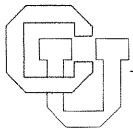
Languages for Representing Software Specifications and Designs

William E. Riddle
Jack C. Wileden

CU-CS-127-78



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

LANGUAGES FOR REPRESENTING SOFTWARE SPECIFICATIONS AND DESIGNS

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

and

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

CU-CS-127-78 May 1978

Abstract:

We consider the nature of software system specifications and designs, then survey the languages used in representing them. We emphasize the utility of language-based representations as a foundation for computerized tools which provide aid during software system development. The survey is based upon a classification of the languages according to their underlying representational constructs.

Introduction

The fundamental activity during the development of a large, complex software system is the successive production of differing descriptions of the system. These successive descriptions differ principally in their orientation, vocabulary and organization. For example, the succession of descriptions typically culminates in a (collection of) computer program(s) forming a description oriented toward a particular (possibly virtual) machine, expressed in the vocabulary of some particular computer language and organized into the units mandated by that language (e.g., control sections, subroutines or procedures). In this paper, we focus upon the first two types of descriptions which arise during the development of a software system, which we designate as specifications and designs, and upon the activity of fashioning the second type of description from the first, which we call the design process.

Normally, the first type of description formulated during a software system development project is a specification for the system. A specification describes the proposed system in terms of the problem(s) which it is to solve, indicating the intended functions of the system and any policies which should govern system behavior. Specifications may also impose economic or performance constraints and suggest desirable attributes for proposed systems. Thus, a specification may be characterized as a problem domain oriented description of a software system. The vocabulary of such a description is generally that common to the problem domain, and a specification is typically organized, if at all, into units which are natural to the problem domain.

The next distinguishable type of system description to emerge during a software system development project may be called a design. As contrasted with specifications, a design describes a software system less in terms of the problem which it addresses and more in terms of a proposed solution to that problem. A design presents this solution as a collection of conceptual processing units, or modules, specifying outlines for their individual activities and indicating the intended interactions among them. Thus, where specifications are problem domain oriented descriptions, designs are implementation or software domain oriented descriptions. The vocabulary of such a description is generally that common to the software and implementation community. So although designs, like specifications, may impose constraints and suggest desirable attributes for a proposed software system, these constraints and attributes normally are specifically related to properties of module activity and interaction. The organization of a design also reflects the underlying orientation toward implementation concerns, since it typically is based upon units (e.g., modules or collections of modules) which are natural to the implementation and software domain.

We use the phrase design process to denote the activity of creating a software system design based upon, and purportedly realizing, the system's specification. This process, whereby a description in the problem domain gives rise to a description in the implementation domain, is at once the most crucial and the least understood of any phase in the development of a software system. Ideally, in order to minimize errors and facilitate production of a design, the process should proceed in a series

of orderly and verifiable steps. Yet, although both specifications and designs may exist and be incrementally elaborated within broad ranges of representational detail (as suggested by Figure 1), the transition from one domain to the other is still typically abrupt and error-prone.

Any systematic approach to software development clearly requires one or more languages in which to formulate unambiguous, well-defined statements of software specifications and designs. These two types of descriptions, with their differing orientations, demand different sets of language capabilities and constructs to support the natural expression of salient properties within their respective domains. When these distinct capabilities and constructs reside within two separate languages, the design process involves producing a description in one language which somehow reflects a description in the other. If both sets of constructs and capabilities are contained within a single language, then the design process can potentially be carried out in a more integrated and gradual manner. In either case, the two sets of constructs should preferably be related, either formally or informally. Such a relationship between constructs is necessary to any verification of design correctness and can also be of use in structuring and organizing the design process.

In the remainder of this paper, we shall discuss a number of languages which have been developed for describing software system specifications and designs. We first briefly consider some of the benefits which accrue through the use of such languages and mention some language attributes which are particularly important for the design process. Then we present a classification of the currently existing specification and design languages,

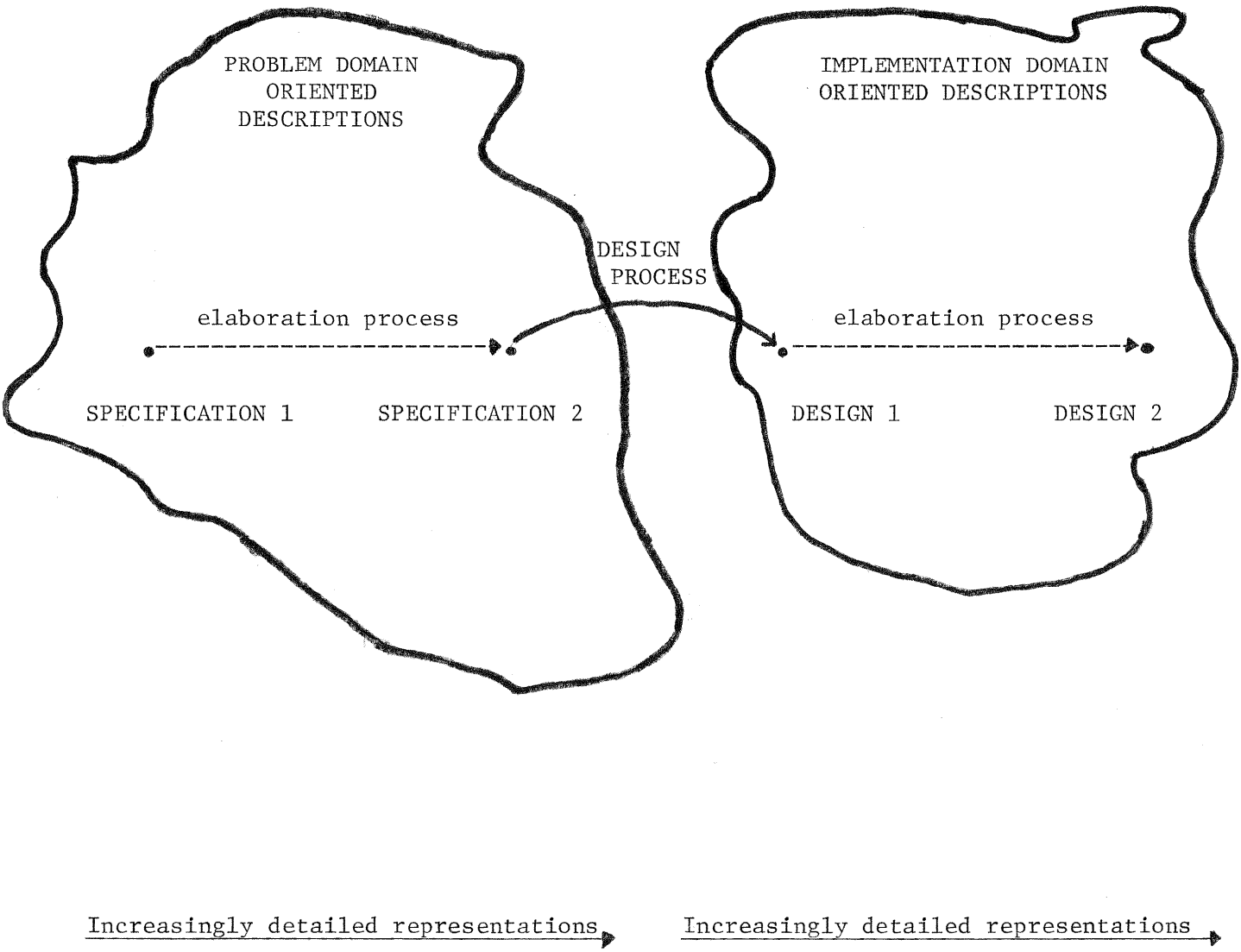

PROBLEM DOMAIN
ORIENTED
DESCRIPTIONS
IMPLEMENTATION DOMAIN
ORIENTED DESCRIPTIONS
DESIGN
PROCESS
elaboration process
elaboration process
SPECIFICATION 1
SPECIFICATION 2
DESIGN 1
DESIGN 2
Increasingly detailed representations
Increasingly detailed representations

Figure 1

showing how several such languages fit within this classification and considering the capabilities and shortcomings of each language.

Software Development Tools

Benefits accrue from languages for the statement of specifications and designs because they provide a basis for a variety of tools which a software system designer or design team may use to ease the design process. Roughly classified, these tools fall into the following categories:

Bookkeeping tool: Provides aid in recording the current specification and/or design, in modifying and augmenting this record, and in returning it to a previous point if decisions are reversed.

Supervisory tool: Provides aid in assuring that design practices and procedures which are considered beneficial are actually followed.

Managerial tool: Provides aid in assessing the progress of the design effort and in focusing attention upon the critical remaining design tasks.

Decision-making tool: Provides aid in measuring and predicting the properties (as implied by the current design) of the system under design for the purpose of detecting errors, gaining confidence in the appropriateness of the already made design decisions, and guiding the further development of the design.

It is possible to define tools in all of these categories without the use of design or specification languages, but the structure of the languages can introduce a desirable rigor into the definition of the tools. For example, the syntactic structure of a language can delineate units of information in a design, such that restrictions on their visibility to different members of a design team constitute a definition of the practice known as information hiding [Pa71].

A set of integrated, computerized tools (a toolbox) may be constructed around a data base system which provides a repository for all of the information concerning the evolving design. The data base itself may be organized according to the syntactic structure of the design and specification languages, with the syntactic structure of the languages serving to delineate the units of information in the data base. Bookkeeping tools may then be provided in the form of text editors which assist in composing description fragments and data base management routines which control the flow of information into and out of the data base. Supervisory tools also take the form of data base management routines, imposing constraints upon what information may be inserted into or retrieved from the data base at any point in time. Management tools include report generators which derive information about the completeness of the information in the data base and analysis routines which help in assessing the implications of this information. Finally, decision making tools take the form of simulation or analysis based routines which process the information in the data base to determine both its consistency and its implications regarding the characteristics of the system under design.

Desirable Language Features

Languages for use during that phase of software system development encompassing the design process are necessarily quite different from the familiar programming languages employed in later developmental phases. In this section, we indicate the nature of this difference and suggest a set of desirable attributes for such languages.

The primary emphasis in this early phase of software development is upon describing the system's intended behavior as opposed to the details of its operation. Languages for use during this phase should, therefore, be behavior oriented. In particular, such languages should be non-prescriptive, capable of describing the designer's intentions for system behavior without prescribing any specific means for achieving those intentions. Non-prescriptiveness serves to prevent the premature selection of implementation mechanisms which can unnecessarily limit the range of potential system realizations. The behavioral orientation of this phase also favors languages in which descriptions orthogonal to the system's eventual implementation description can be formulated. An orthogonal description is one forming associations among the elements of the system which may be completely different from those found in the system's implementation. An orthogonal description may, therefore, express a logical rather than a physical system organization.

While system behavior may be procedurally described using language constructs akin to those appearing in traditional programming languages, this can lead to a more prescriptive, less orthogonal description. Languages for use in the early phases of a software development should, therefore, contain non-procedural constructs which can state behavioral attributes without specifying algorithms or mechanisms for producing those attributes. Language constructs which provide for modelling offer another valuable technique for non-prescriptive, orthogonal behavior description, allowing an abstract, pseudo-procedural representation of intended system behavior.

The descriptions of a software system which appear during the first stages in its development are most valuable if they can serve as the basis for some analysis. Thus, languages for use in stating such descriptions should provide for unambiguous representations. Analysis is also facilitated by languages which permit projection, or the focusing of the description upon various particular aspects of interest. Languages should, in addition, allow and encourage redundant descriptions, since the comparison of multiple descriptions is a valuable analysis technique.

Finally, languages used in conjunction with the design process should obviously facilitate modification to descriptions. Modularity is one useful attribute in this regard, since alterations are simpler and less error-prone when they can be localized. Languages which foster hierarchical representations can similarly ease modifications. Above all, such languages should permit incremental change, so that descriptions can be enhanced and refined as software system development progresses.

Language Classification and Survey

It is possible to distinguish three major types of languages which possess the attributes discussed in the previous section and which support the provision of a design toolbox--state-based languages, event-based languages and relational languages. In the three subsections below, we briefly define these types and survey some representative languages encompassed by each type.

State-based languages. Description schemes in this category provide some means of specifying the set of possible system states and the system's state transition function. Programming languages are state space description

schemes since a program's data and control variables define a state space and the text of the program defines a state transition function. The value of state-based schemes for the task of describing specifications and designs lies in the ability to abstract the state space so that it reflects only interesting details and in the availability of non-programming approaches to the description of the state transition function. For example, a data base could be abstractly modelled in a state space description scheme as an entity which has the states open and closed, and that part of the transition function corresponding to closing the file could be described by the set of transitions {open--> closed, closed--> closed}.

A particularly valuable aspect of state space description schemes is that they admit hierarchical descriptions. A collection of modules may be described as a composite module by demarcating a set of states and delineating the correspondence between one of the states of the composite module and a combination of states of the component modules. The operations which may be invoked upon the composite module may then be described in two ways. First, they may be described in terms of transitions over the states of the composite module. Alternatively, each operation may be procedurally described as an algorithm controlling the invocation of operations upon the component objects. Each module in the resulting hierarchy serves to encapsulate its component modules, providing a "high-level" description of their collective operation without the necessity of detailing their individual operation.

TOPD [He75b] is a program construction toolbox which uses a state space scheme for describing a program's design. In TOPD, fragments of a design describe various aspects of a program's modules such as their states

or the transition definitions of the operations which may be invoked upon them. Bookkeeping tools included within the TOPD system consist of an editor for aiding the composition of design description fragments and a data base management system which provides facilities for augmenting and modifying the design description. TOPD also includes a decision making tool [He75a] which checks the consistency between an operation's transition definition and its procedural definition. The checking facility does not, in and of itself, certify the "correctness" of the procedural definition with respect to the transition definition. Rather, it uncovers inconsistencies and it is the responsibility of the designer to determine the import of this information. This type of analysis, called feedback analysis [Ri77a], is quite appropriate during design when the designer's insight and intuition are needed in the assessment of the design's appropriateness.

A state space description scheme [Ro77] is also the basis for a program development toolbox developed at SRI, International. The description scheme is a variation and extension of that developed by Parnas [Pa72], providing a means for procedural models of the system's components, to which has been added an assertion language, providing a means for non-procedural descriptions. The focus of this toolbox is upon a supervisory tool, in the form of a well-defined method for system development [Ro75a], and a decision making tool which uses theorem proving techniques for ascertaining the consistency between a module's procedural and assertional definitions [Ro75b].

A third toolbox which uses the state space approach to design description is the DREAM system developed at the University of Michigan

[Ri77c,Ri77d]. This system is intended for the development of large-scale software systems which contain (either actual or apparent) parallelism. In DREAM, interactions among the asynchronous, concurrent modules are modelled as either the transmission of messages or the direct sharing of information repositories. The state space therefore reflects the states of the shared information repositories and the state of the message transmissions among the message sources and sinks. Operations upon shared information repositories may be described either procedurally or non-procedurally by means of constructs similar to those in TOPD. The operation of the message sources and sinks may be described procedurally in terms of an algorithm controlling the message transmission operations, the operations upon shared information repositories and the operations modelling the processing of messages. A non-procedural scheme for describing the system's modules is also available and is discussed in the next section. DREAM provides bookkeeping tools which are patterned after those provided by TOPD [Ri77b]. Several decision making tools are under development which will allow the software designer to assess the dynamic, run-time characteristics of the system under design. These tools are intended primarily for the investigation of synchronization properties [Ri76], but will also allow investigation of performance properties [Sa77].

All three of these systems support the hierarchical description of a design, for the following reasons. First, hierarchical description facilitates the incremental development of a design. (All of the systems have a definite orientation toward top-down development, but may also be used with other design methods.) Second, hierarchical description is

provided so that the task of design analysis, with the aid of the decision making tools, may also be carried out incrementally, coincident with the incremental evolution of the design.

Event-based languages. A second set of descriptive techniques potentially useful in describing software specifications and designs may be characterized as event-based. That is, descriptions are stated in terms of orderings upon the occurrence of certain distinguished events in the software system's behavior. The techniques generally represent the collection of desired or acceptable event orderings by combining symbols which stand for event occurrences into one or more formal, algebraic expressions. Such expressions, therefore, offer a means for formulating non-procedural, non-prescriptive descriptions of complex software systems.

The path expression formalism, developed by Campbell and Habermann [Ca74] is one such algebraic technique, originally intended for use as a programming language construct to express desired sequencing and synchronization in concurrent programs. Symbols representing procedure executions are combined in path expressions using operators which indicate the acceptable ordering of procedure occurrences. Several expressions may be used to state various different aspects of the intended sequencing and synchronization, thereby permitting a modular description and facilitating incremental change. A collection of path expressions is normally interpreted as permitting only those procedure invocations which do not violate any of the sequencing constraints imposed by the various members of the collection. Path expressions have been used to construct solutions to classical synchronization problems.

Event sequence expressions [Wi78,Ri77c], an outgrowth of earlier work on event expressions [Ri76], provide a means for non-procedural behavior description within the DREAM system. Event sequence expressions support the definition of a set of distinguished events and the statement of properties, either restrictive or permissive, regarding the intended sequencing and concurrency of those events. A collection of expression operators is provided for use in constructing expressions describing permissible sequences of event occurrences and several constructs are available for the explicit, succinct statement of coordination and synchronization relationships among events. The DDN constructs and their precursors have been used in numerous example software system descriptions and their properties have been extensively studied.

Shaw's flow expressions [Sh77] are very similar to event expressions. Flow expressions are formed using operators similar to those employed in regular expressions, and represent the flow of entities (e.g., control, data items) through components (e.g., instructions, procedures, modules) during system execution. As in the case of event expressions, an arbitrary interleaving of the symbols corresponding to entities represents potential concurrency in the symbol strings described by flow expressions. Certain synchronization operators, which are specialized versions of the event expression synchronization operators, are used to describe intended coordination among or within flow expressions. The flow expression technique has been used to describe several types of software systems from various perspectives.

Grief has proposed a closely related scheme [Gr77] in which the sequencing of event occurrences is constrained by a partial ordering

defined over those occurrences. Both events and sequences of events may be ordered in the Grief approach. In essence, this technique implicitly defines the sets of event sequences which are explicitly defined in the event expression and flow expression formalisms. Grief has demonstrated the use of this scheme in describing solutions to a classical synchronization problem.

Relational languages. Relational description schemes provide a "free-association" approach to description development. In such schemes, it is possible to declare pertinent aspects of a system such as the input to be processed, the output to be produced, the functions to be performed and events which are to happen during the system's operation. Then properties of the system are specified by describing the required or desired relationships among the declared aspects--for example, the relationship "is input to" could be specified as holding between some data entered into the system and one of the functions to be performed by the system. The description scheme itself does not impose any organization upon the order in which aspects must be declared or relationships must be specified.

Two similar toolboxes, ISDOS [Te71] and REVS [Da77], have been developed around a relational description scheme. Both provide a variety of predefined relationships and include bookkeeping tools which aid in the preparation of a system specification with respect to these relationships. Both also include decision making tools which paraphrase the information in a specification and produce reports through which the person preparing the specification can more easily perceive consistencies and potentially discover inconsistencies. In some cases, inconsistencies are uncovered automatically by these systems.

A major difference between these two systems is that in RSL [Tr76], the specification language provided by REVS, it is possible to specify time-related performance criteria. This is accomplished by first specifying abstract algorithms[1] for the various system functions. Then timing relationships between points in these algorithms may be specified by additional algorithms, in the form of "checking programs," which define the desired relationships between the times that control passes these points. The algorithms may be "executed" by a decision making tool which uses a simulator to derive information about the feasibility of achieving the timing relationships specified by the checking programs.

Language Classification and the Design Process

In Figure 1, we suggested that both specifications and designs could exist within broad regions of descriptive detail, being refined and elaborated as software development progressed. We also indicated that the design process involved a transition from the region of specifications, oriented toward the problem domain, to the implementation oriented region of designs. Having now considered three broad classes of languages for use in this phase of software development, we here briefly relate those classes to the spectrum of description stages depicted in Figure 1.

Generally speaking, the languages which we have characterized as relational are most appropriate to descriptions toward the left, or

---

[1] Algorithmic specification of a system is generally not desirable since it is prescriptive of the system's modular structure and eventual implementation. It is, however, consistent with a relational scheme since the algorithm provides a definition of the relationship between system output and system input. Algorithmic specification is consistent with the aims of the specification task, however, only if the algorithms are viewed as non-prescriptive definitions of relationships, possibly orthogonal to the system's eventual modular and algorithmic structure.

specifications, end of the spectrum while those we classify as state-based find their most natural application in descriptions toward the right, or designs, end. Relational schemes typically possess very limited means for combining and structuring subparts of a description, a situation conducive to stating specifications but impeditive to the software domain oriented description found in a design. Similarly, the notion of states, while quite natural to an implementation oriented description, is an encumbrance to problem domain oriented statements of specifications.

The event-based class of languages can conceivably be used anywhere within the spectrum of descriptions associated with the design process. The choice of a set of events and operators largely determines the orientation of a description formulated in such a scheme. In practice, however, the majority of examples employing path expressions, flow expressions, event and event sequence expressions and Grief's formalism have tended toward the design end of the spectrum. While these examples have indicated that event-based languages can be useful in software domain oriented descriptions, the demonstration of their utility for stating specifications awaits the development of appropriate, problem domain oriented, sets of events.

Conclusion

The design of large-scale software systems is a complex task which demands both an orderly approach and some assistance in assessing whether or not the evolving design is acceptably related to system specification. Both these needs are partially satisfied by the use of languages which allow the unambiguous, well-defined statement of specifications and designs.

The languages are even more effective when there are well-defined relationships between constructs in the specification language and constructs in the design language. This permits the definition of techniques for passing from a specification to a design and the development of computer-based tools for assisting in both the use of these techniques and the assessment of the quality of the evolving design.

We have reviewed several schemes for describing software specifications and designs. In the review, we have limited attention to those languages which have been used as a basis for software development toolboxes--a few closely-related languages have also been reviewed although they themselves have not been used in this manner. We have not tried to cover the languages in detail, but have rather tried to indicate the utility of the languages as a basis for software system development tools.

REFERENCES

[Ca74] Campbell, R.A. and Habermann, A.N. The specification of process synchronization by path expressions. Lecture Notes in Computer Science, 16, Springer Verlag, Heidelberg, 1974.

[Da77] Davis, C.G. and Vick, C.R. The software development system. IEEE Trans. on Software Eng., SE-3, 1 (Jan. 1977), 69-84.

[Gr77] Grief, I. A language for formal problem specification. Comm. ACM, 20, 12 (Dec. 1977), 931-935.

[He75a] Henderson, P. Finite state modelling in program development. Proc. 1975 International Conf. on Reliable Software, Los Angeles, April 1975.

[He75b] Henderson, P. et al. The TOPD system. Technical Report 77, Computing Laboratory, University of Newcastle upon Tyne, England, September 1975.

[Pa71] Parnas, D.L. Information distribution aspects of design methodology. Proc. IFIP Congress 71, Ljubljana, August 1971, TA3-26-TA3-30.

[Pa72] Parnas, D.L. A technique for software module specification with examples. Comm. ACM, 15, 5 (May 1972), 330-336.

[Ri76] Riddle, W.E. An approach to software system modelling, behavior specification and analysis. RSSM/25, Department of Comp. and Comm. Sci., University of Michigan, Ann Arbor, July 1976.

[Ri77a] Riddle, W.E. A formalism for the comparison of software analysis techniques. RSSM/29, Dept. of Comp. and Comm. Sci., University of Michigan, Ann Arbor, July 1977.

[Ri77b] Riddle, W.E., Sayler, J.H., Segal, A.R. and Wileden, J.C. An introduction to the DREAM software design system. Software Engineering Notes, 2, 4, (July 1977), 11-23.

[Ri77c] Riddle, W.E. and Wileden, J.C. Behavior modelling during software design. RSSM/56, CU-CS-119-77, Dept. of Comp. Sci., University of Colorado at Boulder, November 1977.

[Ri77d] Riddle, W.E. DREAM--a software design aid system. RSSM/68, Dept. of Comp. Sci., University of Colorado at Boulder, December 1977.

[Ro75a] Robinson, L., Levitt, K., Newmann, P. and Saxena, A. A formal methodology for the design of operating systems software. SRI, Menlo Park, September 1975.

[Ro75b] Robinson, L. and Levitt, K. Proof techniques for hierarchically structured programs. Tech. Rep. CSL-27, SRI, Menlo Park, November 1975.

[Ro77] Robinson, L. and Roubine, O. SPECIAL--a specification and assertion language. Tech. Rep. CSL-36, SRI, Menlo Park, January 1977.

[Sa77] Sanguinetti, J.W. Performance prediction in an operating system design methodology. RSSM/32 (Ph.D. Thesis), Dept. of Comp. and Comm. Sci., University of Michigan, Ann Arbor, May 1977.

[Sh77] Shaw, A.C. Software descriptions with flow expressions. Technical Report #77-10-03, Dept. of Comp. Science, University of Washington, Seattle, October 1977.

[Te71] Teichroew, D. and Sayani, H. Automation of system building. Datamation, August 15, 1971.

[Tr76] Requirements Engineering and Validation System Users Manual, TRW Defense and Space Systems Group, Huntsville, July 1976.

[Wi78] Wileden, J.C. Behavior specification in a software design system. RSSM/43, Dept. of Comp. and Comm. Sci., University of Michigan, Ann Arbor, in preparation.