ABSTRACT MONITOR TYPES

by

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

CS-CU-143-78 October, 1978

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

Abstract

A technique is presented for the description of data repository modules which are shared among a community of concurrent, asynchronous processing modules. The technique is designed to be of use in preparing models of a software system during the architectural phase of its development, when modules, module interfaces and module interactions are being defined. The rigor and precision of descriptions using the technique offers the opportunity to perform analysis during design and several approaches to analysis afforded by the technique are discussed.

I. Introduction

Several systems have been proposed as vehicles for the delivery of computerized support to software system developers ([AmbA77], [BakJ78], [CamI78], [DavC77], [EstG78a], [EstG78b], [GooD77], [HenP75b], [MorM77], [PeaD73], [RidW78d], [RobL75], [TeiD77], [VHoE78]). Though diverse, these systems share two characteristics. First, each provides a software system description language by which models may be prepared which reflect the system under development to an almost arbitrary level of detail. Thus, these support systems foster a modelling approach to development in which the system description gradually evolves as a series of abstract models. Second, the systems provide tools for analyzing models at any point during development. Thus, they allow the integration of analysis and synthesis so that system developers may incrementally develop confidence in a system's appropriateness in parallel with incrementally developing the system itself.

For our own software development support system, called the Design Realization, Evaluation And Modelling (DREAM) system [RidW78d], we have focused upon the architectural design phase of software development. This is that phase which follows the definition of a system's functional performance and economic requirements and precedes the development of the detailed data organizations and processing algorithms to be used in the system. The task during this phase is to specify both a system's gross organization, usually as a hierarchy of subsystems, and a system's overall behavior, usually in terms of the user-perceivable attributes used to express the system's requirements. Descriptions prepared during this phase therefore demarcate modules and indicate their hierarchical organization, their interfaces and their interactions.

In developing the DREAM system we have also focused upon the design of concurrent software systems, that is, those which may be perceived as having components which operate asynchronously and in parallel, even if the eventual system executes in a uniprocessor environment. Not only is the design of this class of software systems notoriously difficult, but viewing a system as decomposible into parallel parts greatly facilitates a "divide-and-conquer" approach to its development.

To allow the specification of models of concurrent software

systems during the architectural design phase of their development, the description language provided by DREAM, called the DREAM Design Notation (DDN), provides three major sets of description constructs. The first of these allows the non-procedural specification of a system's intended behavior and, thus, the expression of the system's requirements in terms that relate to the system's evolving design. These constructs are introduced in [RidW78c] and discussed in depth in [WilJ78]. The second set of constructs provide for the description of collections of sequential processes (these collections are called subsystems in the nomenclature of DDN) which reflect the system's processing strategies and which are perceived as coordinating each other's operation by the interchange of messages. These constructs are also introduced in [RidW78c] and are described in greater detail in [RidW77] and [RidW78b]. The final set of constructs provided in DDN allows the specification of repositories of data which are shared among subsystems and which are therefore subjected to a stream of parallel, and potentially conflicting, demands. These constructs are the subject of this paper.

In the following sections we introduce the data repository description facilities of DDN by a series of examples and indicate the ways in which we have amalgamated and extended the concepts of abstract data types [LisB78] and monitors ([HoaC74], [HowJ76]) in order to provide a practical technique for modelling shared data repositories. We also discuss the approaches to analysis admitted by these DDN description constructs. In the conclusion, we give a brief discussion of the relationship of our work to the work of others and summarize our own assessment of our work.

II. Basic Description Constructs

A model of a data repository module may be constructed with respect to either of two different perceptions of the module's participation in the overall functioning of the system. First, there is the external view, appropriate when defining the support the module lends to other modules, which leads to a definition of the repository's perceivable behavior, generally in terms of its responses to external stimuli. The alternative, internal view is appropriate when defining the repository's manner of operation and involves the definition of the

control algorithms used in the creation of the observable behavior.[1]

In this section we ignore the fact that a data repository may reside within a community of concurrent subsystems. The focus is therefore upon those constructs available in DDN for preparing external and internal descriptions of those aspects of data repositories which do not pertain to the synchronization needed to handle multiple, overlapping external stimuli. These DDN constructs are in essense those developed for the Tools for Program Development (TOPD) system [HenP75b].

II.1. External Descriptions when Concurrency is not Considered

Data repository components are described generically, using the currently popular notion of class that was first introduced in the Simula language [Dah066]. In Figure 1, a class, called [*datum*], of monitor-type objects is defined using the syntax of the DDN language. An object belonging to a class, also called an instance of the class, may be declared and named using the declaration phrase:

x OF [y]

which associates the name *x* with an object which is an instance of the class [*y*]. Note that an identifier is always enclosed in [...] when used to name a class.

Attributes and characteristics of the objects belonging to a class are defined by additional description fragments (called textual units) embedded within the textual unit defining the class. An example is given in Figure 2 where the embedded textual units define operations which may be invoked upon instances of class [*datum*]. All textual units have a header with a keyword indicating the type of the textual unit and a trailer which marks the end of the unit. Nesting of

---

1. The distinction of the roles of these two description orientations is not always as clear cut as that which we use here. Programs in non-procedural programming languages, for example, are external, behavioral descriptions which are prepared in order to define, albeit implicitly, the operation needed to create the behavior. On the other hand, in other parts of the DDN language ([RidW77], [RidW78b]) we have provided constructs by which a seemingly internal, operational description may be prepared in order to describe behavior.

textual units is used to indicate the scope of whatever information they declare.

Figure 2 indicates that the operations which may be invoked upon instances of a class are defined as procedures. Parameters for these procedures are declared as objects belonging to classes, thereby indicating their characteristics as those defined for these classes, and parameter passing mechanisms are denoted as in the AlgolW language [BauH69]. Invocation of an operation upon an object is denoted by the syntax:

$$x.p(a_1,a_2,\ldots,a_n)$$

where p is a procedure defined for the class of which x is an instance and the a_i arguments are instances of the classes of the corresponding parameters. Each object has its own copy of the procedures defined for the class of which it is an instance and, to refer to the copy of a procedure associated with a particular object, we speak of that procedure owned by the object.

To allow the definition of the observable effect of invoking a procedure, objects may be considered to be in some state at any point in time and the effect of procedure invocation may then be described by defining the state transition caused by the procedure's execution. In DDN, the possibly infinite set of states for an object is modelled by specifying a finite set of subsets of the object's state space. An example is shown[2] in Figure 3 for the class [*datum*]. The set of names for the state subsets constitutes a set of "checkable" attributes for an object in the sense that processing may be conditioned upon the state of an object by referencing state subset names in logical expressions.

The effect of procedure invocation is defined by a set of transitions. Each transition defines one of the possible effects in terms of: 1) a condition upon the state of the parameter objects and the object owning the invoked procedure, and 2) a specification of the state of these objects after the procedure's execution. In the example of

---

2. To avoid having to repeat all of the previously defined information when adding new information to a class definition, DDN allows a textual unit to be "inserted" into a class definition by prefacing it with a quoted sequence of identifiers which indicates the textual unit's position within the hierarchy of textual units defining the class.

Figure 4, the transitions for the *assign* procedure indicate that the state subset of the object owning the invoked procedure is referenced by merely referencing one of the state subset names.[3] Thus:

new_value = defined → defined

means "if the object *new_value* is in a state within the *defined* state subset at the beginning of invocation, then the state of the owning object is in the *defined* state subset at the end of invocation." The fact that *new_value* is a value parameter further implies that the state of the argument corresponding to *new_value* does not change. Also, not referencing the object upon which the procedure is invoked within the transition's left-hand side implies that it may be in a state which falls in any of the state subsets. Note that the set of left-hand sides of the transitions defines pre-conditions for the legal invocation of the procedure, and the right-hand sides define post-conditions corresponding to each pre-condition.

II.2. Additional Comments

Before turning to the basic DDN constructs for internal descriptions, we give some additional examples of external descriptions and use these examples to introduce more details concerning the DDN constructs for external description.

In describing objects generically it is frequently desirable to be able to parameterize the description so that specific details may be fixed at the time that an instance is declared. This facility is provided in DDN by qualifier textual units, as indicated in Figure 5, in which identifiers are defined which may be used in the class definition and receive values when an instance is declared. For example:

index OF [integer_range(10)]

would declare an instance in which *high* should be read as *10* wherever it appears in the [*integer_range*] class definition.

Figure 5 also indicates that an ordering may be specified among

---

3. Alternatively, this transition could have been expressed as:
new_value = defined → ME = defined
using the reserved identifier ME to reference the owning object.

the state subsets so that the usual complement of relational operators may be used in conditions upon the state of object. The state ordering textual unit in Figure 5 also indicates that state subsets are not necessarily disjoint. (With non-disjoint state subsets, it is possible that a state ordering is internally inconsistent; the DREAM system will contain an analyzer which checks the consistency of state orderings.)

The transitions for the *increment* procedure indicate that they may be nondeterministic. This greatly facilitates the specification of abstract models since it allows the suppression of detail. It does, however, allow an impreciseness which is potentially detrimental. For example, the name *increment* connotes that the procedure is to increase the "value" of the [*integer_range*] object by one, but the transitions describe many other implementations of the procedure. However, suppose it can be shown that if the procedure's implementation produces the effect defined by the transitions then any algorithm which uses the procedure will function appropriately. Then, any implementation which produces the effect would be acceptable, irrespective of the English-language semantics of the identifier used to name the procedure.

In Figure 6, we indicate that state subsets may be defined through a coordinatization expressed in terms of a set of state variables. This is allowed since it generally makes the task of state subset definition easier. Additionally, it allows the explicit description of the overlap among state subsets. Potential problems again arise because the definition of a state subset may not exactly match that implied by the English-language semantics of the state subset's identifier. But, it is again true that this is not a problem as long as it can be shown that the definitions lead to the delivery of expected or required behavior.

II.3. Internal Descriptions When Concurrency is not Considered

In an internal description, the intent is to represent the manner in which an object is composed of other objects and to describe the algorithms by which the effects of procedure invocation, as described in the external description, may be created by sequences of procedure

invocations upon the internal objects.

In Figure 7, we give an external definition of a class named [*search_routine*] augmented by a textual unit defining the internal components (in this case, there is only one) which comprise each instance of the class. Also in Figure 7 are two definitions of classes referenced within the definition of [*search_routine*]. With the class [*search_routine*] we are specifying a class of objects which are data manipulation objects rather than data repository objects. In this class definition, the DDN description constructs are used to define the class of functions which search an array for a specified item, indicate by their value whether or not the item was found, and return the index of the item in the case that it is found within the array. This class of functions could alternatively have been defined as a procedure for that class of data repository objects which are the arrays being searched, or as a procedure associated with the class of indices returned, or in a variety of other ways. But it is not necessarily natural to associate it with any of the classes of objects upon which it operates.

Being essentially a processing module, the internal objects for the [*search_routine*] class are solely those needed during processing rather than those needed to save values between invocations. In this case, all that is needed is a single object for use in controlling the search of the array and in indicating whether or not the sought-for item was found.

The algorithm used to do the searching is described by giving a body textual unit for the procedure as indicated in Figure 8. The algorithm controls the sequencing of the invocation of operations upon the parameters and internal objects and the controlling conditions are specified as tests upon the states of these objects. DDN provides a full set of control constructs for the go-to-less specification of procedure bodies.

Our presentation of examples has been "bottom-up" in that we have first presented "primitive" classes, like [*datum*], which have no internal components and then presented "higher-level" classes whose instances are composed of instances of the primitive classes. This is an artifact of our order of discussion of the features of DDN. In

actually using DDN to prepare a description during architectural design, class definitions would more likely be prepared in a top-down manner since it is most effective to first define the need for a particular class of objects and delineate the procedures required to manipulate objects in the class, and then subsequently define internal objects and actual control algorithms for this class of objects.

III. Consistency Checking

When both an external and an internal description of a class exists, the question naturally arises: "Do the procedure control algorithms produce the effect described by the transitions?" This question as to the consistency between the internal and external descriptions may be answered by a technique developed in conjunction with the TOPD system [HenP75a]. In this section, we briefly explain the TOPD consistency checking technique since we wish later to discuss the feasibility of extending it to the case of concurrent activation of procedures. We also make some observations about the value of the technique when used in conjunction with a top-down design method.

Central to the TOPD consistency checking technique is an algorithm, called the finite state testing algorithm, which is able to derive the state-transition effect of a procedure body's execution, given an initial state for all of the objects manipulated by the procedure. This algorithm requires transition definitions for each procedure invoked during the execution of the procedure body being analyzed. (And because it uses this information and does not "look inside" the invoked procedures, the algorithm is quite fast.) The algorithm determines the set of possible states for manipulated objects at the termination of the procedure under the assumption that it does terminate. There is a set of possible final states both because there are, in general, branches within the algorithm and because the transitions for the invoked procedures are, in general, nondeterministic.

Consistency checking for a procedure is performed by using the finite state testing algorithm as follows. Each unique left-hand side for the transitions of the procedure being checked is considered in turn. Using information that relates the state of the owning object to

the states of its internal objects, a (set of) initial state(s) is derived. The finite state testing algorithm is then used to find the set of final states which the procedure would produce when invoked with the parameter and internal objects in (one of) the derived initial state(s). The right-hand sides for the transitions involved at this step of the consistency check are then also translated to states upon the parameter and internal objects. Then this set of allowed final states is checked against the set of final states found by the finite state testing algorithm to assure that there is at least one terminal state corresponding to each right-hand side and no terminal states which do not correspond to any of the right-hand sides. If this condition doesn't hold then there is an inconsistency; if it holds for all the unique left-hand sides then consistency has been demonstrated.

As an example consider the consistency checking of the *find* procedure for the class [*search_routine*]. For the left-hand side *item = undefined* the set of initial states may be found directly since there is no reference to the owning object. (The rule for indicating the state of an array is that the array is in state subset *s* if at least one of its element is in state subset *s*.) Applying the finite state testing algorithm will produce a set of terminal states such that in every state in the set, *found_flag* will be in the state subset *false*. Considering the right-hand side of the transition and the equivalency information given in Figure 9, it may be concluded that there are no disallowed terminal states and at least one terminal state which corresponds to the right-hand side. Similar reasoning involving the second transition leads to the conclusion that it too accurately describes an effect of invoking the procedure. Hence, it may be concluded that the transitions and the body are consistent.

Used in conjunction with a top-down design method, the consistency checking technique affords the opportunity to integrate analysis and synthesis during design. If each synthesis step consists of preparing one or more procedure bodies as well as transition definitions for any as-yet-undefined procedures invoked by these procedures, then it may be followed by an analysis step in which the consistency of the new procedure bodies and their corresponding transitions is

checked. If consistency is shown then the designers may proceed to the next synthesis step with increased confidence. If consistency cannot be shown, then the designers may correct the description before proceeding.

IV. Description When Concurrency is Considered

When instances of monitor classes are shared by a community of concurrent subsystems, there is the need to coordinate the multiple, and potentially conflicting, activations of procedures upon each shared object. With respect to internal descriptions, a variety of coordination mechanisms have been proposed and we adopt that defined by Hoare, as the name "monitor class" implies. There are not, however, any extant solutions for the problems arising with respect to external descriptions and so we have developed extensions to the basic DDN external description constructs. We discuss these issues in this section and then, in Section V, suggest that we have preserved the ability to do consistency checking and allowed other types of analysis.

As a basis for our examples in this section we choose the traditional readers/writers problem. We hypothesize a class of objects called [*region*], with states we have already defined in Figure 6. We view each [*region*] object as partitioned into segments, some of which are private to the [*region*] object and some of which are shared among [*region*] objects. Read and write requests may be asynchronously and concurrently directed toward the individuals in a collection of [*region*] objects by invocation of *read* and *write* procedures. A [*datum_id*] parameter to each of these procedures identifies the location which is to be read or written. We wish first to develop internal descriptions of these procedures which exhibit the usual desired characteristics, namely that: 1) writing of a shared segment may not be concurrent with any other operation upon the shared segment, 2) within any single [*region*] object, reading or writing of shared or private segments may not be occurring simultaneously, and 3) across a collection of [*region*] objects, reading or writing of shared or private segments may otherwise occur to any degree of concurrency. Additionally we wish to develop demonstrably consistent external descriptions of these procedures and

be able to rigorously argue that the internal descriptions give rise to the desired characteristics stated above.

IV.1. Internal Description when Concurrency is Considered

As a first step in our description of the class [*region*], we give, in Figure 10, a DDN description of the class of [*segment*] objects. In this description we have used the instantiation control facilities of DDN to indicate that the *locator* subcomponent is unique across all instances of the [*segment*] class. Though perhaps artificial, this allows us to explain how the monitor mechanism is used to control execution. Suppose the *read_segment* procedure for some [*segment*] object is invoked. When execution reaches the invocation of the *find* procedure for the *locator* object, then execution of the *read_segment* procedure is first suspended. If an invocation of the *find* procedure is already in progress (because of an invocation from within a procedure owned by some other object) then knowledge of the invocation is saved on a list of invocations waiting to be done upon the procedures owned by the *locator* object. When the in-progress invocation terminates, control is returned to the procedure which lodged that invocation and one of the waiting invocations is chosen, at random, to be activated.

The internal description of [*region*] objects may then be expressed as in Figure 11. The major subcomponent is the array *data* of [*segment*] objects. The associated array *segment_status* of [*boolean*] objects is used to indicate which of the [*segment*] objects is shared with at least one other [*region*] object; the values in the *segment_status* array are presumably initialized or otherwise set to accurately indicate sharing. The *indicator* object is used to encode the operation being performed at any point in time. And the *segment_number* object is used in accessing the appropriate [*segment*] object.

Control of reading and writing of shared segments is effected through the *waiting* and *commutator* objects. In the solution indicated here, *commutator* is an object shared by all the [*region*] objects (as indicated by the instantiation control textual unit) as well as by some supervisory object (as would be described by an instantiation control

textual unit in some other, unexhibited part of the description). The value of *commutator* is used to give one of the [*region*] objects permission to operate upon a shared [*segment*] object.[4]

The [*CONDITION*] object called *waiting* provides the ability to suspend an invocation of either the *read* or *write* procedure should the [*region*] object want to operate upon a shared [*segment*] object but not have permission. Invocation of the *WAIT* operation upon *waiting* suspends execution until a *SIGNAL* operation is invoked upon the *waiting* object.[5] When such a *SIGNAL* operation is performed, then all suspended invocations (associated with the [*CONDITION*] *waiting*) are transferred to the list of invocations awaiting activation (or re-activation), taking priority over any invocations which have never been activated.

IV.2. External Description when Concurrency is Considered

In the external description of a class of objects, the intent is to describe the state transition effect of each procedure's invocation. As far as the object which invoked the procedure is concerned, the only visible effect is the before/after state subset change since the activation of the invoking procedure is suspended until the invoked procedure terminates. Other concurrently operating objects may, however, make a state inspection during a procedure's activation and the external description must in general, therefore, relate the sequence of state subsets occurring between a procedure's invocation and its termination.

The DDN facilities for external descriptions which reflect state subset transition sequences occurring during procedure activation are illustrated in Figure 12. The concept of pre- and post-conditions is retained but the notation '→' is augmented by the ability to specify a set of state subset sequences, with the implication that any one of

---

4. Thus we are describing a solution in which there is some centralized control allowing only one [*region*] object to be operating upon a shared [*segment*] object at any point in time. This solution may not be ideal, but does provide us with an illustrative example.

5. The suspension effect of a *WAIT* operation propagates up through a hierarchy of procedure invocations. We have chosen this solution to the problem raised in [LisA77] because its effect cannot otherwise be described within DDN.

them may occur during procedure activation. The facilities for describing sets of state subset sequences consist of a set of prefix operators for describing the concatenation of sequences (SEQUENCE), the choice of alternative sequences (OR), the unbounded repetition of a sequence (REPEAT), the arbitrary interleaving of sequences (SHUFFLE), and the arbitrary interleaving of an unbounded number of copies of a sequence (REENTRANT). The set of sequences described by the second transition in the Figure 12 example is:

```
{ unoccupied reading_shared unoccupied,
  unoccupied stalled unoccupied reading_shared unoccupied,
  unoccupied stalled unoccupied stalled
        unoccupied reading_shared unoccupied, ... }
```

and which of these sequences actually occurs depends on the number of times procedure activation is suspended. In the first transition of Figure 12, we retain the '→' with the implication that any sequence of states can possibly be observed during procedure activation.

We do not include the internal and external descriptions of the *write* procedure since they are directly analogous to those given for the *read* procedure.

To close the discussion of this section we want to clarify a perhaps confusing point brought up by our example. The use of the monitor mechanism to assure that only one activation of a procedure is in progress at any point in time gives rise to the following question: "Why is all the machinery of Figure 11 necessary when the indivisibility of the *read_segment* and *write_segment* procedures assures the correct manipulation of shared [*segment*] objects?" A superficial answer is that is is really more than just the activation of these procedures that has been coordinated and that the non-current execution of sections of the *read* and *write* procedures has also been assured. A more accurate and instructive answer is that it is really only subcomponents of [*segment*] objects that are shared between [*region*] objects, as would be described by instantiation control textual units not exhibited in our examples. Thus there are multiple copies of the *read_segment* and *write_segment* procedures which can manipulate the shared subcomponents and we are coordinating the activation of these copies since the monitor mechanism does not, in and of itself, effect coordination between

copies of procedures.

V. Analysis When Concurrency is Considered

With the addition of the ability to describe state subset sequences occurring during procedure activation, we have complicated the checking of the consistency between internal and external descriptions, but we have not eliminated the possibility of performing consistency checking. First, the finite state testing algorithm can be modified to derive, when the effect of suspension is ignored, a regular expression (over state subset names) describing the sequences of state subsets arising from execution of the procedure. Second, use can still be made of state equivalence information, such as illustrated in Figure 13, to transform the specification of a set of state subset sequences to the level of internal and parameter objects. As long as the result of this transformation is also a regular expression (i.e., it does not involve the use of the REENTRANT operator [RidW78e]), then the comparison necessary to check consistency may, in theory at least, be performed.

But the feasibility of consistency checking depends on three conditions implied in the last paragraph's discussion. First, the computational complexity of performing comparison must be reasonable. We feel that the full power of a general algorithm is not necessary and that less-general algorithms, having acceptable complexity characteristics, will suffice. However, it is only through additional experience with DDN that we can determine the types of expressions which occur in describing state subset sequences, and thereby the validity of our intuitive feeling. Second, the REENTRANT operator may not appear. We have not needed it in the descriptions we have prepared so far ([CunJ77a], [CunJ77b], [RidW78a], [SegA77], [StaA77], [WilJ77]). Indeed, since it corresponds to reentrant activation, and state subset sequences pertain to a single activation, it is unlikely that it will ever be required. But, again, only more experience will allow us to determine the validity of this conclusion. Finally, there is the condition that the effect of suspension be ignored. The finite state testing algorithm could potentially be applied recursively, to assure that all procedures return the object to the state subset it was in at the

point of suspension, but there is an unpleasant increase in the amount of analysis necessary. We are continuing our research into ways to efficiently analyze in the face of this problem.

Assuming that consistency checking can be effectively and efficiently performed, there remains the problem of assuring that global properties, such as the non-concurrent reading and writing of shared [*segment*] objects, are exhibited by a proposed internal operation. The approach provided by DDN to solving this problem is based upon event expressions ([RidW78e], [WilJ78]) and event expression analysis [RidW78f] and would proceed roughly as follows. First, the finite state algorithm would be used to derive an expression for the state subset sequences produced by each procedure which affects the property being assessed. In this derivation, special symbols would be used to mark the points of suspension, reflect re-activation via *SIGNAL* operations, and indicate procedure activations. Then an expression for the state subset sequences may easily be formed by using the notation of event expressions to indicate that an overall state subset sequence consists of the interleaving of an arbitrary number of copies of the individual state subset sequences for the procedures. The special symbols constrain the otherwise arbitrary interleaving to reflect the semantics of *SIGNAL* and *WAIT* operations and procedure invocations. Then the resulting expression would be projected onto an alphabet of event names using the state equivalence information and information about the correspondence of an event to a state subset sequence that has been expressed by the event definition facilities available in DDN ([RidW78c], [WilJ78]). The resulting expression would then be compared to the statement of the property, as expressed in similar notation by the desired behavior definition facilities of DDN ([RidW78c], [WilJ78]), to determine whether or not all of the event sequences resulting from system operation are also desired.

There are several problems in the use of this approach. First, a general algorithm for projecting an expression describing the state subset sequences onto the event alphabet probably does not exist. Second, the comparison problem is known to be undecidable in general [RidW78f]. However, it is our feeling, reinforced by our experiences

in using DDN, that the full power of the event expression notation is not needed. We therefore believe that efficiently decidable subcases, that span a wide variety of typically occurring situations, can be identified. We are continuing our research on this topic.

VI. Conclusion

We have proposed a technique for the description of data repository modules which are shared among a community of concurrent, asynchronous processing modules. The technique allows the description of both a module's internal, operational aspects and a module's external, behavioral characteristics. Descriptions using the technique are abstract models of the actual modules and the technique may be beneficially employed during the architectural design phase of software development. The technique provides facilities which lend support to a variety of design methods, most especially top-down methods. The rigor and precision of the descriptions offers the opportunity to perform analysis and this analysis may furthermore be integrated with design, thus providing the opportunity for early error detection. We have suggested several approaches to analysis and identified the problems to be solved in order to produce efficient and effective analysis algorithms.

In this work, we have amalgamated and extended the work of others in order to produce a data repository description technique consonent with the techniques we developed for the modelling of collections of sequential processes ([RidW77], [RidW78b], [RidW78c]) and the description of software system behavior ([RidW78c], [WilJ78]). We have accepted the philosophy behind the concept of abstract data types, relied heavily upon the TOPD description technique which was developed for modelling sequential programs, and used the concept of monitors to provide a coordination mechanism. We have also adopted the TOPD approach to analysis as a basis for some of the analysis techniques we envision. Farther back in the ancestry of our work is the work of Parnas [ParD72] on module description. Our major departure from his work is to not require that state inspection be performed via a procedure and thereby be coordinated with other operations upon a module. We have taken this departure because we feel that modelling is much more easily and conveniently performed under the assumption that (a projection of) the

state of a module is visible to other modules.

We feel confident that the concepts underlying our description technique are important and useful for the modelling task which arises during the architectural design of software. We also feel that the technique provides a basis for a design support system which will deliver beneficial aid to software system design practitioners and that the most effective aid will come from analysis techniques such as discussed here.

VII. References

AmbA77 Ambler, A.L., Good, D.I., Browne, J.C., Burger, W.F., Cohen, R.H., Hoch, C.G., and Wells, R.E. Gypsy: A language for specification and implementation of verifiable programs. Software Engineering Notes, 2, 2 (March 1977), 1-10.

BakJ78 Baker, J.W., Chester, D., and Yeh, R.T. Software development by step-wise evaluation and refinement. SDBEG-2, Software and Data Base Engineering Group, Dept. of Computer Sci., Univ. of Texas, Austin, January 1978.

BaiJ69 Bauer, H.R. Introduction to AlgolW programming. Computer Sci. Dept., Stanford Univ., California, July 1969.

CamI78 Campos, I., and Estrin, G. SARA aided design of software for concurrent systems. Proc. 1978 National Computer Conf. Anaheim, Calif., June 1978, pp. 325-336

CunJ77a Cuny, J. A DREAM model of the RC 4000 multiprogramming system. RSSM/48, Dept. of Computer and Comm. Sci., Univ. of Michigan, Ann Arbor, July 1977.

CunJ77b Cuny, J. The GM terminal system. RSSM/63, Dept. of Computer and Comm. Sci., Univ. of Michigan, Ann Arbor, August 1977.

DahO66 Dahl, O., and Nygaard, K. SIMULA - An Algol-based simulation language. Comm. A.C.M., 9, 9 (September 1966), 671-678.

DavC77 Davis, C.G., and Vick, C.R. The Software development system. IEEE Trans. on Software Engineering, SE-3, 1 (January 1977), 69-84.

EstG78a Estrin, G., and Campos, I. Concurrent software system design, supported by SARA at the age of one. Proc. 3rd International Conf. on Software Engineering, Atlanta, Georgia, May 1978, pp. 239-242.

EstG78b Estrin, G. Application of machine descriptions to design of concurrent systems. In Moneta, J. (ed.), Information Technology, JCIT3/North Holland Pub. Co., 1978.

GooD77 Good, D.I. Constructing verified and reliable communications systems. Software Engineering Notes, 2, 5 (October 1977), 8-13.

HenP75a Henderson, P. Finite state modelling in program development. Proc. 1975 International Conf. on Reliable Software, Los Angeles, April 1975.

HenP75b Henderson, P., Snowdon, R.A., Gorrie, J.D., and King, I.I. The TOPD System. Tech. Report 77, Computing Laboratory, Univ. of Newcastle upon Tyne, England, September 1975.

HoaC74 Hoare, C.A.R. Monitors: An operating system structuring concept. Comm. A.C.M., 17,10 (October 1974), 549-557.

HowJ76 Howard, J.H. Signaling in monitors. Proc. 2nd International Conf. on Software Engineering, San Francisco, October 1976, pp. 47-52.

LisA77 Lister, A. The problem of nested monitor calls. Operating Systems Review, 11, 2 (July 1977), 5-7.

LisB75 Liskov, B.H., and Zilles, S.N. Specification techniques for data abstractions. IEEE Trans. on Software Engineering, SE-1, 1 (March 1975), 7-19.

MorM77 Moriconi, M.S. A system for incrementally designing and verifying programs. ICSCA-CMP-9, Certifiable Minicomputer Project, Inst. for Computing Sci. and Computer Applications, Univ. of Texas, Austin, December 1977.

ParD72 Parnas, D.L. A technique for software module specification with examples. Comm. ACM, 15, 5 (May 1972), 330-336.

PeaD73 Pearson, D.J. CADES - Computer aided design and evaluation system. Computer Weekly, (July/August 1973).

RidW77 Riddle, W.E. Abstract process types. RSSM/42, CU-CS-121-77, Dept. of Computer Sci., Univ. of Colorado at Boulder, December 1977 (revised July 1978).

RidW78a Riddle, W.E. DREAM design notation example: The T.H.E. operating system. RSSM/50, Dept. of Computer Sci., Univ. of Colorado at Boulder, April 1978.

RidW78b Riddle, W.E., Sayler, J.H., Segal, A.R., Stavely, A.M., and Wileden, J.C. A description scheme to aid the design of collections of concurrent processes. Proc. 1978 National Computer Conf., Anaheim, Calif, June 1978, pp. 549-554.

RidW78c Riddle, W.E., Wileden, J.C., Sayler, J.H., Segal, A.R., and Stavely, A.M. Behavior modelling during software design. IEEE Trans. on Software Engineering, SE-4, 4 (July 1978), 283-292.

RidW78d Riddle, W.E., Sayler, J.H., Segal, A.R., Stavely, A.M., and Wileden, J.C. DREAM - A software design aid system. In Moneta, J., (ed.), Information Technology, JCIT-3/North-Holland Pub. Co., August 1978.

RidW78e Riddle, W.E. An approach to software system behavior description. To appear: J. of Computer Languages.

RidW78f Riddle W.E. An approach to software system modelling and analysis. To appear: J. of Computer Languages.

RobL75 Robinson, L., Levitt, K., Neumann, P., and Saxena, A. A formal methodology for the design of operating systems software. In Yeh, R.T. (ed.) Current Trends in Programming Methodology, Vol.I, Prentice-Hall Inc., Englewood Cliffs, N. J., 1977.

SegA77 Segal, A.R. DREAM design notation example: A multiprocessor supervisor. RSSM/53, Dept. of Computer and Comm. Sci., Univ. of Michigan, Ann Arbor, August 1977.

StaA77 Stavely, A.M. DREAM design notation example: An Aircraft engine monitoring system. RSSM/49, Dept. of Computer and Comm. Sci., Univ. of Michigan, Ann Arbor, July 1977.

TeiD77 Teichroew, D., and Hershey, E.A. PSL/PSA: A Computer-aided technique for structured documentation and analysis of information processing systems. IEEE Trans. on Software Engineering, SE-3, 1 (January 1977), 41-48.

VHoE78 Van Horn, E.C. Software evolution using the SEER data base. Digital Equipment Corp., Maynard, Massachusetts, June 1978.

WilJ77 Wileden, J.C. DREAM design notation example: Scheduler for a multiprocessor system. RSSM/51, Dept. of Computer and Comm. Sci., Univ. of Michigan, Ann Arbor, October 1977.

WilJ78 Wileden, J.C. Behavior specification in a software design system. RSSM/43, COINS Tech. Rep. 78-14, Dept. of Computer and Info. Sci., Univ. of Massachusetts, Amherst, July 1978.