Similarly, the final concurrency expression of the example expresses the designer's willingness to allow *private_write* or *private_read* events of one instance of the [region] monitor class to overlap any of the events *private_write*, *private_read*, *shared_write* or *shared_read* of any [region] monitor class instance.

Taken together, the concurrency expressions of the DESIRED BEHAVIOR textual unit in the figure 7 example represent precisely the behavioral specifications which a software designer might wish to indicate regarding the operation of and interactions among the regions of the blackboard. That is, they specify that the writing of a shared subregion must not be concurrent with any other manipulations of the shared subregion (first concurrency expression), while the reading of a shared subregion may be concurrent with other reading but not writing in that subregion (second and fourth concurrency expressions). Further, this DESIRED BEHAVIOR textual unit indicates that reading or writing of private subregions may be concurrent with reading or writing, shared or private, by other instances of the [region] monitor class (fifth concurrency expression), but that at most one of the four operation types may be occurring within any given instance of the [region] monitor class at any given time (third concurrency expression).

DDN constructs for describing desired behavior are discussed in greater detail in [17]. The example of this section serves to indicate that the constructs provide a very general facility for describing a designer's intentions regarding acceptable behavior for the system under design. The example also indicates the ways in which DDN facilitates the non-procedural, modular, perhaps redundant specification of behavior that spans more than one component or relates to a component's behavior over time.

Hierarchical System Description

In this paper, we are primarily interested in the description of the external attributes of software system components. However, we finish our series of examples with that of figure 8 which

[1] POSSIBLY n CONCURRENT(x,y) where n is an integer expression, is a third DDN concurrency expression operator which may be used for describing bounded concurrency situations.

[2] x|y specifies the identifier y which is defined within the definition of the identifier x.

illustrates several DDN constructs for the description of internal attributes. We include this example because it completes our description of the HEARSAY blackboard and indicates that hierarchical descriptions may be constructed by using the external attributes of a set of components to define the internal attributes of another component.

Before commenting on the constructs for describing internal attributes, two comments are appropriate about the textual units in figure 8 which are concerned with the definition of external attributes. First, in the model for the *manager* array of control processes, note the use of nondeterminism in the set-to statement. DDN provides a variety of nondeterministic constructs since such constructs provide a convenient vocabulary for the task of abstraction. Second, in the EVENT DEFINITION textual unit, note the ability to reference events outside of the class definition in order to give an outward-directed specification of behavior as well as the ability to reference internal events in order to give an inward-directed behavior specification.

```
[db_op]:  MONITOR CLASS;
  STATE SUBSETS; read, write, initialize END STATE SUBSETS;
  END MONITOR CLASS;
[blackboard]:  SUBSYSTEM CLASS;
  QUALIFIERS;  #_ks, #_under_surveillance  END QUALIFIERS;
  SUBCOMPONENTS;  reg ARRAY[1::2] OF [region(MY_INDEX)],
                  spy OF [noticer(#_under_surveillance)]
    END SUBCOMPONENTS;
  request:  ARRAY[1::# ks] OF IN PORT;
    BUFFER SUBCOMPONENTS;   op OF [db_op],
                            id OF [entry_id],
                            val_to_write OF [datum]
      END BUFFER SUBCOMPONENTS;
    BUFFER CONDITIONS;
      op=read OR op=write
      END BUFFER CONDITIONS;
    END IN PORT;
  read response:  ARRAY[1::# ks] OF OUT PORT;
    BUFFER SUBCOMPONENTS; val OF [datum]  END BUFFER SUBCOMPONENTS;
    END OUT PORT;
  note:  ARRAY[1::# under_surveillance]  OF OUT PORT;
    BUFFER SUBCOMPONENTS;  notice OF [datum_change]
      END BUFFER SUBCOMPONENTS;
    BUFFER CONDITION;  notice=change  END BUFFER CONDITIONS;
    END OUT PORT;
  manager:  ARRAY[1::# ks]  OF CONTROL PROCESS;
    MODEL;  ITERATE   RECEIVE request(MY_INDEX);
                      IF op(MY_INDEX)=read
                        THEN
          gotread:           SET val(MY_INDEX) TO defined OR errorflag;
          answer:            SEND read_response(MY_INDEX);
                        ELSE
          gotwrite:          NULL;
                        END IF;
              END ITERATE;
      END MODEL;
    LOCAL SUBCOMPONENTS; region_number OF [region_id(2)]
      END LOCAL SUBCOMPONENTS;
    BODY;   ITERATE  RECEIVE request(MY_INDEX);
                     region_number.determine(id(MY_INDEX));
                     IF op(MY_INDEX)=write
                       THEN reg(region_number).write(val_to_write(MY_INDEX));
                       ELSE reg(region_number).read(val(MY_INDEX));
                            SEND read_response(MY_INDEX);
                       END IF;
              END ITERATE;
      END BODY;
    END CONTROL PROCESS;

  EVENT DEFINITION;
    notice: DESCRIPTION;passage of a message out through
                        one of the note ports
                        END DESCRIPTION;
    noticed_write:
      SEQUENCE(OR(reg(1)|write, reg(2)|write), spy|see it),
    some_request:
      OR(gotread, SEQUENCE(gotwrite,notice)),
    serviced_write:
      SEQUENCE([ks]|request_write, gotwrite, notice)
    END EVENT DEFINITION;
  DESIRED BEHAVIOR;
    POSSIBLY CONCURRENT(noticed_write, [blackboard]|noticed_write),
    POSSIBLY CONCURRENT(some_request, [blackboard]|some_request),
    POSSIBLY CONCURRENT(serviced_write, [blackboard]|serviced_write)
    END DESIRED BEHAVIOR;
  END SUBSYSTEM CLASS;
```

Figure 8

With respect to the specification of internal attributes, in figure 8 there are several major components declared for each instance of the class. Three are declared in a straightforward way within the SUBCOMPONENTS textual unit -- the declarations specify the names of the components and their types. (The use of *MY_INDEX* within the declaration of the *reg* array of components indicates that *reg(1)* is of class [region(1)] and that *reg(2)* is of class [region(2)], where the arguments in the class reference specify values for the qualifiers in the class' definition.) The other major components are declared, without reference to a previously defined class, by giving a BODY textual unit for the *manager* array of control processes. This textual unit specifies the algorithm that is to be performed by the control process during subsystem operation. While it is not always the case, in this example the control processes map one-to-one onto components which control the message flow through the ports and the activation of the other components. When this is the case, it is convenient to describe the components directly as the bodies of the control processes.

The DDN facilities for hierarchical description of internal attributes are discussed more fully in [5]. The major point to be noted here is that the description of the class [blackboard] is redundant, with the description of the componentry within a class of subsystems procedurally indicating how the subsystems operate and the rest of the description providing a basically nonprocedural specification of behavior.

The Use of Behavior Specifications in Software Design Analysis

The DDN constructs for describing a software system design present numerous possibilities for analysis of a design at various stages in its evolution. The DDN behavioral specification constructs discussed here are particularly valuable as a basis for simulation and formal or informal consistency checking approaches to design verification.

The transitions and control process models of a DDN software system description provide the basis for simulation of a system at any stage in the design effort. Following the approach described in [18], expected values for timing data and random variables controlling nondeterministic processing can be used to augment the control process model specifications, permitting performance

statistics to be derived for the system as it is described at any stage in its development. Such simulations can uncover potential performance problems at an early point in the design process, rather than allowing their discovery to await a completed system implementation.

The DREAM behavioral specification technique allows for both formal and informal analysis of a software system design based on consistency checking. Informal arguments for correctness of a design may be made by showing the consistency of the various redundant specifications for the designed system. Such arguments may compare internal specifications, such as found in control process bodies, with external specifications, such as given by control process models or stated in the DESIRED BEHAVIOR textual units. Alternatively, informal arguments may demonstrate the consistency among different behavioral specifications, such as control process models and DESIRED BEHAVIOR textual units or possibly the DESIRED BEHAVIOR textual units from several DDN class definitions. Such arguments are clearly not, in and of themselves, sufficient to prove the correctness of a design. However, by helping a designer to consider the proposed design from various viewpoints, they can be very helpful in revealing errors or, when several different descriptions prove consistent, in increasing confidence in the design's correctness.

Formal approaches to the analysis of DDN software system designs can utilize essentially this same type of consistency checking between different descriptions of the system's intended behavior. These formal methods, slated for inclusion in future versions of the DREAM system, are based upon the automatic derivation, using an algorithm similar to that defined in [19], of an event sequence expression describing the behavior represented by each control process model. The behaviors represented by these expressions may then be compared with those specified in desired behavior textual units to determine whether the system as designed would conform to the designer's intentions as expressed in the desired behavior information. In general, this comparison cannot be conducted algorithmically [19], but cases can be found [20] in which algorithmic manipulation is tractable. Alternatively, the derived expressions may be presented to the software system designer, whose knowledge and insight may facilitate the comparison with desired behavior specifications, or who may simply find this alternative representation of the control process' behavior informative. This latter approach is an example of the "feedback analysis" [21] capabilities made possible by the DDN software design language, with information concerning the characteristics of the system under design being presented to the designer who uses human insight to check the correctness of the design.

Conclusions

The DDN design description facilities offer a medium for presenting well-defined, abstract specifications of software system components, describing their behavior without revealing or requiring details of their internal operation. Control process MODEL, procedure TRANSITIONS, EVENT DEFINITION and DESIRED BEHAVIOR textual units can all serve as possibly redundant, abstract behavioral specifications, orthogonal to the operational descriptions of the implementations of the subcomponents. Such specifications are valuable at early stages of design, when they allow for a high-level, non-detailed description of the evolving software system. In later phases of the design effort they are ideally suited for use as external specifications for component behavior, fulfilling an information-hiding function while presenting a complete and formal component description.

We have used DDN to prepare abstract descriptions of several software systems, both hypothetical and real. We have found that DDN assists in the rigorous and formal description of a complex software system design and provides a basis for some analysis of that design at various stages of the design effort. We expect that the form and syntax of DDN will change as we use it in trial design efforts. We also expect that the development of specific analysis algorithms will further shape the language. We have placed the most emphasis upon the naturalness of the language for the task of modelling and feel that we have identified several concepts which are important to carrying out that task.

Acknowledgements

We feel ourselves fortunate to have been able to use the TOPD System ([4], [9]) over the last two years. TOPD was developed as a tool to aid program implementation, but using it for program design tasks greatly facilitated our development of the DDN language and the DREAM System. We would also like to thank Derry Kabcenell and Mark Welter for their contributions to the development of the DREAM System and Jan Cuny, Victor Lesser, Carolyn Steinhaus and Pamela Zave for asking the right questions at the right time.

References

[1] W. E. Riddle, J. Sayler, A. Segal and J. Wileden. An Introduction to the DREAM Software Design System. Software Engineering Notes, 2, 4 (July 1977), 11-23

[2] J. J. Horning and B. Randell. Process Structuring. Comp. Surveys, 5, 1 (March 1973), 5-30.

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

[4] P. Henderson, et al. The TOPD System. Tech. Report 77, Computing Laboratory, Univ. of Newcastle upon Tyne, England, September 1975.

[5] W. E. Riddle. Hierarchical Description of Software System Structure. RSSM/40, Department of Computer Science, University of Colorado at Boulder, Boulder, October 1977.

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

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

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

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

[10] D. L. Parnas. A Technique for Software Module Specification with Examples. Comm. ACM, 15, 5 (May 1972), 330-336.

[11] 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 Calif., 1976.

[12] Barbara Liskov, et al. Abstraction Mechanisms in CLU. Comm. ACM, 20, 8 (August 1977), 564-576.

[13] N. Wirth. The Programming Language PASCAL. Acta Informatica, 1, (1971), 35-63.

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

[15] W. E. Riddle. The Hierarchical Modelling of Operating System Structure and Behavior. Proc. ACM 72 National Conf., August 1972.

[16] W. E. Riddle. Abstract Process Types. RSSM/42, Dept. of Computer Science, Univ. of Colorado at Boulder, Boulder, November 1977.

[17] J Wileden. Behavior Specification in a Software Design System. RSSM/43, Dept. of Computer and Communication Sciences, Univ. of Michigan, Ann Arbor, in preparation.

[18] J. W. Sanguinetti. Performance Prediction in an Operating System Design Methodology. RSSM/32, Dept. of Computer and Communication Sciences, Univ. of Michigan, Ann Arbor, May 1977.

[19] W. E. Riddle. An Approach to Software System Modelling, Behavior Specification and Analysis. RSSM/25, Dept. of Computer and Communication Sciences, University of Michigan, Ann Arbor, July 1976.

[20] M. Welter. Counter Expressions. RSSM/24, Dept. of Computer and Communication Sciences, University of Michigan, Ann Arbor, October 1976.

[21] W. E. Riddle. A Formalism for the Comparison of Software Analysis Techniques. RSSM/29, Dept. of Computer and Communication Sciences, Univ. of Mich., Ann Arbor, July 1977.