Behavior Specification in a Software Design System*

Jack C. Wileden
University of Massachusetts

John H. Sayler
University of Michigan

William E. Riddle
software design & analysis, inc.

Alan R. Segal
NBI, Inc.

Allan M. Stavely
New Mexico Institute of Mining and Technology

A technique for software system behavior specification appropriate for use in designing systems with concurrency is presented. The technique is based upon a generalized ability to define **events,** or significant occurrences in a software system, and then indicate whatever constraints the designer might wish to see imposed upon the ordering or simultaneity of those events. Constructs implementing this technique in the DREAM software design system are presented and illustrated. The relationship of this technique to other behavior specification techniques is also discussed.

INTRODUCTION

Any piece of computer software, coded in some programming language, implicitly specifies the behavior which it will produce when run on a given computing system with a given set of input data. Such a **program-defined** behavior specification, while definitive and theoretically sufficient, is inadequate for most practical problems in software design and analysis, so alternative approaches to software behavior specification have been sought. For example, in their work on formal methods for program verification, Floyd [2], Manna [4], and others have employed relations between expressions in the predicate calculus, called **input-output relations,** as specifications for a program's intended behavior.

Specifying the behavior of a software system is an especially significant and difficult chore when that behavior involves either conceptually or actually concurrent operation. Riddle's event expressions [6], Campbell and Habermann's path expressions [1], Greif's problem specification language [3], and Shaw's flow expressions [15] all represent efforts to define behavior specification schemes for concurrent software systems. Appropriately tailored, such a specification scheme could prove very valuable to designers of complex, concurrent software systems, by providing a mechanism for describing the intended behavior of a proposed software system design long before it is actually implemented. A description of this sort might then be used in guiding a subsequent coding effort, in assessing the resulting software or, perhaps most importantly, in analyzing the proposed design itself for suitability and correctness.

Program-defined behavior specifications are particularly ill-suited for use in designing software systems.

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

Address correspondence to Professor Jack C. Wileden, Computer and Information Science Department, University of Massachusetts, Amherst, MA 01003.

A software system designer generally wishes to specify the behavior of the system being designed as a **prescription** for the eventual program code, a function for which program-defined behavior specification is clearly insufficient. Moreover, a behavioral specification technique useful in designing software should ideally permit a rigorous and formal statement of intended system behavior which can function both as a comprehensible **description** of the software system's possible activity and as a basis for some **analysis** of the design as it is being developed. Its roles in both description and analysis demand a technique capable of **projection,** or the focusing of the description upon certain aspects of the system's behavior. Without such focusing, a system's description tends to be a monolithic description expressing every facet of total system behavior. Projection provides the ability to highlight the **interesting** behavior, at any level of detail and for any number of definitions of "interesting," from out of the overwhelming and often largely irrelevant detail of total system behavior. This ability to highlight various particular aspects of system behavior is essential to both understandable descriptions and tractable analysis, and is not readily available in a program-defined specification.

A behavioral specification's dual roles in description and analysis also require a technique which yields **redundant** specifications, providing a description of the system's behavior which is **orthogonal**[1] to that given by the procedural specification of the program text. This redundancy and orthogonality are especially significant to analysis, since they make possible the comparison of two dissimilar specifications of system behavior. It is clearly impossible for a program-defined behavior specification to possess such redundancy and orthogonality.

This paper presents a set of constructs for behavior specification developed specifically for use by software system designers, particularly those creating systems with concurrently operating subparts. As befits a design tool, the constructs are descriptive rather than prescriptive in nature, i.e., they serve to indicate the designer's intentions for system behavior but do not in any way enforce those intentions as programming language constructs (e.g., path expressions) might. Our behavioral specification technique is based upon a generalized ability to define **events,** or significant occurrences in a software system, and then indicate whatever constraints the designer might wish to see imposed upon the ordering or simultaneity of those events. This technique provides the designer with a formal medium for the rigorous statement of modular specifications for various levels of system behavior. Its constructs facilitate projection and allow the designer to provide redundant and orthogonal specifications of interesting aspects of system behavior. Thus these constructs are useful in both the description and analysis phases of the software system design effort.

The constructs discussed here are part of the design language developed for the Design Realization, Evaluation and Modelling (DREAM) system [7, 10, 11, 12, 18]. DREAM is a tool for the design of large scale software systems, providing the designer with bookkeeping and analysis aid. A description in the DREAM Design Notation (DDN) consists of a collection of (nested) description fragments, called **textual units.** DREAM provides facilities which allow the user (or users, in the case of a design being carried out by a design team) to modify the information in a data base on a textual unit basis.

The focus of this paper is upon the DDN constructs for software system behavior specification; other aspects of DDN are discussed in [8, 9, 13]. Certain parts of DDN which are not primarily for behavior specification are treated minimally here, with references indicating where more complete discussions may be found. Also, we focus here upon the use of the constructs. Some justifications for the constructs are given in this paper, while others lie within the DREAM system's general philosophy which is discussed in [12] and [14].

TYPES OF EVENTS IN DREAM

The notion of an event, i.e., a significant occurrence in a software system, is central to behavior specification in DREAM. We distinguish two broad types of events in DREAM, which we call **endogenous** and **exogenous** events, and also differentiate between **primitive** and **complex** events.

With respect to a DREAM description of a software system design, an endogenous event is one which arises from some activity of the system as it is currently described. Thus, the execution of some procedure or a change of state in some data structure described in the present DDN description of the software system being designed could represent an endogenous event. We call events such as these, which arise as the result of a single (arbitrarily lengthy or complicated) activity within the system, primitive endogenous events.

Not all occurrences which are significant to a software system may be represented in terms of some activity of the system as it is described at a particular point in the evolution of its design. An exogenous event

[1]An orthogonal description is one forming associations among the elements of the system which may be completely different from those found in the system's implementation. An orthogonal description may, therefore, express a logical rather than a physical system organization.

is one which arises as the result of some activity outside the scope of the system's current description but which is relevant to or impinges upon the system as presently described. Thus the execution of some procedure or a change of state in some data structure envisioned for, but not yet described in the DDN description of, the software system being designed could correspond to a primitive exogenous event. Normally an exogenous event of this nature would eventually be supplanted by an endogenous event as further elaboration of the design led to the development of a DDN description for the envisioned component. Alternatively, a primitive exogenous event may arise as the result of an activity completely beyond the scope of the system being designed. Examples of such inherently exogenous events include a procedure execution or data structure state change in some existing piece of software which will interact with that being designed or the recording of some physical event (e.g., change in patient pulse rate or entry of information from an interactive terminal) by a sensor or monitor device. An exogenous event of this nature will remain exogenous in the completed design of the software system, allowing for the description of occurrences relevant but external to the system.

A designer might also wish to regard some sequence of events as itself constituting a significant occurrence. Thus DREAM allows for the definition of complex events, which are events consisting of a sequence of other events. The constituent events in a complex event may be either endogenous or exogenous events and may themselves be either primitive or complex events.[2] Although the distinction is seldom significant, we may call a complex event endogenous if all its constituent events are endogenous and exogenous if all its constituent events are exogenous.

PRIMITIVE ENDOGENOUS EVENT SPECIFICATION IN DREAM

In DREAM, a software system is considered to be decomposable into **objects,** which are themselves decomposable into subobjects, etc. [8]. Each object is an instance of some general **class** of objects and has the attributes specified in the **class definition** for that class. These attributes are each specified by a textual unit. Primitive endogenous events may be defined in DREAM by attaching labels, called **event identifiers,** to various parts of textual units within a class definition.

In DDN, a data object is described by a **monitor class** textual unit [13]. Additional textual units may be nested within the monitor class definition, describing the data objects of the class in terms of state variables (which define a state space), state subsets (which divide the state space into not necessarily disjoint subsets), and state transitions (which describe the effect of an operation in terms of the state changes it causes). Any operations which may be performed upon data objects of the class are defined as procedures by textual units also nested within the monitor class. Event identifiers may be attached to state transitions, procedures, or procedure statements in order to designate the corresponding activity as an event.

As a simple example of primitive endogenous event specification in a monitor class, consider the DDN fragment shown in Figure 1. Within the procedure *get_first,* an execution of which is itself a primitive endogenous event (since procedure names are automatically primitive event identifiers), two transitions have been labeled. The transition associated with the event identifier *cannot* corresponds to the event resulting from an attempt to retrieve an item from an empty list, as indicated by the appearance of the state subset *empty* on the left-hand side of the transition arrow. The event identifier *delete* corresponds to the event resulting from a successful retrieval of an item from the list. Similarly, the event identifier *insert* on the transition within the *put_last* procedure corresponds to the event resulting from the addition of an item to the list.

In the procedure *put_and_get* of the Figure 1 example, both transitions and statements have been labeled. The two transitions represent the two possible cases of performing the *put_and_get* operation on an initially nonempty or on an initially empty list, respectively. The labeled statements in the procedure body permit the designation of either of the two constituent steps in the *put_and_get* operation as an event.[3]

The active components of a software system are represented in DREAM **subsystem** classes, which are viewed as describing collections of potentially concurrent, asynchronous, sequential processes. Subsystem classes have no states and hence no transitions. Rather, their activity is described in terms of **control processes,** which have both **models** and **bodies** to describe their behavior. The distinction between models and bodies is not relevant to the present paper; a detailed discussion of subsystem classes may be found in [9,10]. For our present purposes, it suffices to note that in DDN both

[2]Event definitions may not, however, be recursive, i.e., no complex event may contain itself as a constituent event.

[3]DDN syntax requires that a procedure be applied to an instance of the class in which it is declared. The reserved word ME simply allows a procedure to invoke other procedures defined within the same class, thereby causing them to operate upon the same instance as the invoking procedure.

```
[ready_list]: MONITOR CLASS;
   DOCUMENTATION;
     A data structure shared by the schedulers for
     the various processors in a multiprocessor system.
     Composed of the task control blocks (tcb's) for
     all active tasks in the system.
     END DOCUMENTATION;
   STATE SUBSETS;
     full, empty, other
     END STATE SUBSETS;

   get_first: PROCEDURE;
     PARAMETERS;
       first_tcb RESULT OF [tcb],
       one_there RESULT OF [logical]
       END PARAMETERS;
     TRANSITIONS;
cannot: empty ->
            one_there=false AND first_tcb=undefined,
delete: other OR full ->
            one_there=true AND first_tcb=defined
               AND (empty OR other)
       END TRANSITIONS;
     END PROCEDURE;

   put_last: PROCEDURE;
     PARAMETERS;
       the_tcb VALUE OF [tcb]
       END PARAMETERS;
     TRANSITIONS;
insert: (empty OR other) AND the_tcb=defined ->
            other OR full
       END TRANSITIONS;
     END PROCEDURE;

   put_and_get: PROCEDURE;
     PARAMETERS;
       the_tcb VALUE OF [tcb],
       first_tcb RESULT OF [tcb],
       ok RESULT OF [logical]
       END PARAMETERS;
     TRANSITIONS;
pg1:  other AND the_tcb=defined ->
        other AND first_tcb=defined,
pg2:  empty AND the_tcb=defined ->
        empty AND first_tcb=defined
      END TRANSITIONS;
   BODY;
     put: ME.put_last(the_tcb);
     get: ME.get_first(first_tcb,ok);
       END BODY;
     END PROCEDURE;
   END MONITOR CLASS;
```

Figure 1.

models and bodies consist of statements. Labeling of the statements in the model or body of a control process may be employed to designate the corresponding activities as primitive endogenous events within a subsystem class definition. However, since the labeling of control process statements is, for our purposes here, not significantly different from the labeling of monitor class procedures and the statements within them, we will confine our discussion and examples of endogenous event specification to monitor classes for the remainder of this paper.

PRIMITIVE EXOGENOUS EVENT SPECIFICATION IN DREAM

Unlike endogenous events, definitions of which may appropriately 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. Therefore, a third class type, the **event** class, has been developed in DREAM for use in the definition of exogenous events.

The DDN event class is intended solely for the specification of exogenous events. Aside from the **documentation** textual units which may appear anywhere in a DDN description, only three types of textual units defining its attributes may be nested within an event class. A **qualifiers** textual unit allows for the parameterization of the class, as it does for monitor and subsystem classes [8]. A **desired behavior** textual unit may also appear within an event class; desired behavior textual units are discussed in a later section of this paper. The actual definition of primitive exogenous events occurs within an **event definition** textual unit inside the event class. The events are defined by giving a labeled prose **description** of each primitive exogenous event which is defined for the event class.

As an example of primitive exogenous event specification in DREAM, consider the DDN fragment shown in Figure 2. It, like the previous example, is taken from a DDN design description for the scheduler subcomponent of a multiprocessor operating system [16]. Since the scheduler, in selecting a task to run on an available processor, must manipulate the shared ready list data structure (Fig. 1) in a mutually exclusive fashion, this event class was developed to describe an event, associated with the event identifier *scan_ready_list,* whose occurrence is potentially significant to the scheduler's execution.

EVENT SEQUENCE EXPRESSIONS

Given a vocabulary of event identifiers provided by the specification of primitive endogenous and exogenous events in the manner discussed above, a software system designer may for various reasons wish to represent particular combinations of those events. Such a representation is accomplished in DREAM through the use of **event sequence expressions.**

An event sequence expression is a DDN representation for a set of sequences of events in the same sense that a regular expression is a representation for a set of sequences of symbols which constitutes a regular language. The DDN representation is syntactically functional, consisting of a set of event sequence function operators whose operands may be either event identifiers or event sequence subexpressions. In the remainder of this section, we present an informal, intuitive discussion of the formation and interpretation of event sequence expressions. Examples of their uses in DREAM will be found in subsequent sections of this paper, while a few technical points regarding their interpretation are deferred to Appendix A.

The simplest form of an event sequence expression is that consisting of a single event identifier. Such an expression represents the singleton set of event sequences consisting of just the named event. In addition to designer-defined event identifiers, the DDN reserved word ANY may appear as a simple event sequence expression, representing the set of all possible event sequences over all event identifiers defined within the current DDN design description. Thus, within the event class of Figure 2, either of the following could be an event sequence expression:

```
scan_ready_list
ANY
```

The former represents the single event sequence consisting of one occurrence of the event *scan_ready_list.* The latter represents the set of all possible sequences of defined events. If the [scan] event class[4] happened to be the complete current DDN design description, then ANY would denote the set of event sequences consisting of zero or more occurrences of the event *scan_ready_list.*

More complicated event sequences may be represented in DDN through the use of the various event sequence function operators. Three of these, REPEAT, SEQUENCE, and OR, are exactly analogous to the standard regular expression operators. REPEAT(x) represents the set of sequences consisting of zero or more occurrences of x. SEQUENCE(x,y) represents the set consisting of the event sequence "x followed by y." OR(x,y) represents a two-element set of sequences,

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

one of which is a single occurrence of x and the other of which is a single occurrence of y.[5] Within the event class of Figure 2, any of the following would be possible event sequence expressions:

```
REPEAT(scan_ready_list)
REPEAT(SEQUENCE(scan_ready_list,scan_ready_list))
OR(scan_ready_list,REPEAT(SEQUENCE(scan_ready_list,scan_ready_list)))
SEQUENCE(scan_ready_list,REPEAT(scan_ready_list))
```

The first expression denotes the set of event sequences consisting of zero or more occurrences of the event *scan_ready_list*. Thus it is synonymous with ANY in the case where the [scan] event class is the entire current DDN design description. The second expression represents the set consisting of all even-length sequences of the event *scan_ready_list* (including the zero length sequence), while the third represents that set augmented by the addition of the sequence consisting of a single occurrence of *scan_ready_list*. The fourth expression represents the set of sequences consisting of one or more occurrences of the event *scan_ready_list*.

Two of the DDN event sequence function operators provide the software designer with a means of expressing a qualified "don't care" in an event sequence expression. Whereas the DDN reserved word ANY

[5]Although presented here as binary operators, both SEQUENCE and OR may have any number of operands. REPEAT, however, is a unary operator.

Figure 2.

```
[scan]: EVENT CLASS;
  DOCUMENTATION;
    In anticipation of eventual elaboration of the operator
    interaction functions of the multiprocessor system, an
    event relevant to the scheduler is defined.
    Later elaboration will no doubt convert the exogenous
    event defined here to an endogenous event.
    However, the desired behavior of this event relative to
    the scheduler might well be forgotten by that time.
    This event class serves to record that desired behavior
    while it is fresh in the designer's mind.
    END DOCUMENTATION;


  EVENT DEFINITION;

  scan_ready_list: DESCRIPTION;
    scan_ready_list is an event which occurs when an operator
    wishes to examine the current state of tasks in the system.
    It may well entail the consecutive removal and replacement
    of entries on the ready_list, or could involve selective
    removal and replacement, or might make no actual alterations
    to the list at all.  Its behavior is not yet specified in
    detail.  However, the manipulation of the ready_list must
    be exclusive of the critical sections of objects of class
    [scheduler], hence this event class.
    END DESCRIPTION;

    END EVENT DEFINITION;

  END EVENT CLASS;
```

represents all possible event sequences over all event identifiers defined within the current DDN design description, the function ANY(x,y, . . .) represents the set of all possible sequences of the event identifiers which are its operands.[6] Such a representation is useful when sequences of any number and ordering of a particular set of events are equivalent from the designer's viewpoint. For example, to indicate that an acceptable usage of a file requires that it first be opened, then read and/or written any number of times and finally closed, a designer might use the event sequence expression:

```
SEQUENCE(open,ANY(read,write),close)
```

to represent the set of acceptable event sequences.

The function ANYBUT(x,y, . . .) is the complement to the ANY(x,y, . . .) function. This function represents the set of all possible event sequences over all event identifiers defined within the current DDN design description, except for those which are its operands. This representation is useful when sequences consisting of any number and ordering of all but a few events are equivalent for the designer's purposes. For instance, in the [ready list] example (Fig. 1), the designer might indicate the set of event sequences which could take place during a time period when the list was never empty by using the event sequence expression:

```
ANYBUT(cannot,pg2)
```

representing the set of all sequences containing any number and ordering of occurrences of the events *get_first, put_last, put_and_get, delete, insert, pg1, put,* and *get,* along with any events defined elsewhere in the current design description.

The two remaining event sequence function operators, SHUFFLE and REENTRANT, are used to represent the interleaved activity resulting from concurrent activity within the software system. SHUFFLE, which may take any number of operands, represents the set of all sequences resulting from the arbitrary interleaving of the sets of sequences represented by its operands. For example, the event sequence expression:

```
SHUFFLE(SEQUENCE(x,y),REPEAT(OR(a,b)))
```

represents the set of sequences:

```
{xy,xya,xay,axy,xyb,xby,bxy,xyaa,xaya,
axya,xaay,axay,aaxy,xybb,xbyb,bxyb,xbby,
bxby,bbxy,xyab,xayb,axyb,xaby,axby,abxy,
. . .}
```

that is, all sequences consisting of one x, one y, and any number of a's and b's, in which the x precedes the y. If a software system designer wished to describe the behavior of a system consisting of two concurrent, asynchronous processes, one of which operated in such a way as to cause first an x event and then a y event, while the other's operation could lead to any number of a and b events occurring in any order, the event sequence expression given above would provide exactly the desired description.

The REENTRANT operator, which is unary, represents the set of sequences resulting from applying the SHUFFLE operator to its operand any number of times. For example, the event sequence expression:

```
REENTRANT(SEQUENCE(a,b))
```

represents the set of sequences:

```
{null, ab, aabb, abab, aaabbb, aababb,
                      aabbab, ababab, . . .}
```

that is, all sequences containing an equal number (possibly zero) of a's and b's such that, at any point within any sequence the number of a's to the left of that point is not less than the number of b's to the left of that point. If the event a is interpreted as "process begins execution" while the event b is interpreted as "process completes execution," the event sequence expression given above exactly represents the set of possible behaviors for that process' reentrant execution or, equivalently, for an (arbitrarily large) set of identical, asynchronous, concurrent processes.

There is one other DDN construct relevant to event sequence expressions. A **unique event identifier,** represented in DDN by an integer expression enclosed within <>, may appear anywhere that an event identifier may appear within an event sequence expression. Unique event identifiers are intended to restrict the interleaving of event sequences and thus to represent some synchronization of activities within a concurrently operating software system. A unique event identifier corresponds to a null event, but within an event sequence expression all unique event identifiers having equal-valued integer expressions represent the same time point. Therefore, no interleaving may result in a sequence in which a non-null event separates any pair of equal-valued unique event identifiers. For example, the event sequence expression:

```
SHUFFLE(SEQUENCE(a,b,<1>,c),SEQUENCE(d,
                                   <1>,e))
```

represents the set of sequences:

```
{abdce,adbce,dabce,abdec,adbec,dabec}
```

Sequences such as adebc violate the restriction on unique event identifiers, since the event e would appear

[6]Note that this set could equivalently be represented by the expression REPEAT(OR(x,y, . . .)). Note also that the event expression ANY is thus synonymous with the event expression ANY (x,y,z, . . .) in which all event identifiers defined within the current DDN design description appear as operands of the function operator.

between the <1> following the event d and the <1> following the event b, and are therefore not represented by this event sequence expression. If a designer wished to describe the behavior of two concurrent processes in which the b and d events represented production activities for which the e and c events represented the respective consumption activities, with production necessarily preceding consumption, then the event sequence expression given above would provide precisely the desired description.

COMPLEX EVENT SPECIFICATION IN DREAM

In addition to its use in the definition of primitive exogenous events, the event definition textual unit may be used within a monitor, subsystem or event class definition for the specification of complex events. Within an event definition textual unit, a complex event may be specified in terms of a sequence of state subsets of a monitor class, or in terms of a sequence of statements in a procedure body or in a control process body or model. Event sequence expressions may also be used in specifying a complex event.

The DDN function operator STATE SEQUENCE is used to define a complex event corresponding to the passing of a monitor class object through a particular sequence of state subsets. Labeled with the event identifier chosen for this complex event, the STATE SEQUENCE operator defines the event corresponding to the consecutive occurrence of the state subsets named by its operands. For example, the following event definition textual unit might be added to the monitor class definition of Figure 1:

```
EVENT DEFINITION:
    first_item: STATE SEQUENCE(empty,other),
    list_refilled: STATE SEQUENCE(full,other,full),
    list_refilled: STATE SEQUENCE(full,empty,full)
    list_refilled: STATE SEQUENCE(full,other,empty,other,full)
    END EVENT DEFINITION;
```

defining two new complex events in terms of state subset sequences of the monitor class [ready list].[7]

The DDN reserved word THROUGH (alternatively THRU) may be used to join two event identifiers which are statement labels in order to define a complex event in terms of a sequence of procedure body, control process body or control process model statements. Labeled with the even identifier chosen for this complex event, the THROUGH construction specifies an event corresponding to a sequence of statement executions beginning with the statement bearing the first label and ending with the statement bearing the second label. For example, another event definition textual unit which might be added to the [ready list] monitor class (Fig. 1) is the following:

```
EVENT DEFINITION;
    p_g_execution:  put THRU get
    END EVENT DEFINITION;
```

Although in this instance the new complex event *p_g_execution* is equivalent to the event defined by the procedure label *put_and_get,* in general this construct provides the software system designer with a useful means for specifying complex events.

Event sequence expressions, labeled with an event identifier selected for the complex event they define, provide the final mechanism for complex event specification within an event definition textual unit. For example, the following could be added to the event definition textual unit of the [scan] event class (Fig. 2):

```
one_scan: scan_ready_list,
many_scans: REPEAT(scan_ready_list),
even: REPEAT(SEQUENCE(SCAN_ready_list,scan_ready_list)),
one_or_even: OR(scan_ready_list,even)
```

Note that the complex event *one_scan* defined here is identical to the primitive event *scan_ready_list* defined previously; this ability to provide several synonymous identifiers for events is sometimes convenient for a software system designer. Notice also that complex events defined in the event definition textual unit may themselves appear in event sequence expression definitions of other complex events; the *one_or_even* event definition given here is a compact version of an event sequence expression presented as an example in the previous section.

[7]Notice the use of three different definitions for the complex event *list refilled.* In DDN, multiple definitions corresponding to the same event identifier indicate that the occurrence of any one of them constitutes an occurrence of the named event. Thus, an occurrence of any of the state sequences "full, other, full," "full, empty, full," or "full, other, empty, other, full" would constitute an occurrence of the event *list_refilled.*

DESIRED BEHAVIOR SPECIFICATION IN DREAM

Having considered the DDN mechanisms for event specification, we now describe the facilities provided by the DREAM system for indicating the constraints which a designer might wish to see imposed upon the behavior of the software system being designed. Such constraints are expressed by including **desired behavior** textual units within the DDN description of the system. Within each such textual unit, a list of **concurrency expressions** and/or event sequence expressions is used to specify the various behavior constraints to which the designer wishes the system to conform.

Concurrency expressions are formed using either the MUTUALLY EXCLUSIVE function operator or the POSSIBLY CONCURRENT function operator. A variant of the latter operator which allows an integer expression to appear between the words POSSIBLY and CONCURRENT may be used to express bounded concurrency. The operators for concurrency expressions are all binary, their operands being sets of events.[8] 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. 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.

An example of the use of concurrency expressions for desired behavior specification is the following textual unit, which might be added to the [scan] event class (Fig. 2):

```
DESIRED BEHAVIOR;
    POSSIBLY 2 CONCURRENT(scan_ready_list, [scan]|scan_ready_list),
    MUTUALLY EXCLUSIVE(scan_ready_list,OR( [scheduler]|critical_1, [scheduler]|critical_2)),
    END DESIRED BEHAVIOR;
```

The above textual unit indicates that the designer wishes to allow a maximum of two concurrent occurrences of the event *scan_ready_list,* presumably due to some limitation such as a maximum number of authorized operators. It also indicates that the event *scan_ready_list* must not be permitted to occur during either of two events associated with the (as yet undefined) [scheduler] class, each presumably involving the scheduler's accessing of the ready list data structure.

Event sequence expressions provide a flexible mechanism for specifying constraints on a software system's behavior. Each event sequence expression appearing within a desired behavior textual unit is normally interpreted as a partial[9] specification for the set of acceptable or desirable sequences of occurrences of the events whose identifiers appear within that event sequence expression. Events whose identifiers do not appear within the event sequence expression are not, however, generally constrained by that expression. For example, if the following event sequence expression appeared within a desired behavior textual unit:

```
REENTRANT(SEQUENCE(a,b))
```

it would be interpreted as indicating that no sequence of event occurrences in the system being designed could at any point in time include more instances of event b than it did event a. However, this expression would not be interpreted as restricting what other (non-a and non-b) events could occur between instances of the a and b events, nor as prohibiting the occurrence of any other (non-a and non-b) events. Thus, both of the following event sequences:

```
aababb  and  axbabzayb
```

would be interpreted as acceptable with respect to this event sequence expression specification of desired behavior.

Certain properties of the interpretation of event sequence expressions permit their use as partial desired behavior specifications in the manner described above. We informally describe some of those properties here; a somewhat more complete and technical discussion is found in Appendix A.

The use of the ANY(x,y, . . .) and ANYBUT(x,y, . . .) functions in event sequence expressions is the key to the interpretation of these expressions in desired behavior specifications. While a given expression does not in general imply any constraints on the occurrences of events not explicitly mentioned within that expression, this aspect of its meaning can be altered through use of these two functions. In particular, when ANY(x,y, . . .) appears in an event sequence expression and all of its operands are defined within the same class definition

[8]These sets of events are specified using event sequence expressions. We defer discussion of the details of this representation to Appendix B.

[9]The specification is only partial since other event sequence or concurrency expressions within the desired behavior textual unit may impose additional constraints on the occurrence of some or all of the events named in the event sequence expression. This allows for modular specification of desired behavior, but also introduces the possibility of inconsistent specifications, as mentioned below.

textual unit in which the expression occurs, it denotes all sets of sequences of its operands and of any events defined in any other class definitions. Thus, for example, suppose that the following event sequence expression:

```
SEQUENCE(open,ANY(read,write),close)
```

appeared within a desired behavior textual unit nested inside a class definition wherein the events *open, close, read, write, create,* and *destroy* were defined. This expression would be taken to indicate that between any occurrence of the event *open* and the next subsequent occurrence of the event *close,* no *create* or *destroy* events could occur. The following event sequence:

```
open  read  destroy  close
```

would not then conform to the desired behavior specification given by the event sequence expression, while all of the following sequences:

```
open read read write close
open read x read y write z close
create open read close destroy
```

(where x, y and z are events defined in some other class definition) would conform to the specified desired behavior. Similarly, the appearance of the function ANYBUT(x,y, ...) in an event sequence expression denotes any sequence of events other than those which are its operands. Note that this interpretation implies that, appearing within a desired behavior textual unit, the event sequence expression:

```
SEQUENCE(a,ANYBUT(a,b,c),b,ANYBUT(a,b,c),c)
```

has precisely the same connotation as the event sequence expression:

```
SEQUENCE(a,b,c)
```

Of course, a desired behavior textual unit may contain multiple concurrency and event sequence expressions indicating the intended constraints on various aspects of the system's behavior. It may even happen that no possible event sequence would conform to all the desired behavior specifications within a given design. Such a situation would indicate that the design cannot be realized as it is currently specified, since various aspects of its intended behavior are in conflict. (Recall that the desired behavior specifications in a DREAM design do not in any sense influence the behavior which the design, as specified by bodies, models, and transitions, can exhibit; they are merely indicative of the designer's desires and intentions with respect to the system's behavior.) It would be extremely useful to provide automated checking both of the consistency of the various desired behavior specifications and the potential behavior indicated by the functional and structural (body, model, and transition) specifications. The possibility of providing some such automated checking in the DREAM system remains a topic of research. It is felt, however, that even in the absence of such capabilities, DDN's facilities for rigorously specifying the designer's intentions regarding system behavior, in a manner independent of the system's structural and functional specification, will prove to be most advantageous to the production of correctly operating, reliable software systems.

RELATIONSHIP TO OTHER BEHAVIOR SPECIFICATION TECHNIQUES

The behavior specification technique described in this paper is based upon concepts developed in previous work on event expressions [6], path expressions [1], and constraint expressions [17]. The approach presented here is also related to behavior specification formalisms reported by Shaw [15] and Greif [3].

The most fundamental contribution to our notion of event sequence expressions comes from event expressions, which are the source of the basic set of event sequence expression operators. Event expressions provide a general technique for describing behavior, but they have been used primarily for describing the complete message transmission behavior of software systems formally modeled as collections of concurrent processes. Shaw's flow expressions, aside from notational differences, are extremely similar to event expressions. They have been proposed as "a useful notation that can aid in the design, analysis, and understanding of software sytems" [15]. Greif's problem specification language, in which the sequencing of event occurrences is constrained by a partial ordering defined over those occurrences, is also closely related to event expressions. In essence, the Greif technique implicitly defines the sets of event sequences which are explicitly defined in the event expression and flow expression formalisms.

Campbell and Habermann's work on path expressions suggested a number of the concepts employed in our technique, most notably the specification of behavior in terms of a set of partial behavioral specifications. A significant difference exists between path expressions and DREAM desired behavior specification, however, since path expressions are a programming language construct assumed to enforce the behavior they specify. DREAM desired behavior specifications, as we have noted, do not influence system behavior but merely indicate a designer's intentions for that behavior. Our notation and basic set of operators are also quite different from those used in path expressions, although methods for translating path expressions into event expressions are known [5].

Constraint expressions represent an effort to adapt and particularize event expressions so as to make them suitable for desired behavior specification, assimilating certain ideas from path expressions. As such, they are the most direct precursor to the DREAM technique. The ANY and ANYBUT operators and the projection capabilities of the DREAM approach to behavior specification may be traced to the constraint expressions work.

SUMMARY AND CONCLUSIONS

A technique for software system behavior specification appropriate for use in designing systems with concurrency has been presented. The technique is based upon a generalized ability to define events, or significant occurrences in a software system, and then indicate whatever constraints the designer might wish to see imposed upon the ordering or simultaneity of those events. Constructs implementing this technique in the DREAM software design system have been presented and illustrated. Included among these constructs are event identifiers, event classes, event sequence expressions, and event definition and desired behavior textual units. We have also briefly indicated how this technique is related to constraint expressions, event expressions, flow expressions, Greif's problem specification language, and path expressions.

The behavior specification technique described here possesses the necessary properties to serve as a valuable tool for software system designers. It provides the ability to designate certain selected system occurrences as events, and the ability to specify desired behavior relative to any subset of those events, thus allowing for projection and the focusing of attention upon particular interesting aspects of system behavior. The technique's nonprocedural behavior specification constructs make possible a statement of system behavioral requirements which is redundant and orthogonal to the structural and functional system definition given (in DDN) in terms of procedures and control processes. Thus the technique is valuable as a basis for description and, at least informal approaches to, analysis. Even in the absence of automated analysis methods, the benefits cited here are sufficient to indicate the value of the DREAM behavior specification technique to designers of software systems.

APPENDIX A

We have until now avoided the technical details of event sequence expressions and their interpretation. In this appendix, we provide a brief discussion of some of those details.

Three distinct levels of scope can be defined for an event sequence expression appearing within a given class definition textual unit. The first of these, denoted by R′, encompasses all events defined for any single particular instance of the class defined by the given textual unit. R′ may be viewed as the default scope for event identifiers in an event sequence expression. That is, all event identifiers within the expression which are not qualified[10] with an object name or class identifier refer to events occurring in the same single instance of the class defined by the enclosing class textual unit. An event identifier qualified with the appropriate object name may be used to overcome this default scope and reference an event occurring in a specific object of the given class.

The second level of event sequence expression scope, denoted by R, encompasses all events defined for all objects of the class defined by the given textual unit. The scope of an event identifier qualified with the appropriate class identifier is considered to extend to R.

The final level of scope for event expressions, denoted G, encompasses all events defined for all objects of all classes defined in the current DDN design description. This global scope is never referenced explicitly, but is nevertheless significant to the interpretation of event sequence expressions.

These scope levels are used primarily in assigning a meaning to each occurrence of the function ANY(x,y, . . .) in an event sequence expression. The function always denotes the collection of all event sequences over some set of events. This event set always includes the events which are specifically named as the function's operands, according to the scope rules outlined above. The set may also include other defined events, however, depending upon the scope levels of the function's operands. Specifically, if all of the function's operands have scope R′, i.e., none of the operands is an event identifier qualified with a class identifier, then the set also includes any event occurring in any instance of any class except for the particular instance(s) referred to by the function's operands. However, each class identifier which qualifies one of the function's operands leads to the deletion of all events defined for that class (except those appearing among the operands) from this event set. We express this by the rule:

```
ANY(x,y,z) = ({x,y,z} ∪ (G − Q(x,y,z)))*,
  where Q(x,y,z) = Q(x) ∪ Q(y) ∪ Q(z)
   and Q(w) = R' if w is an unqualified event
                 identifier
              = R' <v> if w is of the form v |eventid
              = R [c ] if w is of the form [c ] |eventid
```

Thus an ANY(x,y, . . .) function connotes sequences over both specifically named events and events whose defining class or instance is *not* specifically referenced among the function's operands. This interpretation is consistent with and may clarify the examples discussed in the paper.

The special event identifier ANY and the function

[10]A DDN qualified identifier [8] is simply an identifier which has been prefixed with either an object name or the class identifier of the class in which it is defined, followed by a "|", e.g., cpu_queue|insert or [ready_list]|insert.

ANYBUT(x,y, . . .) can also be explained in these terms. ANY is simply G*, which is equivalently expressed in the form given in footnote 6 above. The rule for the ANYBUT(x,y, . . .) function is:

```
ANYBUT(x,y,z) = (G − {x,y,z})*
```

This too is consistent with the examples given in the paper.

Finally, the partial behavior specification aspect of event sequence expressions can be described in these terms. The technically correct form for a SEQUENCE function in an event sequence expression is:

```
SEQUENCE(x0,A1,x1,A2, . . .,An,xn)
```

where each Ai is either ANY or one of the functions ANY(x,y, . . .) or ANYBUT(x,y, . . .). This form thus completely specifies the events which are allowed to occur between occurrences of the explicitly named events. However, the operand subsequence:

```
xi,ANYBUT(x0,x1, . . .,xn),xj
```

can be abbreviated by "xi,xj". Again, this is consistent with the discussion and examples presented in the body of the paper.

APPENDIX B

In this appendix, we discuss certain details regarding the interpretation of event identifiers which appear in the operands of concurrency expressions. Throughout this discussion, we will be referring to the desired behavior textual unit shown in Figure 3, which is presumed to be part of the definition of a monitor class called [r_w]. This monitor class is assumed to include the definition of the four events *shared_read, shared_write, private_read,* and *private_write,* where the first two correspond to operations on resources shared among all instances of the monitor class while the latter two correspond to operations on private, unshared resources.

As was stated earlier, 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 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. Thus the first concurrency expression of the Figure 3 example represents the restriction that while some [r_w] monitor class instance is performing a shared write, i.e., its *shared_write* event is occurring, no **other** [r_w] monitor class instance may be performing a shared write and **no** [r_w] monitor class instance may be performing a shared read. Similarly, the second concurrency expression expresses the restriction that while some [r_w] monitor class instance is performing a shared read, i.e., its *shared_read* event is occurring, **no** [r_w] 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 [r_w] monitor class instances simultaneously.) The third concurrency expression indicates the designer's intention that at most one of the *private_read, private_write, shared_read,* or *shared_write* events will be occurring at any time within any given instance of the [r_w] monitor class. The fourth concurrency expression in the Figure 3 example indicates that the designer intends to allow multiple instances of the [r_w] monitor class to be performing shared reads simultaneously, i.e., an instance's *share-*

Figure 3.

```
DESIRED BEHAVIOR;

  MUTUALLY EXCLUSIVE(shared_write,OR([r_w]|shared_write,
                                 [r_w]|shared_read)),

  MUTUALLY EXCLUSIVE(shared_read,[r_w]|shared_write),

  MUTUALLY EXCLUSIVE(OR(private_read,private_write,shared_read,
                        shared_write),OR(private_read,private_write,
                        shared_read,shared_write)),

  POSSIBLY CONCURRENT(shared_read,[r_w]|shared_read),

  POSSIBLY CONCURRENT(OR(private_read,private_write),
                      OR([r_w]|private_read,[r_w]|private_write,
                         [r_w]|shared_read,[r_w]|shared_write))

  END DESIRED BEHAVIOR;
```

d_read event may overlap the *shared_read* event of any instance. And the final concurrency expression of the example expresses the designer's willingness to allow *private_read* or *private_write* events of one instance of the [r_w] monitor class to overlap any of the events *private_read, private_write, shared_read,* or *shared_write* of any [r_w] monitor class instance.

It should be noted that, taken together, the concurrency expressions of this desired behavior textual unit represent precisely the behavioral specifications which a software designer might wish to indicate regarding the use of shared and private data structures. That is, they specify that the writing of a shared structure must not be concurrent with any other manipulations of the shared structure (first concurrency expression), while the reading of a shared structure may be concurrent with other reading but not writing in that structure (second and fourth concurrency expressions). Further, this desired behavior textual unit indicates that reading or writing of private structures may be concurrent with reading or writing, shared or private, by other instances of the [r_w] monitor class (fifth concurrency expression), but that at most one of the four operation types may be occurring within any given instance of the [r_w] monitor class at any given time (third concurrency expression).

REFERENCES

1. R. Campbell and A.N. Habermann, The specification of process synchronization, in *Lecture Notes in Computer Science,* Vol. 16, Springer-Verlag, 1974.
2. R. Floyd, Assigning meaning to programs, in *Mathematical Aspects of Computer Science,* Vol. 19, Am. Math. Soc. 1967.
3. I. Greif, A language for formal problem specification, *Commun. ACM,* 20, 931–935 (1977).
4. Z. Manna, *Mathematical Theory of Computation,* McGraw-Hill, New York, 1974.
5. W. Riddle, The translation of path expressions into message transfer expressions, RSSM/2, Department of Computer and Communication Sciences, The University of Michigan (1974).
6. W. Riddle, An approach to software system behavior specification, *J. Computer Languages,* 4, 29–47 (1979).
7. W. Riddle, An assessment of DREAM, in *Software Engineering Environments* (Hunke, ed.), North-Holland, 1981, pp. 191–221.
8. W. Riddle, J. Sayler, A. Segal, A. Stavely, and J. Wileden, Hierarchical description of software system organization, in *Proc. 13th Hawaii International Conf. on System Sciences,* Honolulu, 1980.
9. W. Riddle, Abstract process types, RSSM/42, Department of Computer Science, University of Colorado at Boulder, 1977, revised July 1978.
10. W. Riddle, J. Sayler, A. Segal, A. Stavely, and J. Wileden, A description scheme to aid the design of collections of concurrent processes, in *Proc. 1978 National Computer Conference, Anaheim,* 1978, pp. 549–554.
11. W. Riddle, J. Wileden, J. Sayler, A. Segal, and A. Stavely, Behavior modelling during software design, *IEEE Trans. Software Engineering* SE-4, 283–292 (1978).
12. W. Riddle, J. Sayler, A. Segal, A. Stavely, and J. Wileden, DREAM—A software design aid system, in *Information Technology: Proc 3rd Jerusalem Conf. on Information Technology,* (J. Moneta, ed.), North-Holland, 1978.
13. W. Riddle, J. Sayler, A. Segal, A. Stavely, and J. Wileden, Abstract monitor types, in *Proc. of Specification of Reliable Software Conference, Boston,* 1979, pp. 37–43.
14. J. Sayler, Philosophy of the DREAM system, RSSM/39, Department of Computer and Communication Sciences, The University of Michigan, 1977.
15. A. Shaw, Software descriptions with flow expressions, *IEEE Trans. Software Engineering* SE-4, 242–254 (1978).
16. J. Wileden, DREAM design notation example: Scheduler for a multiprocessor system, RSSM/51, Department of Computer and Communication Sciences, The University of Michigan, 1977.
17. J. Wileden, *Modelling Parallel Systems with Dynamic Structure,* Ph.D. thesis, Department of Computer and Communication Sciences, University of Michigan (January 1978).
18. J. Wileden, DREAM—An approach to designing large scale, concurrent software systems, in *Proc. ACM National Conf., Detroit,* 1979, pp. 88–94.