DREAM - An Approach to Designing Large Scale, Concurrent Software Systems

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

Introduction

The Design Realization, Evaluation And Modelling (DREAM) system is an automated support system for designers of large-scale, concurrent software systems. DREAM is intended to facilitate the orderly development of such software systems by supporting high-level, abstract design descriptions and the successive modification and elaboration of incomplete descriptions. DREAM also provides a basis for formulating arguments regarding the correctness of an evolving design at any stage during its development.

We begin this paper by presenting a viewpoint which has served to motivate our work in developing the DREAM system. Next we will describe DREAM itself, and its associated design language (the DREAM Design Notation or DDN). This discussion will highlight the major features of DREAM and the approach to software system design which DREAM supports. Finally, we will present a simple example of a DREAM design description. This should serve to illustrate both the specifics of the DREAM Design Notation and also the application of DDN to the description of concurrent software system designs.

A Viewpoint on Software Development

Our work on the DREAM system is predicated upon the fundamental observation that software development consists of the production of a succession of descriptions of the proposed system. Generally speaking, each description in this progression is more detailed than its predecessor. In particular, the final description in this progression should be fully detailed, since it is normally a piece of software intended for execution on some piece of computing hardware.

The various descriptions in the progression offer a succession of differing orientations, vocabularies and concerns in the representation of the proposed system. For instance, the final description in the progression is typically a collection of computer programs. This description may be considered to be oriented toward some particular (perhaps virtual) machine, is generally expressed in the vocabulary of one or more computer languages and is concerned with details of the manipulation of (virtual) machine-level data objects by functional units (e.g., words by registers, variables by arithmetic expressions).

It is both convenient and common practice to group the various descriptions which comprise software development into categories based upon their major orientation. We normally distinguish three such categories and designate them requirements specifications, design specifications and program specifications.

Requirements specifications are characterized as user-oriented. That is, descriptions in this category are essentially oriented toward the problem domain of interest to the eventual user of the software system. These descriptions are phrased in the vocabulary of the problem domain and are principally concerned with the overall functionality of the system, although performance and economic constraints are frequently also addressed.

Design specifications are referred to as implementation-oriented. That is, descriptions in this category are basically oriented toward the implementation domain of conceptual processing units. Such descriptions are formulated using the vocabulary of implementation, being expressed in terms of modules, their functions and their interactions. The central concerns of descriptions in this category are the modularity, module functionality and module interactions of the proposed system, although here, too, performance issues may be of interest.

Descriptions in the program specifications category may be considered execution-oriented. As was mentioned previously, these descriptions are oriented toward a specific (possibly virtual) machine, stated in the vocabulary of one or more programming languages and concerned with the detailed manipulations which embody computational algorithms.

Given that software development consists of producing a succession of descriptions, it is evidently desirable that those descriptions be produced in as orderly a manner as possible. Ideally, the generation of descriptions should proceed through a step-by-step progression with a clearly discernible, perferably formal, relationship between successive descriptions. In any case, orderliness can be maintained only if each succeeding description is thoroughly comprehensible. This demands that the descriptions be expressed using a representation which is precise and unambiguous, although possibly abstract and lacking some degree of detail.

Equally important to high-quality software development is continual assessment of the evolving description of the proposed system. At each step in the progression it is desirable to be able to rigorously inspect the current description for errors and inconsistencies. This makes possible the timely correction of mistakes and can prevent their propagation through to later stages of development where their repair may be significantly more difficult and costly.

DREAM

The Design Realization, Evaluation And Modelling system is intended to support software system development carried out within the framework outlined above. As its name implies, DREAM has been created primarily for use during the design specification phase of development, with the provision of evaluation (i.e., assessment) capabilities as a central goal and with modelling employed as the fundamental technique for design description.

DREAM focuses upon supporting design specification, particularly the specification of designs for large, complex, concurrent software systems. Especially for systems of this kind, the issues of primary importance during the design phase of software development are modularization, component functionality and component interaction. Thus DREAM is geared toward descriptions which highlight the delineation of system components (i.e., modularization), provide specifications of individual components' intended functions and contain a representation of the interaction patterns which those components will utilize in cooperating to achieve the system's overall function.

While offering bookkeeping aid by allowing a designer to store, retrieve or modify descriptions, supervisory aid by automatically enforcing design principles or practices considered beneficial by project supervisors[1] and management aid by supporting the automated generation of progress reports, DREAM is primarily intended to provide designers with analysis aid. Evaluation of even a partial, incomplete design specification can uncover errors at a sufficiently early point in development to permit their relatively easy and inexpensive repair. Thus DREAM seeks to help designers in assessing the appropriateness of decisions which are made and recorded in the descriptions produced during the design phase of development. Of particular interest during this phase are questions concerning modularization (e.g., does the description represent a reasonable decomposition of the system?), component functionality (e.g., are the descriptions of the individual components' functions reasonable? Will their coordinated, joint, concurrent operation produce the intended overall system behavior?) and component interactions (e.g., are the described interaction patterns acceptable? Are there synchronization problems inherent in this pattern of interactions among concurrently operating components?).

The nature of these questions suggests that feedback analysis is the most appropriate approach to design evaluation. Feedback analysis is not a fully automated verification technique and hence avoids the difficulties and complexities of verification. Rather, feedback analysis relies upon the designer to make a final determination as to the appropriateness of the decisions embodied in a particular software system description or description fragment. The technique is based upon having the system, in this case DREAM, first derive some information from the current version of the design specification. The derivation process consists of providing an explicit representation for information which was already implicit in the description. This may be done either through paraphrasing (i.e., generating a restatement of the description in other terms, such as a graphical restatement of a textual description) or animation (i.e., a symbolic or simulated 'execution' of the description). The derived information is then presented to the designer for evaluation. The designer can bring human insight and experience as well as an understanding of the specific problem domain to bear in assessing this feedback and determining the appropriateness of the current design specification. Thus feedback analysis as supported by DREAM attempts to automate only those aspects of evaluation which are relatively routine and mechanical to perform, leaving to the human designer those aspects more amenable to the application of such uniquely human (but little understood) skills as insight and contextualized reasoning.

Modelling is employed as the fundamental technique for design description in DREAM. A model provides an abstract, undetailed, but precise and unambiguous description and hence is appropriate for use in the design phase of software development. Moreover a model provides a conceptual representation, one which is not necessarily indicative of the details of a final implementation. This, too, is appropriate for design specification and is in keeping with our view [7] that a design description should be nonprescriptive, capable of describing the designer's intentions for system behavior without prescribing any specific means for achieving those intentions. Of course, to be useful during the design phase of software development, a model must provide a faithful representation of the modularity and functionality of the proposed software system. It

[1] For instance, documentation standards and practices such as information hiding [6] can be enforced through imposition of appropriate restrictions on the access and modifications which various design team members are permitted to make to the description database.

must also provide a sufficient basis for evaluation of a design specification, but only with respect to issues of functionality and modularity, not details of system composition. An appropriate choice of modelling primitives can insure that these desirable attributes are retained while still achieving a model which is an appropriately abstract representation of a software system design.

DDN

The DREAM Design Notation (DDN) is the language used for representing design specifications in DREAM. DDN is not a programming language, nor an extension of a programming language. Rather, it was expressly developed to support DREAM's focus on the design phase of software development, its concern with evaluation and its modelling approach to design description. The result is a language which facilitates modelling and is well suited to describing modularity, functionality and interaction, but which intentionally inhibits the description of algorithmic details so as to prevent the premature appearance of program specifications in a software development.

All DDN descriptions are expressed in terms of classes. Each class is a generic description of a type of entity, one or more instances of which might appear as components of the overall system being described. A class description is composed of textual units, fragments of descriptive text which specify various individual aspects of the description and which may be nested one within another. A class description may, for instance, be parameterized by a qualifiers textual unit so that individual instances of the class may differ from one another in minor details.

Three varieties of classes may be defined in DDN. A subsystem class is a generic description of a processing entity within a concurrent software system. Each subsystem, which could be thought of as a module, is conceptually a collection of sequential processes. Interaction among subsystems is represented as message transmission. This is, however, intended merely as a model for interaction and does not necessarily imply a message-based implementation, although message transmission is becoming increasingly popular as a basis for concurrent programming [1,4]. Functionality of subsystems is represented by abstract descriptions of message movement through the subsystem. Again, this description, which amounts to a specification of the input/output behavior for a subsystem, is a conceptual representation which does not necessarily reflect the actual details of an eventual implementation.

The second variety of DDN class is the monitor class. DDN monitors, which are generic descriptions of shared data repositories in a concurrent software system, are based on the monitor concept of Hoare [5]. Interactions with a monitor are represented by procedure calls, while a monitor's functionality is described abstractly in terms of states and the state transitions produced by procedure executions. Here again, DDN's monitor-based representation is purely a conceptual model and is in no way intended to prescribe the use of monitors as an implementation construct.

The final variety of DDN class is the event class. Event classes are used in describing aspects of the proposed system's behavior without reference to particular processing entities or data repositories. Such descriptions may relate to occurrences which are relevant to but not produced by the software system itself. Alternatively, they may be associated with components of the system which have not yet been described in terms of subsystems and monitors or may relate to occurrences not easily associated with any particular entity. Thus they enhance the flexibility and descriptive power of DDN by permitting the representation of facets of system behavior not representable using classes of the other two varieties.

Fully detailed presentations of the various constructs and features of DDN have appeared elsewhere [8,9,10,11,16]. Rather than replicate that information here, we will proceed to illustrate a variety of aspects of the language by means of a simple example of its use. The example also serves to demonstrate the DREAM approach to software design description.

An Example

We present a DREAM description for part of a simple airline ticket reservation system. The components which we describe here may be considered to constitute some fraction of the overall data processing system for an airline.

As the first step in designing our example system, a designer might choose to focus upon the interactions which will take place between the system and the ticket sales agents who will be using it. A possible result of considering this aspect of the system's design is the [ticket_manager] subsystem class shown in figure 1. The [ticket_manager] (DDN syntax requires that all class names be enclosed in square brackets) represents an initial design for the controller of the ticket reservation facet of the airline's data processing operation and for the system's interface to the various agents using the ticket reservation system. As indicated by the qualifiers textual unit, various instances of the ticket manager might differ from one another according to how many agents they were capable of serving. Thus in a completed DDN description of the ticket reservation system we might find several instantiations of the [ticket_manager], each serving a different number of agents and all conceptually operating concurrently.

The interfaces to the [ticket-manager] are specified using port definition textual units. A port may best be thought of as a hole through which messages may flow into or out of a subsystem. Thus the *agent_req* ports represent sources of agent request messages to the [ticket_manager] while the *agent_resp* ports provide the interface through which the [ticket_manager] may send response messages back to the agents, (Arrays are used here and elsewhere in DDN descriptions, just

```
[ticket_manager]:  SUBSYSTEM CLASS;
  QUALIFIERS; number_of_agents END QUALIFIERS;
agent_req:  ARRAY [1::number_of_agents]
                  OF IN PORT;
  BUFFER SUBCOMPONENTS;
    req_type OF [request_code],
    req_info OF [request_data]
    END BUFFER SUBCOMPONENTS;
  BUFFER CONDITIONS;
    req_type = ticket_req
    END BUFFER CONDITIONS;
  END IN PORT;
agent_resp:  ARRAY [1::number_of_agents]
                   OF OUT PORT;
  BUFFER SUBCOMPONENTS; answer OF [tm_response]
    END BUFFER SUBCOMPONENTS;
  END OUT PORT;
manager:  ARRAY [1::number_of_agents]
                OF CONTROL PROCESS;
  MODEL;
    ITERATE
      RECEIVE agent_req (MY_INDEX);
      SET answer (MY_INDEX) TO ok OR error;
      SEND agent_resp (MY_INDEX);
      END ITERATE;
    END MODEL;
  END CONTROL PROCESS;
END SUBSYSTEM CLASS;
```

Figure 1.

as they are typically used in other computer-related languages, as a notational shorthand for describing a collection of entities with identical attributes.) Buffer subcomponent textual units add detail to the interface description by specifying of what type (i.e., monitor class) each subfield of any message passing through the port must be. Buffer conditions further specify the designer's intentions by indicating the acceptable states for a subfield. Thus, agent request messages entering a [ticket_manager] subsystem through an *agent_req* port should only have subfields which are instances of the monitor classes [request_code] and [request_data] (both presumably described elsewhere in the DDN ticket reservation system description). Moreover, the designer has indicated that only a request of type *ticket_req* (i.e., a [request_code] subfield whose state is *ticket_req*) is acceptable in an incoming message. This information could later be used by the DREAM system to determine that the [ticket_manager] was being used correctly in the overall design, e.g., that it was not ever sent messages regarding such other airline data processing operations as payroll or maintenance scheduling.

Taken together, the various textual units within the port descriptions represent a complete (at the present level of detail), conceptual specification of the possibilities for interaction between a [ticket_manager] subsystem and other components in the airline data processing system. Thus they begin to delineate the nature of the new system's interfaces and its modularization, providing an important first step toward the design of the airline ticket reservation system.

Having begun with the specification of its interfaces and modularization, the designer might wish to continue development of the ticket reservation system by providing an initial, abstract, high-level description of its intended operation, i.e., by indicating the intended functionality of the [ticket_manager] subsystem. The functionality of a subsystem is described in DDN using control processes. A control process is an abstract representation of the potential flow of messages through a subsystem. As illustrated in figure 1, a control process description may include a model textual unit, which describes the subsystem's message transmission behavior as it might appear to an outside observer of the subsystem's activity. According to the model shown here, each *manager* control process indefinitely repeats a sequence of conceptual actions whose first step is to await the arrival of a message through the *agent_req* port associated with that particular control process. (The DDN keyword MY_INDEX is used to uniquely associate a pair of ports with each individual control process.) Next, the control process nondeterministically (from the point of view of an outside observer) chooses to return a response of either 'ok' or 'error', then sends a message containing that response and proceeds to repeat the sequence. Thus the model presents an abstract description of the subsystem's functionality, i.e., says what the subsystem does, without any details as to how that functionality is achieved. Such a description may be termed outward-directed, in that it offers a description of subsystem behavior which can be relied upon by others (e.g., designers of other components which might interact with a [ticket_manager]) while hiding information [6] about the internal operations which generate that behavior.

Together, the various textual units of the [ticket_manager] subsystem class shown in figure 1 comprise a description which focuses on precisely the issues of interest in a design specification. The description offers an abstract representation of component functionality and serves as an initial step toward a description of component interaction and system modularization. Of course, further elaboration of this description would be required before a design specification sufficient to serve as a guide to implementation would be achieved; in particular, more information is needed regarding the relationship of inbound and outbound messages. We will in fact indicate how such an elaboration might proceed later in this example.

At this point, however, the designer might decide that the next interesting design questions to be addressed are the form and function of the data repositories used in the airline reservation system, and thus might leave the [ticket_manager] subsystem for a while in order to address these issues. The [flight_seats] monitor class shown in figure 2 could be the designer's initial representation of the ticket system's information on seat availability for a particular flight. Its qualifiers textual unit indicates that individual instances of [flight_seats] will differ only as to the specific flight to which they correspond. (Of course, later elaboration may introduce additional differences such as number of seats available, but at the current level of detail such distinctions are irrelevant.) In all probability

```
[flight_seats]:  MONITOR CLASS;
  QUALIFIERS; flight_# END QUALIFIERS;
  STATE SUBSETS; full, not_full END STATE SUBSETS;
  begin:  PROCEDURE; END PROCEDURE;
  select:  PROCEDURE; END PROCEDURE;
  confirm: PROCEDURE;
    TRANSITIONS;
      not_full --> full,
      not_full --> not_full
      END TRANSITIONS;
    END PROCEDURE;
  reject:  PROCEDURE; END PROCEDURE;
  done:  PROCEDURE; END PROCEDURE;
  EVENT DEFINITIONS;
    choice:  SEQUENCE (select, OR
                       (confirm, reject)),
    trans:  SEQUENCE (begin, REPEAT (choice),
                      done),
    END EVENT DEFINITIONS;
  DESIRED BEHAVIOR;
    POSSIBLY CONCURRENT (trans,
             [flight_seats]|trans),
    MUTUALLY EXCLUSIVE (choice, choice)
    END DESIRED BEHAVIOR;
  END MONITOR CLASS;
```

Figure 2

there will be numerous instances of [flight_seats] in the completed ticket reservation system design, all potentially being accessed concurrently.

Interfaces to the monitor class are described by procedure textual units, each procedure being assigned a name. (Parameters may also be associated with monitor procedures, although that aspect of DDN is not illustrated in the present example.) The usual conventions for mutually exclusive usage of a monitor procedure, as defined by Hoare [5], apply to DDN monitor procedures. Interaction with a monitor is then described by procedure invocations, as we shall see later in this example.

Monitor functionality is described in terms of states and state transitions. At the current level of detail, the [flight_seats] monitor class can be described using two state subsets, *full* and *not_full*. It is not necessary that DDN state subsets be disjoint, although the two in this example presumably are. It is, however, possible to further coordinatize a monitor's state space using state variables, each of which can assume one of a disjoint set of values [11].

An abstract, conceptual and outward-directed description of each monitor procedure can be formulated using the transitions textual unit. Transitions may be viewed as a precondition/post-condition specification for procedure behavior. The left-hand sides of the transitions stipulate the set of state subsets which the monitor class may be in when the procedure is invoked; invocation when the monitor is in any other state subset is not anticipated by the designer. The right-hand sides of the transitions indicate what state subsets the monitor may be put into as a result of a procedure activation. In keeping with their role as outward-directed representations of procedure behavior, transitions may be nondeterministic. In our figure 2 example, for instance, invocation of the *confirm* procedure is only expected when the [flight_seats] monitor class is in its *not_full* state subset, but such an invocation might result in the monitor's either remaining in the *not_full* subset or transiting into the *full* state subset.

The figure 2 example also provides an illustration of DREAM's nonprocedural behavior description capabilities, which are based upon the definition of events. Events may be defined in a number of different ways in DREAM [16]. In particular, every procedure definition in a DDN description implicitly defines an event whose name is the same as the procedure's name and which corresponds to an activation of the procedure. Thus the [flight_seats] monitor class implicitly defines the events *begin*, *select*, *confirm*, *reject* and *done*. Events may also be defined using an event definitions textual unit. In particular, within an event definitions textual unit events may be defined by event sequence expressions, which are expressions describing one or more sets of sequential or concurrent occurrences of other events. For instance, in figure 2 the *choice* event is defined to consist of an occurrence of the *select* event followed by an occurrence of the *confirm* event or an occurrence of the *select* event followed by an occurrence of the *reject* event. Similarly, the *trans* event is defined to consist of an occurrence of the *begin* event followed by any number of occurrences of the *choice* event followby an occurrence of the *done* event.

The principal use of events in DREAM descriptions occurs in desired behavior textual units. A designer may use a desired behavior textual unit to express intended restrictions on the behavior of the system being designed. These restrictions are in no way enforced by DREAM; they merely reflect the designer's intentions and provide for the possibility of assessing the design description by checking to see whether those intentions were obeyed in the actual behavior described by the subsystems and monitors. In the example of figure 2, the designer has indicated two intentions with respect to usage of [flight_seats] monitors. The first concurrency expression in the desired behavior textual unit indicates the designer's willingness to permit seats on different flights to be reserved simultaneously (i.e., a *trans* event in one [flight_seats] monitor may occur concurrently with a *trans* event in any other [flight_seats] monitor). The second concurrency expression indicates the intended restriction that two *choice* events within a single [flight_seats] monitor should not be allowed to occur simultaneously (i.e., actual decisions regarding seats on a particular plane must be made in a mutually exclusive fashion). These intended restrictions are stated by the designer of the [flight_seats] monitor, the person in the best position to know what restrictions are required (the person who would, for example, know that mutually exclusive occurrence of the entire transaction is not necessary). However, these behavioral restrictions must actually be realized by designers whose components use the [flight_seats] monitor, since desired behavior descriptions are nonprescriptive. DREAM can help in ascertaining whether or not the

intentions so expressed have been honored, but will not in any way actually enforce those intentions.

We continue our example by illustrating how the designer might carry out one step in an elaboration of the airline ticket reservation system design. The step begins with the three minor modifications to the [ticket_manager] subsystem class shown in figure 3. The first change (quoted prefixes are used in DREAM to indicate what part of the current design description is to be modified) is the addition of a subcomponents textual unit to the [ticket_manager] description. A subcomponents textual unit describes the internal composition of objects in a class, in this case indicating that each [ticket_manager] contains a collection of [flight_seats] monitors, one for each flight on which it is able to reserve tickets. Similarly, the second change shown in figure 3 is the addition of a local subcomponents textual unit to the *manager* control process of the [ticket_manager], indicating that each control process in the subsystem will have available to it a monitor which can be used essentially as an integer variable taking values in the range 1 to #_of_flights. Both these changes may be termed inward-directed since they reflect details of the internal operation of the subsystem rather than providing purely behavioral information about the [ticket_manager] class. The third change shown in figure 3 is the replacement of the qualifiers textual unit of the [ticket_manager] class by an updated version which recognizes that #_of_flights is now a parameter of the subsystem class.

```
'[ticket_manager]:  SUBSYSTEM CLASS' SUBCOMPONENTS;
    list:  ARRAY [1::#_of_flights] of
                 [flight_seats (MY_INDEX)];
 END SUBCOMPONENTS;

'[ticket_manager]:  SUBSYSTEM CLASS;
                    manager:  CONTROL PROCESS'
                    LOCAL SUBCOMPONENTS;
    fl_# of [1::#_of_flights]
 END LOCAL SUBCOMPONENTS;

'[ticket_manager]:  SUBSYSTEM CLASS'
  QUALIFIERS; number_of_agents, #_of_flights
    END QUALIFIERS;
```

Figure 3

The elaboration step continues with the addition of the control process body shown in figure 4 to the [ticket_manager] subsystem class description. A control process body provides the inward-directed description which corresponds to the outward-directed description of a control process model. That is, the body indicates what manipulations of the subcomponents of the subsystem are required to produce the behavior specified by the model. Thus, for instance, we find invocations of the *begin*, *select*, *confirm*, *reject* and *done* procedures applied to specific instances of the [flight_seats] monitor class in the *list* subcomponent.

We conclude our example by briefly considering representative samples of the type of design

```
'[ticket_manager]:  SUBSYSTEM CLASS;
                    manager:  CONTROL PROCESS'
    BODY
      ITERATE;
        RECEIVE agent_req(MY_INDEX);
        IF req_info(MY_INDEX) = garbled
          THEN SET answer(MY_INDEX) TO error;
          ELSE req_info(MY_INDEX).get_fl(fl_#)
            list(fl_#).begin;
            ITERATE;
              list(fl_#).select;
              MAYBE list(fl_#).confirm;
                ELSE list(fl_#).reject;
              END MAYBE;
            END ITERATE;
            list(fl_#).done;
            SET answer(MY_INDEX) TO ok;
        END IF;
        SEND agent_resp(MY_INDEX);
      END ITERATE;
    END BODY;
```

Figure 4

analysis supported by DREAM. As indicated earlier, the fundamental approach is feedback analysis based upon consistency checking. Thus, for instance, DREAM can be used to ascertain that the behavior resulting from the operations specified for the *manager* control process body (figure 3) corresponds to the description of the *manager* control process' behavior as given by its model (figure 1). By reporting the discovery of this consistency between the inward-directed and outward-directed descriptions of a design component, DREAM can help to bolster the designer's confidence in the correctness of this elaboration step in the design of the airline reservation system.

A second form of feedback analysis is based upon checking for consistency between the evolving description of a design's components and the designer's intentions regarding behavioral restrictions as expressed through desired behavior textual units. Using techniques similar to those presented in [13], an event sequence expression representing the behavior of the *manager* control process in terms of *trans* and *choice* events can be derived from the control process body. Upon comparing this derived expression with the concurrency expressions of the desired behavior textual unit associated with the [flight_seats] monitor class (figure 2), the designer would discover that the mutual exclusion between *choice* events which was recorded as desirable has not been realized in the [ticket_manager] subsystem. Thus feedback provided by DREAM permits the designer to discover an inconsistency, thereby exposing an oversight or an error in the design as it is currently described. Uncovering the flaw at this early stage in the development of the airline reservation system makes possible a timely and relatively easy correction of the problem, preventing it from persisting through further developmental stages to become a major repair job later.

Conclusion

In this paper we have attempted to show how the Design Realization, Evaluation And Modelling system and the DREAM Design Notation can facilitate the orderly development of large, concurrent software systems. Based upon numerous example usages of DDN [2,3,12,13,14,17], we believe that the notation is well-suited for use in designing such systems and that the assessment facilities of DREAM can be of significant value to the developers of large-scale, concurrent software.

Acknowledgements

The DREAM system was developed jointly by William E. Riddle, John H. Sayler, Alan R. Segal and the author. Allan M. Stavely was also instrumental in DREAM's development, while Mark Welter and Dirk Kabcenell made significant contributions at various stages in that development. DREAM has also benefited from discussions which we have had with Jan Cuny, Victor Lesser, Carolyn Steinhaus and Pamela Zave.

References

[1] A. Ambler et al. GYPSY: A Language for Specification and Implementation of Verifiable Programs. ICSCA-CMP-2, Certifiable Minicomputer Project, Univ. of Texas, Austin, January 1977.

[2] J. Cuny. A DREAM Model of the RC4000 Multiprogramming System. RSSM/48, Dept. of Computer and Comm. Sciences, Univ. of Mich., Ann Arbor, July 1977.

[3] J. Cuny. The GM Terminal System. RSSM/63, Dept. of Computer and Comm. Sciences, Univ. of Mich., Ann Arbor, August 1977.

[4] J. Feldman. High Level Programming for Distributed Computing. Comm. ACM, 22, 6 (June 1979), 353-368.

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

[6] D. Parnas. Information Distribution Aspects of Design Methodology. Proc. IFIP Congress 71, Ljubljana, August 1971, pp. TA-3-26-TA-3-30.

[7] W. Riddle and J. Wileden. Languages for Representing Software Specifications and Designs. Software Engineering Notes, 3, 5 (October 1978), pp. 1-5.

[8] W. Riddle, J. Wileden, J. Sayler, A. Segal and A. Stavely. Behavior Modelling During Software Design. IEEE Transactions on Software Engineering, SE-4, 4 (July 1978), 283-292.

[9] W. Riddle. Hierarchical Description of Software System Structure. RSSM/40, Dept. of Computer Science, Univ. of Colorado at Boulder, November 1977.

[10] W. Riddle. Abstract Process Types. RSSM/42, Dept. of Computer Science, Univ. of Colorado at Boulder, December 1977 (revised July 1978).

[11] W. Riddle, J. Sayler, A. Segal, A. Stavely and J. Wileden. Abstract Monitor Types. Proc. Specifications of Reliable Software, Boston, April 1979.

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

[13] W. Riddle. An Approach to Software System Modelling and Analysis. J. of Computer Languages, to appear.

[14] A. Segal. DREAM Design Notation Example: A Multiprocessor Supervisor. RSSM/53, Dept. of Computer and Comm. Sciences, Univ. of Mich., Ann Arbor, August 1977.

[15] A. Stavely. DREAM Design Notation Example: An Aircraft Engine Monitoring System. RSSM/49, Dept. of Computer and Comm. Sciences, Univ. of Mich., Ann Arbor, July 1977.

[16] J. Wileden. Behavior Specification in a Software Design Aid System. RSSM/43, Dept. of Computer and Information Science, Univ. of Massachusetts, August 1978.

[17] J. Wileden. DREAM Design Notation Example: Scheduler for a Multiprocessor System. RSSM/51, Dept. of Computer and Comm. Sciences, Univ. of Mich., Ann Arbor, October 1977.