RSSM/84

AN EVENT-BASED DESIGN METHODOLOGY SUPPORTED BY DREAM

by

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

CU-CS-140-78 October, 1978

Abstract

A methodology for the architectural design of software systems is composed of three interrelated facilities. First, there must be some means of capturing the requirements for the system in some primarily non-procedural specification. Second, there must be some means for describing potential modularizations of the system in some primarily pseudo-procedural design which captures the essential detail concerning the modules' interfaces and their interactions. Finally, there must be some means of determining whether a system design appropriately meets the system specification. In this paper, we present a design methodology based on the use of event and event sequence descriptions. We first give a brief definition of the Design Realization, Evaluation and Modelling (DREAM) system and its description language as they relate to an event-based design method. Then we define the design method and give a simple example.

I. Introduction

Many methodologies have recently been advanced for the disciplined, orderly development of software systems. Each has as its basis a development method (or process for the gradual, evolutionary development of a software system) which serves to decompose the overall synthesis task into a sequence of smaller, more manageable steps. Many methodologies lend additional help to software developers in the form of guidelines, maxims or techniques that serve to support the method which they provide. A few offer help for the analysis which must be done to assure that the developed system delivers the functional capabilities which are required and does so within the performance and economic constraints which have been levied against the system.

A particularly effective way in which a development methodology may be delivered to software design practitioners is as a computerized support system. Systems of this type (e.g., [AmbA77], [BakJ78], [CamI78], [DavC77], [EstG78a], [EstG78b], [GooD77], [HenP75b], [MorM77], [PeaD73], [RidW78c], [RidW78d], [RobL75], [SnoR76], [TeiD77], [VHoE78]) provide a language for the precise, but abstract, description of systems at intermediate points in their development when completely detailed descriptions cannot be given. This language then allows the definition of various development techniques and the provision of computerized aids supporting these techniques.

In our own development support system, called the Design Realization, Evaluation And Modelling (DREAM) system [RidW78d], we have focused upon providing help for the development of concurrent software systems (that is, those systems having parts which may be perceived as operating synchronously and in parallel, even if the system actually executes in a uniprocessor environment). We have additionally focused upon providing analysis aid which is highly integrated with a top-down design method, so that system developers may gradually develop their confidence in the appropriateness of a system in tandem with developing the system itself.

In this paper we discuss the DREAM system and one of the development methods which it supports. In the next section, we narrow the scope of consideration to that phase of the software development life-cycle for which DREAM has been developed. We then discuss description languages in general and the DREAM description language in particular. Then, after outlining the DREAM system and discussing the types of computerized aid it provides, we outline a variant of the traditional top-down design method which the DREAM system provides and give a short example of its use.

II. Architectural Design Phase

The software system development process may be divided into three major phases.[1] Chronologically first among these is the requirements definition phase during which the users' requirements are expressed in terms of the system's expected functional capabilities as well as in terms of performance and economic constraints upon the system. Chronologically last is a phase which may be called the algorithm design phase, during which the system's processing algorithms and internal data structures are developed to the level of detail needed to permit compilation.

Intermediate to these two phases is a phase which we call architectural design. During this phase the system's gross organization is specified in terms of a hierarchically structured collection of modules, each of which plays some well-defined role in the delivery of the system's functional capabilities. In addition to the delineation of the modules, their interfaces and interactions are defined in order to specify the coordination needed to assure the delivery of the expected functional capabilities and the observance of the constraints which have been levied upon the system.

The systhesis task during design[2] is therefore the development of a modularization for the system and the definition of strategies for interactions among the modules. The associated analysis task is to

1. The distinction of phases which we use here is a simplified version of the many, diverse distinctions that have been made by others, for example by [PetL78].

2. For the remainder of this paper we use the term design to refer to architectural design.

assess the strategies with respect to the system's requirements with the intent of certifying that if the modules operate and interact as specified then the requirements will be satisfied.

III. Software Design Description Languages

The description task during architectural design is a modelling task that is only superficially similar to the traditional programming task encountered during the algorithm design phase. Modelling shares with programming the need to specify some details concerning the processing performed by the system. But this specification should be abstract with respect to specific mechanisms for implementing the processing. Stated differently, the specification should be expressed in requirements-oriented terms which reflect the _effect_ of system operation rather than implementation-oriented terms reflecting the system's actual operation and, therefore, the _cause_ of the observable effects. There is the need, therefore, for languages which are considerably different from programming languages and, in this section, we discuss the attributes of such _design description languages_.

Because of the need to highlight module boundaries and interactions, a design description language needs to be _behavior-oriented_ rather than operation-oriented. A behavior-oriented language supports the abstraction of a module by allowing the description of the effect of the module's operation without the description of the manner in which this effect is caused. One approach to abstraction is to allow descriptions that are _projections_ of the actual modules within the system. In a projection, the operational details of the module are suppressed, resulting in a description which focuses upon the module's overall behavioral characteristics. Another approach to abstraction is to allow a description which is _orthogonal_ to the system's implementation description in the sense that the description may form associations among the elements of the system that are completely different from those formed by the system's internal, physical organization. Thus, the modules may be merely logical entities rather than physical entities. A final approach to behavioral abstraction is to allow descriptions that are _non-proce-_

dural in that they describe the effect of the module's processing without specifying an algorithm for achieving the effect.

> Example: Assertions [ManZ74] are non-procedural projections of the modules they describe. They are orthogonal to the implementations of the modules in that they do not necessarily imply that one procedure must exist for each set of input/output relations defined by an assertion -- one generalized module could conceivably implement many sets of input/output relations.

The need to be able to investigate the interactions among the modules and the interplay among the various strategies implies that design description languages should be analysis-oriented. An obvious criterion is that descriptions be unambiguous, so that formal analysis techniques may be defined which derive information about the overall operation of a collection of interacting modules. Descriptions should also be outward-directed, specifying the characteristics of a module that are pertinent to its interactions with other modules, again so that characteristics of the overall operation of a collection of modules may be uncovered. Finally, descriptions should be redundant, specifying a behavior either from different points of view -- for instance, from the point of view of both the supplier or user of some processing facility -- or with respect to different sets of concerns -- for instance, with respect to deired properties of the overall operation as well as with respect to the operation of the individual modules.

> Example: Several recently defined programming languages -- such as Euclid [PopG77], CLU [LisB77], and Alphard [WulW75] -- satisfy many of these criteria. They each have a well-defined semantics and hence lead to unambiguous descriptions. Redundancy is allowed either through the incorporation of assertions which allow formal verification, such as in Alphard, or the use of data abstractions and strict typing which lead to useful compile-time checks, as in CLU. Outward-directed descriptions are generally allowed through axioms, as in Alphard and Euclid, which allow the effect of a sequence of operations to be deduced.

Design description languages should also be modification-oriented so that descriptions may be easily augmented, as new decisions are made, or returned to a previous state should decisions be reversed. Descriptions should be modular so that self-contained descriptions of different aspects of the system may be separately specified. They should also be hierarchical so that the descriptions may be iteratively elaborated. It is also usually desirable to allow incremental descriptions so that fragments of the overall description may be prepared in an arbitrary order. This is sometimes contradictory with the desire to have hierarchical descriptions, but only when it is required (sometimes unnecessarily) that the order in which portions of the description are developed corresponds to the nesting structure of the hierarchy.

> Example: Algolic programming languages admit hierarchical, modular descriptions and the current trend toward incorporating data abstraction constructs, as for example in CLU, enhances programming languages with respect to these criteria. Incremental descriptions are not easily achieved in block-structured languages (and are somewhat inconsistent with the idea of structured programming), but can be achieved in languages which use the idea of guarded commands [DijE76].

Finally, design description languages should be guidance-oriented in that they should allow descriptions which help guide the further design (and eventual implementation) of the system but do not overly constrain the design (or implementation) decisions remaining to be made. Descriptions should therefore be non-prescriptive such that they capture the decisions already made but do not prescribe the manner in which the resulting properties of the system should be achieved. Although non-prescriptive, descriptions should also be inward-directed, helping the designers to discover the options that are possible and feasible for ensuing decisions. It is sometimes beneficial to allow attribute descriptions in which various properties of the system are described positively by indicating the values that they may assume or negatively by indicating the values that are not to be allowed.

> Example: Assertions are non-prescriptive and frequently inward-directed since they describe a procedure's function as the conjunction of constraints and can therefore indicate different cases which should be handled.

Assertions could also be used to give attribute descriptions which specify the range of permissible values for each attribute.

The examples have indicated that contemporary programming languages have moved in the direction of design description. Partially this is because programming is also algorithm design; partially, it is in recognition of the need to perform analysis in order to produce reliable programs. But programming languages are rarely acceptable design languages because they are inherently operation-oriented and specify behavior only implicitly through procedures for achieving the behavior.

IV. The DREAM Design Language

As a basis for the DREAM system, which we discuss in the next section, we have developed a design description language called the DREAM Design Notion (DDN) which possesses many of the characteristics delineated in the last section. In this section, we discuss the major facilities available in DDN and indicate how these give rise to the various characteristics. The discussion here is brief and without examples. Some brief examples are given later -- others and more detailed discussions of the DDN facilities are given in the cited references.

DDN descriptions are of <u>classes</u> of components within a system [RidW77b]. Class-oriented descriptions were introduced in the SIMULA language [DahO66] and have been incorporated in a wide variety of computer-oriented description schemes ([HenP75a], [LisB75], [LisB77], [ParD72], [SnoR76], [WirN71b], [WulW75]). DDN allows parameterized descriptions so that some of the details of a member (or <u>instance</u>) of the class may be specified when the member is created. This facility allows the specification of many different types of variation among members of a class. For example, it allows the description of system components which have a variable number of subcomponents (such as the class of multiple-task programs), a variable number of output linkages (such as the class of message transmitters in a computer network), or variable types of subcomponents (such as the class of stacks).

Class definition constructs facilitate the hierarchical definition of a software system since the designers are encouraged to think of the system as decomposible into subcomponents which are members of a set of classes and which are themselves decomposible into subcomponents which are members of other classes. They also facilitate projection and orthogonality since a class serves to collect together all of the information pertinent to a collection of components and hence leads to descriptions which highlight those characteristics common to the collection rather than particular to its members. Finally, class definition facilities allow modular descriptions which may be incrementally developed since fragments of a class definition may be prepared in any order. In sum, class definition facilities primarily enhance the modification orientation of the language.

As implied above, a system must be hierarchically decomposed into its subcomponents to be described in DDN -- components are composed of sub-components which are composed of sub-subcomponents, etc. The tree-like system organization which results is often natural in describing the details of a system's algorithmic processing as indicated by the usefulness of structured programming ([DahO73], [WirN71a]) for program description. The tree-like organization is not appropriate, however, in situations in which sharing occurs either because of the nature of the system (as in shared database systems such as online reservation systems) or because of a desire to layer the system into levels of virtual machines (as in many current operating systems).

To allow non-tree-like organizations, DDN contains instantiation control constructs [RidW76b]. Using these constructs, the designers may specify that components which are described, for clarity of description, as distinct are actually a single component in the final system configuration. These constructs are similar in effect to equivalencing constructs such as found in the Fortran programming language, but do not allow components of differing types to be "overlaid". They are also similar in effect to the aliasing constructs typically provided by parameter passing mechanisms and capability-based addressing schemes, but are not intended to be effected by the transmission of addresses at run-time. Thus, the constructs provide for the description of sharing relationships that are determined by the system's configuration and which do not vary during system operation.

The instantiation control constructs contribute primarily to the inward-directedness of DDN since they allow explicit description of the system's eventual physical configuration. They also contribute by allowing the implicit description of all of the contexts in which a given component is to function, thus leading the designer of the component to consider how the component should be implemented so as to efficiently and effectively function in the various contexts.

The instantiation control constructs also lead to orthogonal, modular descriptions since logically different parts of the system may be separately described even though they share common subparts. The constructs may then be used to separately describe the sharing relationships amont the parts. This also tends to lead to incremental descriptions.

Components in a DDN description are considered to execute concurrently and asynchronously and there are two major types of components [RidW78c]. Subsystems are those components which (logically at least) operate concurrently, interact freely but in a way acceptable to both parties to any interaction, and collectively provide the system's functional capabilities ([RidW77c], [RidW78b]). Monitors also operate concurrently, but provide data storage capabilities and possess built-in mechanisms that may be used to synchronize the potentially conflicting demands placed by other components [RidW78c]. The DDN view of a system is therefore that it is composed of hierarchically organized collections of sequential processes which interact through shared data objects which individually contain the necessary synchronization code.

This parallel-world view of systems enhances the clarity of system descriptions in the following way. It is generally recognized that complex systems may be more easily comprehended if they can be broken into simpler, smaller parts which interact in well-defined ways [SimH62]. When this is done, then an understanding of the system's operation may be obtained by first understanding the parts and then understanding the interactions among the parts. As evidenced by recent texts on operating systems ([BriP77], [HabA76]) and recently developed methods for structuring artificial intelligence systems [FenR77], decomposition of a complex software system is easier when the parts operate concurrently and interact asynchronously.

In terms of the characteristics delineated in the previous section, the parallel-world view primarily facilitates modularization. It also allows abstractions to be more easily constructed, since it facilitates projection of the intermodule interactions and the overall effect of the interactions by allowing the suppression of detail concerning how the interaction is accomplished. Finally, it allows outward-directed descriptions which focus upon overall properties of the system which derive from the interactions among the components.

In collections of concurrently operating components which interact via shared data structures, it is common for the shared data structures to be used for message exchange. This form of interaction is distinguished in DDN and constructs are provided for explicitly describing the message generation and utilization characteristics of subsystems and the message flow among the subsystems. This message transfer view leads to descriptions which are orthogonal, projective and outward-directed, allowing as it does focus upon the logical characteristics of module interdependencies and interactions.

The final set of DDN description facilities are those for the non-procedural specification of overall system operation through the definition of behavior by the algebraic specification of sets of sequences of events [WilJ78]. Arbitrary events may be defined such as "program added to schedule queue " or "legal access made to database" and sequences of events which the designer wishes to allow may be specified by constructs ([RidW78f], [WilJ78]) which are extensions to those found in regular expressions and similar to those defined for path expressions ([CamR74], [HabA75]) and flow expressions [ShaA78].

The event sequence expression constructs of DDN have been included primarily to allow analysis. They provide a redundant, outward-directed description which may be checked for consistency against the operational descriptions of the component interactions [RidW78g]. As a descriptive scheme, however, they also provide for the non-procedural, non-prescriptive, projective and orthogonal description of the operation of collections of subsystems.

V. The DREAM Design Support System

Precisely-defined design description languages of the sort discussed in the previous section can serve as a basis for the delivery of several types of aid to software design practitioners. Tools, or computerized techniques providing this aid, are most effectively delivered as a set of coherent, integrated facilities within a design support system. These tools may be classified as follows:

<u>bookkeeping</u> tools provide aid in recording the current state of the development, in modifying and augmenting this record, and in returning it to a previous state if decisions are reversed

<u>supervisory</u> tools provide aid in assuring that practices and procedures that are deemed beneficial are actually followed

<u>management</u> tools provide aid in assessing progress and making resource allocation decisions

<u>feedback</u> tools provide aid in measuring the characteristics of the system under development for the purpose of detecting errors, gaining confidence in the appropriateness of the decisions that have been made, and guiding the further development of the system

The DREAM system is organized as depicted in Figure 1. (This organization has been patterned after that developed for the Tools for Program Development (TOPD) system [HenP75b].) Central to the system is a database in which information is stored in textual form, organized into <u>textual units</u> or fragments of description which individually specify different aspects of the system. Textual units are organized hierarchically -- for example, a textual unit describing a class of subsystems is composed of textual units which describe the subcomponents comprising each subsystem, the interfaces through which messages flow, the message transmission activities performed, and other aspects.

Several bookkeeping tools are provided as information insertion and retrieval mechanisms. Mechanisms for augmenting and modifying the information in the database on a textual unit basis facilitate the gradual evolution of the system's design while the current description is constantly available. Textual units which are to be added to the database (using ENTER) are prepared with the aid of a text editing facility (EDIT) and may be altered versions of units which have been copied from the database (using VIEW). Modifications to the database never result in the deletion of textual units which were previously entered -- thus designers have all of the previous versions of the design available and may back up to any previous point merely by retrieving the description which corresponds to that point and re-entering it, thus making it the current description. Problems caused by having an ever-expanding database are eliminated by an archiving mechanism (ARCHIVE) which may be used to copy the database to external, offline storage, retaining only the latest version of each textual unit in the database itself.

DREAM does not currently provide any supervisory aids since one of the intents of the system is to allow the experimental determination of the efficacy of various aids and hence none were pre-defined in the system. DREAM has, however, been designed with the aim of allowing these aids to be easily added to the system as mechanisms (within MANAGE) which control the flow of information into and out of the database. For example, documentation standards can be enforced by requiring that each textual unit be accompanied by a documentation unit when it is entered. Or, the principle of information hiding could be enforced by permitting designers access to only some of the information regarding components which they did not personally design. Or, a particular design method could be enforced by requiring that items in a design description be entered in a particular order.

Nor does DREAM currently provide any management aids, primarily because it is not clear what these aids should be. DREAM does provide a basis for management aids (such as GENERATE) as long as they can be formulated in terms of summary reports concerning the completeness or rates of change of the information in the database.

It is in the area of analysis aid that DREAM has been developed to provide the most aid. The analysis aids (such as ANALYZE) that are under development provide feedback analysis [RidW77a] in which information concerning the characteristics and properties of the system under design is derived and presented to the designers to be used in formulating rigorous arguments about the design's correctness or incorrectness. The only aid of this sort provided in the current DREAM system is a syntax checking mechanism, providing a check that is a necessary pre-requisite to any other analysis. The aids that are contemplated will span a broad spectrum (discussed in detail in [StaA77b]), from simple ones such as cross-reference generators to sophisticated ones which derive predictions of run-time characteristics by either analytic [RidW78g] or simulation [SanJ77] techniques.

VI. An Event-based Design Method

With the DREAM design support system and the DDN design description language, we have attempted to lend support to a variety of design styles rather than enforce our own style. A major reason is that we wish to be able to use the DREAM system to quantitatively measure the differences among various approaches to design. An equally important reason, however, is that we are not confident that we could define a method that was universally applicable across the full spectrum of systems which may be addressing using the DREAM system.

Our predilection toward top-down design methods is, however, quite evident in DREAM and DDN. We feel that these methods are the most effective to use since they allow designers the opportunity to introduce detail as warranted by consideration of a system's requirements rather than by consideration of aspects of the processing domain.

In using DDN to describe a variety of existing software systems ([CunJ77a], [CunJ77b], [RidW78a], [SegA77], [StaA77a], [WilJ77]), we have found that it admits an interesting and novel variation of the traditional top-down design method. We call this variation the <u>event-based design method</u> and it is the purpose of this section to describe it in some detail and make some observations as to its value.

As with most design methods, the event-based design method consists of a design step which is applied iteratively. This basic step is graphically represented in Figure 2. The first task is to identify, on the basis of the partial design prepared by previous steps, events which are pertinent to the aspect of the system which is to be addressed at this design step. Then, the partial design is used to develop constraints upon the occurrences of these events which are necessary to assure that the system operates as required or intended. The next task is then to define system components which can produce the delineated events, and the final task is to develop the interactions among the components which lead to the observance of the constraints.

Each design step therefore consists of the initial specification of the required behavior to be exhibited by the part of the system under consideration at this step. After this specification is prepared, the step is then completed by defining modules and module interactions which produce this required behavior.

As an example, consider the design of an on-board system for the in-flight monitoring of an aircraft's engines, and specifically that step at which we detail the interactions needed to call the attention of the pilot to the heating up of one of the engines. We hypothesize a partial design that includes the requirements of interest at this step, namely: there are four engines; a hot engine is to be signalled to the pilot by sounding an audio-alarm device; and signalling the existence of one hot engine should not block the recognition of another engine heating up and the signalling of this to the pilot. Notice that at this step we cannot focus exclusively on only the software parts of the system but must also consider non-software parts.

The events identified at this design step are defined, using the DDN language, in Figure 3. They correspond to four interesting "happenings" of concern at this step and reflect aspects of the system that an external agent could observe during system operation. In addition to three "primitive", undecomposed events, we have defined the overall event of handling a hot engine as a sequence of instances of the three primitive events.

The constraints of interest at this point are recorded in Figure 4, again using the DDN description technique. First, we have established that every time an engine heats up, this is noticed by the monitoring system and the alarm is sounded. The second constraint reflects the requirement that the handling of hot engines be able to be carried out concurrently in the case that another engine heats up when one hot engine is being handled. In this description, the constraints are expressed in terms of desirable properties of the sequences of events which occur during system operation. In approaching the description task this way, the desired effect is non-procedurally and non-prescriptively being specified.

Figure 5 gives a DDN definition of the components suggested by the events that have been defined. Notice the components are ones which operate concurrently and that both hardware and software components are specified.

The interactions among the components are defined, in message transmission terms, in Figure 6. Parts are defined and "plugged" together to establish the communication pathways, and the control process models specify the message transfer among the components. It should be emphasized that message transmission is being used to model the interactions among the components and that the interactions may actually take place using some other mechanism. Note that programming-language-like constructs are used to define the interactions. Also note that models of the hardware are specified to record the capabilities that are required -- the engines, for example, must have sensors and be able to "send out" status signals upon demand. Finally, note that the events identified at the beginning of this design step have been related to specific points during the operation of the system by labelling statements with the event identifiers.

We feel that the event-based design method outlined in the previous discussion formalizes the general practices of many software designers and that this formalization leads to several benefits. The greatest benefit comes from the provision of an important additional facility -- the definition of events and constraints. This facility allows the gradual reduction of requirements to precise statements oriented to the system's evolving modularization.

Another benefit, which we only briefly argue here, is the ability to verify the design as it evolves. The constraints developed at each step provide a redundant specification against which the newly designed operation of (part of) the system may be checked. In our example, we could use the pseudo-procedural descriptions given in Figure 6 as input to a simulator and thereby derive sequences of event occurrences. These sequences may then be compared to the desired sequences as stated in Figure 4. Of course, simulation is not in general sufficient since one needs to know all possible sequences of events that may arise. It is possible to derive a description of all possible sequences but the comparison is not, in general, feasible [RidW78g]. We are continuing our research in this area, seeking algorithms for subcases which span a wide variety of naturally occurring situations.

VII. Conclusion

We have described a variation of the traditional top-down design method that utilizes the concepts of event and event sequence definition. At each step, criteria for system modules being elaborated at that step are extracted from the existing design and precisely stated using the event and event sequence constructs. The step is then completed by detailing the interactions among newly-designed and existing modules necessary to satisfy the criteria. The opportunity then exists to analyze the new, partial design to assure that the criteria are met before proceeding to the next design step.

The event-based design method arose in the process of our assessment of the effectiveness pf the DREAM development support system and its description language, DDN. We have not extensively used it in design experiments, but have conducted two simple experiments in order to refine its definition. We have found it natural and easy-to-use, and feel that it both formalizes the practices of design practitioners and provides a basis for the integration of analysis with synthesis during software system design.

VIII. References

AmbA77 Ambler, A.L., Good, D.I., Browne, J.C., Burger, W.F., Cohen, R.M., Hock, 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.

BriP77 Brinch Hansen, P. The Design of Concurrent Processes. Prentice-Hall, Englewood Cliffs, N.J., 1977.

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.

CamR74 Campbell, R.A., and Habermann, A.N. The specification of process synchronization by path expressions. In Lecture Notes in Computer Science, 16, Springer Verlag, Heidelberg, 1974.

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.

DahO73 Dahl, O., Dijkstra, E., and Hoare, C.A.R. Structured Programming, Academic Press, N.Y., 1973.

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

DijE76 Dijkstra, E. A Discipline of Programming. Prentice-Hall, Englewood Cliffs, N.J., 1976.

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. 230-242.

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

FenR77 Fennel, R., Lesser, V.R. Parallelism in artificial intelligence problem solving: A case study of HEARSAY II. IEEE Trans. on Computers, C-26, 2 (February 1977).

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

HabA75 Habermann, A.N., Path expressions. Computer Sci. Dept., Carnegie-Mellon Univ., Pittsburgh, June 1975.

HabA76 Habermann, A.N. Introduction to Operating System Design. SRA, Chicago, 1976.

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.

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.

LisB77 Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C. Abstraction Mechanisms in CLU. Comm. ACM, 20, 8 (August 1977), 564-576.

ManZ74 Manna, Z. Mathematical Theory of Computation. McGraw-Hill, New York, 1974.

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, IS, 5 (May 1972), 330-336.

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

PetL78 Peters, L.J., and Tripp, L.L. A model of software engineering. Proc. 3rd International Conf. on Software Engineering, Atlanta, May 1978, pp. 63-70.

PopG77 Popek, G.J., et al. Notes on the design of Euclid. Software Engineering Notes, 2, 2 (March 1977), 11-18.

RidW77a Riddle, W.E. A formalism for the comparison of software analysis techniques. RSSM/29, Dept. of Computer and Comm. Sci., Univ. of Michigan, Ann Arbor, July 1977.

RidW77b Riddle, W.E. Hierarchical description of software system organization. RSSM/40, CU-CS-120-77, Dept. of Computer Sci., Univ. of Colorado at Boulder, November 1977.

RidW77c 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. Abstract monitor types. RSSM/41, Dept. of Computer Sci., Univ. of Colorado at Boulder, October 1978.

RidW78f Riddle, W.E. An Approach to software system behavior description. To Appear: J. of Computer Languages.

RidW78g 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.

SanJ77 Sanguinetti, J.W. Performance prediction in an operating system design methodology. RSSM/32 (Ph.D. Thesis), Dept. of Computer and Comm. Sci., Univ. of Michigan, Ann Arbor, May 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.

ShaA78 Shaw, A.C. Software descriptions with flow expressions. IEEE Trans. on Software Engineeering, SE-4, 3 (May 1978), 242-254.

SimH62 Simon, H.A. The architecture of complexity. Proc. Am. Phil. Soc., 106, (December 1962), pp. 467-482. Also in Simon, Sciences of the Artificial, MIT Press, Cambridge, 1969.

SnoR76 Snowdon, R. Interactive use of a computer in the preparation of structured programs. Thesis, Univ. of Newcastle upon Tyne, England, 1976.

StaA77a 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.

StaA77b Stavely, A.M. A survey of feedback aids in software design aid systems. RSSM/60, Dept. of Compuer and Comm. Sci., Univ. of Michigan, Ann Arbor, November 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.

WirN71a Wirth, N. Program development by step-wise refinement. Comm. ACM, 14, 4 (April 1971), 221-227.

WirN71b Wirth, N. The programming language PASCAL. Acta Informatica, 1, (1971), 35-63.

WulW75 Wulf, W.A., London, R.L., Shaw, M. Abstraction and verification in Alphard. In Schuman, S.A. (ed.), New Directions in Algorithmic Languages, IRIA (1975).

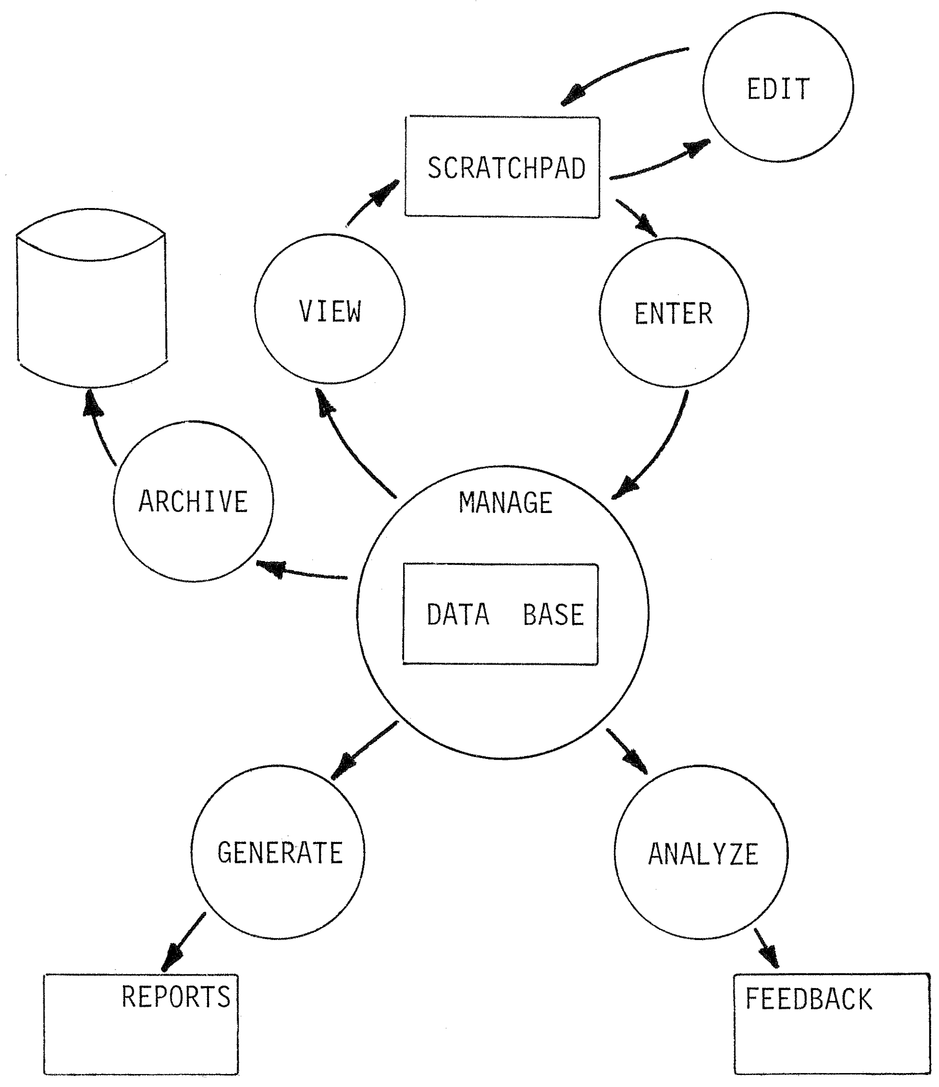

Figure 1.
Organization of the DREAM System

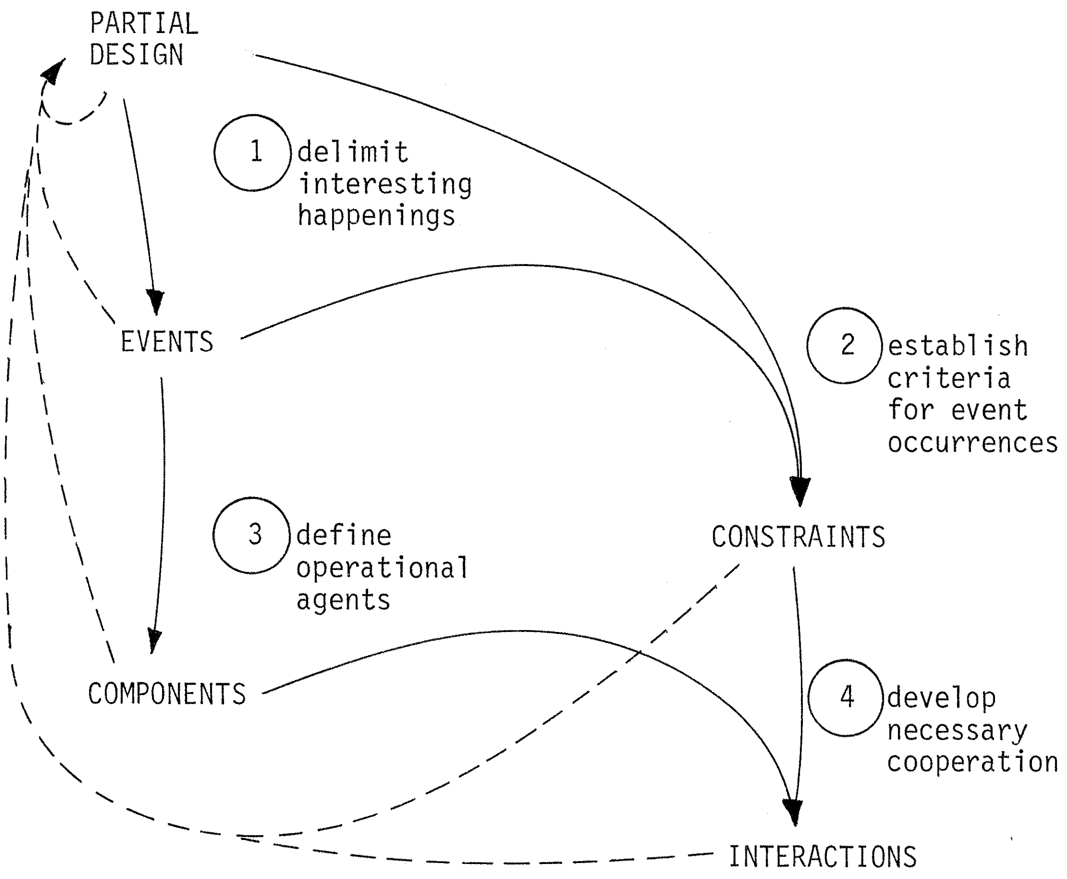
PARTIAL DESIGN
1 delimit interesting happenings
EVENTS
2 establish criteria for event occurrences
CONSTRAINTS
3 define operational agents
COMPONENTS
4 develop necessary cooperation
INTERACTIONS

Figure 2.

Basic Steps in Event-based Design Method

```
EVENT DEFINITION;
   heat_up: DESCRIPTION; temperature of an engine becomes greater
                 than or equal to 150 degrees F END;,
   ring: DESCRIPTION; audio-alarm device sounds END;,
   notice: DESCRIPTION; monitor recognizes that engine has heated
                 up END;,
   handle_hot_engine: SEQUENCE(heat_up, notice, ring)
   END EVENT DEFINITION;
```

Figure 3.

```
DESIRED BEHAVIOR;
   SEQUENCE(heat_up, notice, ring),
   POSSIBLY 4 CONCURRENT
     (SEQUENCE(heat_up, notice, ring))
   END DESIRED BEHAVIOR;
```

Figure 4.

```
SUBCOMPONENTS;
   engines ARRAY[1::4] OF [engine],
   monitors ARRAY[1::4] OF [engine_monitor],
   alarm OF [audio_device]
   END SUBCOMPONENTS;
```

Figure 5.

```
[engine_monitor]: SUBSYSTEM CLASS;
   request_status: OUT PORT;  END OUT PORT;
   receive_status: IN PORT;
      BUFFER SUBCOMPONENTS; signal OF [status_signal]
         END BUFFER SUBCOMPONENTS;
      END IN PORT;
   sound_alarm: OUT PORT;  END OUT PORT;
   observe: CONTROL PROCESS;
      MODEL;
         ITERATE SEND request_status;
                 RECEIVE receive_status;
                 IF signal = is_hot
                    THEN notice: NULL;
                                 SEND sound_alarm;
                    END IF;
                 END ITERATE;
         END MODEL;
      END CONTROL PROCESS;
   END SUBSYSTEM CLASS;

[status_signal]: MONITOR CLASS;
   STATE SUBSETS;  is_ok, is_hot  END STATE SUBSETS;
   END MONITOR CLASS:

[engine]: SUBSYSTEM CLASS;
   status_request: IN PORT;  END IN PORT;
   status_report: OUT PORT;
      BUFFER SUBCOMPONENTS; signal OF [status_signal]
         END BUFFER SUBCOMPONENTS;
      END OUT PORT;
   report: CONTROL PROCESS;
      MODEL;
         ITERATE RECEIVE status_request;
                 MAYBE heatup: SET signal TO is_hot;
                   ELSE        SET signal TO is_ok;
                   END MAYBE;
                 END ITERATE;
         END MODEL;
      END CONTROL PROCESS;
   END SUBSYSTEM CLASS;

[alarm]: SUBSYSTEM CLASS;
   ring_request: IN PORT;  END IN PORT;
   ringer: CONTROL PROCESS;
      MODEL;
         ITERATE RECEIVE ring_request;
                 ring: NULL;
                 END ITERATE;
         END MODEL;
      END CONTROL PROCESS;
   END SUBSYSTEM CLASS;
```

```
CONNECTIONS;
   FOR ALL i IN [1::4];
      PLUG (engines[i]|status_request, monitors[i]|request_status);
      PLUG (engines[i]|status_report,  monitors[i]|receive_status);
      PLUG (engines[i]|sound_alarm,    alarm|ring_request);
      END FOR;
   END CONNECTIONS;
```

Figure 6.