# Procedural Approaches to Software Design Modelling

William E. Riddle

CU-CS-150-79

RSSM/86

# PROCEDURAL APPROACHES TO SOFTWARE DESIGN MODELLING

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

CU-CS-150-79 April 1979

## Abstract

Several approaches to the modelling of a software system during its design are discussed. Existing software design modelling techniques are categorized and it is concluded that the category of procedural techniques, which utilizes many of the same concepts as programming languages, offers the most promising opportunity for providing an effective design support system as an integrated collection of tools.

## Introduction

In developing a software system which appropriately realizes a solution to a problem one must pass through many levels of conceptualization of the system. During the initial stages of development, these conceptualizations are oriented toward the user or customer community. Gradually, during development, the conceptualizations change to an orientation toward the environment in which the system will execute.

At every conceptual level, the system is abstractly described so as to record what is known about the system, allow the recognition of what remains to be determined about the system, and provide the ability to answer questions for the purpose of assessing the validity and reasonableness of the system. Therefore, these abstract descriptions are each, in the full sense of the word, a model of the system.

Software system development is thus a process of model construction and elaboration. The ease with which this may be done depends primarily upon the modelling "materials" provided to software development practitioners and the tools provided these practitioners to aid in both the formulation of models using these "materials" and the assessment of the models. The development process is further facilitated by providing a support environment in which the model formulation and assessment tools are highly integrated, complementing each other and easily usable in tandem.

In this paper, we discuss several software design modelling formalisms, i.e., approaches to providing modelling "material" to software system developers during the design phase of development. We then focus on one particular formalism, characterize a number of existing techniques for bringing this formalism to design practitioners, and argue that it provides the best opportunity, of the formalisms discussed here, for the provision of an integrated collection of tools.

## Modelling During Software Design

By design we mean that activity during development which follows an initial phase of specification (also known as requirements definition) and precedes the implementation phase. Thus, the system's required capabilities have been detailed and recorded before the design activity starts and this record serves as a definition of acceptance criteria for the design. Also, specific details of the data processing and organization will be developed after the design phase. The intent during design is therefore to transform the system's overall requirements into specific, detailed requirements for the system's components.

The goal during design is the preparation of a gross, conceptual blueprint, or schematic, for the system. One task is therefore to modularize the system and demarcate its major processing and data repository components. Another task is to define the interfaces among the components. The final task is to describe the interactions among the components.

It is through the description of the interactions that the purpose of design, the transformation of system requirements into component requirements, is primarily achieved. The interactions are an embodiment of strategies that the designer has chosen for the component cooperation necessary to achieve the policies set forth in the system's specification. Thus the interaction description may be vague as to the algorithmic data manipulation that must be performed in support of the information exchange required by the strategies. But it must be specific as to the actual information exchange so that it can be determined that the strategies actually achieve the required policies and so that the requirements for information preparation and transmission levied against the components can be determined.

Because the individual components' processing capabilities and their interactions are the aspects of interest during design, the system description prepared during the design phase should explicitly describe the functionality of the system's modules. Focusing upon the explicit description of functionality generally precludes the simultaneous, explicit description of other system characteristics; but these other characteristics should be able to be easily derived from the description. (This sometimes requires that additional information, not necessarily pertinent to the functional characteristics, be provided in the model.)

## Software Design Modelling Formalisms

We have established that the goal of the design phase of software development is to model the system's modularity and the interactions among the modules. The aspect of primary interest is functionality and the audience is primarily the designer but some part of the design is eventually presented to the implementors as a definition of the functional capabilities of the components which they must prepare. Also, there is an assessment task, namely to assure that the system as designed has some chance of producing the required overall system capabilities. Another secondary audience is therefore composed of those persons, perhaps identical to the designers, who will assess the design with respect to the system specification. Before reviewing existing techniques for software design modelling and making several observations concerning the various ways in which they satisfy this goal, we introduce, in this section, some general modelling terms which will help in classifying and commenting on existing software design modelling techniques.

A model is a description prepared according to the rules of some modelling formalism. A modelling formalism, itself, consists of three parts. First there is a basic vocabulary of concepts which are primitive, i.e., well-defined and not (necessarily) definable using the modelling scheme. Second, there is a set of composition rules which define the ways in which models may be validly constructed. Finally, there is a set of derivation rules by which the effect of composing the primitive concepts in the ways allowed by the composition rules may be determined -- these rules allow the derivation of the overall properties of a model itself.

As one example of a modelling formalism, consider a traditional, sequential programming language. The base vocabulary consists of the concepts of variables, constants, data types, operations, assignment, etc. -- the base vocabulary is roughly related to the lexicographic, micro syntax of the language. The composition rules are closely related to, but not identical with, the language's syntax; they include the rules by which execution flow may be controlled, structures of data may be composed, elements may be selected from data structures, etc. Finally, the derivation rules are related to, but again not identical with, the language's semantics. Very roughly, they allow one to determine the effect of the model in terms of the changes in the values of variables over time.

As indicated, the concepts of base vocabulary, composition rules and derivation rules are akin to, but distinct from, the seemingly similar concepts of lexicographic, syntactic and semantic rules for a language. The distinction is that the modelling concepts relate to the approach that is used to capture the system being modelled whereas the related language concepts pertain to the statement of the model in some well-defined description technique. We are thus making a distinction between the "world view" which is captured in the modelling formalism and a description technique which provides one way (probably of many) to state models using this "world view."

To clarify this distinction, consider a Newtonian formalism for modelling a gas. This formalism has a base vocabulary consisting of the concepts of moving and stationary objects, composition rules governing the containment of a collection of moving objects within an organized and structured collection of stationary objects, and derivation rules provided by the principles of Newtonian motion and two-body interaction. This modelling formalism is quite distinct from the many "languages" which may be used to represent models in the formalism, languages which range from English to graphical techniques to mathematical formalisms.

In the remainder of this paper, we use the term modelling technique when referring to a language by which models may be represented using some modelling formalism. We emphasize that there are many modelling techniques corresponding to a particular modelling formalism, just as there are many algolic languages.

The composition and derivation rules provide the opportunity to check the validity of a model. The validity of the model's "form" may be checked by assessing its adherence to the composition rules -- alternatively, anything constructed using solely the composition rules is a legally formed model. Considering programming languages, an example of this type of check is assuring that the type restrictions upon operations and assignment are not violated. The derivation rules permit a different type of check, namely one that relates to aspects concerning the effect of the model. An example from the domain of programming languages is the check that subscript selection values are within range. While the distinction between violations of the composition rules and violations of the derivation rules is not precise, the major difference is that the composition rules relate to static properties while the derivation rules relate to dynamic properties.

While the composition and derivation rules allow the determination of the validity of the model with respect to the rules for the proper formation of models, they do not, by themselves, allow one to assess the correctness of the model as a representation of the modelled system. For this purpose, it is necessary to have an _interpretation_ which relates the characteristics of the model to those of the modelled system. It is through the interpretation that a model has meaning with respect to the system being modelled, and it provides a "window" into the model that relates some of the characteristics of the model to some of the characteristics of the modelled system. The interpretation is therefore determined by three things: the modelling formalism, the modelled system and the characteristics of the modelled system which are of interest.

It is common to use a higher-level modelling formalism to represent the system characteristics which are of interest, in which case the interpretation may be captured as a mapping between models, relating the characteristics of the lower-level model to those of the higher-level model, and thus the modelled system. In terms of the previous programming language example, the higher-level modelling formalism could be that for which the language of predicate calculus may be used as the modelling technique. In this case, the interpretation consists of the rules of verification condition generation which allow a model in the higher-level domain to be extracted from one in the lower-level domain. For the previous non-software example, a higher-level modelling formalism for gases is provided by thermodynamics and the interpretation is provided by statistical mechanics.

When an interpretation is captured as a mapping between modelling formalisms, the role of the interpretation, namely to allow the verification of the correctness of the model with respect to the characteristics of the modelled system, is achieved in a two-step process. First, the formally defined interpretation rules are used to transform the model from the lower-level formalism to the higher-level formalism, thus inferring the modelled system's characteristics as they are represented by the model. Then the composition and derivation rules of the higher-level formalism are used to check the validity of the higher-level model. Thus the question of correctness of the model is reduced to the question of the validity of another model.

## Types of Existing Design Modelling Formalisms and Techniques

Using these concepts, it is possible to categorize existing modelling formalisms and techniques and comment on some of their differences and similarities. In this section, we form a rough categorization of some existing approaches to software design modelling to illustrate some of the possibilities which have been tried. We then focus upon one category, procedural modelling formalisms, and subdivide it so as to be able to make some observations about various techniques that have been used. Our observations concerning the various techniques appear in the next section.

A dichotomy of existing software design modelling formalisms can be made on the basis of the formalism's basic vocabulary. One broad category of formalisms, called event-based formalisms, relies upon the idea of significant events which occur during system operation and the use of sequences of events to relate system behavior. These formalisms are reviewed elsewhere [ShaA79].

The other broad category of existing formalisms, called decompositional formalisms, uses the concepts of data processing and storage components and control and accessing relationships among them. These formalisms are quite natural for the modelling of software systems as they use concepts similar to well-defined ones from the implementation phase -- in some instances the techniques have been obtained by the generalization of programming languages.

This category may be refined by considering the primitive concepts used to denote the coupling among the entities in the system. One set of formalisms, which can be called _relational formalisms_, uses the formal notion of a relation to describe the data usage, data organization and control interactions among the components of a system. Some of these formalisms (e.g., ISDOS [TeiD77], SREM [DarC77]) were developed for use during the specification phase of software development. But they allow the system's modularity and interactions to be exposed and so are equally useful during the design phase. Others (e.g., TELL [HebP79], MIL [DreF76]) were developed expressly for the purposes of design. Regardless of their genesis, these techniques all require that the inter-relationships among the entities in the system be modelled by the use of (predefined or user-defined) relations such as "contains", "uses", "activates", "causes", etc.

A second subcategory, called _functional formalisms_, relies upon the concept of function composition in order to denote coupling. Entities are considered to be represented by functions and system modularity, both of a physical and logical nature, is represented by the composition of these functions. Some techniques falling in this subcategory are described in [SmoS79], [TonA77], and [ZavP79]; most of them are based upon the same general principles as those underlying the Lisp programming language.

The third, and last, major subcategory, called _procedural formalisms_, is similar to the second but utilizes the concept of procedure activation to reflect component interactions, and hence coupling. These are perhaps the most natural for use by designers who are experienced programmers. Techniques falling in this category are currently the most numerous and are the ones of interest in the remainder of this paper.

One major class of techniques useful for this formalism subcategory are oriented towards the modelling of sequential programs. Some of these are notations which may be used to highlight the control flow (e.g., PDL [CaiS75]) or the data flow (e.g., HIPO [KatH76]) through the system. These generally consist of a small "programming" language which retains the pertinent constructs from traditional sequential programming languages but allows the designer to express processing details in English.

Other techniques falling in this class are based on the concept of data abstraction. These force the designer to structure the model into collections of procedures which serve, as unit, to define an abstract data type in terms of the operations that can be performed upon it and the effect of these operations both in isolation and in conjunction with each other. Many of these techniques (e.g., CLU[LisB77], Alphard [WulW75]) have been developed as extensions to traditional programming languages, but have resulted in the elevation of these languages to the realm of design because they allow the focus of attention to be directed toward the interactions between modules and away from processing details.

The third set of techniques within this class are closely related to the second as they, too, rely upon the notion of data abstraction. But these also include the concept of the state of the system's components. This additional concept makes these techniques less like programming languages and more clearly modelling languages. Some of these (e.g., Parnas' method [ParD72], HDM [RobL75]) recognize a dichotomy of procedures into those which change the state of objects and those which merely inspect the state of objects -- these are quite similar to traditional programming languages. Others (e.g., TOPD [HenP75] and STESD [BakT78] incorporate the additional notation of state transitions in order to allow the direct description of pre-conditions and post-conditions upon procedure invocation and the effect of procedure invocation itself.

The other major class of techniques falling within the procedural modelling formalism subcategory are those which are oriented toward the description of concurrent systems. This orientation requires that the techniques provide for the description of synchronization among asynchronous components as well as the sharing of data repositories. (Provision of constructs for the modelling of shared data is theoretically not necessary as it can usually be modelled by use of the synchronization interaction constructs; but less obscure models generally result from the inclusion of constructs specific to shared data.) The basic vocabulary of these techniques (e.g., DREAM [RidW78], SARA [AmbA77] and Gypsy [CamI78]) generally include the concepts of processes and monitors, or concepts very similar. The most popular means of describing synchronization interactions is by message communication and this is a primitive concept in most techniques. Some include the concept of collections of processes so that entities which are capable of internal parallelism may be directly represented.

It is interesting to note that, until recently, these techniques where clearly for the purposes of modelling as they used a synchronization mechanism, namely message transmission, which was sometimes inefficient for use in the actual system. But, recent proposals ([AndG79], [BriP78], [FelJnd], [HoaC78]) for concurrent programming languages useful in the domain of distributed systems have used many of the same notions and thus it is no longer as clear where modelling stops and programming begins.

## Some Observations Concerning Procedural Modelling Formalisms and Techniques

Having categorized various formalisms for software design modelling and very roughly classified some techniques in terms of the basic vocabularies of the procedural modelling formalisms upon which the techniques are based, we turn to some observations concerning the nature of procedural modelling techniques.

The first is that what is good for modelling is not necessarily good for programming. Given recent developments in programming languages, it is perhaps more accurate to say that what is good for modelling may be good for programming but that usefulness for programming is not an important goal when developing a modelling technique. Instead, there are two more important goals which are often in conflict with those goals present during the development of programming techniques.

One of these goals is that the models be projective representations of the modelled system. A basic attribute of a model is that it is abstract, meaning that it suppresses some detail in order to highlight other detail. Thus, it is important that a modelling technique utilize a set of primitive concepts that allows the succinct and natural expression of the details which are to be highlighted and the suppression of unnecessary details. As an example, non-determinism is an important concept in the modelling of software systems, but is generally held to be an anathema with respect to programming techniques.

Another goal is that the modelling technique permit descriptions which are non-prescriptive with respect to the details of the modelled system. This is generally hard to achieve for procedural modelling techniques, because many of the basic concepts have strong connotations in the realm of programming. Take, for example, the use of message transmission to represent synchronization within concurrent systems. The recent proposals for distributed system programming languages indicate that this synchronization mechanism is one which can plausibly and profitably be used in the actual system implementation. But it may also be used to represent synchronization which is perhaps more effectively accomplished, during implementation, by other mechanisms such as semaphores.

Because of the fact that there are differing goals for modelling and programming, the use of concepts and constructs in a modelling technique which are similar or identical to those found in programming languages results in a confusing similarity between the model and the system it models. This confusion is, however, in the eye of the beholder and can generally be clarified by consideration of the differing aims of modelling and programming. On the other hand, the confusion can be capitalized upon, as done in HDM and, to some extent, in Gypsy so that the act of model construction results in the development of control algorithms to be used, verbatim, in the eventual system.

Another observation is that there is a difference between the organization of the model's description and the organization of the model (and the modelled system) itself. Algolic languages suffered from the lack of this distinction with respect to variable accessibility and hierarchical organization of program text using begin...end blocks.

Recently, languages (such as Mesa [GesC77]) have been developed in which the hierarchical structure of the text implies nothing about the accessibility of the variables declared at various levels of the hierarchy.

The goal in developing a modelling technique for use during software design should be to allow hierarchically organized descriptions that can be easily prepared incrementally. The hierarchical structure of the textual description tends to foster an approach to design which relies upon the gradual evolution of the model. The ability to incrementally develop the text supports the general, and natural, tendency to "jump around" the system during design. Taken together, the result is the ability to prepare an organized description as a result of a somewhat disorganized process.

For procedural modelling formalisms, however, there seems to be general agreement that the technique, and the formalism which it represents, should permit the development of hierarchical models. In some cases, for example HIPO , there is the restriction that the model hierarchy coincide with the hierarchical structure of the text of the model description. In most cases, however, there is some provision for organizing the model itself differently from the organization of the model's textual description. There is also general agreement that the organization of the model should coincide with the organization of the modelled system -- which is reasonable given that the intent is to capture the modularity of the system.

A final set of goals that can be extracted from considering existing software modelling techniques concern the ability to perform assessment during design. First, the modelling technique should be based upon a well-defined, unambiguous underlying formalism so that the rules for composition, derivation and interpretation are well formulated. For procedural modelling techniques, this is relatively easily achieved because of the similarity to programming languages and thus the ability to utilize simulation and analytic techniques already developed for the validation and verification of programs. But other underlying formalisms exist and have been used in some techniques, for example SARA and DREAM.

Another goal related to the ability to perform analysis is that the procedural modelling technique should be closely related to some higher-level modelling technique so that questions concerning the correctness of the model can be handled by "reducing" them to questions concerning the validity of a higher-level model obtained algorithmically from the model expressed in the procedural technique. Most of the procedural modelling techniques cited above are related to higher-level, usually event-based, formalisms.

## Models, Tools and Methods

Modelling formalisms and their associated representational techniques form a basis for a variety of software design tools:

- bookkeeping tools provide aid in preparing, augmenting and modifying a model's description,
- supervisory tools enforce design methods, principles and guidelines which are deemed beneficial,
- management tools allow the assessment of the extent of progress and the determination of how best to allocate design team efforts so as to speed progress,
- decision-making tools help designers determine the validity and reasonableness of design decisions as well as identify, and rank order, decisions yet to be made.

In addition to their purpose, tools may vary according to the extent to which they incorporate knowledge about the details of the modelling technique and the modelling formalism upon which they are based. A text manipulation bookkeeping tool, for example, may be independent of the modelling technique and formalism and view descriptions solely as unstructured sequences of characters; or the tool could be highly dependent on the technique and formalism and utilize knowledge of the organization of models and their descriptions to guide or monitor the text manipulations. Note, however, that some tools cannot be defined independehtly of the modelling technique and formalism - the above list of tool types is arranged in terms of increasing dependence of the tools upon the modelling techniques and formalism.

Variance among tools may also stem from the <u>design method</u>, i.e., the approach used to elaborate a model's textual description. Supervisory tools generally, as a class, exhibit a high degree of dependence upon the design method as they typically are the embodiment of the method. But a supervisory tool which enforces the information hiding design principle [ParD71] is totally independent of the design method. Similar variance may be found in the other tool classes defined above.

Finally, there is a variance among tools stemming from the design methodology, or approach to model construction, which they reflect. The "rules" which constitute the methodology, such as "check the validity of each model elaboration before proceeding to the next elaboration," not only place specific capability requirements upon the tools but also generally require that there be certain interrelationships among these capabilities.

## Tool Integration

Collections of tools may be characterized according to the breadth of their collective coverage of the spectrum of functional capabilities and the extent to which they are collectively dependent upon a particular modelling formalism, modelling technique, design method, or design methodology. Before assessing the effectiveness of the procedural modelling formalism for allowing the development of a highly integrated and effective collection of design tools, we demarcate two major categories of tool collections and comment on the quality of the aid which can be delivered by collections in these categories.

In the toolbox category, major emphasis is placed on achieving a breadth of functional capabilities which little or no dependence upon any particular approach to modelling or to design. Collections in this category are akin to operating systems in that they are able to support a variety of languages, and therefore modelling techniques and formalisms, and place very few restrictions upon the sequence in which the various tools may be used.

In the other broad category of tool collections, called support systems, there is a high degree of dependence of the tools upon a specific approach to modelling or design, or both. In this type of collection, one particular language is emphasized, namely the language of the modelling technique. If the support system is additionally dependent upon a particular design method or methodology, there are also restrictions upon the manner in which the tools may be used either in isolation or in tandem.

These categories are not exhaustive of the possibilities. Rather, they are at the end-points of a spectrum defined by the degree of relationship between the tools in the collection and the modelling or design approach being supported.

Toolboxes are relatively more easy to prepare because of the high degree of independence of the tools; because the tools are not oriented toward any particular modelling or design approach, each may be a "stand alone" module. This allows the toolbox to be organized relatively loosely and means that the command language which allows the user to activate the individual tools may be relatively simple.

Support systems, on the other hand, are harder to prepare because of the need to account for the modelling or design approach in both the system's organization and its command language. The payoff for this increase in effort is higher quality help to the users. While toolboxes can be utilized according to some modelling or design approach which the users impose upon themselves, in a support system there can be tools specific to the modelling or design approach being used and these can provide effective help more directly related to the tasks being performed.

The point is that while the restrictions which are part of a particular modelling formalism or design methodology complicate the task of developing a support system, they also facilitate the preparation of a design and thus the level of aid provided by the collection of tools.

## Conclusion

We have discussed the strong relationship between software design and software modelling, developed a framework for discussing alternative software design modelling formalisms and techniques, used the framework to roughly categorize existing software design modelling formalisms and techniques, and made some general observations about the nature of one particular modelling formalism, the procedural modelling formalism. We then turned to the issue of providing an integrated collection of software development tools, noting the dependencies of the tools upon both the modelling formalism and the design methodology and arguing that higher quality help can be provided, at the cost of additional effort, by a support system where the orientation of the collection of tools is toward a particular modelling formalism or design methodology. The question addressed in this final section is: What modelling formalism is best to adopt as the basis for a software design support system?

The answer is that, <u>at this point</u>, the procedural modelling formalism seems best suited to providing high quality support systems. The major reason is that more techniques upon which to base tools are available because of the similarity of procedural modelling formalisms to programming formalisms. This does not necessarily hold with respect to bookkeeping tools because of the relative independence of this type of tool from the underlying formalism even in the case of support systems. But for supervisory tools, the extensive body of knowledge concerning the rules by which clear, succinct, modifiable descriptions should be evolved can be utilized directly in controlling the preparation of procedurally-defined design models. And in the realm of decision-making and management tools, both analytic and simulation techniques which have been developed for the assessment of programs may be used for the assessment of designs. For none of the other formalisms discussed here are both the rules governing good structure of a model or its description and the techniques for assessment as extensively developed.

A second reason is that the procedural formalism provides a set of primitive concepts which are quite natural for describing system modularity and module interactions.

It also appears that this conclusion is not just an artifact of the current state of the art. Because the procedural formalism has a direct and obvious relationship to detailed, program-level description formalisms and there are well-defined techniques for interpreting procedural models in terms of characteristics of interest, these formalisms appear best suited for use in stating descriptions which fall "mid-way" between user-oriented specifications and machine-oriented programs. The other formalisms discussed here perhaps admit more easy interpretation but are "farther away" from the realm of programming and thus not as easily related, algorithmically or non-algorithmically, to the eventual program which is the product of design.

We venture the summary conclusion, therefore, that the procedural modelling formalism is, and will continue to be, the most adequate basis for effective design support systems.

## Acknowledgements

Many "audiences" have been subjected to somewhat rambling discourses while I was formulating and honing the ideas presented here, and I am appreciative of their patience and help. I would like to acknowledge the particularly beneficial comments by Jack Wileden and the students in my graduate-level software design workshop course.

## References

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

AndG79 Andrews, G. Synchronizing resources. Dept. of Comp. Sci., Cornell Univ., February 1979.

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 Comp. Sci., Univ. of Texas, Austin, January 1978.

BriP78 Brinch Hansen, P. Distributed processes: A concurrent programming concept. Comm. ACM, 21, 11 (November 1978), 938-941.

CaiS75 Caine, S.H., and Gordon, E.K. PDL - A tool for software design. Proc. 1975 National Computer Conf., June 1975, pp. 271-276.

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.

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

DReF76 DeRemer, F., and Kron, H. Programming-in-the-large versus programming-in-the-small. IEEE Trans. on Software Engineering, SE-2, 2 (June 1976), 80-86.

FelJnd Feldman, J. A programming methodology for distributed computing (among other things). TR9, Computer Sci. Dept., Univ. of Rochester, n.d.

GesC77 Geschke, C.M., Morris Jr., J.H. and Satterwaite, E.H. Early experience with Mesa. Comm. ACM, 20, 8 (August 1977), 540-553.

HebP79 Hebalker, P.G. and Zilles, S.N. TELL - A system for graphically representing software design, Proc. Compcon Conf., San Francisco, 1979.

HenP75 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.

HorC78 Hoare, C.A.R. Communicating sequential processes. Comm. ACM, 21, 8 (August 1978), 666-677.

KatH76 Katzen, H. *Systems Design and Documentation: An Introduction to the HIPO Method*. Van Nostrand Reinhold, New York, 1976.

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

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

ParD71 Parnas, D.L. Information distribution aspects of design methodology. *Proc. IFIP Congress 71*, Ljubljana, August 1971, pp. TA3/26-TA3/30.

RidW78 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.

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.

ShaA79 Shaw, A. Expression-based approaches to software description. Appears in this volume.

SmoS79 Smoliar, S. Using applicative techniques to design distributed systems. *Proc. Specification of Reliable Software Conf.*, Boston, April 1979.

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.

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).

YonA77 Yonezawa, A. Specification and verification techniques for parallel programs based on message passing semantics. MIT/LCS/TR-91, Lab. for Computer Sci., Mass. Inst. of Tech., Cambridge, December 1977.

ZavP79 Zave, P. Functional specification of asynchronous processes and its application to the early phases of system development. Dept. of Comp. Sci., Univ. of Maryland, March 1979.