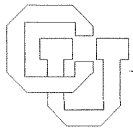
Modelling and Simulation in the Design of Complex Software Systems

William E. Riddle
John H. Sayler

CU-CS-130-78



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

MODELLING AND SIMULATION IN THE DESIGN OF COMPLEX SOFTWARE SYSTEMS

by

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

John H. Sayler
Computer and Communication Sciences Dept.
University of Michigan

CS-CU-130-78 July 1978

Prepared for presentation at the Symposium on Modelling and Simulation Methodology, Weizmann Institute of Science, Rehovot, Israel, August 13-18, 1978

I. INTRODUCTION

There is no single attribute or characteristic which distinguishes a software system as complex - rather, software system complexity may be assessed only in terms of a complicated combination of a variety of quantitative and qualitative measurements. Typical quantitative measures are size, number of components and length of the construction period - generally these can be artificially increased by inexperienced or inept designers or programmers. Measures of a more qualitative nature are ones such as understandability and modifiability - these are generally dependent upon the experience and ability of those trying to measure the system's complexity.

One measure does stand out, however, as hightly correlated with general, subjective assessments of software system complexity. This measure is *connectivity*, the degree to which interdependencies exist among the system's components. Frequently, the interdependencies manifest themselves in terms of aspects of the system which may be easily measured; an example is the connection established between two components when one invokes the other through the subroutine-call facilities of the language in which they are programmed. At least equally frequently, connectivity is *not* manifest in easily measurable aspects of the system; an example is the typical situation of an initialization component which establishes the environment in which other components operate.

Recent research in the areas of design and maintenance of software systems has mainly been directed toward discovering ways in which connectivity can be controlled and reduced. Much of this research has been directed toward small, sequential programs in the belief that programming in the small must be thoroughly understood before one can appropriately attack large-scale software systems.[1] But large-scale systems differ qualitatively from small-scale ones, primarily in the nature of the connectivity among the components. Instead of the simple data referencing and control flow connections typical of small-scale systems, there are more complex data sharing and control synchronization connections.

There are two basic approaches to coping with connectivity, and

---

[1]This viewpoint has been espoused by many, perhaps the most notable being E. W. Dykstra ([1],[2]).

hence complexity, within large-scale software systems. On the one hand, system organizations and operation modes may be sought which lead to clean, clear connections among the components within the system. Alternatively, description schemes may be sought which permit the concise, clear description of component connectivity so that an understanding of the connectivity and its implications concerning the operation of the system as a whole may be quickly grasped. In either case, the superstructure and synchronization of the components must be defined clearly in order to avoid cacophony and this orchestration of the components is best approached at an appropriately abstract level, where the system's structure and behavior can be investigated directly without concern for specific implementation details.

In this paper we discuss an approach to coping with software system complexity of the second type delineated above. Specifically, we review a software modelling scheme, and approaches to software design which are admitted by this scheme, which allows designers to explicitly and succinctly describe the structural and behavioral connectivity among a software system's components. We argue that through the explicit description of connectivity and the concomitant ability to reason (via simulation or analytic techniques) about its appropriateness, designers may more effectively function during the design process.

In the next two sections, we discuss software modelling in general, indicating its relationship to other work directed toward the production of correctly functioning systems and its relationship to programming. Then we review several software modelling schemes and give an overview of our own, followed by a discussion of the relationships between our scheme and the theory of general systems. We then turn to analysis techniques admitted by our modelling scheme, discussing first the role these techniques play in the general design methodology based upon the modelling scheme and, second, the specific simulation and analytic techniques which we have investigated. In closing, we attempt to give some direction to future work in this area.

Modelling and the Production of Correctly Functioning Systems.

Numerous investigators have studied the problems of proving the correctness of programs since Floyd's famous paper [3]. Some

understanding of the principles involved is now required of most computer science students and at least two current textbooks [2,4] postulate the necessity of proving programs correct as they are being developed. While the future usage of these techniques is open to debate, it is clear that the majority of working programmers are not now engaged in formally proving the correctness of their programs, and this seems unlikely to change in the near future. There are two issues here pertinent to the development of large systems: one, it is presently too difficult and time consuming to develop proofs for each small piece of a large system, much less integrate these proofs to achieve a proof of the total system; second, the language of the assertions required in correctness proofs is inimical to the algorithm and data structuring constructs used in the abstract description of a large-scale software system.

Another major research area relating to the production of correctly functioning systems is the area of program comprehensibility. Included under this heading are structured programming, code format conventions, and programming language constructs. The complexity of current software systems make it imperative that their understanding require less effort than their original development, and program comprehensibility research attacks this problem.

A final major area of research concerns questions regarding the development of software systems - what sorts of environments are best for teams of programmers ([5],[6]), and what sorts of "methodologies" should be applied during development ([7],[8],[9])? While the intuitive notions developed through these investigations are useful, they lack the precision of formal specifications and hence function as quidelines only.

There is little doubt that the research mentioned has significantly improved the practice and understanding of the programming process. However, it has not been concerned with the design phase as such, but rather with the program which is the result of the design phase. In particular, research on the above themes focuses on questions of the following sort (see for example [10]):

What programming language features produce the "cleanest" code?

What programming language features have good proof rules associated with them?

What sorts of things can be checked at compile time and which must be delayed until run time?

How can one enforce strong typing?

What are good practices for programmers to follow so that the products of their labor are understandable, correct and modifiable?

The central thesis of this paper is that it is highly profitable to consider large software systems from a more abstract point of view, that of a general systems theorist. From this viewpoint, some different questions are addressed and different goals arise, for example:

What are the central characteristics of large software systems, i.e., what characteristics are essential to modelling such systems?

How can these characteristics be quantified, in particular to allow comparison of different systems or different designs for the same system?

What sorts of mathematical models are appropriate to the study of large software systems?

What homomorphisms can be used to relate different levels of a system design?

Can our mathematical models yield any formally justified design principles?

The ultimate goal of our work is to develop a software system design methodology, based on answers to these types of questions, which provides a vehicle for employing *program development* quidelines and practices in the *design* of large_scale software systems.

Modelling Versus Programming

Software system modelling and software system programming are superficially similar but differ qualitatively in three respects. First, during the programming phase the intent is to produce efficient, well-structured code for each of the system modules. Hence the emphasis is upon the definition of data structures and the algorithms manipulating them. During the design phase the aims are more global - one is interested in preparing a complete specification and doing so in an incremental fashion rather than en masse. Second, system properties which are of concern during design may cut across module boundaries whereas during implementation the system properties of interest are usually local to modules. Finally, during implementation, behavior is defined implicitly whereas during design one wishes to give explicit, nonprocedural specifications of behavior.

The constraints in effect during the two phases are quite different also. During implementation one must account for limited resources, the operation of real synchronization primitives provided by the programming language, and error detection and recovery. During the design phase these concerns are not as important because one is more concerned with global, and partial, descriptions. Thus, during the design phase it is acceptable to assume an infinite supply of resources and idealized synchronization primitives. Further, error correction is defined implicitly by nonprocedural definitions of the behavior that the system *must* realize, and error detection may be modelled by the inclusion of "error" states as post conditions of certain actions of the system.

Recent advances in the design of programming languages have tended to raise their level toward the domain of system modelling. For example, abstract data types [11] have recently emerged as an important facility for the specification of sequential programs. While they are convenient for the description of a software system's data storage components, they are not convenient for the succinct description of those system components which are more for the processing than the storage of data. This is particularly true when the components operate concurrently.[1] The major problem is that abstract data types are oriented towards describing components as structures of data which are operated upon via procedure calls. Many components (e.g., a text editor in an operating system or a file system in a multiprocessor computing facility) are not naturally described in this manner.

Software Modelling

Thus software system modelling is a task which is distinct from software system programming. Not only are the intents of modelling different from those of programming, but the constructs needed in a modelling language are somewhat different in nature from those of programming languages. While there have been surprisingly few efforts to develop software system modelling languages, some of the essential constructs of these languages have been developed.

---

[1]By "concurrent" we mean parallelism which may be actually achieved by executing the system in a multiprocessing environment or which may be only apparent at abstract levels of system description and never achieved during system execution.

The SIMULA language [12] introduced the class concept which has subsequently found a home in the TOPD system (see below) and the DREAM system, and is central to the definition of abstract data types. Several other features of the SIMULA language make it useful for performance modelling of software systems [13].

The TOPD system ([14],[15]), developed at the University of Newcastle upon Tyne, England, introduced the concept of finite state modelling of sequential programs. In TOPD the "values" of data objects are partitioned into "states". Procedures which operate on these objects effect state transitions which may be specified via pre- and post-conditions for the procedure's invocation. One may "run" a TOPD model and receive a listing of the possible states of objects at each statement of the program. In addition, TOPD can perform some checking of the internal consistency of the model description.

Petri nets and equivalent schemes (reviewed in [16]) have been used for the formal modelling of systems with parallelism. These schemes accurately portray the detailed action sequences of processes operating in parallel, but have strong drawbacks as a general modelling scheme for the design of complex systems: first they are not language based; second, they are generally quite cumbersome to use because of the low (essentially machine) level of description; and third, they generally have no explicit behavioral component, i.e., the action sequences may be obtained only through model simulation.

The PPML modelling scheme [17] was developed as a natural, but still formally defined, modelling tool for systems with parallelism. In PPML the component processes of a system are described in a high level modelling language rather than a graph or mathematical representation of the potential activity of the system. The component processes of the system communicate via message transmissions. The communication is mediated by link processes which serve to effect all necessary message transmission synchronization. Since overall system coordination is modelled by the transmission of messages, this activity is of paramount importance in understanding PPML-modelled systems. Thus, the PPML scheme allows one to algorithmically derive [18] a closed-form representation of the message transmission behavior of the system,

which may be inspected for incorrect functioning of the system.[1]

Software Modelling in the DREAM System

The considerations outlined above guided us in our development of the Design Realization, Evaluation And Modelling (DREAM) system ([19]-[22]) and its associated modelling language, the DREAM Design Notation (DDN). Our specific aims were to develop a system which would allow a designer to iteratively develop a model of an intended system by providing both a modelling language for the description of the system and a data base for retention and extraction of design fragments. In addition we wished to provide both simulation and analytic techniques which designers could use to incrementally bolster their confidence in the validity of the evolving design. In this section we describe the basic modelling constructs provided by DDN; analysis techniques provided by DREAM will be treated in a later section.

In DDN a system is defined to be a network of hierarchically decomposable components which execute concurrently and asynchronously. The overall purposes of the system are achieved via the internal processing of the components as coordinated by communication among them. In DDN, communication is used both as a modelling construct and as a construct available for specifying implementation details. There are two mechanisms, message transfer and shared data, for communication description which are oriented toward these two different purposes. Message transmission is analogous to control signal processing in the hardware domain, and is primarily a modelling concept for software. In DDN, message handling models those aspects of component communication in which components require some knowledge of the rest of the system since the message pathways must be explicitly defined and fixed and the message types must be known to the participating processes. The shared data mode of communication is less of a modelling concept, and nearer to implementation (although it is a modelling concept used in general systems). This mode of communication does not require knowledge of how the rest of the system operates nor how the rest of the system

[1]This is obviously not as simple as it sounds, and research continues on ways to extract particular features of interest from this behavioral description. This is discussed more fully in a later section.

utilizes the data, although constraints may be imposed within the data definition itself.

The key modelling concepts underlying DDN are the class notion, the thorough distinction between the structure and the behavior of a system, and the concept of isolated knowledge ("need to know", "information hiding"). The class notion presumes that a system is comprised of many similar units, hence a model of a system should allow description of templates for the system parts. In DDN these templates (classes) have associated qualifiers (or parameters) which allow slight distinctions to be incorporated in class instances.

The designer of a large system exhibiting parallelism is constantly plagued by confusion between the structure and the behavior of the target system. The difficulty arises from the different levels of description the designer wishes to use, and from the different levels of abstraction inherent in a system description. For example, an operating system may be viewed as an object that has as its parts a scheduler, an interrupt handler, managers for primary and secondary storage, an I/O handler, and user jobs. But a description of any of these sub*objects* is in fact a template of a potential *process*. DDN provides the capability of illuminating these differences via formal specifications of structure and concomitant descriptions of behavior, either endogenous (associated with the structural specification) or exogenous (not linked to specific structure).

The concept of information hiding pervades modelling activities in many domains, under an assorted collection of pseudonyms. Basically, the principle states that component A should be (or is) only knowledgeable about the function(s) provided by component B, and should be (or is in fact) ignorant of the internal mechanisms by which component B performs. This concept is also a meta-modelling one, since designers should not be exposed to the inner workings of components other than the one currently being designed, else they are likely to take advantage of this *structure* when they are to be guided only by the behavior of other components. For the same reason, programmers should be guided by this principle [23], and only be given as much (behavioral) description of the rest of the system as is necessary to complete their part

of the system. DDN provides information hiding capabilities in several ways, notably via explicit behavioral description of processes, and by the definition of only certain aspects of a component to be "observable" to the rest of the system.

More specifically, in DDN descriptions a software system is decomposed into components of two types. *Subsystems* are those components which control and guide the performance of the system's processing, which operate (conceptually at least) in parallel and asynchronously with respect to other components, and which are individually capable of performing several activities at once.[1] *Monitors* are those components which also operate concurrently and asynchronously with respect to other components but which serve primarily as repositories of shared information and are individually capable of performing only a single activity at any point in time.[2] An auxilliary component of DDN descriptions is the *event* class. This is used for describing part of the system's operation in terms of a behavioral description which is exogenous to the system. One could, for example, give a description, in terms of several events, of an operating system from an external user's point of view, thus providing a description which is redundant with the internal description and orthogonal to the internal specification in the sense that it may establish associations among activities which occur within physically different parts of the system.

Structural descriptions are given in DDN by specifying the componentry and the operation of the components. For subsystems this consists of giving fragments, called *textual units*, of design description which describe the ports through which the subsystem sends and receives messages, the subcomponents (instances of other subsystem classes and of monitor classes) which comprise an instance of this class, and the algorithms for control processes which control the flow of messages among

---

[1]Those system components which execute concurrently and manipulate shared data objects are usually considered to be sequential processes, as defined in [24]. A subsystem is a more general object, being essentially a collection of sequential processes.

[2]The monitors of DDN are essentially those defined by Hoare [25]. To the usual definition of monitors, we have added constructs for behavior specification, patterned after constructs developed for the TOPD system [14].

the subcomponents and/or ports. As a very simple example[1], consider a data base which we will call a blackboard. Values stored in the blackboard may be inspected and modified. The blackboard serves as a repository for information and has processing components which "observe" the changes made to particular entries in the blackboard and notify the outside world whenever the entry is the subject of a modify operation. One possible DDN description of such a data base is:

```
[blackboard]:  SUBSYSTEM CLASS;

    QUALIFIERS; #_of_entries END QUALIFIERS;

    request:  IN PORT;
      BUFFER SUBCOMPONENTS;
        request_type OF [bb_request_type]
        END BUFFER SUBCOMPONENTS;
      END PORT;

    answer:  OUT PORT;
      BUFFER SUBCOMPONENTS;
        done_signal OF [bb_request_answer]
        END BUFFER SUBCOMPONENTS;
      END PORT;

    signal:  OUT PORT;
      BUFFER SUBCOMPONENTS;
        value OF [possible_values]
        END BUFFER SUBCOMPONENTS;
      END PORT;

    SUBCOMPONENTS;
      entries:  ARRAY [1::#_of_entries] OF [data_entry],
      watchers:  ARRAY [1::#_of_entries] OF[watcher]
      END SUBCOMPONENTS;

    END SUBSYSTEM CLASS;
```

The qualifier *#_of_entries* serves to parameterize the class definition to produce a description of a generic class of entities. The blackboard receives the commands requesting the inspection or modification of entries through the *request* port and notifies the outside world that the requested operation has been completed through the *answer* port. The condition checking portion of the blackboard sends notification that the "observed" entry was the subject of a modification operation out through the *signal* port. Each of the ports has a buffer associated with it where messages are stored on the way in and out. The

[1]More extensive examples are given in [20], [21], [26]-[31].

subcomponents are structures of instances of other classes which indicate the essential subparts of an instance of this class. The definition could also include the description of a control process which channels messages which come in through the *request* port to the appropriate entry in the *watchers* array as indicated by the content of the request message.

Not all of this definition would normally be visible to a member of the design team other than the designer who prepared it. Port and qualifier definitions would be visible but subcomponent definitions and control process bodies are not visible except when specifically made so by using the attribute "visible" in their declaration. This enforces the desirable result that in defining the use of a subsystem one may normally not rely upon any knowledge of its internal operation.

Monitors are described by defining their subcomponents, their states, and the actions which may be performed on the data objects they model. The states description in DDN contains a great deal of structural information. State variables are defined, and state subsets may then be defined as subsets of the cross product of the values of the state variables. (State subsets are always visible, but state variables are visible only when indicated by the attribute "visible".) In addition one may define an ordering relation on states and an equivalence relation between states of the monitor class and sets of states of the subcomponents of the monitor class. The definition of the monitor's actions or procedures includes the definition of the local subcomponents, the sequence of computation steps, the parameters of the procedure, and the transitions of the procedure, i.e., the state transitions that will occur as a result of the procedure's invocation.[1] As a simple example, the following definitions specify the monitor classes referenced in the previous example:

```
[possible_values]:  MONITOR CLASS;
  STATE SUBSETS; value1, value2, value3 END STATE SUBSETS;
  END MONITOR CLASS;
```

[1]It should be noted that all of the textual units of a DDN description are optional, thus a designer may specify as much or as little detail as desired in any given design step.

```
[bb_request_type]:  MONITOR CLASS;
  STATE SUBSETS; inspect, modify  END STATE SUBSETS:
  END MONITOR CLASS;

[bb_request_answer]:  MONITOR CLASS;
  STATE SUBSETS; value,done  END STATE SUBSETS;
  END MONITOR CLASS;
```

For each, all that has been defined are the externally observable states (which are essentially the same, in this case, as values for an enumerated type in Pascal).

In addition to structural descriptions in terms of templates for the subsystem and monitor classes, designers may additionally define the sharing of system components and how the ports of the components are connected together. Sharing may be described by instantiation control textual units which serve to indicate the "equivalence" of subcomponents which are otherwise (for ease or clarity of description) described as distinct. Connection textual units may be used to describe message communication pathways in terms of "transmission lines" between ports. (More extensive discussion of these constructs may be found in [32] and [33].)

Behavioral aspects of a system may be specified both pseudo-procedurally and non-procedurally. The major pseudo-procedural means of describing a subsystem's behavior is by giving models which define the subsystem's operation, in terms of message flow through the ports, as seen by external subsystems. For example, the operation of [blackboard] subsystems may be modelled as:

```
'[blackboard]:'
  servicer:  CONTROL PROCESS;
    MODEL;
      ITERATE
        RECEIVE request;
        IF request_type = modify
          THEN value SET TO value1 OR value2 OR value3;
               SEND signal;
               done_signal SET TO done;
          ELSE done_signal SET TO value;
          END IF;
        SEND answer;
        END ITERATE;
      END MODEL;
   END CONTROL PROCESS;
```

Nondeterministic control constructs are also provided in DDN so that models such as these may be even more abstract.

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

The most elementary method for defining endogenous events is to simply attach a label, called an *event identifier*, to some portion of the DDN description - an example appears later. Exogenous events definitions may not be associated with any monitor or subsystem definitions and are therefore defined via the event class textual unit. Once a set of events has been defined, a software system designer may specify intended behavior by describing the possible sequencing and simultaneity of event occurrences which would be acceptable during system operation. (A more complete description of DDN constructs for event definition and sequence specification is given in [34].)

Comparison With Formal Modelling Concepts

In this section we will relate the modelling concepts of DDN to the theory of modelling and simulation developed by Zeigler [46].

Each of the four basic concepts in that theory will be presented in terms of Zeigler's definition, its definition with respect to software systems, and its realization in DDN.

The *real system* is, in Zeigler's scheme, the source of data; moreover the data or the behavior is all that we can know directly about the system. In particular, we can have no direct knowledge of the structure of the system. Ignoring the profound epistomological implications of this approach, we note that this allows a precise distinction to be made between the system and a model of that system. The situation is somewhat murkier with respect to software. Considering a software system to be a collection of computer programs, we must infer that a software system is in fact purely a model of some other system since the code has no behavior - only a prescription of behavior.[1] The appropriate real system is the combination of the software and some hardware upon which the software instructions are executed.[2]

There remains another difference, namely our knowledge of the system. Since we have available the entire structure of the system (the code and the hardware principles of operation), it would appear that our systems are vastly different from those of Zeigler. The complexity of the software systems under discussion vitiates this point. In terms of human understanding, the one million lines of code of the IBM OS/360 operating system are as unknowable as a large ecosystem. In fact, it is because of this complexity that software systems may benefit from application of modelling theory!

In DDN, the real system is the target system being designed, i.e., the real system is a tabula rasa. We know something about the behavior of the system because we generate an exogenous behavior for it, but we do not know the structure of the target system, only a DDN model of it which admits of numerous realizations.

---

[1]This is a bit pedantic, but in fact there is some confusion in the literature regarding this point.

[2]We are still faced with a considerable dilemma since to be precise we would have to specify a particular hardware machine and we wish to consider only the software. Hence we assume hereon the existence of a universal order code into which all programs will be translated.

An *experimental frame* is a set of circumstances under which the real system can be observed or can be experimented with. It usually corresponds to some set of questions posed about the real system. Since knowledge about the real system is purely behavioral, an experimental frame may be defined by the entirety of I/O observations from it. In software, since one has the structure of the system to examine (the code), there exist static as well as dynamic experimental frames. Statically, one can examine scope of variables, scope of control, the flow graph of a program or the program schema. Dynamically, one may for example trace variables or resources, or concentrate on the working set or the page movement. In DDN, static experimental frames could include any subset of textual units across the system. Dynamic experimental frames could include any specified exogenous behavioral attributes pertinent to some subset of the subsystems. Note that (as in PPML) one may derive behaviors of the system (in terms of events and event sequences) without the necessity of exercising the system, and these derived behaviors could constitute an experimental frame, i.e., the endogenous behavior description on one level may be viewed as an exogenous behavioral description for the next lower level of detail. The pseudo-procedural models of control processes provide an alternative means in DDN of defining experimental frames since these serve as externally visible definitions of the behavior of a class of subsystems.

The *base model* is a model that accounts for all of the I/O behavior of the system. It is a model that is "valid" (see below) in all permissible experimental frames. The base model is never fully knowable since the set of possible experimental frames is infinite. In software the base model of the system is the code (see above) which is in principle fully known. In DDN the base model is also the code, but is never fully knowable because the DREAM machine is an abstract machine.[1]

A *lumped model* is an abstraction of the base model obtained from the base model by simplification, lumping of variables, or suppression of detail. The lumped model concept provides a framework for discussing

[1]The modelling language of DREAM, DDN, assumes the existence of a hardware machine with capabilities necessary to the modelling process, such as infinite resources including number of processors. As discussed previously, modelling is very different from programming.

the validity of system models. A lumped model is tested for validity with respect to an experimental frame, and is valid with respect to some experimental frame if it can generate all of the behavior of the experimental frame to within some specified tolerance. Thus many different lumped models may be valid with respect to the same experimental frame, and a given lumped model may be valid in some experimental frame and not in another one. In software, lumped models may be informal prose descriptions of the processes of the system and/or hierarchical organization charts or they may be formal predicate assertions defining system purpose. In DDN a lumped model of the system is a collection of textual units describing the internal operation of subsystems and monitors - its most important constituents are body textual units for control processes and procedures. Which textual units are chosen depends on the system properties under investigation, i.e., upon the experimental frame. Since DDN supports redundant system descriptions it is possible to have different lumped models which are valid with respect to a selected experimental frame. Note that this is absolutely necessary for a design system, in order to allow a designer to investigate alternative structures of the target system. The natural relationship between experimental frames and lumped models developed by Zeigler is of great value for an understanding of the modelling and design processes.

The modelling scheme advanced in [46] has proven to be very useful in guiding our thinking about software modelling. There are, however, two major distinctions between the (natural) systems discussed therein and software systems. The first is the nature of the *real system*, as already discussed. The other difference is that DDN is primarily a modelling system for the synthesis of systems rather than for abstraction from existing systems.[1]

A Design Methodology

DREAM is a design methodology - a collection of tools and procedures - useful for the design of complex, concurrent software systems. The intent in constructing DREAM has been to provide tools which support

[1]So far, our examples of the usage of DDN ([20],[21],[26]-[31]) have been in the traditional role of abstraction for understanding rather than for synthesis.

a variety of design procedures - one of the criteria has been to develop a medium for the experimental evaluation of the viability and efficacy of different design procedures. Most of the tools prepared so far, however, tend to make DREAM most useful in conjunction with a top-down design method in which designers iteratively prepare the DDN text describing a system by gradually elaborating the detail of the system's organization and operation. The completed design consists of a set of templates for the components of the system, a description of the behavior exogenous to the system, and an initial configuration of the system: the connectivity and the instantiation control. At each design step, the designer uses DREAM to either modify or augment the existing design retained in a DREAM database by changing, adding, or deleting textual units. It should be noted that the design language admits a hierarchical design, and that DREAM allows a system design to exist at varying levels of detail simultaneously.

DREAM provides a variety of tools to assist in managing this elaboration. Primary among these are an editing facility to assist in text preparation and a data base management facility to assist in the augmentation and modification of the design description. The other tools provided by DREAM allow the designers to obtain information about the characteristics of the system under design that are not explicitly stated in the text of the design description. The intent of these tools is to provide designers with some means of analyzing their design in tandem with its development.

Analysis During Design

A major reason for preparing models of software systems in some formally defined modelling scheme is the concomitant ability to algorithmically assess the system's characteristics. This is all the more important during design, since the analysis provides a preview of the characteristics of the eventual system and allows the designers to incrementally gain confidence in the validity of their decisions. In this section we discuss some approaches to analysis during design which are supported by a modelling scheme of the sort discussed in the previous section.

The analysis schemes that we have in mind are not necessarily fully

algorithmic. At any point during design, there exists a partially complete description, D, of the system under design. In addition, the designers have some idea of the system's characteristics along some dimension and we may denote this as C_i. Analysis at this point in the design is then representable as:

$$\rho_i(C_i, \delta_i(D))$$

δ_i derives the system's actual characteristics along the dimension i, to the extent that these are represented by the partial design D. ρ_i compares the system's actual and desired characteristics to determine whether or not they are acceptably related. In the analysis schemes that we discuss here, we require that δ_i be algorithmic but admit ρ_i that cannot be embodied in an algorithm.

The non-algorithmic nature of the analysis may arise from one of two sources. First, the comparison that needs to be done can be theoretically undecidable and thus an algorithm for ρ_i may not exist. Second, it may be impossible for the designers to specify C_i to the level of detail needed to apply the ρ_i comparison algorithm. In either case, the designers must use their intuition, experience and skill at formulating a logical argument in order to prepare a demonstration that the actual characteristics and the desired characteristics are or are not acceptably related.

This type of analysis may be called *feedback analysis* to connote that information concerning the system's characteristics is derived and presented to the designers for their assessment of the implications and subsequent application of any necessary corrective measures. This general class of approaches to analysis may be subdivided according to, first, the characteristic under analysis and, second, the approach used to implement δ_i. This leads to three major types of feedback analysis: feedback analysis of system organization, simulation-based feedback analysis of system behavior, and analytic feedback analysis of system behavior.

Feedback Analysis of System Organization

The simplest form of feedback analysis concerns the organization of the system, its physical characteristics. The point of this analysis is to present the designers with a description of the overall system

organization, derived from the designer-prepared description which explicitly describes only the components and their local inter-relationships.

The DDN description scheme supports two feedback analysis techniques of this type. The first derives a *connectivity* graph which shows the communication pathways which exist for message transfer among the components. The definition of this graph may be easily derived using the information contained in the port and connection textual units. But actually drawing the graph is not an easy task because of the problem of determining node placement so that a graph with a near-minimal number of arc crossings is produced.

The value of this feedback analysis technique lies in the fact that message communication connections are used in DDN descriptions to model the potential dependency of one component's operation upon the operation of another component. Thus the *physical* connections for message transfer actually reflect the *logical* connections for processing inter-dependencies.

The second feedback analysis technique supported by DDN for the analysis of system organization focuses upon the hierarchical organization of the system's componentry. This technique produces an *instantiation graph* in which arcs represent "part of" relationships among the components - the graph is essentially a map of the elaboration process followed in preparing the design. The definition of an instantiation graph is also easily derived, in this case from the subcomponents and instantiation control textual units. The graph itself is not difficult to prepare since it is a directed acyclic graph[1] rather than a general network.

The value of this feedback analysis technique is that it pictorially represents the component sharing within the designed system. For ease and naturalness of describing individual components, DDN allows the sharing of ports (and therefore buffers) and subcomponents to be described separately from the description of the components themselves. The instantiation graph therefore provides feedback which is helpful in discovering both system organization errors and potential conflicts due to sharing.

---

[1]DDN does not allow recursive subcomponent definitions.

Feedback analysis of system organization is primarily of value in checking that the design (at the current level of elaboration) has been correctly described in DDN. The DDN description scheme relies heavily upon relationships among the components such as "sends message to" and "is part of". Graphically presenting these relationships to the designers affords the opportunity to check that the local connectivity with respect to these relationships gives rise to a global system organization which coincides with the desired one.

Simulation-based Feedback Analysis of System Behavior

More critical than an understanding of global system organization is an understanding of the system's overall behavior. The designers have prepared a definition of each component which specifies how it should interact with other components. It is the intent of feedback analysis of system behavior to derive information concerning the overall behavior which results from the local interactions.

One way in which this information may be used is in verifying that the overall behavior is indeed what is required or intended. Another use is in checking the consistency of two descriptions of an interface, one in each of two interacting components. For either of these possible uses, it is necessary to gain information about the dynamic, run-time operation of the system and one way to do this is by simulation.

To effectively perform simulation, the designers must supply estimates of the time consumption characteristics of modelled operations and the behavioral characteristics of nondeterministic operations. This can be done in DDN after the scheme is extended[1] in a relatively straightforward manner. For example the description of the control process within the blackboard class can be augmented as follows:

---

[1]This extension is patterned after one used as the basis for a recent thesis on performance assessment during design [35].

```
servicer:  CONTROL PROCESS;
  MODEL;
    ITERATE
      RECEIVE request;
      NULL //100//;
      IF request_type = modify
        THEN value SET TO //3//
               value1 OR value2 OR value3 (.33,.33,.33);
             SEND signal;
             done_signal SET TO //1// done;
        ELSE done_signal SET TO //1// value;
        END IF;
      SEND answer;
      END ITERATE;
    END MODEL;
  END CONTROL PROCESS;
```

The notation //n// associates with a modelled operation a time, quoted in some arbitrary time units, for its execution. In general, a time distribution would be specified and techniques developed for simulation languages could be used for this purpose. Also, a more general scheme would allow the specification of timing characteristics of SEND and RECEIVE operations - for simplicity, we assume that this time is a constant known to the simulator. The notation (p,q,...) indicates the probabilities with which various options in a nondeterministic operation should be chosen.

With descriptions augmented in this manner, the model of the system under design may be exercised by a simulator to obtain an estimate of the distribution of derived statistics concerning the run-time characteristics of the system. In addition, the timing and probability estimates may be varied to parametrically investigate the sensitivity of these derived statistics to changes in the estimates.

Simulation-based feedback analysis can be used in several ways in addition to obtaining estimates of the time-related behavioral characteristics of a system. First, it can be used to investigate the effect of bounding the number of messages sent out through a port, such as a [blackboard]'s *signal* port, which have not been forwarded to a receiver. (This would, of course, require expanding our example description to reflect this bound.)

Second, simulation could be used to check for the violation of conditions upon the messages that may flow among the components. For

example, DDN would allow the description of the *signal* port to be augmented as follows:

```
signal:  OUT PORT;
  BUFFER SUBCOMPONENTS;
    value OF [possible_values]
    END BUFFER SUBCOMPONENTS;
  BUFFER CONDITIONS;
    value = value1 OR value = value2
    END BUFFER CONDITIONS;
  END PORT;
```

indicating that only messages with *value* being either *value1* or *value2* may validly flow out through the port. During simulation, observance of this condition could be checked whenever a SEND operation is done involving the *signal* port.

Finally, simulation may be used to investigate the effect of the system's eventual processing environment upon the system's time-related behavioral characteristics. The DDN description scheme is built upon the assumption that resources are infinite in the execution environment - memories are assumed to be unbounded and it is assumed that there is a sufficient number of processes for all the components to run in parallel, each on a dedicated processor. These assumptions are fine for the purposes of modelling and force the designers to conscientiously consider the control needed to operate in a restricted resource environment since no restrictions are levied by the modelling scheme itself. The simulator may be implemented so that it accepts the definition of a run-time environment, in terms of memory bounds and the numbers and types of processors, and takes account of the overhead incurred by sharing within the defined, resource-impoverished, run-time environment.

There are two major problems with a simulation-based approach to feedback analysis of system behavior during design. The most serious is the lack of data upon which to base the timing estimates and, to some extent, the probability estimates. Because the analysis is being done during design, the designers must rely upon their intuition, experience and knowledge to develop reasonable estimates. But, even good estimates of mean values will have large variances and these will cascade to give derived statistics with large variances.

This problem is somewhat aleviated by the fact that as the design

progresses, the designers will be able to get better estimates as the design moves to the more detailed level, closer to primitive operations about which the designers have better intuition. The designers may check that these better estimates are consistent with those used previously and revalidate previous designs if there is a significant inconsistency. This iterative approach to behavior assessment is consistent with typical approaches to software design.

The cascading of variances is sometimes tolerable when the designers are only interested in the derived statistics for the purposes of comparing design alternatives. In these cases, the variances of estimates used in corresponding portions of the alternative structures will typically be roughly the same and it will be the timing distribution means which differ significantly. Therefore, it may be possible to relate differences in the derived statistics to differences in the means of timing distributions and designers may therefore often make correct assessments as to the sensitivity of the system's characteristics to the operation of the individual components and correctly assess the differences among alternative system structures.

The effect of cascading variances can sometimes also be tolerated when absolute judgments rather than relative comparisions are being made. This can occur, for example, when the system is a real-time one and its specifications indicate some constraints that must be observed. In this case, if the ranges of the derived statistics lie within the constraint, then the magnitude of the variances makes no difference and the designers may validly conclude that the system observes the real-time constraints. When the ranges fall outside the constraints, it may be possible for the designers to parametrically assess the relationship of the variances of the individual timing estimates to determine local constraints which lead to satisfaction of the global constraints.

The second problem with simulation-based approaches to system behavior feedback analysis is not too severe and may be avoided by careful design of the simulator - this is the problem of simulating a multiple processor system. During simulation, operations will generally be sequentialized when in the eventual system they may be simultaneous. Problems stemming from this sequentialization may be avoided by having the simulator nondeterministically or pseudorandomly choose among

simultaneous operations. This is effective only if it is permissible to interpret "simultaneous" as "sequential but unordered" - this is most usually a valid interpretation in the case of software systems.

Analytic Feedback Analysis of System Behavior

Simulation is an important part of the theory advanced by Zeigler and others. DDN, however, is *not* a simulation language in the conventional sense for several reasons. First, current simulation languages do not allow behavioral specification, thus there can be no static analysis of behavior. That is, simulations produce behavior which is then examined, rather than allowing derivation of behavior from the structural description, as is possible in DREAM. Second, simulations are typically used to investigate performance characteristics rather than yes/no questions like system deadlock which should be answerable from a closed form behavioral description rather than from a (possibly infinite) number of simulation runs. Thirdly, simulation languages address process synchronization questions implicitly or when explicity by (simple) tie-breaking rules, and synchronization questions are at the heart of many problems with complex software systems, e.g., operating systems.

The analytic approaches supported by DDN are based upon the ability to derive from the structural model a description of behavior in the form of an algebraic expression over the set of event names for the system.[1] To give an example of this, we must augment the blackboard description so that there is a description of that part of the rest of the system which serves to activate the blackboard:

---

[1]One version of this derivation procedure is given in [18]. It is a variant of the Brzozowski method [36] for deriving a regular expression for the language recognized by a finite-state automaton.

```
[driver]:  SUBSYSTEM CLASS;

  ask:  OUT PORT;
    BUFFER SUBCOMPONENTS;
      request OF [bb_request_type]
      END BUFFER SUBCOMPONENTS;
    END PORT;

  listen:  IN PORT;
    BUFFER SUBCOMPONENTS;
      disposition OF [bb_request_answer]
      END BUFFER SUBCOMPONENTS;
    END PORT;

  exerciser:  CONTROL PROCESS;
    MODEL;
      ITERATE
        request SET TO inspect OR modify;
        RECEIVE listen;
        SEND ask;
        END ITERATE;
      END MODEL;
    END CONTROL PROCESS;

  END SUBSYSTEM CLASS;
```

Processes of this class interact with the blackboard in a coroutine fashion, at each interaction requesting either an inspection or a modification operation.

We must also add the definition of some events to the control process within the blackboard definition:

```
    servicer:  CONTROL PROCESS;
      MODEL;
        ITERATE
    hear:  RECEIVE request;
           IF request_type = modify
             THEN value SET TO value1 OR value2 OR value3;
activate:         SEND signal;
                  done_signal SET TO done;
             ELSE done_signal SET TO value;
             END IF;
 respond:  SEND answer;
           END ITERATE;
         END MODEL
       END CONTROL PROCESS;
```

By labelling some of the statements, we have demarcated events by which the behavior of the system may be described.

Suppose there are two processes of class [driver] and one of class [blackboard] and they are connected so that the *ask* ports of the drivers are connected to the *request* port of the blackboard and the *listen* ports of the drivers are connected to the *answer* port of the blackboard. Then we have one possible configuration for the system and may ask: what is the system's behavior in terms of the sequences of events which arise from its operation? An answer may be algorithmically obtained once an initial configuration of messages in the ports has been specified. In our example, an appropriate initial configuration is that the *listen* port has a single message and the other ports are empty. Using a notation called event expressions [18], the behavior of the system could be expressed as:

(hear(activate $\cup$ $\underline{\lambda}$)respond)*

This indicates that the *servicer* loops through the sequence of events: receive a request (hear); possibly send a message out through the *signal* port (activate); respond to the request (respond). (This particular behavior description is easy to derive by inspection, but that is because of the simplicity of the example.)

With this information, the designers are able to draw inferences as to whether the system is appropriately designed or not. They might, for example, not have realized that the blackboard is constructed so that it operates as a subroutine (with the undesirable effect that responses may be received by the wrong driver process) and wish to redefine it as a subsystem with two components, one for each of the drivers. In general, the designers may use the derived information to confirm that the system will behave as they intend. Of course, the level of confidence that they can attain in this way depends on the completeness of the design and the extent to which they have defined events that capture the behavior they are interested in analyzing.

A second use for the behavior expression derivation technique is in the guidance of the design process itself. In this case, the designers use the derived information to guide future design decisions. A particularly important use along these lines is the choice of synchronization mechanisms for the efficient coordination of parallel processes. A large number of synchronization mechanisms have been developed ([25],[37],[38])

and they are all essentially equivalent in power. The choice of a particular mechanism is frequently done on the basis of a designer's experience or the seeming appropriateness of the mechanism. But a wrong choice can lead to a much higher than necessary overhead in the system's operation. A totally appropriate choice can be made only if the designers know how the mechanism is *actually* used as well as how it is *intended* to be used.

In our example, the intended usage of the *signal* port would indicate that some bounded message transmission mechanism, such as communication semaphores [39], would be appropriate. But if event expression analysis were to uncover that this port was never used to send other than *value1* messages, then a more efficient solution would be to use a normal semaphore mechanism which merely counts the number of messages rather than actually storing them. This solution may later prove to be an incorrect one and the designers must be careful to periodically check that the solution is valid.[1] But at the point in design represented by our example, there is no reason to choose a more general, less efficient solution merely to be rid of the responsibility to carefully assess the effect of subsequent design decisions.

This is an example of a larger class of problems concerning the choice of appropriate strategies, policies and algorithms for controlling the interactions among a system's parallel components. By using DDN in conjunction with a top-down design method, the designers will have a much clearer idea of how a facility will actually be used before they design the components which provide the facility. They will therefore be able to tune the implementation of a facility to its use at the same time as designing its basic organization and operation.

A final way in which the expression derivation technique may be utilized is in performance assessment. To give an example, we must first create a description of a closed system by adding the definition of the

---

[1]Whenever a design is modified or elaborated, some check must be made to assure that the design description is internally consistent. The DREAM approach is to provide tools by which advisory information may be obtained and used in assessing the implications of a change in the design - these tools are not automatically applied when a change is made but are rather selectively applied by the designers.

class of processes that are activated by the blackboard when an "observed" entity is assigned a new value:

```
[handler]:  SUBSYSTEM CLASS;

  ask:  OUT PORT;
    BUFFER SUBCOMPONENTS;
      request OF [bb_request_type]
      END BUFFER SUBCOMPONENTS;
    END PORT;

  listen:  IN PORT;
    BUFFER SUBCOMPONENTS;
      disposition OF [bb_request_answer]
      END BUFFER SUBCOMPONENTS;
    END PORT;

  start:  IN PORT;
    BUFFER SUBCOMPONENTS;
      next_value OF [possible_values]
      END BUFFER SUBCOMPONENTS;
    END PORT;

  handle_it:  CONTROL PROCESS;
    MODEL;
      ITERATE
        RECEIVE start;
        ITERATE 0 OR MORE {{5}}TIMES
          request SET TO //1// inspect;
          SEND ask;
          RECEIVE listen;
          END ITERATE;
        END ITERATE;
      END MODEL;
    END CONTROL PROCESS;

  END SUBSYSTEM CLASS;
```

In the nondeterministic ITERATE statement, the notation {{m}} indicates the expected number of times that repetition will occur.

An event expression could then be derived which contains the timing and probability information [35]. We do not give this expression for two reasons. First, it is fairly complex and uses some constructs which we do not want to have to explain here. Second, and more important, in and of itself it does not explicitly give any information about the overall timing characteristics of the system - it is really just another representation of the system that happens to account more directly for the interactions among the parts of the system. In DREAM, the derived

expression is not displayed to the user but rather there are facilities for deriving summary statistics, such as means and variances. Through the interactive use of these facilities the designer could determine, for instance, that if the objects of class [handler] operate as described and if the timing estimates are an accurate reflection of the real system, then the operation of the object of class [driver] would be slowed down by about 20 percent, due to the interference introduced by the interactions of an instance of class [handler] with a [blackboard]. That this is true is not immediately obvious from inspection of the descriptions, but is rather easy to deduce from statistics about the execution and waiting times of the processes.

A Look to the Future

We have argued throughout that there exists a need to view complex software systems from a modelling and systems viewpoint, and discussed one rapprochement. There is some evidence that other research groups are beginning to use this approach to solving software design problems (e.g., [40]-[44]). We believe that there are many benefits to both the modelling and the software communities by collaboration.

The modelling community will find that software is rich in problems to be solved by formal means, and that software systems are readily available for study at all levels of system complexity. The software community could particularly benefit from answers in the following areas:

What is a natural domain for expressing the semantics of software description languages?

Can we describe a hierarchy of software system models à la Zeigler [46], and morphisms between them?

How may we formalize notions of subsystem connectivity and strength?

What sorts of analyses of behavior can we perform?

It should also be apparent that the primitives of DDN (or similar languages) are well suited for the description of systems other than software. The concepts of rigorous specification of subsystems, inter-system communication mechanisms, and behavioral specification are particularly interesting in this regard.

Conclusion

General systems theory has traditionally worked on two fronts,

system characterization, and the unification of models of systems. The DREAM system furnishes a testbed for the investigation of both of these issues with respect to complex software systems. There does not currently exist a formal theory of software system synthesis (or any other type of system for that matter)[1] nor a formal theory of software system organization, nor even a commonly accepted set of definitions or vocabulary for the discussion of such systems. It is hoped that DREAM is a rich enough system to provide the beginnings of answers to these questions (or at least a framework for the discussion of such issues), in addition to its intended practical value.

[1]This has been recognized as one of the major problems that general systems theorists should attack [45].

Bibliography

1. E. W. Dijkstra. Notes on Structured Programming. In Structured Programming by O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. Academic Press, 1972.
2. E. W. Dijkstra. A Discipline of Programming. Prentice-Hall, 1976.
3. R. W. Floyd. Assigning Meanings to Programs. Mathematical Aspects of Computer Science, J. T. Schwartz (ed.), 19, AMS, Providence, RI, 1967.
4. S. Alagic, and M. A. Arbib. The Design of Well-Structured and Correct Programs. Springer-Verlag, 1978
5. F. T. Baker. Structured Programming in a Production Programming Environment. IEEE Trans. on Software Engineering, SE-1, 2 (June 1975), 241-252.
6. G. Weinberg. The Psychology of Computer Programming. Van Nostrand Reinhold, 1971.
7. F. W. Zurcher and B. Randell. Iterative Multi-level Modelling - A Methodology for Computer System Design. Proc. IFIP Congress 68, Edinburgh, August 1968.
8. Tutorial on Software Design Techniques. P. Freeman and A. Wasserman (eds.). IEEE Computer Society, 1977.
9. N. Wirth. Program Development by Step-wise Refinement. Comm. ACM, 14, 4 (April 1971), 221-227.
10. Proceedings of an ACM Conference on Language Design For Reliable Software. D. B Wortman (ed.). SIGPLAN Notices, 12, 3, (March 1977).
11. B. H. Liskov and S. N. Zilles. Specification Techniques for Data Abstractions. IEEE Trans. on Software Engineering, SE-1, 1 (March 1975), 7-19.
12. O. J. Dahl and K. Nygaard. SIMULA--an Algol Based Simulation Language. Comm. ACM, 9, 9 (September 1966), 671-678.
13. B. Unger. Programming Languages For Computer System Simulation. SIMULATION, (April 1978).
14. P. Henderson, et. al. The TOPD System. Tech. Report 77, Computing Laboratory, Univ. of Newcastle upon Tyne, England, September 1975.
15. P. Henderson. Finite State Modelling in Program Development. Proc. 1975 International Conf. on Reliable Software, Los Angeles, April 1975.
16. J. L. Peterson and T. H. Bredt. A Comparison of Models of Parallel Computation. Proc. IFIP Congress 74, Stockholm, August 1974.
17. W. E. Riddle. The Hierarchical Modelling of Operating System Structure and Behavior. Proc. ACM 72 National Conf., August 1972.
18. W. E. Riddle. An Approach to Software System Modelling, Behavior Specification and Analysis. RSSM/25, Dept. of Computer and Communication Sciences, University of Michigan, Ann Arbor, July 1976.

19. W. E. Riddle, J. H. Sayler, A. R. Segal and J. C. Wileden. An Indroduction to the DREAM Software Design System. Software Engineering Notes, 2, 4 (July 1977), 11-23.

20. W. E. Riddle, J. C. Wileden, J. H. Sayler, A. R. Segal and A. M. Stavely. Behavior Modelling During Software Design. IEEE Trans. on Software Engineering, SE-4, 4 (July 1978).

21. W. E. Riddle, J. H. Sayler, A. R. Segal, A. M. Stavely and J. C. Wileden. A Description Scheme to Aid the Design of Collections of Concurrent Processes. Proc. Natn. Computer Conf., Anaheim, Calif., June 1978, pp. 549-554.

22. W. E. Riddle, J. H. Sayler, A. R. Segal, A. M. Stavely and J. C. Wileden, DREAM--A Software Design Aid System. Proc. 3rd Jerusalem Conf. on Info. Tech., Jerusalem, August 1978.

23. D. L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules, Comm. ACM, 15, 12 (December 1972) 1053-1058.

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

25. C. A. R. Hoare. Monitors: An Operating System Structuring Concept. Comm. ACM, 17, 10 (October 74), 549-557.

26. J. Cuny. A DREAM Model of the RC 4000 Multiprogramming System. RSSM/48, Dept. of Computer and Communication Sciences, Univ. of Michigan, Ann Arbor, July 1977.

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

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

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

30. J. Cuny. The GM Terminal System. RSSM/63, Dept. of Computer and Communication Sciences, Univ. of Michigan, Ann Arbor, August 1977.

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

32. W. E. Riddle. Hierarchical Description of Software System Organization. RSSM/40, CU-CS-120-77, Dept. of Computer Science, Univ. of Colorado at Boulder, November 1977.

33. W. E. Riddle. Abstract Process Types. RSSM/42, Dept. of Computer Science, Univ. of Colorado at Boulder, December 1977.

34. J. C. Wileden. Behavior Specification in a Software Design System. RSSM/43, Dept. of Computer and Comm. Sciences, Univ. of Michigan, Ann Arbor, in preparation.

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

36. J. A. Brzozowski. Derivatives of Regular Expressions. J.A.C.M., 11, 4 (October 1964), 481-494.

37. E. W. Dijkstra. Cooperating Sequential Processes. In Programming Languages, Genuys (ed), Academic Press, 1968.

38. P. Brinch Hansen. A Comparison of Two Synchronization Concepts. Acta Informatica, 1, (1972), 190-199.

39. H. J. Saal and W. E. Riddle. Communicating Semaphores. CS-71-202, Comp. Sci. Dept., Stanford Univ., Stanford, Calif., February 1971.

40. G. Estrin. Concurrent Software System Design, Supported by SARA at the Age of One. Proc. Third International Conference on Software Engineering, Atlanta, Georgia, May 1978.

41. G. Estrin. A Methodology for the Design of Digital Systems - Supported by SARA at the Age of One. Proc. Natn. Computer Conf., Anaheim, Calif., June 1978, pp. 313-324.

42. I. M. Campos and G. Estrin. SARA Aided Design of Software for Concurrent Systems. Proc. Natn. Computer Conf., Anaheim, Calif., June 1978, pp. 325-336.

43. A. A. Ambler. Gypsy: A language for Specification and Implementation of Verifiable Programs. Proc. of an ACM Conference on Language Design for Reliable Software, Software Engineering Notes, 2, 2 (March 1977), 1-10.

44. J. W. Baker, D. Chester and R. T. Yeh. Software Development by Step-wise Evaluation and Refinement. SDBEG-2, Dept. of Comp. Sci., Univ. of Texas, Austin, January 1978.

45. G. J. Klir. Trends in General Systems Theory. Wiley-Interscience, 1972.

46. B. P. Zeigler. Theory of Modelling and Simulation. Wiley-Interscience, 1976.