Algebraic Techniques for the Analysis of Concurrent Systems

George S. Avrunin*
Jack C. Wileden**

COINS Technical Report 81-11
May 1981

*Department of Mathematics and Statistics
University of Massachusetts
Amherst, Massachusetts 01003

**Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

Abstract

We have developed algebraic techniques for the analysis of certain special classes of concurrent systems based on a description of the systems in a suitable modelling scheme. Both the modelling scheme and these analytic techniques have been specifically tailored for use in the design phase of concurrent software system development. The modelling scheme and analytic techniques are described and an illustrative example is given.

Introduction

It is often very difficult to understand, and hence to reason about, the behavior of a computing system in which there is concurrent activity. The behavior of even relatively small and simple concurrent systems is frequently beyond the comprehension of their designers, as evidenced by the appearance of such incorrect concurrent programs as [HYMA66]. Yet the trend is clearly toward large and complex concurrent systems. In particular, the declining cost of computer hardware, the proliferation of distributed computing systems, and the demand for increasingly sophisticated applications all portend a dramatic increase in the development of large, complex, concurrent software systems. New aids for understanding and analyzing concurrent systems must be found if there is to be any hope of producing reliable and robust concurrent software.

Although improved techniques for analyzing completed concurrent programs could certainly prove useful, approaches to pre-implementation analysis, especially analysis during the design phase of a concurrent software system's development, offer even more promise. The majority of software faults can be traced to design errors [BOEH73], and correcting a faulty design prior to implementation is much more economical than attempting to compensate for it in a completed, erroneous program. Therefore, techniques for understanding and analyzing the design of a concurrent software system, prior to its implementation, could greatly facilitate the concurrent software development process.

The role that we envision for these techniques in pre-implementation concurrent software development is illustrated by the following scenario: A designer, early in the development of a large, complex, concurrent software system, conceives a modularization for the system. The modularization is described by identifying the individual processes comprising the system and specifying how those processes will interact. Continued development of the system, eventually culminating in an implementation, will involve a great deal of time and effort, all of which would be wasted if any error has been made at this early pre-implementation stage. Therefore, before proceeding with the development, the designer employs analysis techniques expressly tailored for pre-implementation use to check for design flaws. Specifically, these techniques can be used to determine whether or not certain patterns of behavior occur, given the specified processes and process interactions. The patterns of interest may represent desirable properties of system behavior, such as mutually exclusive utilization of a shared resource, or graceful degradation and continued operation following the failure of one or more system components. Alternatively, the patterns might represent pathological behaviors such as deadlocks. Through use of the appropriate analysis techniques, the designer could gain confidence in the suitability of a design before proceeding to later stages of the software development process. Our research is directed at developing precisely the kind of analytic techniques that might be used for this purpose.

Due to the complexity of large concurrent software systems, techniques for understanding and analyzing them must provide both a basis for rigorous reasoning about them and a means of focusing on their important features while ignoring a welter of details. An appropriate formalism for describing concurrent computation would offer the needed rigor and capability for abstraction. To be appropriate for application in the concurrent software development process such a formalism must be relatively easy to understand and use. Specifically, it must be amenable to use by software development practitioners who may have little or no training in advanced mathematics or theoretical computer science. Ideally, it should be possible to provide an automated version of the formalism to permit its use in a concurrent software development environment [CLAR81]. Moreover, an appropriate formalism for use in the development process should bear a reasonable relationship to standard software specification and design techniques and should be able to answer the types of questions that arise most crucially during specification and design. Finally, a formalism can only be appropriate for use during concurrent software development if it is applicable to a wide range of distributed system organizations.

We have adopted a formalism that we believe meets these criteria successfully and have begun to develop techniques for the analysis of concurrent systems based on this formalism. Our work relies on a formal modelling scheme that permits the natural description of concurrent systems with a wide range of organizations, including those with dynamic structure

[WILE80]. In addition, the scheme provides a closed form description of the possible behaviors of a modelled system as strings over an alphabet of symbols representing events in the system. Our approach to analysis is to first interpret interesting features of the behavior of the system as patterns of event symbols in these strings. We then ask whether strings containing particular patterns actually occur among those corresponding to the modelled system. To answer these questions, we use the closed form descriptions of possible behavior strings to generate algebraic relations involving the numbers of occurrences of particular symbols in certain segments of the strings. We then attempt to determine whether a string satisfying these relations can contain the pattern under consideration. This can be viewed as a generalization of the technique employed by Habermann [HABE72] in analyzing a semaphore solution to a producer-consumer problem.

The remainder of this paper expands upon our approach to the understanding and analysis of concurrent systems. In the next section we describe alternative approaches and indicate why we find them less suitable for use in the analysis of concurrent systems. We then describe the formalism that we have adopted and outline our approach to the analysis of concurrent systems.

Related Work

A good deal of work has been done that is relevant to the problem of understanding concurrent systems. In this section we briefly survey the major approaches taken in this area and discuss their applicability to pre-implementation analysis in concurrent software development. We distinguish three broad categories of related work: proof techniques, formal semantics, and formal models.

Techniques for proving properties of programs, also known as verification techniques, have received a great deal of attention. By attaching logical predicates, called assertions, to specific points within a program and then using the program's statements to direct a formal reasoning process, it is sometimes possible to prove that a given relationship exists between input values and output values, or that the program terminates after a bounded number of statements have been executed [FLOY67,MANN74]. The formal reasoning process involved is often guided, for a given programming language, by a set of proof rules [HOAR69], one for each type of statement in the language. Particularly when applied to small sequential programs employing only simple data types, this proof rule approach is very appealing in that it produces rigorous formal statements about program properties. Adequate formalizations of the properties of complex data structures, such as arrays and lists, have proven elusive, however. Moreover, even with the aid of sophisticated and powerful theorem provers [BOYE75], proofs about realistically large programs remain computationally infeasible.

The proof rules approach has been applied to concurrent programs by Owicki and Gries [OWIC76], Lamport [LAMP77], and others (e.g., [APT80]). Although it may eventually provide a useful method for analyzing concurrent programs, we do not feel that it is an appropriate approach for understanding and analyzing concurrent system specifications and designs. One reason for this is the complete dependence of the proof rules on the specific programming language being used. More importantly, we feel that the predicate calculus underlying this approach is not a formalism that software developers will find easy to work with. Finally, the construction of proofs in the predicate calculus remains a challenging problem with no acceptably general or powerful automated aid likely to be available soon. Although some or all of these shortcomings may eventually be surmounted, we feel that, taken together, they warrant the pursuit of alternative approaches.

Formal semantics for programming languages, particularly the denotational semantics of Scott [TENN76], attempts to give a rigorous mathematical statement of the meaning of any program written in the language. Programming language statements and constructs are mapped to sophisticated mathematical structures and the result is taken to be a machine-independent representation of the program's meaning.

Milner and his colleagues have investigated denotational semantics for concurrent systems [MILN79]. They first define a model of communicating processes, then develop a flow algebra for describing the individual processes and their composition into a concurrent system. The solution of a powerdomain equation provides the mathematical basis for their approach.

Formal semantics does not seem to provide an appropriate foundation for pre-implementation analysis of concurrent software systems. The description of even a simple concurrent system in a formalism such as Milner's is very complicated and can be understood only with substantial effort. The mathematical structures underlying such approaches are far too sophisticated for most software development practitioners to understand or use. Most importantly, the powerdomain formalism, while it may be useful as a descriptive device, seems to offer no provision for analyzing a concurrent system once its description has been formulated.

The third category of previous work relevant to understanding concurrent systems can be loosely classified as formal models. This includes such things as Petri nets [PETE77], dataflow models [ADAM68,KAHN74], and many other automata-theoretic models. These models, particularly Petri nets, have been extensively studied in an effort to learn more about fundamental properties of concurrent computation. Typically, each type of model corresponds to one particular organization for concurrent computing systems. In some instances, attempts have been made to provide analysis techniques applicable to these models. The primary example of this is the use of vector addition system analysis in studying Petri nets and related models.

Despite some efforts to employ these various formal models in analyzing concurrent software (e.g., [KELL76]), we have not found any of these models to be truly satisfactory for use in formulating analysis capabilities appropriate to pre-implementation software development. Some simply do not allow the description of a suitably wide range of system organizations. For instance, dataflow models cannot be conveniently used to represent the sharing of data objects, and Petri nets and related models cannot succinctly represent systems with dynamic structure. In particular, these limitations make these models ill-suited for analyzing the survivability of a distributed system, i.e., its behavior when certain shared components cease to operate correctly. Moreover, most such models describe concurrent systems in terms that do not naturally correspond to software system features, and thus are not well suited for software specification and analysis. For example, a Petri net would not normally appear in a top-down development of a software system. For these reasons, we have based our work on the Dynamic Process Modelling Scheme (DPMS) [WILE78], a formal model that provides more natural descriptions for a wide range of system organizations.

The Dynamic Process Modelling Scheme

One component of DPMS is a modelling language, called DYMOL, that can be used to formulate precise, high-level, procedural descriptions of constituent processes in a concurrent system [WILE80]. A second component of the modelling scheme, called constrained expressions, is a closed form, non-procedural, representation for all the possible behaviors that could be realized by some concurrent system. For an important subset of dynamically-structured concurrent systems these two components of DPMS are related by an effective procedure for deriving the constrained expressions describing the potential behavior of any given DYMOL description of a system. In previous work the Dynamic Process Modelling Scheme has been used as a basis for concurrent software design methods, including informal analysis of the potential behavior represented by a design [WILE79,WILE80], and has been employed in studying formal properties, such as decidability questions, regarding the class of concurrent systems with dynamic structure [WILE78]. Our most recent efforts have been aimed at increasing the usefulness of DPMS as an aid to software system developers by strengthening and formalizing the analysis capabilities associated with the scheme. In the remainder of this section we summarize the relevant features of the Dynamic Process Modelling Scheme. We first describe the computational model on which DPMS is based, then discuss the DYMOL language and constrained expressions.

In DPMS, a dynamically-structured concurrent system is considered to be composed of individual sequential processes, communicating with one another by means of message transmission. Each individual process is an instance of one of a finite number of distinct classes of potential processes. Each class is described by a template, i.e., a generic program written in DYMOL. This DYMOL template precisely specifies the ways in which processes of the class may interact with other processes, through message transmission or by creating or destroying processes, but only abstractly describes the local, internal activities of the process itself. Thus, DPMS descriptions focus on process organization and interaction, which is the appropriate orientation for design description and analysis, rather than on internal process activity.

Message transmission as modelled in DPMS is both a communication and a synchronization mechanism. A process may, using an appropriate DYMOL instruction, send a message through an outbound port into a link associated with that port. The link is essentially an unbounded, unordered repository that is used to mediate the asynchronous message transmission activity of DPMS processes. Having sent a message, the sending process may continue with subsequent activities as described by its DYMOL program. The message will reside in the link until some process requests receipt of a message, using another DYMOL instruction, through one of its inbound ports that currently is connected to the link by a channel. At that point, assuming no competing requests have been lodged in the interim, the message will be removed from the link and placed into the buffer of the requesting process. If no messages are currently residing in any of the links currently connected to the designated inbound port when a receive request is lodged, the requesting process simply waits. The wait continues at least until a message becomes available in a link connected to the designated inbound port, or until a link containing a message is connected to the designated port by a newly established channel. (Both of these obviously must result from activities of processes other than the waiting process.) Neither the appearance of a message nor the opening of a channel will necessarily end a wait, however, since competing requests might be lodged in the interim and requests need not be serviced in the order in which they were made. Clearly, a process could wait for receipt of a message indefinitely.

The DYMOL language is a simple programming-like language whose syntax is based on Algol 60. Among its features are instructions for message transmission (SEND and RECEIVE), and a standard set of control flow constructs. Dynamic structure can be described using DYMOL instructions for communication channel manipulation (ESTABLISH and CLOSE) and process creation and destruction (CREATE and DESTROY). Decisions based upon internal process computation are modelled as non-deterministic choices (e.g., IF INTERNAL TEST ... or WHILE INTERNAL TEST DO ...). An example of a DYMOL description appears below in Figure 1 while further details on DYMOL can be found in [WILE80].

Constrained expressions are a closed form, non-procedural representation of concurrent behavior in the same sense that regular expressions [KLEE56] are a closed form, non-procedural representation of the behavior of finite state machines. In fact, the operators used in constrained expressions include the standard regular expression operators (concatenation, alternation, transitive closure) as well as two operators (interleaving and its transitive closure) used to represent concurrent activity. A constrained expression is formed by using these operators to combine symbols from an alphabet of <u>events</u> in the system being described into a collection of subexpressions, one subexpression for each process in the modelled system. The interleave of these subexpressions then represents the unconstrained set of possible system behaviors, ignoring such fundamental properties as the necessity of a message's being sent before it can be received or a channel's being opened before it can be used in message transmission. The required fundamental properties are formally described by a second collection of subexpressions, called the <u>constraint set</u>. Then the set of behaviors (or, in formal terms, the language over the event alphabet) described by the overall constrained expression is just what remains after the unconstrained set of behaviors is filtered by the constraint set. (This filtering process can be formally defined as a set intersection [WILE78].) Although constrained expressions are a general mechanism for describing concurrent system behavior, for our purposes we use a particular alphabet of events focusing on message transmission activities, in keeping with the DYMOL orientation. Specifically, we formulate expressions in terms of such events as send events (e.g., the sending of a message through outbound port x, denoted s(x)), receive events (e.g., the receiving of a message through inbound port y that was sent through outbound port x, denoted r(x,y)), and wait events, where the wait event w(y) occurs when a process waits indefinitely to receive a message through its inbound port y. Examples of constrained expressions appear both in [WILE78] and later in this paper.

Since DYMOL bears a strong resemblance to a programming language, DPMS models are easy to understand and have a natural relationship to standard software specification and design techniques. Because its primitives are message transmission and the creation and destruction of processes, DPMS is suitable for describing a wide range of concurrent system organizations. DPMS focuses on process organization and interaction, and therefore addresses precisely those issues most crucial during specification and design. For these reasons we believe that the Dynamic Process Modelling Scheme is an appropriate basis for both describing and analyzing designs of concurrent software systems.

Analysis Techniques

With DPMS, we can formulate questions about the system under consideration as questions about the occurrence of particular symbols in a behavior. For example, the question as to whether a particular process can become blocked becomes the question of whether there is a behavior with a wait at one of the inbound ports of that process. Similarly, the question of survivability of the system after a certain process is destroyed can be answered by determining the set of behaviors that include the appropriate destroy symbol. To answer such questions, we use the derived expressions corresponding to the processes to generate a collection of equations and inequalities involving the number of occurrences of various symbols in various segments of a behavior. Using these relations, we attempt to establish whether a behavior with the specified set of symbols exists.

These algebraic relations are generated in the following way. Certain fundamental properties of the underlying computational model, including those expressed by the constraint set, can be regarded as conditions on the collection of events preceding the occurrence of a given event. In terms of behavior strings, these conditions produce a set of equations and inequalities involving the numbers of occurrences of various symbols in the segment of the string preceding a given symbol. To determine whether there is a behavior containing a specified pattern of symbols, we begin by assuming that these symbols occur in a string, and then generate the relations for the segments that would precede them. These relations in turn involve occurrences of other symbols, and we generate new relations on the segments preceding these. Continuing in this fashion, we attempt to determine whether the relations are inconsistent, in which case no behavior contains the specified pattern. If the relations are consistent, we use them in an attempt to produce a behavior containing the pattern.

To make this procedure clearer, we describe below how the equations and inequalities are generated in a restricted version of the modelling scheme, and use them to determine the possible behavior of a particular system. For simplicity, we will assume that there is no dynamic structure and that the only control construct is WHILE INTERNAL TEST DO. In this case the derived expressions corresponding to the system's processes are themselves made up of subexpressions of the form A1 or (A1 A2...An)*, where the elements Ai are either send, receive, or wait symbols, or unions of these symbols. We refer to subexpressions of the first type as <u>simple phrases</u> and those of the second as <u>compound phrases</u>, and we speak of the occurrence of any symbol from Ai in a string as an occurrence of the element Ai.

Now consider the collection of derived expressions corresponding to the processes of a given system. A behavior is a string composed of occurrences of elements from the phrases in those derived expressions. We focus our attention upon an (arbitrarily selected) occurrence within the behavior of the i<u>th</u> element from the j<u>th</u> phrase in some derived expression. The properties of the computational model underlying DPMS lead to the following conditions on the symbols appearing in the segment of the string preceding this occurrence.

(1) Simple phrases in this derived expression preceding the jth phrase must appear exactly once.

(2) All elements in a compound phrase in this derived expression preceding the jth phrase must appear the same number of times. (This number can, of course, be zero.)

(3) No symbol from a phrase in this derived expression following the jth phrase can appear.

(4) If the jth phrase is (A1...An)*, Ai,Ai+1, ..., An must each appear exactly k times and A1, ..., Ai-1 must each appear exactly k+1 times for some $k \geq 0$.

(5) No w(y) or STOP from this derived expression can appear.

(6) If the symbol occurring is an r(x,y), y must be connected to x and the number of appearances of s(x) must be greater than the total of the number of appearances of symbols r(x,z) as z ranges over the inbound ports connected to x.

(7) If the symbol occurring is a w(y), the total number of appearances of symbols s(x), where x ranges over the outbound ports connected to y, must equal the total number of appearances of symbols r(x,z), where x ranges over the outbound ports connected to y and z ranges over all the inbound ports.

The behavior must also satisfy the conditions

(8) If a w(y) occurs in the behavior, the equality of (7) must hold for the behavior considered as a whole.

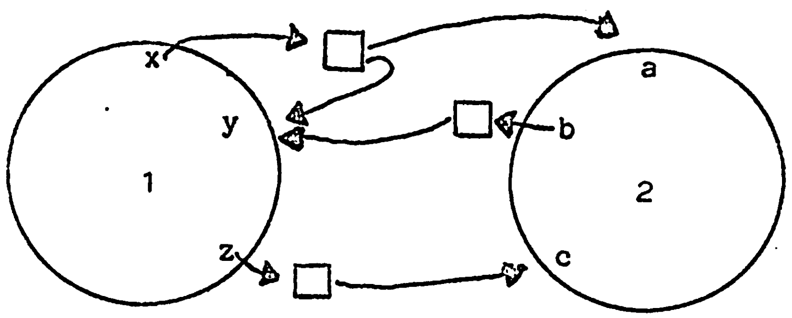
and

(9) Exactly one of the symbols w(y) and STOP from each derived expression appears.

The first five conditions insure that symbols in the behavior are taken in the appropriate order from each derived expression, and no further symbols are taken if the corresponding process becomes blocked. Condition (6) is simply the requirement that there be a message to be received when an r(x,y) appears. Condition (7) says that a process may not become blocked if there are messages it can receive, while the eighth condition requires that there be no messages left at the end of a behavior which could be received by a blocked process. The last condition says that a process must be blocked or terminate normally.

Now consider the system of Figure 1. The derived expressions are (s(x) (r(x,y) u r(b,y) u w(y)) s(z))* STOP1 and ((r(x,a) u w(a)) (r(z,c) u w(c)) s(b))* STOP2. Conceivably, this system could reach a deadlock state where process 1 is attempting to receive at y before sending at z and process 2 is attempting to receive a message from z at c before sending at b. We will use conditions (1)-(9) to see that this can only occur in the behaviors s(x)r(x,a)w(y)w(c) and s(x)r(x,a)w(c)w(y). In fact, we show that this is the only way the second process can become blocked waiting for a message through its inbound port c.

Suppose that process 2 does become blocked at c, i.e., that a w(c) appears in a behavior. By (8), we must have the number of appearances of s(z) in the string equal to the number of appearances of r(z,c); we write this as $|s(z)|=|r(z,c)|$ and use subscripts on the cardinality symbols to distinguish counts in different segments of the behavior where this is necessary to avoid confusion. Conditions (4) and (9) together imply that $|r(z,c)|=|r(x,a)|-1=|s(b)|$, so $|s(z)|=|s(b)|$, and that



```
PROCESS 1:                          PROCESS 2:

WHILE INTERNAL TEST DO              WHILE INTERNAL TEST DO
   BEGIN                               BEGIN
      SEND x;                             RECEIVE a;
      RECEIVE y;                          RECEIVE c;
      SEND z                              SEND b
   END                                 END
```

The arrows indicate channels connecting outbound port x to inbound ports a and y, outbound port z to inbound port c, and outbound port b to inbound port y. Square boxes represent links.

Figure 1

$$|s(z)|=|r(x,y)|+|r(b,y)|=\begin{cases} |s(x)| & \text{if no } w(y) \text{ appears in the behavior} \\ |s(x)|-1 & \text{if a } w(y) \text{ appears in the behavior} \end{cases}$$

By (6), $|s(x)|+|s(b)|-|r(x,a)|\geq|r(x,y)|+|r(b,y)|=|s(z)|$. Now if w(y) does not appear, we have $|s(x)|=|s(z)|$ so $|s(b)|-|r(x,a)|\geq 0$. But we saw that $|s(b)|=|r(x,a)|-1$. Thus, a w(y) must appear in the behavior.

Condition (6) implies that $|s(x)|-|r(x,a)|\geq|r(x,y)|$. Since a w(y) occurs, the arguments above show that $|r(x,a)|-1=|s(b)|=|s(z)|=|s(x)|-1$, whence $|r(x,a)|=|s(x)|$. Since all counts are nonnegative, this implies that $|r(x,y)|=0$. The occurrence of a w(y) also implies, by (8), that $|s(x)|-|r(x,a)|+|s(b)|=|r(x,y)|+|r(b,y)|$, so $|s(b)|=|r(b,y)|$.

Suppose that $|s(b)|\geq 1$. Let $|\text{symbol}|_1$ denote the number of occurrences of that symbol in the segment preceding the first s(b). Condition (4) says that $|r(z,c)|_1=1$, and then condition (6) says that $|s(z)|_1\geq 1$. It then follows from (4) that $|r(x,y)|_1+|r(b,y)|_1\geq 1$. We have seen that $|r(x,y)|=0$, so $|r(x,y)|_1=0$ and $|r(b,y)|_1\geq 1$. Thus, an r(b,y) occurs in the segment preceding the first s(b), which violates (6). We conclude that $|s(b)|=0$. The possible behaviors are then s(x)r(x,a)w(y)w(c) and s(x)r(x,a)w(c)w(y).

This approach extends easily to less restricted versions of the modelling scheme allowing more than one message type, certain types of dynamic structure, and additional control constructs. In each case, we can give a general set of conditions which, together with the derived expressions, are used to generate a system of algebraic relations involving the numbers of appearances of various symbols in segments of a behavior. These relations are used in turn to determine the class of behaviors which include particular sets of symbols.

Conclusion

We believe that a suitable formalism for describing and analyzing concurrent systems is a prerequisite to useful tools supporting the work of concurrent software system developers. In this paper we have discussed alternative approaches, indicating why we find them inappropriate, and then described our approach to concurrent software system description and analysis. Our descriptive formalism is based on the Dynamic Process Modelling Scheme, whose DYMOL language abstractly describes concurrent systems as collections of sequential processes communicating via message transmission. Our analysis technique involves generating a collection of equations and inequalities from a DYMOL description. These equations and inequalities involve the number of occurrences of various events in various segments of a posited behavior, and can be used to establish whether a behavior consisting of the specified events can possibly occur. Both the descriptive formalism and the analysis technique have been chosen for their suitability as foundations for useful concurrent software development tools.

The analysis techniques described here cannot yet be applied to systems whose descriptions require the full generality of the DPMS scheme; the creation of processes presents the major remaining difficulty. Our current work is directed to a solution of this problem, and to the efficient implementation of these techniques as a concurrent software development tool.

Bibliography

[ADAM68] D. A. Adams, A Computation Model with Dataflow Sequencing, Computer Science Department, Stanford University, Technical Report CS-117 (December 1968).

[APT80] K. Apt, N. Francez, and W. DeRoever, "A Proof System for Communicating Sequential Processes, ACM Transactions on Programming Languages and Systems (July 1980), pp. 359-385.

[BOEH73] B. Boehm, "Software and Its Impact: A Quantitative Assessment," Datamation, vol.19, No.5 (May 1973), pp. 48-59.

[BOYE75] R. Boyer and J. Moore, "Proving Theorems About LISP Functions," Journal of the ACM (January 1975), pp. 129-144.

[CLAR81] L. Clarke, R. Graham, and J. Wileden, "Thoughts on the Design Phase of an Integrated Software Development Environment," Proceedings of the 14th Hawaii International Conference on Systems Science, Honolulu (January 1981).

[FLOY67] R. W. Floyd, "Assigning Meaning to Programs," Proceedings of Symposia in Applied Mathematics, Mathematical Aspects of Computer Science (1967), pp. 19-32.

[HABE72] A. N. Habermann, "Synchronization of Communicating Processes," Communications of the ACM (March 1972), pp. 171-176.

[HOAR69] C. A. R. Hoare, "An Axiomatic Basis of Computer Programming," Communications of the ACM (October 1969), pp. 576-580, 583.

[HYMA66] H. Hyman, "Comment on a Problem in Concurrent Programming Control," Communications of the ACM (January 1966), p. 45.

[KAHN74] G. Kahn, "The Semantics of a Simple Language for Parallel Processing," Information Processing 74, North-Holland, Amsterdam, 1974.

[KELL76] R. M. Keller, "Formal Verification of Parallel Programs," Communications of the ACM (July 1976), pp. 371-384.

[KLEE56] S. Kleene, "Representations of Events in Nerve Nets and Finite Automata," Automata Studies, Ann. Math. Studies no.34, Princeton University Press, 1956, pp. 3-41.

[LAMP77] L. Lamport, "Proving the Correctness of Multiprocess Programs," IEEE Transactions on Software Engineering (March 1977), pp. 125-143.

[MANN74] Z. Manna, Mathematical Theory of Computation, McGraw-Hill, (1974).

[MILN79] G. Milne and R. Milner, "Concurrent Processes and Their Syntax," Journal of the ACM (April 1979), pp. 302-321.

[OWIC76] S. Owicki and D. Gries, "Verifying Properties of Parallel Programs: An Axiomatic Approach," Communications of the ACM (May 1976), pp. 279-285.

[PETE77] J. Peterson, "Petri Nets," ACM Computing Surveys (September 1977), pp. 223-252.

[TENN76] R. Tennent, "The Denotational Semantics of Programming Languages," Communications of the ACM (August 1976), pp. 437-453.

[WILE78] J. Wileden, "Modelling Parallel Systems with Dynamic Structure," Department of Computer and Information Science, University of Massachusetts, COINS Technical Report 78-4 (January 1978).

[WILE79] J. Wileden, "Relationships Between Graph Grammars and the Design and Analysis of Concurrent Software," Lecture Notes in Computer Science, vol.73, Spring-Verlag, Berlin, 1979.

[WILE80] J. Wileden, "Techniques for Modelling Parallel Systems with Dynamic Structure," Journal of Digital Systems, 4,2 (Summer 1980), pp.177-197.