Talk Notes

on

Mathematical Studies of Parallel Computation

Raymond E. Miller

Research Report #89

October 1976

Supported in part by the Office of Naval Research under Grant N00014-75-C-0752.

I. Introduction

In this talk I first want to discuss some of the motivation for studying parallel computation from a formal or mathematical point of view. Then I hope to briefly review some of the different types of studies of parallel computation in order to place into perspective the role played by mathematical models of parallel computation. Finally, I will introduce a number of mathematical models of parallel computation, with a threefold purpose in mind; first, to discuss some of the basic properties that have been isolated and studied; second, to highlight some general approaches to proofs that have proved useful; and third, to describe relationships between some of the different models. A fairly large selected bibliography is included in these notes as an aid to those who might wish to delve deeper into parallel computation.

II. Motivation for Studying Parallel Computation

Probably the most obvious reason for looking at parallel computation is the hope of obtaining significant speedup in large computations. Certainly there is some basis for this. Some special parallel machines have been designed, and even some built, which show dramatic speed advantages on particular types of problems over more standard general purpose machine designs. Also, even though most machine speedup has been accomplished through dramatically increasing the speed of the raw computer circuits, rather than additions of parallelism, it has been long said that the limits of direct circuit speedups are being reached. Thus parallelism may play an ever increasing role in future computer speedup.

Another related reason for studying parallelism is the increased appearance of parallel machine architectures. With such machines here, or nearly here, there is a challenge to learn how to use them efficiently, even though most programming languages only have, at most, rudimentary facilities to express parallel sequencing. This challenge spans a broad spectrum from inventing new algorithms which are parallel in nature, to introducing more flexible facilities to utilize parallelism both in terms of programming statements and hardware implementations.

Even though these reasons are compelling, there is a probably much more fundamental reason for studying parallelism in computation. This is simply that it is not well understood, has so far defied precise quantification, and if understood could not only improve parallel computation but could even be helpful in improving sequential calculations.

Questions of the following variety arise. The fundamental arithmetic operations used in calculations are well understood. The arithmetic complexity of computations can thus be studied and in some very important cases is already starting to be well understood. The blossoming area of theoretical computer science called complexity theory is essentially this area of study. Yet, there is more to computation than pure arithmetics. These include data structures and parallelism. Appropriate sequencing, and parallelism could further reduce the time and space complexity of a problem. What, however, are the fundamental operations for sequencing or parallelism control? Or, for that matter, is there even any fundamental set of operations for this? These questions will probably have to be settled before any significant progress can be made on understanding complexity issues in parallel computation. However, so far essentially no progress has been made. Another basic question, on which some progress has been made, is the isolation of inherent properties of parallelism to describe the amount of parallelism and correct behavior in parallel computation. Some of these topics are the main part of my talk. The properties have acquired names like deadlocks, determinism, boundedness, safeness, etc. Before we get into this, however, we should show where studies of these formal properties fit into the more general framework of parallel computations.

III. Types of Studies

In order to make parallel computation a reality one needs both parallel machines and parallel solutions to problems. A recent survey of associative and parallel computers is given by Thurber [141]. Although this survey stresses associative processing techniques, a number of parallel machine architectures are also described. The most obvious way to realize parallelism in a machine is to have a direct implementation of vectors and arrays by having a set of processors each of which operates on one component in the array or vector with the same arithmetic or logical operation. This allows a simplified machine control requiring only a single instruction stream to control all processors. Also, since vector and matrix notation is commonly used for expressing large computational problems it gives the hope that natural parallel algorithms will follow for such machine structures. Many such machines have been proposed [14,22,23,48,50,105, 131,132] with probably the most widely known development being the ILLIAC IV [14] which has actually been built and has demonstrated the feasibility of such machines for classes of problems that fit well into such a format. However, such machines appear to be rather special purpose, since there can be considerable degradation of performance on problems that do not match well with the structure and size of the particular machine. Rather than an array of processors some machines "pipeline" the operations through processors, to gain their speed; e.g. [5,24,73,132,138]. Since all of these machines are restricted to essentially one stream of instructions, they are somewhat restrictive in how they can exploit parallelism. Various ideas using multiinstruction streams have been proposed [23,28, 40,42,84,131], and even more drastic proposals in which the actual computer structure could dynamically change, restructuring itself to fit the problem, have been proposed [34,95,118,124,137,139] but these seem further from reality.

Another area of study is programming constructs and analysis for parallelism. If one is going to allow a program stream to break into several program streams and recombine in various ways then constructs are needed in the programming language to facilitate this. Various ideas have been proposed: FORK and JOIN [28,36], parallel DO's, TASKING, etc. Also, along with this, operations for sequencing of parallel, and possibly asynchronous, events have been proposed: LOCK-UNLOCK, SEMAPHORES [38], etc. Now with the possibility of multiple instruction streams a number of questions arise. Given a program, could it be made "more parallel?" (See e.g. [15,45,75,81,83,100,117,140]). How does one schedule and allocate facilities to the program both for efficient processing and efficient use of facilities? (See e.g., [9,26,31,32,45,46,70,120]). Will the program run correctly or could it deadlock somewhere in the process? (See e.g., [63,111]). Much work has been done on these problems. Solutions to particular problems have been proposed, such as for the mutual exclusion problem [39,78]. Also, these problems are formalized and studied in the mathematical modelling studies.

A third type of study in parallelism involves the development of parallel algorithms for various classes of computational problems such as polynomial evaluation [104], systems of difference equations [70], iterative methods [144], etc. Miranker [101] gives a survey of numerical analysis type of work in this area, and much has been done subsequently. Most of this work assumes a machine with multiple units that can be doing different operations simultaneously, and that the results done by one processor are readily available to any other processor. This idealization certainly simplifies the question, and would be what one would ideally desire for a machine, but as we have seen from the parallel machine discussion it does not match actual machines which constrain what can be done in parallel as well as the way in which data can flow from one processor to another. Nevertheless, many important results have been obtained, particularly about the inherent limitations of speedup obtainable from parallel operation.

In contrast to these first three areas of study of parallelism that deal with more or less practical matters of parallel computation; that is, designing parallel computers, programming them, and finding "good" algorithms for them, the studies of theoretical models of parallel computation and the complexity in parallel computation are considerably less applied. To make an analogy, the models of parallel computation are to parallel computation like the finite state machine model is to sequential circuits. In the models one hopes to be able to represent classes of behaviors; classifying them, understanding the basic properties of these behaviors, and using them to model practical siuations. Thus, for example, within a model one might be able to specify precisely what "more parallel" means and derive procedures to obtain a more parallel form. The other questions of scheduling, allocation, correct operation, etc. also can be made precise and studied. For example, one such thing concerning correct operation which has been so isolated and studied is determinism. Determinism is now recognized as one of the important and inherent properties of parallel computation, and various means for insuring determinism have been obtained. Before this was isolated, through the study of theoretical models, there seemed to be considerable confusion over ideas such as races, conflicting sequencing situations, errors in interlocking, etc. All of this became much clearer through the formulation of the notion of determinism. The use of the theoretical models to represent practical problems has also been fruitful in isolating errors in proposed solutions and in clarifying the practical sequencing problems. Thus, the role of the theoretical modelling studies of parallel computation is to provide a formalism or "language" to represent and study practical problems of parallel computation, supplying both an understanding of various types of behaviors and properties and of providing a means of finding equivalent but, in some sense, "better" realizations of the same problem. Now let's look at some of these models and see what we know so far.

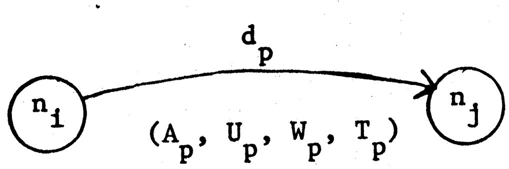
IV Mathematical Models of Parallel Computation

The theoretical models for parallel computation, which started to be developed in the early 1960's, still do not provide a unified framework for studying parallel computation. Some models are graph theoretic in nature, providing a flowchart-like representation. Others include more automata-like ideas with the analysis using the idea of the instantaneous descriptions. Also, some of the production systems of logic (such as Post systems and λ-calculus) can be viewed as types of parallel computation formalisms. Some work has been started to unify these approaches, but this has not progressed far enough to provide a uniform treatment of these models and their relationships to one another.

A. Various Types of Models

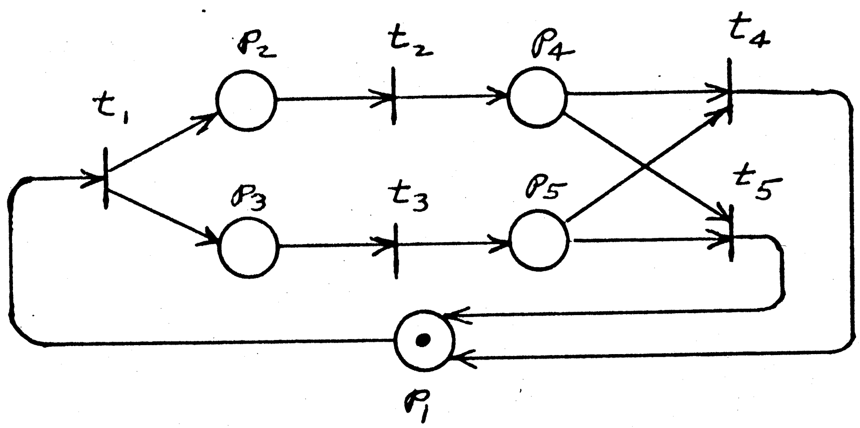
Graph Models

In the early 1960's a group at UCLA under Professor Estrin worked on the notion of improved computer structure through a "fixed plus variable" structure [41]. This led to the study of parallelism which they modelled, for the study of scheduling and allocation, via acyclic graph structures. At M.I.T., Rodriguez developed a graph model [123]. Karp and Miller [69], through studying possible additions to speedup computers by having special purpose devices perform macroinstructions, developed a graph model now commonly called a computation graph in which nodes were used to represent individual operations and directed edges were used to represent fifo queues of working data. Parameters attached to the edges specified the queue behavior. Thus if an edge d_p was directed from node n_i to n_j it had four parameters A_p, U_p, W_p, T_p



where A_p represented the initial number of items in a queue, U_p the number of items added to the queue each time n_i fired, W_p the number of items removed from the queue by each firing of n_j, and T_p the number of items needed (the "threshold") by n_j for it to be activated. Computation graphs do not allow conditional branching, due to the firing rules, but it was shown that they necessarily provided determinate computation and algorithms were obtained for their termination and queue length values. Also special scheduling results were obtained [120].

Petri [116], in the early 1960's also developed a graph model now commonly called a Petri net, and this has received considerable study. A simple Petri net is shown in the figure.



The graph has two types of nodes depicted by circles and bars called "places" and "transitions" respectively. Places can hold "tokens" and these control the firing of the transitions. A transition can fire if all of the places entering it have tokens. The firing of a transition removes a token from each incoming place and adds a token to each outgoing place of the transition. Thus, in this example t_1 is the only fireable transition (the dot indicates a token) and when t_1 fires it removes the token from p_1 and places tokens in p_2 and p_3. Then t_2 and t_3 are fireable so "parallel computation" is represented. The structure p_4, p_5, t_4, t_5 shows a "conflict" situation. When a token is in both p_4 and p_5 then both t_4 and t_5 are fireable. However, not both can fire since in firing each requires a token in both p_4 and p_5 to be removed. Thus the global rule is imposed that a token can be used in only a single firing, and in this example this means that an arbitrary choice must be made on whether to fire t_4 or t_5. Petri nets have been studied extensively

[4,11-13,27,35,51-59,61,62,66-68,74,79,99,102,108-116,133,134,137,145] and are still not completely understood. A rather comprehensive survey of Petri nets is given by Peterson in [113]. A simplified Petri net called a marked graph restricts each place to have exactly one input transition and one output transition. These turn out to be special kinds of computation graphs [96] in which $U_p = W_p = T_p = 1$ for each edge, and are well understood.

One of the very intriguing aspects of Petri nets is the simple and illustrious way in which they represent parallel sequencing. Some researchers have enriched the model by various techniques. For example, by providing tokens of different colors, by inhibitor edges, and by timings [2,3,9,65,93,137]. It appears that any such addition, although quite helpful for representing certain behaviors, turns the model into one that can simulate a Turing machine, and in that sense makes it hopeless to completely analyze.

Schemata Models

Two basic types of schemata models exist. One is based on having a finite set of operations operating on a common memory, and whose control of the operations is done by some sort of automata theoretic construct [72,75,90,136]. Thus we have a schema $\mathcal{S} = (M,A,\mathcal{T})$ where M is the memory, A is the set of operations and $\mathcal{T}$ is the control. The models are usually uninterpreted models or partially uninterpreted models meaning that the particular functions and decisions associated with the operations are not specified.

A second type of schemata model is based upon elementary operation schemas (usually a finite set of them) which are interconnected to form a data-flow schema [37,80,129]. In these, rules of interconnection are often specified in order to insure determinacy of the interconnected schema. That is, we have sufficient conditions for determinacy. In contrast, in the (M,A,$\mathcal{T}$) schemata one develops constraints on the schemata (usually global in nature) from which necessary and sufficiency of determinacy follow.

The more purely automata type models vary anywhere from finite automata forms [17,18] to parallel random access programmed machine in nature. A special iterative form has been studied [87,88] in which some complexity types of results have been obtained.

B. Basic Properties and Proof Techniques

As we have remarked earlier determinacy is one of the better understood properties of parallel computation. It takes several different forms in the different models, but in essence it means that the outcome of the computation is unique and does not depend upon the particular relative times that operations are allowed to be performed. The computation graph is by its structure always determinate, as are some of the data-flow schemata. In terms of schemata one can envision different types of determinacy. The one studied in [72] is a very strong type of determinacy. It means that for any memory location the complete sequence of values that appear in the location during computation under a given interpretation is independent of how the individual operations were sequenced. Necessary and sufficient conditions are developed for such determinacy and they are shown to be essentially the Bernstein conditions [15] on overlap on domain and range locations of operations. Also, for a broad class of schemata, namely repetition-free, lossless, persistent, commutative, counter schema it is shown that determinacy is decidable. The technique for showing this is a more-or-less standard sliding argument which is used in Church-Rosser type theorems which allows one to slide symbols of one sequence of operations to match another sequence without changing memory values. Another aspect of the proof involves vector addition systems of which we say more later. A rather surprising aspect of the decidability of determinacy (as well as other properties) is its lack of "stability." It has been shown [94] that if the single property of repetition-free is removed from the hypothesis then determinacy becomes undecidable. This boundary between the decidability and undecidability can be viewed as the most rudimentary measure of complexity, although some of the properties are known to be quite complex [85] even though they are decidable.

Normally, this strong form of determinacy is more than really desired. Often one would only require the final values (assuming termination) of two computation sequences to match on either all, or a specified subset, of memory. The strong determinacy of course implies this weaker "output determinacy" but little is known how to obtain output determinacy without requiring determinacy throughout the sequence.

The determinacy property does not arise directly in terms of Petri nets. This is because the Petri net does not have interpreted functional operations, nor does it have a formal way, like interpretations for schemata, of adding them. Thus any such questions must be dealt with outside the Petri net model. The conflict situation in Petri nets does, however, give rise to an obvious situation that looks like it would lead to indeterminacy. Also, it has been shown to be intimately connected with deadlocks.

Other properties of interest include: termination, i.e. how many times the operations of the model are performed; boundedness, i.e., the number of operation performances that can be done concurrently; and the number of control states that are reachable in computations. For schemata all of these properties are decidable in a manner similar to determinacy, and become undecidable without repetition-freeness assumed. For computation graphs rather straightforward algorithms for boundedness and termination can be derived. In Petri nets boundedness is defined in terms of the maximum number of tokens that can reside in any place at any moment. A net is called "safe" if this bound is one. Termination is expressed by the term "liveness" in a Petri net. A transition in a Petri net is called "live" if from any reachable token distribution it is possible to reach a situation in which the transition is fireable. Boundedness and safeness follow directly from the decidability of a problem in vector addition systems whereas liveness is equivalent to the "reachability problem" in vector addition systems.

Vector addition systems first arose in parallel program schemata [72] and later were seen to be, in some ways, equivalent to Petri nets [54,99,115]. Since they are a simple mathematical construct, and since they underlie many problems concerning parallel computation, I will diverge for a few moments to discuss vector addition systems.

A vector addition system in r-dimensions consists of a pair $\mathcal{W} = (d,W)$ where d is on r-dimensional vector of nonnegative integers, and W is a finite set of r-dimensional integer vectors.

The reachability set $R(\mathcal{W})$ is the set of points in the first orthant that can be reached from d by successively adding vectors in W such that the path of points so formed always remains in the first orthant.

In [72] it was shown that it was decidable, given $\mathscr{W}$ and a point x, whether there existed some point $y \geq x$ for which $y \in R(\mathscr{W})$. The decidability of this "simple" problem provided decidability of determinacy, boundedness, and termination for schemata and boundedness and safeness for Petri nets. Rabin [13] showed, however, that given two vector addition systems $\mathscr{W}$ and $\mathscr{W}'$ it was undecidable whether $R(\mathscr{W}) \subseteq R(\mathscr{W}')$. Later, using a Petri net construction, Hack [51] showed that the question of whether $R(\mathscr{W}) = R(\mathscr{W}')$ was also undecidable. The "reachability problem" for a vector addition system is; given x is $x \in R(\mathscr{W})$? This problem remained open for a number of years. Hack showed that this was equivalent to the liveness question for Petri nets. It appears that some recent work of Sacerdote and Tenney has succeeded in proving that the reachability problem is decidable, but that their technique gives an upper bound substantially above Lipton's lower bound [85]. The decidability of this problem would substantially aid in the analysis of Petri nets.

Returning to the basic properties of models, probably one of the most basic questions is that of determining whether two models of a certain type, say two schemata, are equivalent. For schemata this was shown to be undecidable by encoding the Post correspondence problem in schemata.

Still other properties are of interest. Can one determine a maximum parallel version for an instance of a model? Keller [76] and Slutz [136] have studied maximum parallelism in terms of types of schemata. Scheduling has been studied for very restricted situations [31,32,70,120] but much still remains open. Modifying memory use to either economize on memory or increase parallelism was studied by Logrippo [89,90] under the term renamings in schemata. Finally the composition and decomposition within a model has been studied [19,77,97] in an attempt to allow one to model program pieces in one stage which are later interconnected at a later stage. The questions of synchronous versus asynchronous operation are still ill understood. Many of the models are basically asynchronous in nature having a lot in common with Muller's asynchronous switching circuits [102]. On the other hand, scheduling and allocation work usually assumes a form of synchronous computation. In [87,88] a bounded time asynchronism was introduced for a linear iterative type of structure. In this way the relationship between synchronous versus asynchronous computation could be studied, and a bound between the two types of computation was obtained. A similar result is also possible directly in terms of vector addition systems but this work has not yet been completed.

C. Applications of the Models

The theoretical models for parallel computation are finding their way into applications in several different ways. Petri nets and their generalizations have been used to model both hardware and programming systems [34,35,52,65,74,93,102,108,109,110,114,121,122,133,134,137,145]. Schemata based models, although more complex have also been applied to various problems [1,16,31,32,73,80,125-127].

[1] Adams, D.A., "A model for parallel computations," in Parallel Processor Systems, Technologies, and Applications, L. C. Hobbs, et al., Ed. Washington, D.C. :Spartan, 1970, pp. 311-333.

[2] Agerwala, T., "A complete model for representing the coordination of asynchronous processes," Hopkins Computer Research Report #32, Computer Science Program, The Johns Hopkins University, (July 1974),

[3] Agerwala, T., "An analysis of controlling agents for asynchronous processes," Hopkins Computer Research Report #35, Computer Science Program, The Johns Hopkins University, (August 1974).

[4] Agerwala, T. and Flynn, M., "Comments on capabilities, limitations, and 'correctness' of Petri nets," in Proceedings of the 1st Annual Symposium on Computer Architecture, Lipovski, G.J. and Szygenda, S.A. (Eds.) University of Florida, (December 1973), pp. 81-86.

[5] Anderson, D.W., F.J. Sparacio, and R.M. Tomasulo, "Machine philosophy and instruction handling," IBM J. Res. Develop., Vol. 11, Jan. 1967, pp. 8-24.

[6] Aschenbrenner, R.A.; Flynn, M.J.; and Robinson, G.A., "Intrinsic multiprocessing," Proc. AFIPS, 1967 Spring Jt. Computer Conf., 30, AFIPS Press, Montvale, N. J., 1967, pp. 81-86.

[7] Baer, J.L. and E.C. Russel, "Preparation and evaluation of computer programs for parallel processing systems," in Parallel Processor Systems, Technologies, and Applications, L.C. Hobbs, et al., Ed. Washington, D.C.: Spartan, 1970, pp. 375-415.

[8] Baer, J.L., D.P. Bovet, and G. Estrin, "Legality and other properties of graph models of computations," J.Assoc.Comput. Mach., 17, July 1970, pp. 543-552.

[9] Baer, J.L., "A survey of some theoretical aspects of multiprocessing," ACM Computing Surveys, Vol. 5, No. 1, March 1973, pp. 31-80.

[10] Bährs, A.A., "Operation patterns (an extensible model of an extensible language)," Int'l Symp. Theoretical Programming, Novosibirsk, USSR, Aug. 7-11, 1972, Lecture Notes in Computer Science, Vol. 5, Springer-Verlag, 1974, pp. 217-246.

[11] Baker, H.G., "Petri nets and languages," Computation Structures Group Memo 68, Project MAC, M.I.T. (May 1972).

[12] Baker, H.G., "Equivalence problems of Petri nets," S.M.Thesis, Department of Electrical Engineering, M.I.T., (June 1973).

[13] Baker, H.G., "Rabin's proof of the undecidability of the reachability set inclusion problem of vector addition systems," Computation Structures Group Memo 79, Project MAC, M.I.T., (July 1973).

[14] Barnes, G.T., et al., "The ILLIAC IV computer," IEEE Trans. Comput., C17 pp. 746-757, August 1968.

[15] Bernstein, A.J., "Analysis of programs for parallel processing," IEEE Trans. Electron. Comput., EC15, pp. 757-763, October 1966.

[16] Bradshaw, F.T., "Structure and representation of digital computer systems," Ph.D. dissertation, Case Western Reserve Univ., Cleveland, Ohio, Jan. 1971.

[34] Dennis, J.B. and D.P. Misunas, "A Computer Architecture for Highly Parallel Signal Processing," Proceedings ACM Annual Conference, November 1974, pp. 402-409.

[35] Dennis, J.B., "Modular asynchronous control structures for a high performance processor," in Rec. Project MAC Conf. Concurrent Syst. and Parallel Computation. New York: Assoc.Comput.Mach., 1970, pp. 55-80.

[36] Dennis, J.B., and E.C. Van Horn, "Programming semantics for multiprogrammed computations," Comm. Assoc. Comput. Mach., 9 pp. 143-155, Mar. 1966.

[37] Dennis, J.B., J.B. Fosseen, and J.P. Linderman, "Data flow schemas," Int'l Symp. on Theoretical Programming, Novosibirsk, USSR, Aug. 7-11, 1972. In Lecture Notes in Computer Science, Vol. 5, Springer-Verlag, 1974, pp. 187-216.

[38] Dijkstra, E.W., "Co-operating sequential processes," in Programming Languages, F. Genuys, Ed. New York: Academic, 1968.

[39] Dijkstra, E.W., "Solution of a problem in concurrent programming control," Comm. Assoc. Comput. Mach., 8, pp. 569-570, September 1965.

[40] Dill, F.H., "Alternative computer architectures using LSI." IBM Research Report RC 5555, June 1976.

[41] Estrin, G., B. Bussell, R. Turn, and J. Bibb, "Parallel processing in a restructurable computer system," IEEE Trans. Electron. Comput., EC-12, pp. 747-755, Dec. 1963.

[42] Flynn, M.J., A. Podvin, and K. Shimizu, "A multiple instruction stream processor with shared resources," in Parallel Processor Systems, Technologies, and Applications, L.C. Hobbs, et al., Ed. Washington, D.C.: Spartan, 1970, pp. 251-286.

[43] Gill, S., "Parallel programming," Comput J., pp. 2-10, April 1958.

[44] Goldstine, H.H., L.P. Horwitz, R.M.Karp, and R.E. Miller, "On the parallel execution of macroinstructions," IBM Research Report RC-1262, August 17, 1964.

[45] Gonzales, M.J. and C. V. Ramamoorthy, "Recognition and representation of parallel processable streams in computer programs," in Parallel Processor Systems, Technologies, and Applications, L.C. Hobbs et al., Ed. Washington, D.C.,:Spartan, 1970, pp. 335-371.

[46] Gonzales, M.J. and C.V. Ramamoorthy, "Program suitability for parallel processing," IEEE Trans. Comput., C-20, pp. 647-654, June 1971.

[47] Gosden, J.A., "Explicit parallel processing description and control in programs for multi- and uni-processor computers," in 1966 Fall Joint Comput. Conf., AFIPS Conf. Proc., 29. Washington, D.C.: Spartan, 1966, pp. 651-660.

[48] Graham, W.R., "The parallel and the pipeline computers," Datamation, pp. 68-71, April 1970.

[49] Graham, W.R., "The impact of future developments in computer technology," presented at the Joint Air Force and Lockheed Aircraft Conf. Comput. Oriented Analysis of Shell Structures, Aug. 13, 1970.

[50] Gregory, J. and R. McReynolds, "The SOLOMON computer," IEEE Trans. Electron. Comput., EC-12, pp. 774-781, Dec. 1963.

[51] Hack, M., "The equality problem for vector addition systems is undecidable," Computation Structures Group Memo 121, Project MAC, M.I.T., 1975, pp. 1-32.

[52] Hack, M., "Analysis of production schemata by Petri nets," S.M.Thesis, Department of El. Eng., MIT; also MAC TR-94, Project MAC, MIT, (Feb. 1972), Errata Hack, M., "Corrections to 'Analysis of production schemata by Petri nets'," Computation Structures Note 17, Project MAC, MIIT, (June 1974).

[53] Hack, M., "A Petri net version of Rabin's undecidability proof for vector addition systems," Computation Structures Group Memo 94, Project MAC, MIT, (Dec. 1973).

[54] Hack, M., "Decision problems for Petri nets and vector addition systems," Computation Structures Group Memo 95-1, Project MAC, MIT, (Aug. 1974).

[55] Hack, M., "The recursive equivalence of the reachability problem and the liveness problem for Petri nets and vector addition systems," Computation Structures Group Memo 107, Project MAC, MIT, (Aug. 1974), 9 pp; also in 15th Symposium on Switching and Automata Theory, IEEE, New York.

[56] Hack, M., "Petri net languages," Computation Structures Group Memo 124, Project MAC, MIT, (June 1975).

[57] Hansal, A. and G.M. Schwab, "On marked graphs III," Report LN 25.6.038, IBM Vienna Labs, Vienna, Austria, (Sept. 1972).

[58] Henhapl, W., "Firing sequences of marked graphs," Report LN 25.6.023, IBM Vienna Labs, Vienna, Austria, (Feb. 1972).

[59] Henhapl, W., "Firing sequences of marked graphs II," Report LN 25.6.036, IBM Vienna Labs, Vienna, Austria, (June 1972).

[60] Harper, S.D., "Automatic parallel processing," Proc. Computing and Data Processing Society of Canada, Second Conference, (June 1960), 321-331.

[61] Holt, A.W., et al., "Applied Data Res. Inc., Rep. AD676972, Inform. Syst. Theory Project Final Rep., Rome Air Devel. Cen., Contract AF30(602)-4211, Sept. 1968.

[62] Holt, A.W. and F. Commoner, "Events and conditions," in Rec. Project MAC Conf. Concurrent Syst. and Parallel Computation. New York: Assoc. Comput. Mach., 1970, pp. 3-52.

[63] Holt, R.C., "On deadlocks in computer systems," Ph.D. dissertation, Cornell Univ., Ithaca, Jan. 1971; also Dept. Comput. Sci. Tech. Rep. 71-91.

[64] Horwitz, L.P., R.M. Karp, R.E. Miller and S. Winograd, "Index Register Allocation," IBM Research Report RC-1264, August 20, 1964. ACM Journal, Vol. 13, No. 1, pp. 43-61, January 1966.

[65] Irani, K.B. and C.R. Sonnenburg, "Exploitation of Implicit Parallelism in Arithmetic Expressions for an Asynchronous Environment," Dept. of Elec. and Computer Engineering, Univ. of Michigan Report, Ann Arbor Michigan, 1975.

[66] Izbicki, H., "On marked graphs," IBM Lab., Vienna, Austria, Rep. LR 25.6.023, Sept. 1971.

[67] Izbicki, H., "On marked graphs II," Report LN 25.6.029, IBM Vienna Labs, Vienna, Austria, Jan. 1972.

[68] Jones, N.D., L.H. Landweber, and Y.E. Lien, "Complexity of some problems in Petri nets," 1976.

[69] Karp, R.M. and R.E. Miller, "Properties of a model for parallel computations; determinacy, termination, queueing," IBM Research Report RC-1285, September 1964. Also, SIAM J, Vol. 14, No. 6, pp. 1390-1411, November, 1966.

[70] Karp, R., R. Miller, and S. Winograd, "The organization of computations for uniform recurrence equations," IBM Research Report RC-1667, 1966. Also JACM, Vol. 14, No. 3, July 1967, pp. 563-590.

[71] Karp, R. and R. Miller, "Parallel program schemata: A mathematical model for parallel computation," IEEE Conf. Record 8th Annual Symposium on Switching and Automata Theory, pp. 55-61, October, 1967.

[72] Karp, R.M. and R. E. Miller, "Parallel program schemata," IBM Research Report RC 2053, 1968. JCSS 3, pp. 147-195, May, 1969.

[73] Keller, R.M., "Look-ahead processors," ACM Computing Surveys, 7, No. 4, December 1975, pp. 177-195.

[74] Keller, R.M., "Vector replacement systems: A formalism for modeling asynchronous systems," Princeton University, E.E. Technical Report No. 117, December, 1972. Revised Jan. 1974.

[75] Keller, R.M., "Parallel program schemata and maximal parallelism," J.ACM 20, 3 (July 1973) 514-537; and J.ACM 20, 4 (Oct. 1973), 696-710.

[76] Keller, R.M., "On maximally parallel schemata," in Conf. Rec., 1970 IEEE 11th Annu. Symp. Switching and Automata Theory, pp. 32-50.

[77] Keller, R.M., "On the decomposition of asynchronous systems," in Conf. Rec., 1972 IEEE 13th Annu. Symp. Switching and Automata Theory pp. 78-89.

[78] Knuth, D., "Additional comments on a problem in concurrent programming control," Comm. Assoc. Comput. Mach., Vol. 9, pp. 321-322, May 1966.

[79] Kosaraju, S.R., "Limitations of Dijkstra's semaphore primitives and Petri nets," Technical Report 25, The Johns Hopkins University, (May 1973), also in Operating Systems Review, Vol. 7, No. 4, (Oct. 1973), pp. 122-126.

[80] Kosinski, P.R., "A data flow programming language," IBM T.J. Watson Research Center Report RC-4264, Yorktown Heights, N. Y., March 1973.

[81] Kotov, V.E., and A.S. Maringani, "On transformation of sequential programs into asynchronous parallel programs" in Proc. IFIPS Congr., 1968, pp. J37-J45.

[82] Kotov, V.E., "Towards automatic construction of parallel programs," Int'l Symp. on Theoretical Programming, Novosibirsk, USSR, Aug. 7-11, 1972. In Lecture Notes in Computer Science, Vol. 5, Springer-Verlag 1974, pp. 309-331.

[83] Kuck, D.J.; Muraoka, Y.; and Chen, S.C., "On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speed-up." IEEE Trans. Computers, C-21, 12 December 1972, 1293-1309.

[84] Lehman, "A survey of problems and preliminary results concerning parallel processing and parallel processors," Proc. IEEE, Vol. 54, Dec. 1966, pp 1889-1901.

[85] Lipton, R.J., "The reachability problem requires exponential space," Yale Univ., Computer Science Dept., Research Report #62, Jan. 1976 (to appear in Theoretical Computer Science J.).

[86] Lipton, R.J., L. Snyder, and Y. Zalcstein, "A comparative study of parallel computation," Proceedings, 15th Annual IEEE Symposium on Switching and Automata Theory, October, 1974.

[87] Lipton, R.J., R.E. Miller, and L. Snyder, "Introduction to linear asynchronous structures," to appear in Proc. of Symposium on Petri Nets and Related Methods, M.I.T., Cambridge, Mass., July 1-3, 1975.

[88] Lipton, R.J., R.E. Miller, and L. Snyder, "Synchronization and computing capabilities of linear asynchronous structures," in Proceedings of the Sixteenth Annual Symposium on Foundations of Computer Science, Berkeley, Cal., October 13-15, 1975, pp. 19-28. Also full version to appear in JCSS.

[89] Logrippo, L., "Renamings in program schemas," in Conf. Rec. 1972 IEEE 13th Ann. Symp. Switching and Automata Theory, pp. 67-70.

[90] Logrippo, L., "Renamings in parallel program schemas," Ph.D. dissertation, University of Waterloo, Waterloo, Canada, February 1974.

[91] Luconi, F.L., "Output functional computational structures," in Conf. Rec., 1968 IEEE 9th Ann. Symp. Switching and Automata Theory, pp. 76-84.

[92] Martin, D.F., and G. Estrin, "Models of computations and systems -- Evaluation of vertex probabilities in graph models of computations," J. Assoc. Comput. Mach., Vol. 14, pp. 281-299, Apr. 1967.

[93] Merlin, P.M., "A Methodology for the Design and Implementation of Communication Protocols," IEEE Trans. on Communications, Vol. COM-24, No. 6, June 1976, pp. 614-621.

[94] Miller, R.E., "Some undecidability results for parallel program schemata," IBM Research Report RC 3371, May, 1971. Also, SIAM Computing Journal, Vol. 1, No. 1, pp. 119-129, March, 1972.

[95] Miller, R.E., and J. Cocke, "Configurable computers: A new class of general purpose machines," IBM Research Report RC 3897. Invited paper presented at the Symposium on Theoretical Programming, Novosibirsk, USSR, August, 1972. In Lecture Notes in Computer Science, Vol. 5, "International Symposium on Theoretical Programming," Springer-Verlag, New York, 1974, pp. 285-298.

[96] Miller, R.E., "A comparison of some theoretical models of parallel computation," IBM Research Report RC-4230. Also IEEE Transactions on Computers, Vol. C-22, No. 8, pp. 710-717, August, 1973.

[97] Miller, R. E., and W.A. Brinsfield, "Insertion of parallel program schemata," Proc. of the 7th Annual Princeton Conference on Information Sciences and Systems, March, 1973.

[98] Miller, R.E., "Eight Lectures on Parallelism: I, Configurable Computers and the Data Flow Model Transformation; II, Computation Graphs and Petri Nets; III-VII, Parallel Program Schemata; VIII, Relationships between Various Models of Parallelism and Synchronization." Presented at CIME International Mathematical Summer Center on "Theoretical Computer Science," June 9-14, 1975, Bressanone, Italy. In Proceedings. pp 5-63.

[99] Miller, R. E., "Relationships among models of parallelism and synchronization," (Revision of RC-5074) to appear in Proceedings of Symposium on Petri Nets and Related Methods, M.I.T., Cambridge, Mass, July 1-3, 1975.

[100] Miller, R.E.. and J.D. Rutledge, "Generating a data flow model of a program," IBM Tech. Disclosure Bull., Vol. 8, pp. 1550-1553, 1966.

[101] Miranker, W.L., "A survey of parallelism in numerical analysis," SIAM Rev., Vol. 13, pp. 524-547, Oct. 1971.

[102] Misunas, D., "Petri nets and speed independent design," CACM, 16, No. 8, August 1973, pp. 474-481.

[103] Morris, D.; and Treleaven, P.C., "A stream processing network," Sigplan Notices 10, 3, (March 1975), 107-112.

[104] Munro, I. and M. Paterson, "Optimal algorithms for parallel polynomial evaluation," in Conf. Rec., 1971 IEEE 12th Ann. Symp. Switching and Automata Theory, pp. 132-139. Also JCSS, Vol. 7, No. 2, pp. 189-198.

[105] Murtha, J.C., "Highly parallel information processing systems," Advances in Computers, pp. 1-116, 1966.

[106] Narinyani, A.S., "Looking for an approach to a theory of models for parallel computation," Int'l Symp. on Theoretical Programming, Novosibirsk, USSR, Aug. 7-11, 1972. In Lecture Notes in Computer Science, Vol. 5, Springer-Verlag, pp. 247-284.

[107] Nash, B.O., "Reachability problems in vector addition systems," The American Mathematical Monthly, 80, 3, 292-295, March, 1973.

[108] Noe, J.D., "A Petri net model of the CDC 6400," Report 71-04-03, Computer Science Dept., Univ. of Washington, (1971); also in Proc. of the ACM SIGOPS Workshop on System Performance Evaluation, ACM, New York, (1971), pp. 362-378.

[109] Noe, J.D. and G.J. Nutt, "Macro E-nets for representation of parallel systems," in IEEE Trans. on Computers, Vol. C-22, no. 8, (Aug. 1973) pp. 718-727.

[110] Patil, S.S. and Dennis, J.B., "The description and realization of digital systems," Computation Structures Group Memo 71, Project MAC, M.I.T. (Oct. 1972); also in Sixth Annual IEEE Computer Society Int'l Conference Digest of Papers, IEEE, (1972).

[111] Patil, S.S., "Coordination of asynchronous events," Ph.D. dissertation, M.I.T., Cambridge. (Project MAC Rep. TR-72, Sept. 1967).

[112] Patil, S.S., "Closure properties of interconnections of determinate systems," in Rec. Project MAC Conf. Concurrent Syst. and Parallel Comput., New York: Assoc. Comput. Mach., 1970, pp. 10-116.

[113] Peterson, J.L., "Petri Nets," U. of Texas msc., July 1976.

[114] Peterson, J.L., "Modeling of Parallel Systems," Ph.D. Thesis, Elec. Eng. Dept., Stanford University, January, 1974.

[115] Peterson, J.L., and T.H.Bredt, "A comparison of models of parallel computation," Inform. Processing 74, Proceedings IFIP Congress 1974, 466-470, August, 1974.

[116] Petri, C.A., "Communication with automata," Suppl. 1 to Tech. Rep. RAD C-TR-65-337, Vol. 1, Griffiss Air Force Base, New York, 1966. (Translated from Kommunikation mit Automaten, Univ. Bonn, Bonn, Germany, 1962.)

[117] Ramamoorthy, C.V. and M.J. Gonzalez, "A survey of techniques for recognizing parallel processable streams in computer programs," in 1969 Fall Joint Comput. Conf., AFIPS Con. Proc., Vol. 35, Montvale, N.J.: AFIPS Press, 1969, pp. 1-15.

[118] Reddi, S.S. and E.A. Feustel, "A restructurable computer system," Report, Laboratory for Computer Science and Engineering, Rice Univ. Houston, Texas, March 1975.

[119] Reigel, E.W., "Parallelism exposure and exploitation," in Parallel Processor Systems, Technologies, and Applications, L.C. Hobbs et al., Ed. Washington, D.C.:Spartan, 1970, pp. 417-438.

[120] Reiter, R., "Scheduling parallel computations," J. Assoc. Comput. Mach., Vol. 15, pp. 590-599, 1968.

[121] Riddle, W.E., "The modelling and analysis of supervisory systems," Ph.D. Thesis, Computer Science Dept., Stanford Univ., (March 1972),

[122] Riddle, W.E., "The equivalence of Petri nets and message transmission models," SRM 97, The Univ. of Newcastle upon Tyne, (Aug. 1974).

[123] Rodriguez, J.E., "A graph model for parallel computation," Ph.D. dissertation, M.I.T., Cambridge, Sept. 1967. (Also M.I.T., ESL, and Project MAC Rep. ESL-R-398, MAC-TR-64, Sept. 1969.)

[124] Rohrbacher, D.L., "Advanced computer organization study," Rome Air Devel. Corp. Tech. Report RADC-TR-66, 7 (2 vols.) AD 631 870, and 631 871 (April 1966).

[125] Rose, C.W., "A system of representation for general purpose digital computer systems," Ph.D. dissertation, Case Western Reserve Univ., Cleveland, Ohio, Sept. 1970.

[126] Rose, C.W., "LOGOS and the software engineer," in 1972 Fall Joint Comput. Con., AFIPS Conf. Proc, 41. Montvale, N.J.: AFIPS Press, 1972, pp. 311-323.

[127] Rose, C.W., and F. T. Bradshaw, "The LOGOS representation system," Case Western Reserve Univ., Cleveland, Ohio, Rep., Oct. 1971.

[128] Russell, E.C., "Automatic program analysis," Ph.D. dissertation, Univ. California, Los Angeles, 1969.

[129] Rutledge, J.D., "Parallel processes, schemata and transformations," IBM Res. Rep. RC 2912, June 1970.

[130] Rutledge, J.D., "Program schemata as automata, part I," in Conf. Rec. 1970 IEEE 11th Annu. Symp. Switching and Automata Theory, pp. 7-24.

[131] Schwartz, J., "Large parallel computers," J. Assoc. Comput. Mach., pp. 25-32, Jan. 1966.

[132] Senzig, D.N. and R.V. Smith, "Computer organization for array processing," in 1965 Fall Joint Comput. Conf., AFIPS Conf. Proc., Vol. 27. Montvale, N.J.: AFIPS Press, 1965, pp. 117-128.

[133] Shapiro, R.M. and H. Saint, "The representation of algorithms," Applied Data Res., Inc., Rome Air Develop. Cen., Tech. Rep. TR-69-313. Vol. 2, Sept. 1969.

[134] Shapiro, R.M. and H. Saint, "The representation of algorithms as cyclic partial orderings," Meta Information Applications, Inc., NASA Final Rep., Contract NASW-2097, July 1971.

[135] Slutz, D.R., "Flow graph schemata," in Rec. Project MAC Conf. Concurrent Syst. and Parallel Computation. New York: Assoc. Comput. Mach., 1970, pp. 129-141.

[136] Slutz, D.R., "The flowgraph schemata model of parallel computation," Ph.D. dissertation, M.I.T., Cambridge, September 1968.

[137] Sonnenburg, C.R., "A configurable parallel computing system," Ph.D. Dissertation, University of Michigan, Ann Arbor, October 1974.

[138] Stone, H.S., "A pipeline push-down stack computer," in Parallel Processor Systems, Technologies, and Applications. Spartan Books, Washington, D.C., 1970, pp. 235-249.

[139] Syre, J.C., "From the single assignment software concept to a new class of multiprocessor architectures," Report, 1975 Department d'Informatique, C.E.R.T. BP4025, 31055 Toulouse Cedex, France.

[140] Tjaden, G.S. and M.J. Flynn, "Detection and parallel execution of independent instructions," IEEE Trans. Comput., Vol. C-19, pp. 889-895, Oct. 1970.

[141] Thurber, K.J., "Associative and Parallel Processors," Computing Surveys, Vol. 7, No. 4, Dec. 1975.

[142] van Leeuwen, J., "A partial solution to the reachability-problem for vector-addition systems," Proceedings, 6th Annual ACM Symposium on Theory of Computing, 303-309, May, 1974.

[143] Vantilborgh, H. and A. van Lansweerde, "On an extension of Dijkstra's semaphore primitives," Information Processing Letters, 1, 181-186, October, 1972.

[144] Winograd, S., "Parallel iterative methods," in Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, Ed. New York: Plenum, 1972.

[145] Yoeli, M., "Petri nets and asynchronous control networks," Applied Analysis and Computer Science Research Report CS-73-07, University of Waterloo, Waterloo, Ontario, Canada, April, 1973.

Today: Parallel and Pipelined Machines

Next Time: Data Flow Transformation and Configurable Computers

References: Thurber [141]

Keller [73]

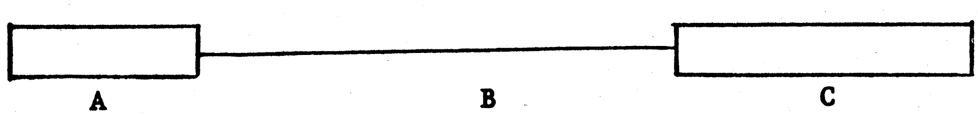
Chen [22]*

* Numbers refer to Bibliography in "Math. Studies...".

The Idea of Pipelining

A simplistic view of how a program is executed on a computer is that in any one moment of time only a single instruction of a single program is in some phase of its operation. This view is often sufficient to analyze a program and it is close to how early computers actually operated. Many techniques have been introduced, however, for overlap and look-ahead to increase the rate of processing instructions. These include interleaved memory, local buffer registers, cache memories, and tagged bus systems. The earliest use of instruction overlap consisted of doing the instruction fetch and other preparation of the $i + 1^{st}$ instruction at the same time as doing the functional execution of the i^{th} instruction.

This created a speedup of almost a factor of two; as expected when the instruction preparation and execution times are equal. Pipelining is carrying this notion of overlap even further. For example, the instruction preparation could be viewed as a sequence of three separate events.

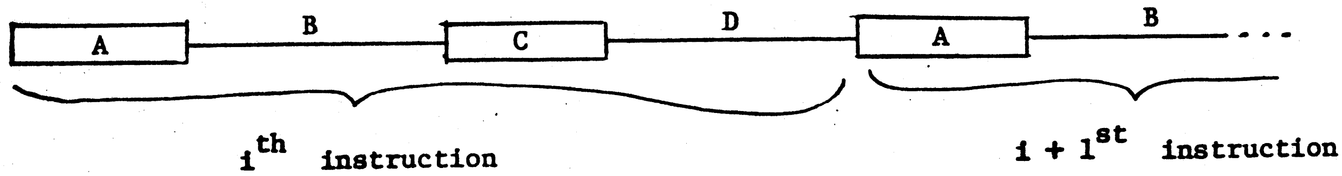


Where A = instruction address generation

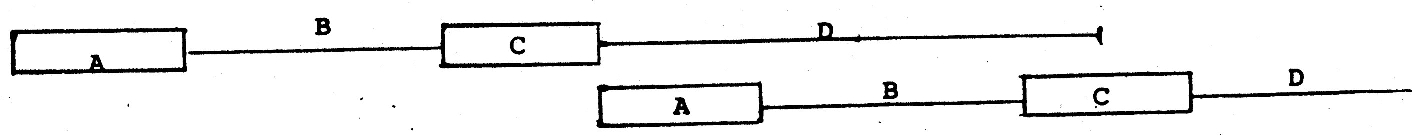
B = instruction fetch

C = insruction decode and operand address generation.

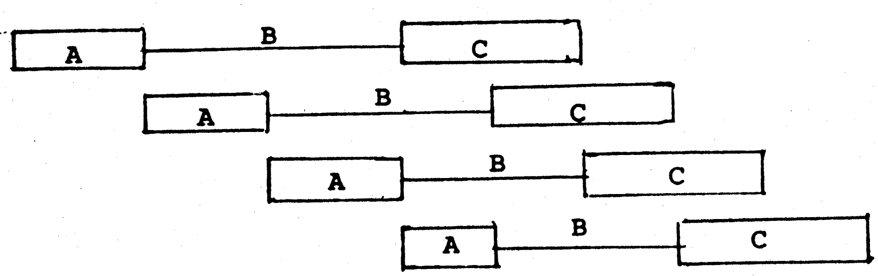
This would then be followed by a time, say D, for instruction execution. For serial operation we would get a time chart of the form:



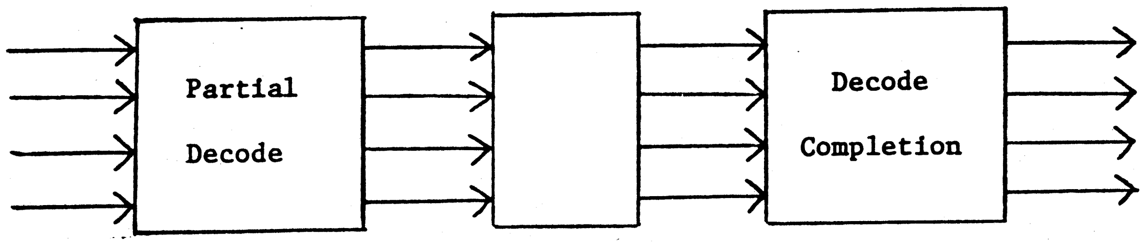
For the simple overlap of instruction preparation with instruction execution we get:



Now we can overlap or pipeline further, such as:



Here we always are generating some instruction address; several instruction fetches are going on simultaneously, which may be possible using interleaved memory; and there are points in time where two instructions are being decoded. To accomplish this decoding the decoding circuit could be split into two serial stages, as:



with the center circuit, called a "station," being a buffer to isolate the two stages. Then instruction i could be in the completion decode stage at the same time as instruction $i + 1$ is in the partial decode stage.

Similarly other circuits, including the instruction execution circuits, could be pipelined or multiple units used to accomplish this pipelining of instructions.

Naturally, there are limitations upon how much pipelining can be done. We must consider:

1. Some independence between instructions is required to accomplish pipelining:

   E.g. (a) a conditional branch instruction must be performed before the next instruction in the stream is known.

   (b) an operand can only be fetched if it is known that no other currently executing instruction is still computing that as a result.

2. An isolation station between stages of a pipeline requires some time to react, thus increasing the number of pipeline stages also increases the overall instruction execution time.

3. If k is the depth of the pipeline, i.e., one can have instructions i, i + 1, ...i + k - 1 concurrently being executed in the pipeline, then as k increases the probability of interference of type 1 increases.
4. The operation times for each stage must be matched throughout the pipeline, and it is convenient to have this unit of time some submultiple of the storage access time.
5. Some functions, such as adders, multipliers, decoders, etc., have a natural number of stages beyond which it is inconvenient to partition them.

Each of these considerations limit and constrain the amount of pipelining that is possible. Also, considerable analysis of the program is required to insure the required independence between instructions. In pipelined machines this is done by having a set of buffer registers to store a stack of instructions, and this "window" of instructions is analyzed. That is, a local analysis of the program is done dynamically during execution. Data dependencies and conditional branches can cause inefficient use of the pipeline. Some examples of pipelined computers are the CDC 7600, CDC Star, and the IBM 360/95 and 195.

A variety of techniques have been developed for attempting to maintain an even flow of instructions through pipelined computers. A simplified view of such a structure is shown in Figure 2 (next page). Here the operation and operand buffers isolate the AU's from storage and control. They can operate in a semiautonomous way, assuming that these buffers have suitable encoding to associate the operations and

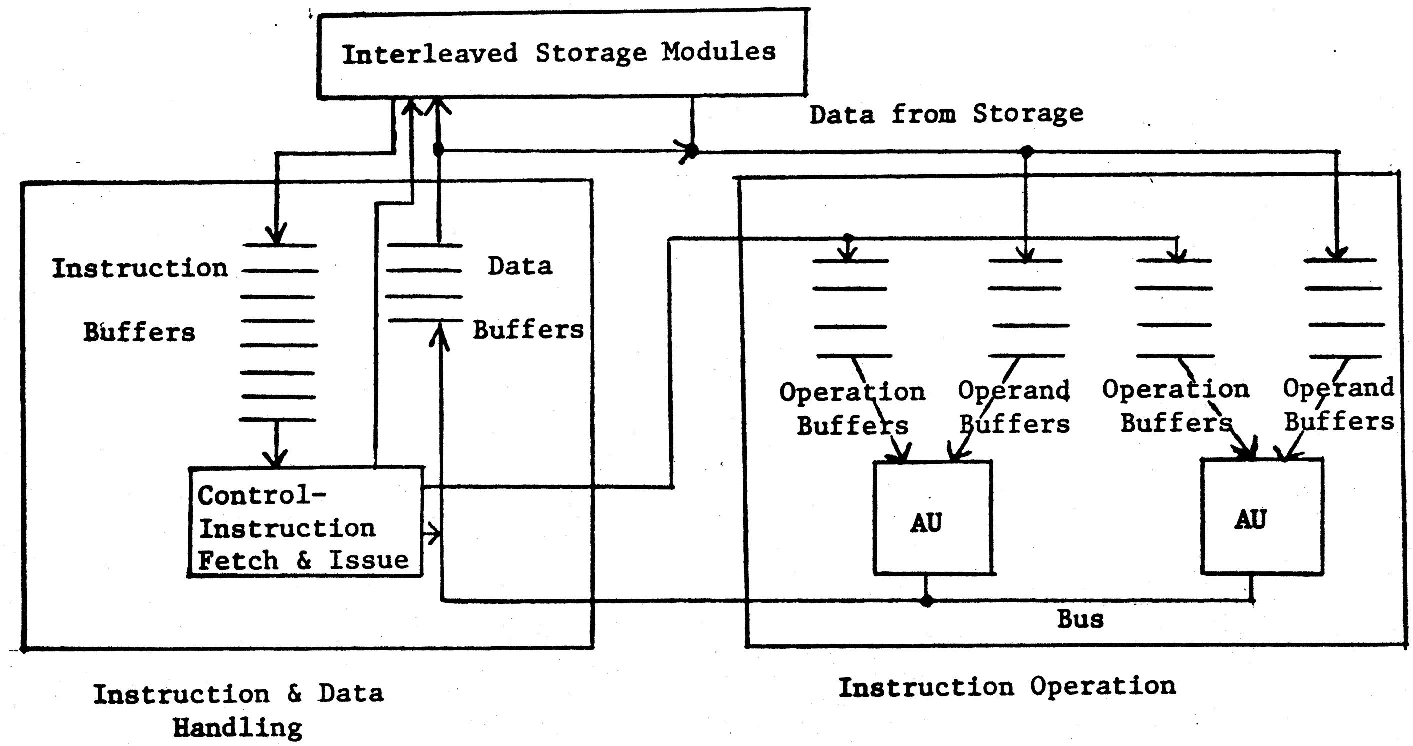


Figure 2: A Simplified View of Pipelined Machine

operands. From an AU point of view it is operating only on inputs from the operation and operand buffers and supplying results only to data buffers. Each AU itself might be a pipelined unit. The instruction and data handling section controls how data is accessed in memory and how instructions are loaded into the instruction buffers. Naturally, with a stack of instruction buffers one now has the capability of storing a whole loop in the buffers. Detection of this during operation is worthwhile since in such a case further instruction fetching can be disbanded during loop execution and this decreases the traffic to memory. Another feature is the possible prefetching and partial decoding of instructions along one or both paths exiting the loop so that an even flow of instruction executions could continue even on loop exit.

Parallel Machines

Suppose we make a simple assumption about a computational job that it takes τ_s units of time on a single processor but that it can be broken into two parts with $\tau_s = \tau_1 + \tau_2$, where the τ_1 part could be done in parallel with n processors in time τ_1/n but the τ_2 part could not be speeded-up by parallelism. Then on an n-processor machine the time required for τ_n is: $\tau_n = \tau_1/n + \tau_2$.

The ratio of $R = \frac{\tau_n}{\tau_s} = \frac{\tau_1/n+\tau_2}{\tau_1+\tau_2}$ so $R = 1 - \frac{\tau_1(\frac{n-1}{n})}{\tau_1 + \tau_2} = 1 - \frac{\tau_1}{\tau_s}(1-\frac{1}{n})$.

The speed-up factor is: $S = \frac{1}{R} = \frac{\tau_s}{\tau_n}$

$$S = \frac{1}{R} = \frac{1}{1 - \frac{\tau_1}{\tau_s}(1 - \frac{1}{n})}$$

If we assume $\frac{\tau_1}{\tau_s} = .9$ and $n = 32$ then $S \approx \frac{1}{.13} < 8$. Plotting the speed-up factor vs $\frac{\tau_1}{\tau_2}$ values we get the dilution curve shown in Figure 3.

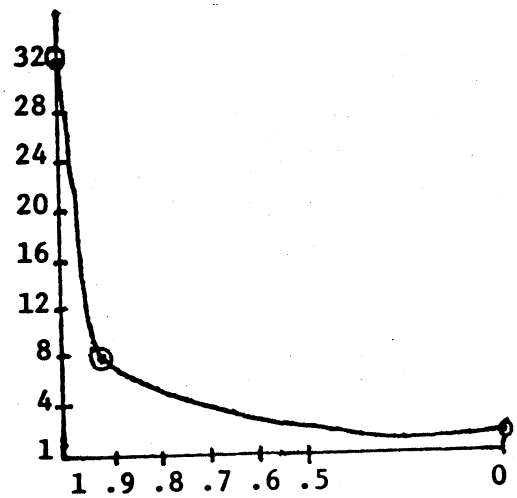


Figure 3: Parallelism Dilution Curve τ_1/τ_s

Similar curves can be drawn for any assumed value of n. This simple argument illustrates the problem that even with a very little non-parallel part to the computation any parallel machine can lose considerable efficiency. Here at $\tau_1/\tau_s = .9$ we are fully utilizing only somewhat less than 8 of the 32 processors. Thus, for fuller utilization an attempt must be made to mold the computing problem into a suitable computation. Otherwise, it could be better to run independent problems on each of the n processors.

The vector and array type of parallel machine are based on natural parallelism in data structures. The VAMP machine [132] (for Vector Arithmatic Multi-Processor) was an early such proposal. The arithmatic unit (or mill) is depicted in Figure 4.

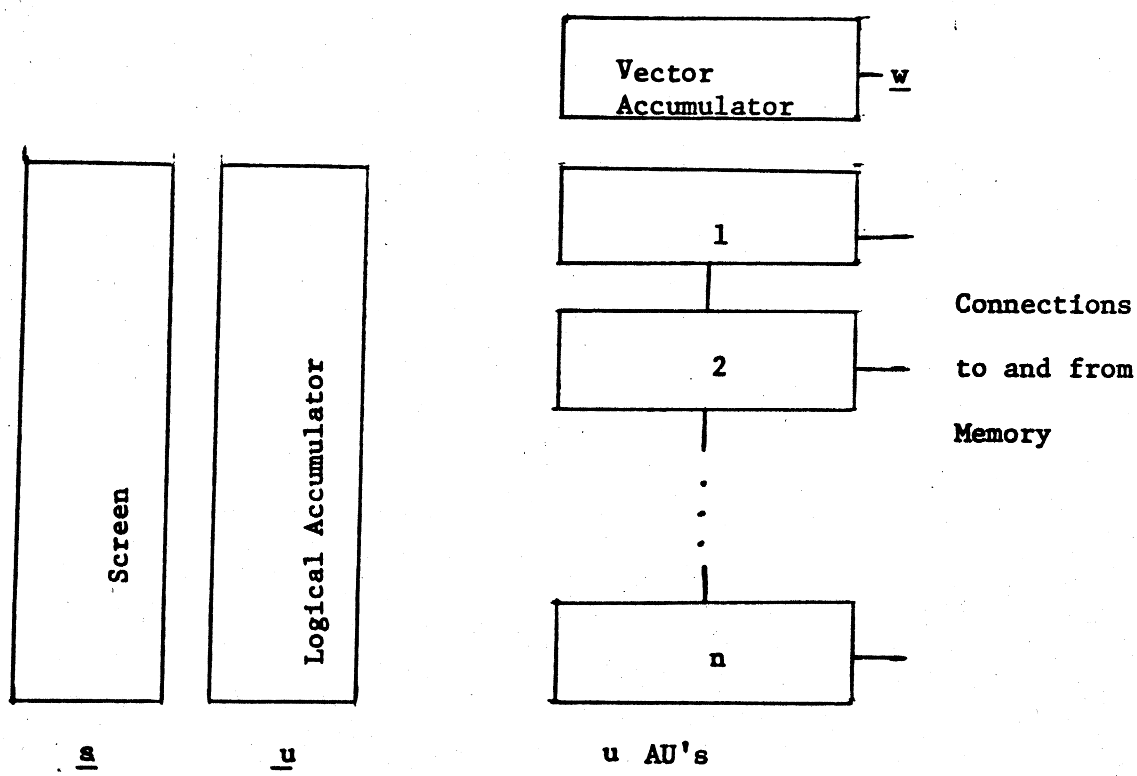


Figure 4: VAMP mill

The mill has n arithmetic units, two n-bit registers s and u and one k-bit accumulator register w, where the word length is assumed to be k. The n arithmetic units are assumed to perform the same operation on n data items simultaneously. Each AU^i has a floating point adder, a storage register z^i of k-bits attached to the memory but not program addressable, an accumulator register X^i of 2k-bits. Instructions for loading, doing arithmatic and Boolean operations in vector form are basic to the machine. The s register has a bit s_i associated with each AU^i. If $s_i = 1$ then AU^i is active and will perform the instruction being executed. If $s_i = 0$ then AU^i is inhibited. This allows different parts of arrays to be treated in different ways, and allows one to fit the problem appropriately into the machine. The $\underline{u}$ register records results of logical operations and tests, and is connected to the $\underline{s}$ register so that the subsequent screen can be the result of a previous logical operation or test. The vector accumulator $\underline{w}$ allows one to form the sum, product, min or max of the values in the X^i registers. Also, various APL operations are included as instructions for restructuring of data.

The memory of VAMP uses interleaving and has both vector direct and indirect modes of access.

Some mention is made of using pipelined units to simulate the behavior of the n-AU's.

CS314a Lecture #3 September 21, 1976

Today: Illiac IV, Data Flow Transformation and Configurable Computers

Next Time: Language Constructs for Parallelism

The Illiac IV array processor [14] consists of an array of 64 processing elements. Each processing element P.E. has an arithmetic unit and a memory for data. PE_i can only directly access it's own memory, but in one step can access words in the logically surrounding PE's: $PE_{(i-1)}$, $PE_{(i+1)}$, $PE_{(i-8)}$ and $PE_{(i+8)}$. This is depicted in Figure 1.

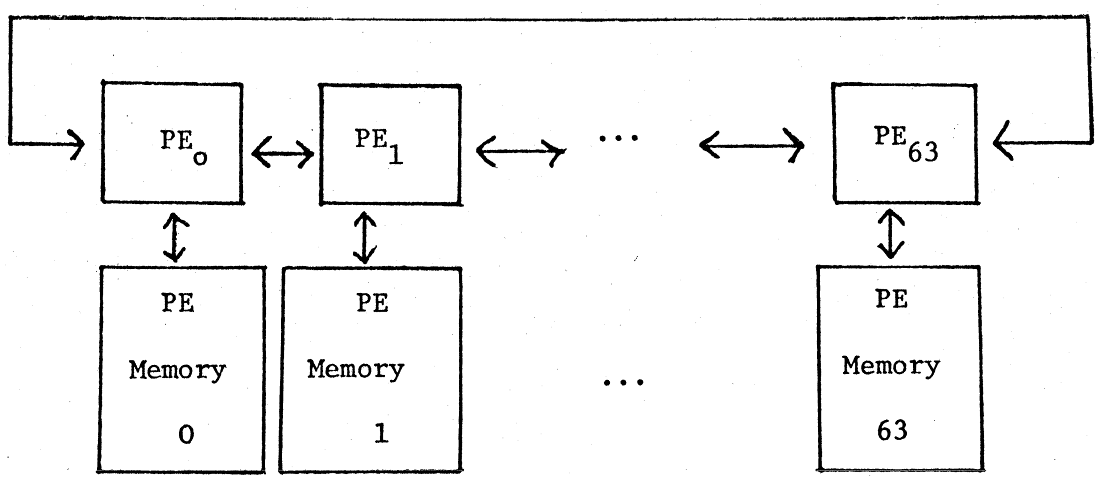


Figure 1 Illiac IV PE structure.

Access to other PE's must be accomplished through a sequence of such routing steps. The separate PE memories differ substantially from VAMP's single memory directly accessable to all arithmetic units. As with VAMP it has a single instruction stream control and bits which can inhibit particular PE's. Orginally Illiac IV was designed to have four 8 x 8 arrays of PEs, each with its own instruction control, (Figure 2) but only one of the four arrays was constructed.

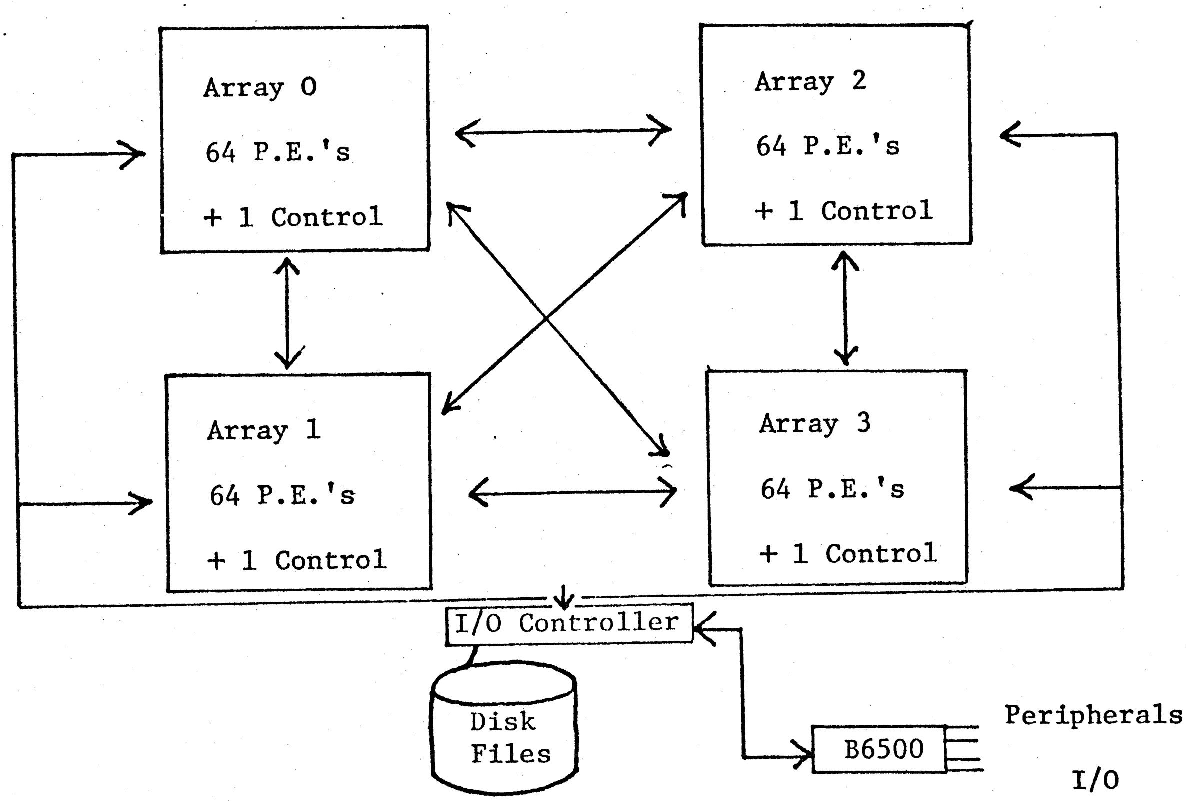


Figure 2 Early Illiac IV proposed organization

The individual memories associated with each P.E. is an organization that provides fast and easy access by PE_i of data in PE $Memory_i$. Memory address generation is easier than in VAMP and there is less chance of memory conflict. However, the routing steps necessary to bring data from one PE memory to another can require as many as seven routing instructions. Thus considerable effort can be expended in trying to mold the data structures to fit well into the Illiac array structure.

Various multi-instruction stream machines have also been proposed [42, 48, 84, 131]. Here, the problems of detecting and controlling the multi-instruction streams and the queues of tasks to be performed lead to some complex control problems. In [141] Thurber also describes a number of parallel systems based on associative processing techniques. The notions of data dependencies have led to data flow type proposals for machines also, which in some cases have the machine dynamically change their structure [34, 95, 118, 124, 137, 139]. To explain these briefly we first introduce the notion of transforming a program into a data flow structure.

Data Flow Structures and Related Machine Organizations

(See related references [34, 37, 80, 81, 95, 98, 100, 118, 137].)

We first describe a simple approach to transforming a sequential program into a parallel form in which the sequencing depends upon operand and result availability rather than instruction sequencing [98, 100].

The basic steps of the transformation are:

1. Partition the program into "basic blocks" and move each block. A basic block is a contiguous segment of code which can be entered only through the first instruction of the block, must be executed by executing each successive instruction in order, and can be exited only from the last instruction of the block. That is, it is a "straight line" segment of code.

2. Determine the immediate predecessors and successors of each block. This is done through the transfer instruction labels plus the normal assumed sequencing of the sequential program, and produces a flowchart - like structure for the program.

3. Generate a "data flow segment" for each block.

A data flow segment consists of:

(i) An input list which consists of the variables needed as inputs by the block.

(ii) An output list which consists of the names of results at the end of block execution that were produced by the block.

(iii) An interconnection of modules for the block which are the operations and flow of data from result to operand locations as formed by block execution.

Step 3 can be performed for each block in arbitrary order. A dictionary of the language is used which specifies for each instruction what the inputs and outputs of the instruction are, what the operation module is, if any, and what registers are used by the operation. Thus Step 3 is concerned only with structure internal to the blocks.

4. Interconnect data flow segments. This step uses the predecessor and successor information from Step 2 to perform interconnections from output lists to input lists, through a matching of names. Also, it updates input and output lists for data that is available at the input of a block but is not used by the block. This is required so such data that is "passing through" a block may be required by some succeeding block.

Without attempting a formal definition of this transformation we simply illustrate it by a very simple example.

An Example

As an example program we consider the problem of evaluating the function $f(x) = a^x + bx + c$. We assume that x, a, b, and c are inputs stored in the symbolic locations x, a, b, and c respectively, and also assume that x is a positive integer. A simple program to perform this evaluation is shown below. The program language used is simple and should be self-explanatory.

Statement #	Program	Comments
1	CLA x	set accumulator to x.
2	STO COUNT	put x in location COUNT.
3	CLA a	put a in accumulator.
4	TRA 6	transfer to statement 6.
5	MPY a	multiply accumulator by a.
6	Decrement COUNT	decrease COUNT by 1.
7	Branch on COUNT (to 5 on $\neq$ 0)	conditional transfer.

Statement # (continued)	Program	Comments
8	STO T	store a^x in T.
9	CLA x	place x in accumulator.
10	MPY b	form bx in accumulator.
11	ADD T	form a^x+bx in accumulator.
12	ADD C	form a^x+bx+c in accumulator.
13	STO R	store result in R.

The basic blocks for Step 1 can be directly determined using the following definition.

Definition: A basic block is a continuous segment of code whose last instruction is one of the following types:

(a) a branch instruction,

(b) an "end" instruction, or

(c) the predecessor of an instruction with more than one predecessor.

The first instruction of a basic block is the first instruction preceding the last instruction which is one of the following types:

(a) a starting point,

(b) an immediate successor of some branch instruction, or

(c) an instruction with more than one predecessor.

Applying the definition of basic block to this program we find that instruction 4 is the end of a basic block because it is a branch instruction. Instruction 5 is the end of a basic block because it is the predecessor of instruction 6 but instruction 6 has more than one predecessor, namely 4 and 5.

Instruction 7 is the end of a basic block because it is a branch instruction and instruction 13 is the end of a basic block because it is the "end" instruction for this program. Working back from these final instructions for basic blocks we find instructions 1,2,3,4 form a basic block with instruction 1 being the start of the program. Similarly instructions 6,7 form a basic block, 8,9,10,11,12,13 form a basic block and 5 alone forms a basic block. These blocks are depicted and named BB1 through BB4 in the following diagram.

	1	CLA x
	2	STO COUNT
BB1	3	CLA a
	4	TRA 6
BB3	5	MPY a
	6	Decrement COUNT
BB2	7	Branch on COUNT (to 5 on $\neq$ 0)
	8	STO T
	9	CLA x
	10	MPY b
BB4	11	ADD T
	12	ADD c
	13	STO R

Step 2 of the algorithm determines immediate successors and immediate predecessors as shown in Figure 3.

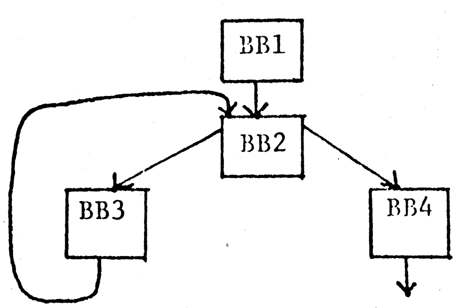


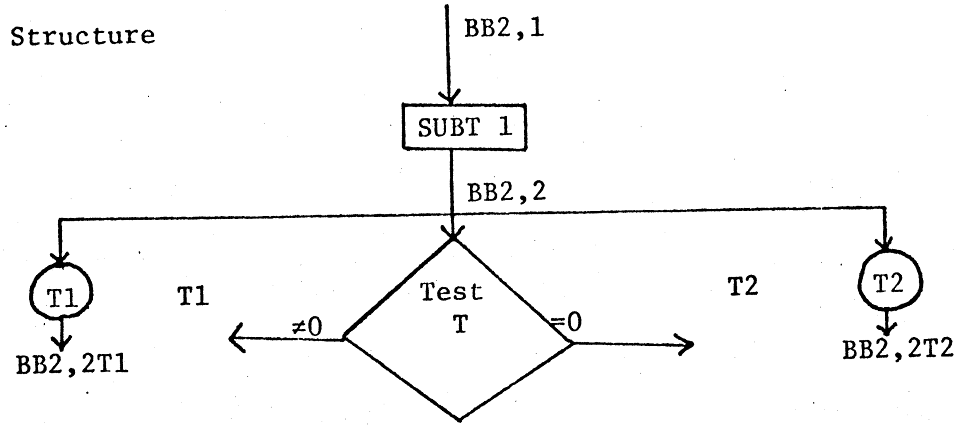
Figure 3.

Step 3 of the algorithm generates a "data flow segment" for each basic block. The idea here is to generate a list of items needed as inputs to the block, the outputs created by the block and the operations used to create these outputs along with the flow of data between the operations within the block. These items have names associated with them through the program language definition, these names we call "source names." During generation we assign "local data names" to items also. Consider, for example, basic block 2, (BB2). This block starts with instruction 6 -- Decrement COUNT. By definition this instruction needs an input with source name "COUNT" and produces a new output also called "COUNT" which has a value one less than the original value of COUNT. Thus COUNT is placed on the input list, and we associate a local data name with this input value. We use the name BB2,1; i.e., the first local data name in BB2. In the output list we then have COUNT with a new value and name this new value BB2,2 as a local data name. The operation performed is a SUBTRACT 1 so this operation gets placed in the module structure with input BB2,1 and output BB2,2. The second instruction of BB2 is Branch on COUNT. This instruction uses the current value of COUNT, namely BB2,2 and tests it for =0 or $\neq 0$. This is indicated in the output list as changing item COUNT-BB2,2 to two values COUNT BB2,2T1 and COUNT BB2,2T2 for the outcome of the test either being outcome T1 or T2. A test module is added to the module structure -- we call it test T -- with the two indicated outputs. This completes Step 3 for BB2. Our result is summarized below.

<u>BB2 Data Flow Segment</u>

Input List	COUNT - BB2,1
Output List	~~COUNT - BB2,2~~
	COUNT - BB2,2T1
	COUNT - BB2,2T2

Module Structure



Similar calculations are done for each of the other basic blocks producing the following results.

<u>BB1 Data Flow Segment</u>

Input List

x - BB1,1

a - BB1,2

Output List

~~ACCUM---BB1,1~~

COUNT - BB1,1

ACCUM - BB1,2

no modules.

<u>BB3 Data Flow Segment</u>

Input List

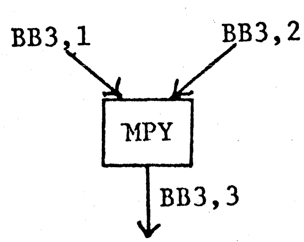
ACCUM - BB3,1

a - BB3,2

Output List

ACCUM - BB3,3

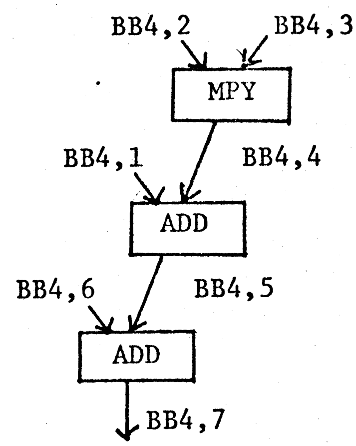
Module Structure



BB4 Data Flow Segment

Input List	ACCUM	- BB4,1
	x	- BB4,2
	b	- BB4,3
	c	- BB4,6
Output List	T	- BB4,1
	~~ACCUM~~	~~- BB4,2~~
	~~ACCUM~~	~~- BB4,4~~
	~~ACCUM~~	~~- BB4,5~~
	ACCUM	- BB4,7
	R	- BB4,7

Module Structure



Step 4 of the transformation interconnects these module structures by using successor and predecessor information and making output to input connections through common source names. The result of making these interconnections and inserting test points T1 and T2 for places where data passes only conditionally on the outcome of test T is shown in Figure 4. Note that even in this simple example some possibilities for parallelism are exhibited. For example, the two multiplications and the subtract 1 operations could all be performed concurrently.

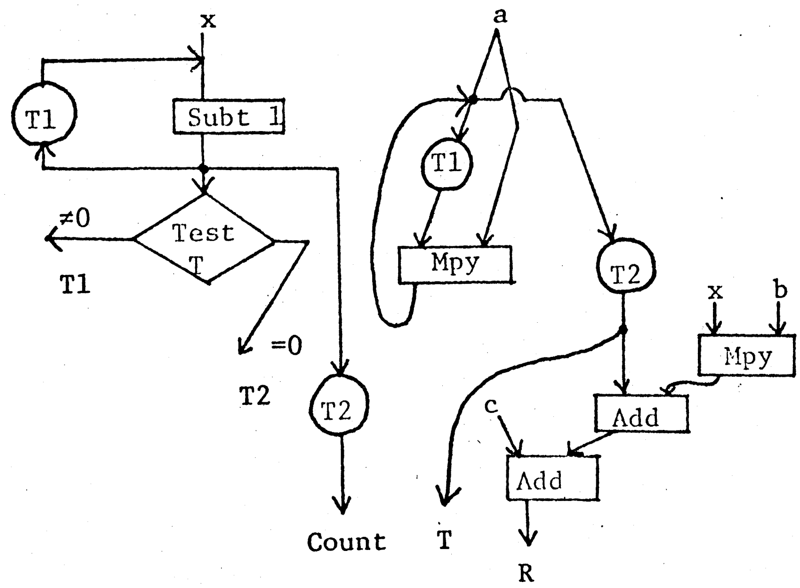


Figure 4: Data Flow for the Example

Even though this transformation detects potential parallelism in the algorithm it is unlikely that ordinary computers could make use of this parallelism since their basic operation is predicated on perscribed instruction sequencing. The configurable computer concepts described next, however, are perfectly suited for such data-flow programs.

Configurable Computers

One of the main features of digital computers since their inception has been the concept of their automatic control through a program stored in memory. We discuss here a major departure from this concept. Rather than running directly under program control a configurable computer has the machine structure change to conform to the structure of the algorithm being performed. We briefly describe two approaches to attaining this structure change;namely a so-called "search mode" and an "interconnection mode." Many variants of these approaches seem possible and some have been studied to some extent. Some of the advantages of configurable computers over other forms of parallel machines should be evident even now. For example, fast operation over a broad range of problems, and utilization of standard programming followed by automatic transformation to data-flow form for machine execution.

The basic organization for a search mode configurable computer is shown in Figure 5.

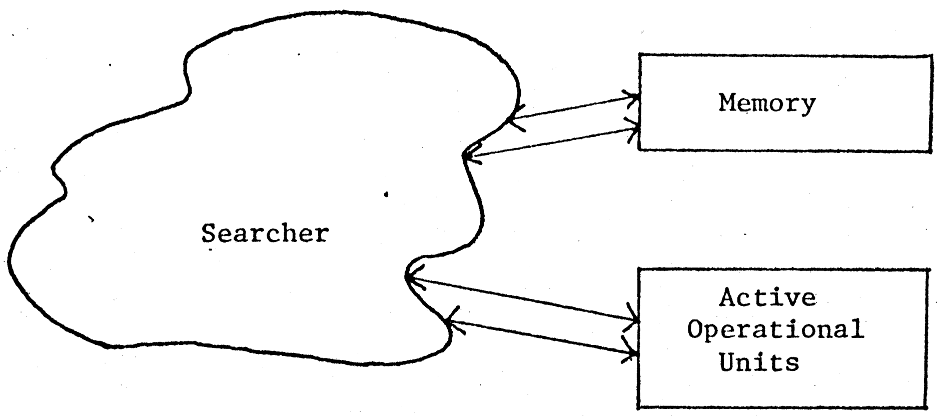


Figure 5: Search Mode Configurable Computer

The operational units consist of general or special purpose units which perform computational tasks, condition determinations, and data generation. The units are "active" in the sense that when one of them completes a task it requests the searcher to find another task for it to perform. The searcher is thus a different type of computer control unit. Upon being asked to supply a new task to an operational unit it inspects memory to find such a task. The memory is thus assumed to be organized in such a way that it stores task specifications, their operands, and their state of readyness to be performed. That is, a task is ready to be performed when its operands are available -- a data flow organization. An example machine instruction to be stored in memory is shown in Figure 6. This should aid in understanding how the searcher would inspect memory.

Operation Code	Status Bits	First Operand	Second Operand	"Address" for Result

Figure 6: Search Mode Instruction Format

Here we show the format for an arithmetic operation with two operands and one result. The operation code bits specify the type of operation to be performed. Thus, when the searcher is looking for an operation to be performed for an operational unit, if the operational unit is not general purpose it would have to check that the operation code specified an operation that could be performed by the operational unit. The status bits keep a record of whether operands currently reside in the operand locations of the instruction and whether the location for the result is available for storing a result. Thus the searcher also must inspect the status bits. The operand fields actually hold operand values and the address for the result field specifies where the result is to be stored, i.e. as an operand of one or more subsequent instructions. Clearly such an instruction format eliminates the need for the normal form of instruction sequencing. The sequencing is "data driven" and, at any particular moment in time there may be many instructions that are ready to be performed.

The operation of the searcher requires some additional discussion since it is clear that wrongly implemented it could create a bottleneck in the operation. A search for a single task could require a long sequence of memory accesses until a ready task was found. To circumvent this difficulty one might use a high speed cache memory which can be searched associatively over the operation code and status bits. Also, one might build up within the searcher a queue of ready operations for each class of operational unit so that all that is needed to satisfy a request from an operational unit is a popping of the appropriate queue. We should note in passing that the readiness status of an instruction goes normally from completely not ready (no operands available) to a monotonic build-up of operands. Each time an operand is stored the instruction is accessed. Thus the status bits might be inspected at these times, and it is at the insertion of the last operand when the instruction becomes "operand" ready to be performed. The complete readiness, of course, also depends upon result storage availability which may not fall into this nice semi-monotonic form.

Another form of configurable computer called the interconnection mode configurable computer attempts to implement the data flow form in a more direct interconnection of operational units through electronic switching networks somewhat like telephone switching networks. A block diagram of such an organization is shown in Figure 7.

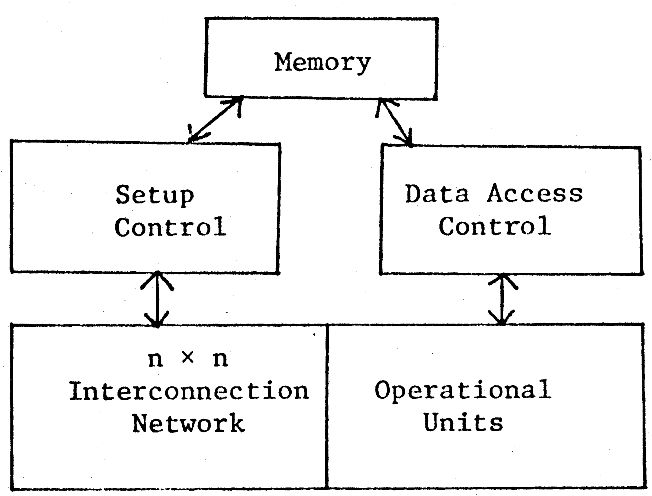


Figure 7: Interconnection Mode Configurable Computer

In this organization the interconnection network is used to directly interconnect outputs of one operational unit to inputs of another operational unit as was depicted by the data flow form of the algorithm. The basic steps for using such a machine, assuming programs given in a normal programming language are:

1. Decompose the program into blocks of appropriate size sò that each block can be set up as a single interconnection on the machine.
2. Transform each block into a data flow form.
3. Store the blocks, so transformed, in memory as set-up instructions for the interconnection network.
4. Choose a block to be performed (to start with this is the block containing the start of the program and subsequently the block is specified by the exit taken from the previous block) and set up the interconnections.
5. Perform the block execution. Note that during execution only data has to be fetched and stored in memory. No instructions are required. Also, many temporary results never get stored in memory, they are simply transferred from the output of an operational unit to the input of another operational unit through the interconnection network.
6. Termination of a block specifies the next block, return to 4.

Both the search mode and interconnection mode structures provide machine organizations well suited to flexible data flow operation. The natural parallelism of algorithms can be exploited. Also, as languages develope for directly representing parallel operation they should be readily implemented on the configurable machines. These machines have the potential speed advantages of special purpose devices, especially the interconnection mode machine that at any point in operation actually is special purpose at that moment. Other features are also interesting. The failure of a single operational unit, if known, could be tolerated by just not assigning that unit any tasks or by not including it in the interconnections. Complex control units are not required for the machines as seem to be required for highly pipelined machines. Finally, much of the data flow form of analysis is identical to that developed for code optimization in compilers. Thus the knowledge gained through compiler optimization can be directly applied to obtaining better algorithms for forming data flow programs.

Today: Language Constructs for Parallelism

Next Time: Start discussing graph Theoretic Models of Parallel Computation

Classification of Types of Parallelism

Parallelism can be classified in various ways; for example, in terms of the algorithms or in terms of the machine. A machine approach of Flynn is reported in Thurber and Wald [141]. If we define an "algorithm" to be a set of "procedures" that operate on data structures in some prescribed sequence, then we can classify parallelism of an algorithm in terms of the procedure and data structures as:

1) Parallelism within a procedure:

   a) Within a data structure:

      Here each element of the data structure is treated identically. Examples of this include vector and matrix operations and machines of the VAMP and Illiac IV type are specifically designed to exploit such parallelism.

   b) Local parallelism:

      More than one addition, multiplication, etc. in a single procedure can be performed concurrently, where the concurrent operations need not be identical. Pipelined machines exploit such parallelism within a local region of the procedure through the instruction stack. Also, the data flow thransformation displays such parallelism globally over the program.

2) Parallelism between instances of a procedure:

The same procedure may be used on several different sets of data structure values, as exemplified by the multiple use of a subroutine. Questions of interferences between instances of the process, and of correct interlocks arise here.

3) Parallelism between procedures:

a) Independent procedures or algorithms:

Examples include multiprogramming and timesharing.

b) Dependent procedures:

Here several procedures can be in operation if suitably interlocked. A classical example is the mutual exclusion problem.

The classification in terms of machines is based upon the procedure and data streams.

1) Single instruction stream single data stream (SISD). This is the classical uniprocessor, single port memory type of computer.
2) Multiple instruction stream single data stream (MISD). This is exemplified by some pipelined processors.
3) Single instruction stream multiple data stream (SIMD). Both the

Illiac IV and VAMP are examples of this type of machine.

4) Multiple instruction stream multiple data stream (MIMD). Configurable computers, as well as multiprocessor systems, can be viewed as MIMD. Also, the Carnegie-Mellon C.mmp machine which is on interconnection of PDP-11's is of this form.

Other classifications of parallel machines are also described by Thurber and Wald. The reason for giving these two here is to illustrate that there is a broad spectrum of possible types of parallelism, and that given any particular machine the problem is to try to fit problems (or algorithms) into a format suitable for the machine.

Instructions for Parallelism

One of the early constructs for parallelism was proposed by Conway [28] and later discussed by Dennis and Van Horn [36]. These are FORK and JOIN constructs which enable one to initiate and merge multiple instruction streams. Figure 1 illustrates a simple flow chart FORK and JOIN construction.

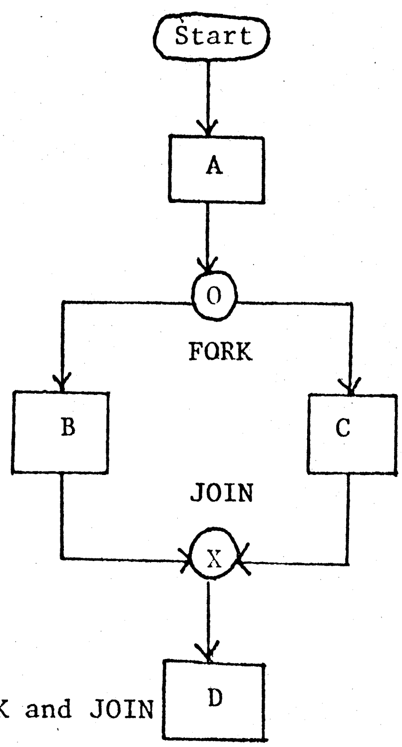


Figure 1: Simple FORK and JOIN

The FORK instruction is to occur at the completion of procedure A, and it is used to initiate two instruction streams, here illustrated by procedures B and C. The JOIN is to indicate that procedure D can be initiated only after both B and C are completed. This explains the logical flow of control that the FORK and JOIN constructs are trying to implement. There are several forms of FORK and JOIN instructions. To produce two instruction streams from one a simple single address FORK can be used of the form: FORK t.

If this instruction is statement m of a program then the two streams enabled by the fork are: one stream starting with statement m+1 of the program and the other starting with statement t. In general, n-1 such FORK instructions can be used to start n streams. Examples for n = 4 are shown in Figure 2.

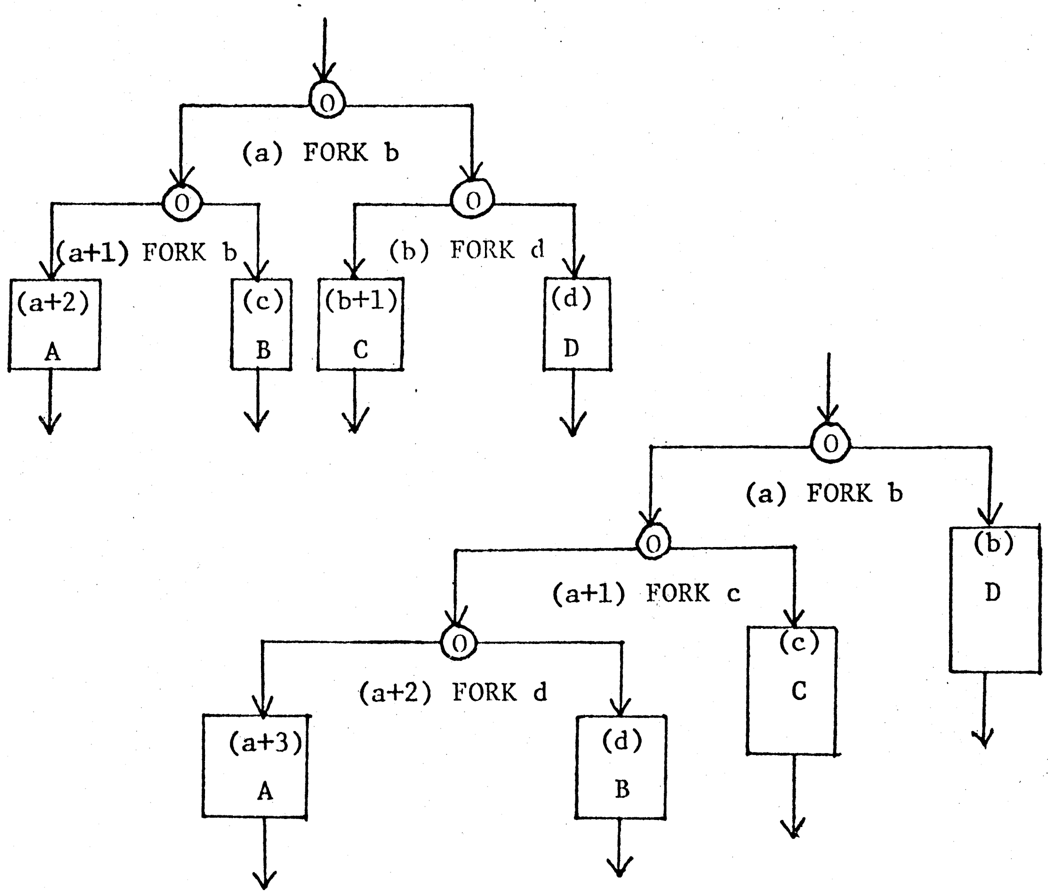


Figure 2: 4 streams started by FORKS.

The JOIN instruction realized in a single address format as : JOIN c assumes a counter to be implemented in location c. The JOIN c tests the value of the counter. When a JOIN c instruction is performed it tests the value in c. If this value is greater than 1 (indicating that more than one parallel stream is still in progress) then one is subtracted from location c and the procedure in which JOIN c was executed is terminated. If the value of location c is 1 then it is set equal to zero and a new procedure starting at location c+1 is allowed to start. This form of JOIN then assumes that the counter value in location c is initially correctly set to the number of streams being joined. For example, in Figure 1, in A or the start of the program a set c = 2 instruction would be required, and JOIN c instructions could be placed at the end of both B and C, with procedure D starting in location c+1. The reader may wish to consider how JOINS could be used to merge the Figure 2 cases of four streams together into a single stream. In some cases the recombination of a stream is not required so a STOP or QUIT instruction could be used. Also, if for the case of Figure 1 it is known that procedure C will always finish before procedure B then a QUIT instruction (rather than a JOIN) can be placed at the end of C and D can be made to directly follow B with no JOINS required.

Some variants of FORK and JOIN are:

1) (a) FORK b,c,v

specifies a FORK to locations a+1 and b to start two streams and also sets up a counter in location c to value v. This eliminates the need for presetting the counter value as required above for the JOIN instructions.

2) (a) JOIN c,d

specifies the same sort of JOINing condition as before except that if this instruction is encountered and the value of c is > 1 then the procedure starting in location d is executed.

3) (a) FORK b,c

specifies a FORK to locations a+1 and b and increases by 1 the counter value located at c. This type of FORK is useful when the number of streams can be variable depending on which side of a conditional branch is taken.

An early form of interlocking procedures is given in Dennis and Van Horn [36] in terms of LOCK and UNLOCK instructions. Some such type of control may be necessary when several processes have access to common data like in the mutual exclusion problem (see Dijkstra [39]). Suppose, for example, that we have two procedures, the first updates a file and the second reads the file. In such a case it is unwise to allow simultaneous access to the file by both procedures since the second procedure may end up reading a mixture of old and new file information. To interlock procedures we introduce the two instructions LOCK w and UNLOCK w. Here w is assumed to be either 0 or 1, and it is assumed that only a single LOCK or UNLOCK instruction can be accessing w at any time.

LOCK w operates as follows:

If w = 1 the procedure waits at the instruction (continually testing w) until w = 0. If w ≠ 1 then the instruction sets w to 1 and the procedure continues.

UNLOCK w simply sets w = 0.

Application of these instructions to the two procedure example we discussed earlier is depicted in Figure 3.

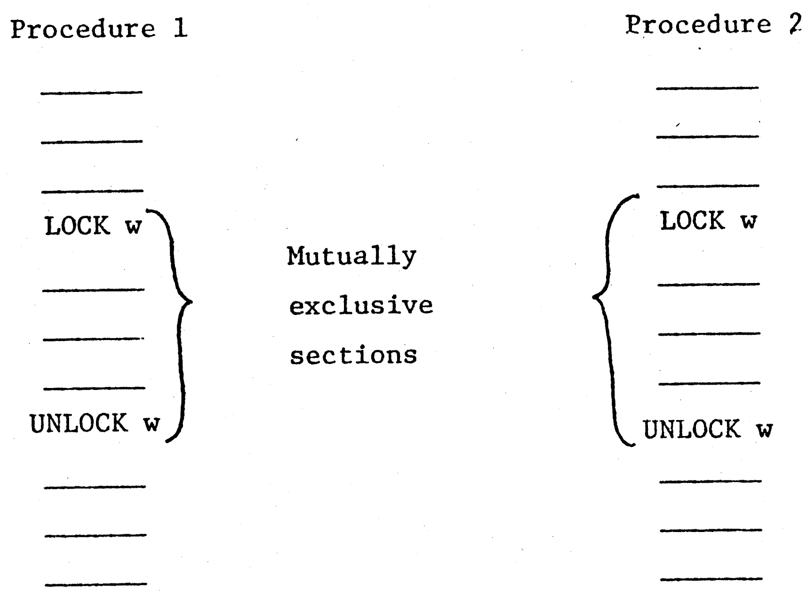


Figure 3: Interlocking procedures.

The concept of semaphores (which are somewhat similar to the lock bit w just discussed) was introduced by Dijkstra [38] to provide a more flexible means for coordinating the sequencing of cooperating sequential processes. Such problems as mutual exclusion, the readers-writers problem, and producer-consumer problems were easily controlled by semaphores.

A semaphore s is a nonnegative integer valued variable which can be accessed by program processes only through two specialized types of instructions P(s) and V(s) as defined below.

P(s) is an indivisable operation on a semaphore s. P(s) at location L is defined as:

L: if $s < 1$ go to L else $s \leftarrow s-1$.

V(s) is an indivisable operation on a semaphore s defined as:

$s \leftarrow s+1$

The indivisability is like the assumed access to a lock bit w. Once either a P or V operation is started on s then that operation must be completed without interaction or interference from any other P or V operation for s. The situation can be somewhat more general, however. For example, if $V_1(s)$ and $V_2(s)$ were both attempting to act on the same semaphore s "simultaneously" this could be allowed as long as the value of s were increased by 2 after the completion of both $V_1(s)$ and $V_2(s)$. If, on the other hand, a semaphore value was 1 and two P operations were attempting to operate on it, only one of the two (arbitrarily determined) would be allowed to proceed and that would decrease the value to 0. The other P operation would have to wait until the semaphore value was again increased before it could operate, and at that time it would have to compete with other P operations on that semaphore that were now attempting to be performed. If the original value of a semaphore were k, then up to k P oerations could proceed simultaneously, as long as the ending value was a decrease by exactly how many P operations were allowed to continue. The two procedure example shown in Figure 3 would now appear, using a single semaphore s as shown in Figure 4, where the preset value of s was s = 1.

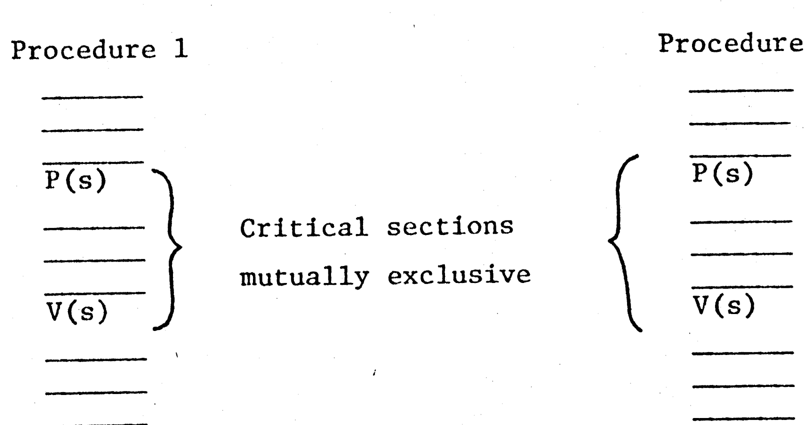


Figure 4: Semaphore for interlocking

In this simple example the semaphore value only ranges over the values 0 and 1, however in more complex synchronizing situations, like for example multiple readers-writers problems, semaphore values greater than 1 are encountered.

Quite a few variations on semaphores have been proposed in the literature. First, one could have P's and V's that change a semaphore value by more than 1. For example, P(n,s) and V(n,s) to indicate a change of value of $\pm$ n to semaphore s. Another variant is a P(T,W,S) defined as:

L: if $s < T$ go to L else $s \leftarrow s-W$.

And constraining $W \leq T$ implies that the semaphore s would not be allowed to become negative. Often it is convenient to have conditions in several semaphores be required to hold for a process to be initiated. One could propose to have a sequence of P operations on the various semaphores to accomplish this, but this runs the danger of creating deadlock situations in complex sequencing problems. For this reason P's and V's on sets of semaphores have been proposed. For example, we could have $P(s_1,s_2,...,s_k)$ and $V(s_1,s_2,...,s_k)$ be operations to indicate indivisible testing and setting of the set $\{s_1,s_2,...,s_k\}$ of semaphores. One of the problems with these types of semaphores, as well as other synchronization primitives, is insuring in the system the requirement of indivisibility of the operations. If one is testing or setting a single semaphore, for example, it might be feasible to assume indivisibility simply from the fact that all this could be performed in a single access cycle to the semaphore stored in memory and there is only a single access port to the variable. However, this may no longer hold when more complex operation on the variable is proposed or when sets of variables need to be treated as a single entity.

Another annoying factor is the requirement for waiting, and continually testing when a P(s) operation is encountered with s = 0. This "busy wait" state seems logically totally unnecessary. It would be more desirable to free the system of "busy waits" so that it could do other necessary processing.

Some of these problems have led to proposals for other synchronizing primitives. In (2) Hoare proposes the notation $\{Q_1||Q_2||\ldots||Q_n\}$ to denote parallel operation for processes $Q_1, Q_2, \ldots, Q_n$. Essentially this notation means we have a FORK-JOIN pair on the n processes. In (2) additions are made to this notation to allow one to specify a common resource to be used by the processes and also to have conditional entry to critical regions. In (1) Hansen proposes a similar scheme. The instruction

var v: shared T

is used to declare a shared variable of the type T, where concurrent processes can only refer to, and change, shared variables within critical regions. A critical region is defined by

region v do S.

This associates the statement S with the shared variable v. Thereby critical regions referring to the same shared variable must be run in a mutually exclusive fashion. A "conditional critical region" is specified by a statement:

region v when B do S.

This specifies that the critical region (S) using shared variable v is to be executed only when condition B holds. B can be a complete Boolean expression, rather than a simple semaphore test as we had for P's on semaphore. Thus this provides quite a general means for specifying conditional entry into a region. The testing of B, and entry into the region, are supposed to be controlled as follows, as described by Hansen. When the conditional critical region statement is encountered B is tested. If B is true the process is allowed to execute S. If B is false, the process leaves the critical region and is delayed until another process has successfully completed a critical region using the same shared variable. At this point the delayed process is again allowed to evaluate B. This cycle continues until B is true and S is executed.

It is through this delaying of the process that one attempts to circumvent the "busy wait" condition, since there is no sense in continually testing a condition B until one or more of its variables (the shared variables) has been changed.

(1) Hansen, Per Brinch, "A Comparison of Two Synchronizing Concepts", Acta Informatica, Vol. 1, 1972, pp190-199.

(2) Hoare, C.A.R., "Towards a Theory of Parallel Programming", in <u>Operating Systems Techniques</u>, Academic Press, 1971, pp61-71.

<u>Today & Next Time</u>: Graph Models of Parallel Computation

Some References: [52, 56-59, 61, 62, 66, 67, 69, 74, 92, 96, 98, 99, 102, 113, 120, 145]

Directed graphs are a natural model for representing computation. The most common such representation being the flow chart for sequential programs. We will concentrate first on computation graphs and Petri net models of parallel computation, but there are many other models as well. The Petri net model is the most studied of all models of parallel computation. We define this first and look at some of its properties.

Petri Nets

<u>Definition 1</u>: A <u>Petri net</u> $P = (\Pi,\Sigma,R,M_o)$ consists of:

(i) a finite set Π called <u>places</u>,

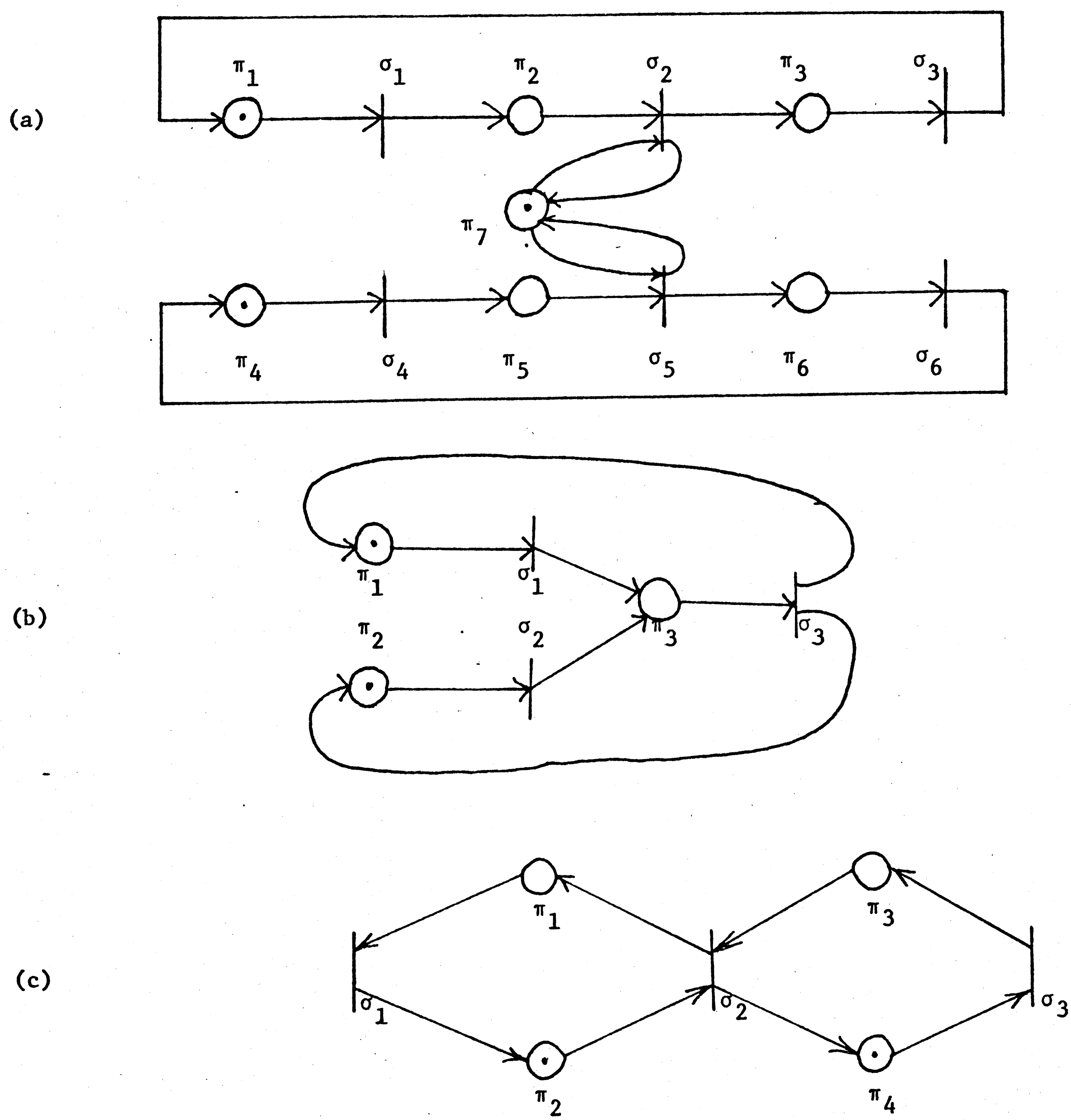
(ii) a finite set Σ called <u>transitions</u>,

(iii) a relation $R \subseteq (\Pi \times \Sigma) \cup (\Sigma \times \Pi)$, and

(iv) a mapping $M_o: \Pi \to N$, called the <u>initial marking</u>, where N represents the set of nonnegative integers.

Usually a Petri net is represented by a graph in which places and transitions are represented by nodes, R is represented by directed edges, and M_o is represented by dots in the place nodes. To distinguish the place and transition nodes, circles ○ are usually used for places and bars | are used for transitions. If $\pi \epsilon \Pi$ and $\sigma \epsilon \Sigma$ where $\pi x \sigma \epsilon R$, then $\pi x \sigma$ is represented by an edge directed from the node for π to the node for σ. Similarly for a $\sigma x \pi \epsilon R$ by an edge from σ to π. Places are used to hold markers called tokens and M_o assigns an initial number of tokens to each place. Several examples of Petri nets are shown in Figure 1.

For a given place π those transitions σ_i for which $(\sigma_i,\pi)\epsilon R$ are called the input transitions of π and those σ_i for which $(\pi,\sigma_i)\epsilon R$ are called the output transitions for π. Similarly, for a given $\sigma\epsilon\Sigma$, those π_i for which $(\pi_i,\sigma)\epsilon R$ are called the input places of σ and those π_i for which $(\sigma,\pi_i)\epsilon R$ are cllled the output places of σ. The Petri net is thus a fixed graphical structure which is supposed to represent the allowed sequencing of parallel processes. Usually the transitions are viewed as processes and the tokens on the input places of a transition are used to control the initiation of the process. A transition σ is called active or fireable if and only if each of its input places contains one or more tokens. An active transition σ may fire, and this can be interpreted as the execution of the process represented by σ. When σ fires it reduces by 1 the number of tokens in each of its input places, and increases by 1 the number of tokens in each of its output places. The firing of a transition thus changes the distribution of tokens on places. Such a distribution of tokens is called a marking. Through the marking change other transitions may become active. It is the sequence of transition firings that is used to represent the computation sequence in a Petri net. A sequence of transition firings is called a firing sequence. It also defines, given an initial marking, a marking sequence. Since a given place may be in the set of input places for more than one transition it is possible that a single token in a place causes more than one transition to be fireable. To prevent the number of tokens upon transition



<u>Figure 1</u>; Petri net examples

firing to become negative it is assumed that a token is used in only a single transition firing. This is assumed formally in yet another way, namely by defining firing sequences to be a sequence of transition labels, implying that even though several transitions are simultaneously fireable, no simultaneous firing is allowed in the formal study. Thus the next element in a firing sequence is one of the transition labels as picked arbitrarily from the current set of fireable transitions.

The examples of Figure 1 are helpful in understanding these definitions and conventions. In part (a) the set of places is $\Pi = \{\pi_1, \pi_2, \ldots, \pi_7\}$ and the set of transitions $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_6\}$. The initial marking is $M_o(\pi_1) = M_o(\pi_4) = M_o(\pi_7) = 1$ and zero elsewhere. From the initial marking both σ_1 and σ_4 are fireable. If σ_1 fires then the new marking M has $M(\pi_1) = 0$ and $M(\pi_2) = 1$, with all other places marked as before. Now both σ_4 and σ_5 are fireable. If σ_4 fires then tokens appear in π_2, π_5, and π_7 and both σ_2 and σ_5 are fireable. Now, however, σ_2 and σ_5 cannot fire simultaneously since π_7 has only a single token, thus their firing must be sequential. This differs from the case of σ_1 and σ_4 being initially fireable since in that case no single token was playing a role in causing the two transitions to be fireable. For the case when σ_2 and σ_5 are simultaneously active we say that σ_2 and σ_5 "conflict" under this marking. We say in general that a pair of transitions σ_i and σ_j are in <u>conflict under a given marking M</u> if both σ_i and σ_j are active in M and there is some place p_k belonging to the input places of both σ_i and σ_j with $M(p_k) = 1$. It is precisely under the conflict situation that although both transitions are simultaneously active they cannot simultaneously fire.

The Petri net of part (b) of Figure 1 is an example that shows that the number of tokens may grow unboundedly in a place. Here a single firing of both σ_1 and σ_2 causes π_3 to have two tokens. A single firing of σ_3 places tokens back in π_1 and π_2 leaving one token in π_3. Repeating this cycle of transition firings causes the number of tokens in π_3 to grow to as large a number as desired.

Part (c) of Figure 1 gives an example of a very special kind of Petri net. A Petri net P is called a marked graph if and only if each place π of P has exactly one input transition and exactly one output transition. When this restriction is made on Petri nets the graph can be simplified by absorbing each place into an edge and then letting the place marking be represented by a marking on the edge.

Similarly, restricting a Petri net so that each transition has exactly one input place and one output place gives a special class of Petri nets called state machines. This is readily seen by simplifying the graph as done by letting each transition now be represented by a directed edge from its input place to its output place. This then assumes the structure of a transition diagram of a finite state machine, but here the edges are not labelled. If one assumes an initial marking now as a single token in a single place (representing the start state) then state to state transitions correspond to transition firings. The analogy is too obvious to belabor. Both the marked graphs and state machines are subclasses of Petri nets that are considerably easier to analyze than general Petri nets. Other subclasses of Petri nets have also been defined and extensively studied. A number of properties of Petri nets are of interest and worth defining. First we note that any marking M of a Petri net P with n places can be viewed as an n-dimensional vector in which the value of the i^{th} coordinate of the vector is the number of tokens in the i^{th} place of P.

Definition 2: The reachable set of markings $R(P,M_o)$ of a Petri net $P = (\Pi,\Sigma,R,M_o)$

is defined as:

$$R(P,M_o) = \{M \mid \exists \text{ a marking sequence starting with } M_o \text{ and ending with } M\}.$$

<u>Definition 3</u>: A Petri net P is called <u>safe</u> if $M \in R(P,M_o)$ implies that each coordinate of M is either zero or one.

Thus a safe net is a net in which the number of tokens in any place never exceeds one. This property is of interest when for some practical considerations one is interpreting the Petri net to represent a set of interrelated events and conditions, where conditions are represented as places. A condition is interpreted as holding if the place contains a token, and as not holding if the place does not contain a token. For such situations it is senseless to have more than one token in a place, so one wants to know that the net representing events and conditions is a safe net.

A natural extension of safeness is k-bounded or k-safe.

<u>Definition 4</u>: A Petri net P is called <u>k-safe</u> if $M \in R(P,M_o)$ implies that each coordinate of M takes on values from the set $\{0,1,2,...,k\}$.

A second property of Petri nets is related to the current or eventual fireability of transitions.

<u>Definition 5</u>: A transition σ of a Petri net $P = \{\Pi,\Sigma,R,M_o\}$ is called <u>live</u> if and only if for every $M \in R(P,M_o)$ there is some firing sequence continuing from M which fires σ. The transition σ is called <u>dead with respect to M</u> if there is no firing sequence continuing from M which fires σ.

<u>Definition 6</u>: A Petri net P is called <u>live</u> if every transition of P is live.

The relavence of the property of liveness is evident when one interprets the transitions of the Petri net as representing processes. Liveness of a transition means that there is no way in which a sequence of process executions can cause the system to get into a state from which the given process can never again be executed. Thus both the liveness and deadness properties of Petri nets are related to the concept of deadlocks in operating systems.

Given any Petri net P we would like to know how to determine if P is safe, k-safe, live, or what transitions are dead, and with respect to what markings. We will approach these problems via vector addition systems introduced in [72].

Last Time: Petri Nets

Today: Vector Addition Systems and their Relation to Petri Nets

Last time we gave a definition for conflicting transitions that only held for the special case of pairs of transitions. The following easy generalization covers conflict for any set of transitions.

Definition: A subset $\{\sigma_1,\sigma_2,\ldots,\sigma_k\}$ of transitions of a Petri net, where $k\geq 2$, is said to be in conflict under a marking M if there exists a place π in each input set of places of $\sigma_1,\sigma_2,\ldots,\sigma_k$ and $M(\pi)\leq k-1$.

Vector Addition Systems

We introduce vector addition systems as a purely mathematical object and later show how it is related to Petri nets. Later still, we will use vector addition systems for schemata and possibly other problems in parallel computation.

Definition 1: An r-dimensional vector addition system is a pair $\mathcal{W} = (d,W)$ where d is an r-dimensional vector of nonnegative integers, and W is a finite set of r-dimensional integer vectors.

The <u>reachability set</u> $R(W)$ is the set of all vectors of the form $d+w_1+w_2+\ldots+w_s$ such that $w_i \epsilon W$ for $i = 1,2,\ldots,s$, and $d+w_1+w_2+\ldots+w_i \geq 0$ for $i = 1,2,\ldots,s$. That is, $R(W)$ is the set of points that can be reached from d by successively adding elements of W such that the path of points so formed always remains in the first orthant.

A simple example: $r = 2$, $d = (1,1)$, $W = \{(-2,1),(0,1),(3,-1)\}$. Note that $(4,2) \epsilon R(W)$ since $(4,2) = (1,1) + (3,-1) + (0,1) + (0,1)$ and the successive points $(4,0),(4,1)$ and $(4,2)$ are all in the first orthant.

We use the following terminology:

(1) For r-dimensional vectors $x \leq y$ if and only if $x_i \leq y_i$ for $i = 1,2,\ldots,r$.

(2) We sometimes use 0 to denote the r-dimensional vector of zeroes.

(3) ω is a symbol such that if n is an integer then $n<\omega$ and $n+\omega = \omega$. In some sense ω intuitively means "as large as desired."

(4) A <u>rooted tree</u> is a directed graph with some designated node, δ, called the <u>root</u>, which has no edges directed into it, each other node has one edge directed into it, and each vertex can be reached through a directed path from the root. If ξ and η are distinct nodes of the rooted tree having a directed path from ξ to η, then we say $\xi \prec \eta$. If there is a directed edge from ξ to η then η is called the <u>successor</u> of ξ. If η is a node with no edge directed out of it, then η is called an <u>end</u>.

For W we construct a rooted tree $T(W)$ with labelled nodes $\ell(\xi)$ for each node ξ, where $\ell(\xi)$ is an r-dimensional vector label having components from $N \cup \{\omega\}$.

<u>Definition 2</u>: $T(W)$ consists of:

(1) a root δ with label $\ell(\delta) = d$.

(2) let η be a node of $T(W)$

(a) if for some vertex $\xi \prec \eta$ $\ell(\xi) = \ell(\eta)$ then η is an end.

(b) otherwise successors of η are formed (one for each $w \in W$ for which $\ell(\eta) + w \geq 0$).

Let ηw denote the successor of η associated with $w \in W$. Then $\ell(\eta w)$ is determined as follows:

(i) if there is a $\xi < w$ such that $\ell(\xi) \leq \ell(\eta) + w$ and $(\ell(\xi))_i < (\ell(\eta)+w)_i$ then $(\ell(\eta w))_i = \omega$.

(ii) if no such ξ exists, then $(\ell(\eta w))_i = (\ell(\eta)+w)_i$.

This is a complicated definition which needs some explaining. The recursive form of definition of $T(W)$ provides a means for recursively constructing $T(W)$ starting with the root with label d. Given any node ξ of $T(W)$ that has not yet been shown to be an end we first construct trial successors to ξ, one for each $w_i \in W$ with temporary label $\ell(\xi) + w_i$. If $\ell(\xi) + w_i < 0$ then it is not a node of $T(W)$, otherwise parts 2b(i) and (ii) of the definition are used to obtain the permanent label for this node, component by component. Having the permanent label one can check to see if the node is an end. The initial portion of the tree $T(W)$ for our example vector addition system is shown in Figure 1.

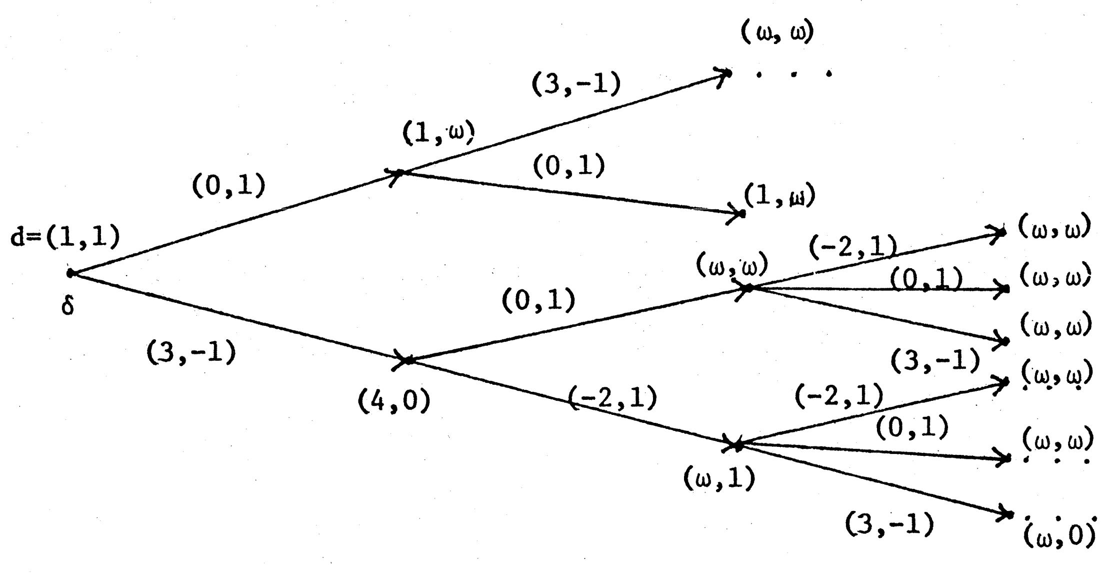


Figure 1: $T(W)$ for example W.

The crucial fact about $T(W)$ that makes it useful is stated in the next theorem.

Theorem 1: For any vector addition system W, $T(W)$ is finite.

Proof: Assume $T(W)$ contains an infinite path from its root, $\delta, \eta_1, \eta_2, \ldots$. Then $\ell(\delta), \ell(\eta_1), \ell(\eta_2), \ldots$ is an infinite sequence of labels with coordinate values taken from $N \cup \{\omega\}$. This must contain an infinite subsequence of labels $\ell(\eta_{i1}), \ell(\eta_{i2}) \ldots$; with $i_1 < i_2$ such that $\ell(\eta_{i1}) \leq \ell(\eta_{i2}) \leq \ldots$. (Such a sequence can be found by first extracting an infinite subsequence which is nondecreasing in the first coordinate, then from this one nondecreasing in the second coordinate, etc.). Since none of these nodes is an end we never have $\ell(\eta_{ij}) = \ell(\eta_{j+1})$. Thus $\ell(\eta_{ij+1})$ has at least one coordinate greater than $\ell(\eta_{ij})$, and by the $T(W)$ definition, 2b(i), this coordinate must equal ω. Now, the number of coordinates is finite, and at least one coordinate must change to ω each time in the subsequence, thus no such infinite path can exist. Also, by construction the number of edges leaving any node is $\leq |W|$ so it is finite. An appeal to König's lemma shows that $T(W)$ is finite. Q.E.D.
The statement of König's lemma which we use is:

König's Lemma: Let T be a rooted tree in which each vertex has only a finite number of successors and there is no infinite path away from the root. Then T is finite.

Before continuing we note that $T(W)$ encodes some information about the reachability set $R(W)$. If $T(W)$ contains a node ξ and $\ell(\xi)$ is finite in all components then the path from δ to ξ shows how the vector $\ell(\xi)$ can be reached from d by successively adding elements from W such that the path always remains in the first orthant. If some coordinates of a node ξ are ω, this in some sense means that by successive application of some subsequence of vectors this coordinate value can be made "as large as desired," or can be "pumped." Since several ω's in ξ can interact with each other care must be taken in such pumping. With a careful analysis (see proof in [72] of Theorem 4.2) we obtain the following theorem.

Theorem 2: Let x be an r-dimensional vector of nonnegative integers. Then the following statements are equivalent:

(1) there is a $y \in R(\ )$ such that $x \leq y$;

(2) there is a node $\eta \in T(W)$ such that $x \leq \ell(\eta)$.

Now, since $T(W)$ is finite, and can be recursively constructed we obtain a number of decidable properties for vector addition systems.

Corollary 1: It is decidable of a vector addition system W and a point x whether $R(W)$ contains a point $y \geq x$.

Corollary 2: It is decidable of an r-dimensional vector addition system and a set $\Theta \subseteq \{1,2,\ldots,r\}$ whether the coordinates in Θ are simultaneously unbounded.

Corollary 3: It is decidable of a vector addition system W whether $R(W)$ is finite or infinite.

Relationship between Generalized Petri Nets and Vector Addition Systems

If we let each place of an n place Petri net be represented by a coordinate it seems possible to represent the reachable set of markings of a Petri net by an n-coordinate vector addition system. Here the initial marking vector M_o would correspond to the initial vector d of $\mathcal{W} = (d,W)$ and each transition σ would give rise to an element of W in which the coordinates for the input places for σ would have -1 entries, and the coordinates for the output places for σ would have $+1$ entries, and all other coordinates would be zero. This is intuitively the rough idea of a possible correspondence but we wish to be more precise. First we would like to allow entries in $w_i \epsilon W$ to be other than $+1,-1$ or 0 so we generalize the notion of Petri net so that transition firings can remove and add more than single tokens, we call such a structure a generalized Petri net (see Keller [74] and Hack [54]).

Today: Relationship between VAS and Generalized Petri nets.

In the rough relationship between vector addition systems and Petri nets that we discussed last time we saw that transitions of a Petri net were represented by elements of W in a vector addition system which had entries of only 0, -1, or +1. We generalize Petri nets so that this restriction no longer holds.

Definition 1: A generalized Petri net $P = (\Pi,\Sigma,R,M_0,\Delta_I,\Delta_0)$ consists of:

(i) a finite set Π called places,

(ii) a finite set Σ called transitions,

(iii) a relation $R \subseteq (\Pi\times\Sigma)\cup(\Sigma\times\Pi)$,

(iv) a mapping M_0: $\Pi \to N$, called the initial marking, and

(v) two functions Δ_I: $(\Pi\times\Sigma) \to N$ and Δ_0: $(\Sigma\times\Pi) \to N$, where for $\pi\in\Pi$ and $\sigma\in\Sigma$, $\Delta_I(\pi,\sigma) = 0$ if and only if $(\pi,\sigma)\notin R$ and $\Delta_0(\sigma,\pi) = 0$ if and only if $(\sigma,\pi)\notin R$.

A generalized Petri net is like a Petri net (conditions (i) through (iv)) with added functions Δ_I and Δ_0. These functions define the amount by which the number of tokens on a place π change by the firing of a transition σ. A transition is called active or fireable in a generalized Petri net if each input place π to σ contains at least $\Delta_I(\pi,\sigma)$ tokens. The firing of an active transition σ changes the number of tokens on a place π by the amount $\Delta_0(\sigma,\pi) - \Delta_I(\pi,\sigma)$. We use the same terminology and concepts developed for

Petri nets to discuss generalized Petri nets. The only extension being generalizing the removal and addition of tokens by transition firing to be other than single token changes. See [54, 74, 99] for further discussion of generalized Petri nets.

The next two definitions describe structural restrictions on generalized Petri nets.

Definition 2: Two transitions $\sigma \neq \sigma'$ of a generalized Petri net P are called equivalent transitions if and only if, for all $\pi \in \Pi$, $\Delta_I(\pi,\sigma) = \Delta_I(\pi,\sigma')$ and $\Delta_0(\sigma,\pi) = \Delta_0(\sigma',\pi)$.

Definition 3: A generalized Petri net P is called irreflexive if and only if there does not exist any $\pi \in \Pi$ and $\sigma \in \Sigma$ such that $\Delta_I(\pi,\sigma) > 0$ and $\Delta_0(\sigma,\pi) > 0$.

Suppose P is an irreflexive generalized Petri net without equivalent transitions, where $\Pi = \{\pi_1,\pi_2,\ldots,\pi_n\}$ and $\Sigma = \{\sigma_1,\sigma_2,\ldots,\sigma_t\}$. A system $W(P) = (d,W)$ corresponding to P is defined as follows:

(1) d is an n-coordinate nonnegative integer vector:

$$d = (M_0(\pi_1),M_0(\pi_2),\ldots,M_0(\pi_n)).$$

We also use M_0 to represent this marking vector.

(2) W is a set of t vectors, one for each transition of P. Let w_j denote the vector for transition σ_j and $(w_j)_k$ the kth coordinate value of w_j, then define

$$(w_j)_k = \Delta_0(\sigma_j,\pi_k) - \Delta_I(\pi_k,\sigma_j).$$

Lemma 1: $W(P) = (d,W)$ is an n-coordinate vector addition system.

Proof: Immediate.

Lemma 2: $s = \sigma_{i1}, \sigma_{i2}, \ldots, \sigma_{ik}$ is a firing sequence for P if and only if each of the points $d, d+w_{i1}, d+w_{i1}+w_{i2}, \ldots, d+w_{i1}+w_{i2}+\ldots+w_{ik}$ is in $R(W(P))$.

Proof: The proof here is also trivial. A firing sequence gives a marking sequence starting with the initial marking in which each marking is a nonnegative n-vector. A typical transition is from a marking M_{ij-1} to a marking M_{ij} by σ_{ij}. For σ_{ij} to be active M_{ij-1} must contain at least $\Delta_I(\pi,\sigma_{ij})$ tokens in each place π, and by definition M_{ij} coordinates are related to M_{ij-1} coordinates by a change $\Delta_0(\sigma_{ij},\pi) - \Delta_I(\pi,\sigma_{ij})$. Thus if the coordinate value for any π in M_{ij-1} is nonnegative, so is that coordinate value for M_{ij}. It must be at least $\Delta_I(\pi,\sigma_{ij})$ to start with and at most $\Delta_I(\pi,\sigma_{ij})$ is subtracted. Thus inductively a firing sequence creates a reachable path in the manner claimed. By similar reasoning a reachable path creates a firing sequence as claimed.

Without going into detail it should be readily seen that for any vector addition system one can construct a corresponding irreflexive generalized Petri net without equivalent transitions. This, plus Lemmas like 1 and 2 give us the result:

Theorem 1: There is an isomorphism between irreflexive generalized Petri nets without equivalent transitions and vector addition systems which provide an isomorphism between firing sequences and reachable paths.

The reader may wish to provide the details for these constructions and results which have been omitted here. Note that the irreflexive and equivalent transition restrictions are important to have the simple isomorphism results. If a Petri net had equivalent transitions σ_i and σ_j then the construction of $W(P)$ would give the same vector for w_i and w_j. Since W is a set the information about equivalent transitions is lost in the mapping from the

the generalized Petri net P to $W(P)$. Thus there would no longer be an isomorphism between firing sequences and reachable paths. The irreflexive property of P means that in transforming a vector addition system to a generalized Petri net that a nonzero entry $(w_j)_k$ in $w_j \epsilon W$ immediately indicates the interconnection of place π_k with transition σ_j. If $(w_j)_k = 0$ there is no direct connection. If $(w_j)_k = a > 0$ then π_k is in the output set of places for σ_j and has $\Delta_0(\sigma_j,\pi_k) = a$. If $(w_j)_k = a < 0$ then π_k is in the input set of σ_j and has $\Delta_I(\pi_k,\sigma_j) = {}^-a$. Irreflexitivity insures that no confusion can exist from the general relation $\Delta_0(\sigma_j,\pi_k) - \Delta_I(\pi_k,\sigma_j)$.

From Theorem 1 relating reachable points in $W(P)$ and reachable markings in P we immediately obtain:

<u>Corollary 1</u>: For any irreflexive generalized Petri net without equivalent transitions $R(W(P)) = R(P,M_0)$.

Thus many properties about generalized Petri nets can be studied via the corresponding vector addition system. For example, the coordinate values of reachable points in $R(W(P))$ determine safeness and K-safeness.

For the remainder of this lecture when we use P for Petri net we will mean an irreflexive generalized Petri net without equivalent transitions. A simple restatement of safeness now is:

<u>Corollary 2</u>: P is safe if and only if each reachable point in $R(W(P))$ has coordinate values that lie in the set $\{0,1\}$, and P is k-safe if and only if the coordinate values lie in the set $\{0,1,\ldots,k\}$.

<u>Corollary 3</u>: The properties safe and k-safe for P are decidable.

Proof: Inspect nodes of $T(W(P))$. For safeness labels on the tree must have coordinate values only from $\{0,1\}$, and for k-safeness from $\{0,1,\ldots,k\}$. Naturally all of $T(W(P))$ may not have to be constructed to prove that a given P is not safe or not k-safe.

A much less immediate corollary, which was shown by Hack [55] through a complex series of constructions using Petri nets, is:

Corollary 4: The questions of liveness of a Petri net P and of whether $x \in R(W)$ in a vector addition system are recursively equivalent.

The corollaries stated for vector addition systems -- using the $T(W)$ tree -- are also directly translated into results for Petri nets. Namely, for any marking M it is decidable whether there is an $M' \geq M$ in $R(P,M_0)$. It is decidable, for any subset of places, whether markings can be reached where the number of tokens in these places are simultaneously unbounded. It is decidable whether $R(P,M_0)$ is finite or infinite.

Consider now the property of whether a given transition σ is dead with respect to a particular marking M. A simple modification of $W(P)$ allows one to decide this. Construct $W'(P) = (M',W')$ exactly like $W(P)$ but add one extra coordinate to represent the firing of σ. Let M' be the initial marking which is equal to M, and with 0 the extra coordinate value. Now for the $w \in W'$ representing σ let the extra coordinate value equal one, and for all other $w \in W'$ that coordinate value is set equal to zero. Now σ is dead with respect to M if and only if there is no $p \in R(W'(P))$ with a value in the extra coordinate greater than zero. This can be tested by inspection of $T(W'(P))$. This technique of adding coordinates to count or test certain properties is useful for testing other properties as well.

A slight generalization of vector addition systems was made by Keller [74] called vector replacement systems. These systems use a pair of vectors (u_i, v_i) rather than a single vector w_i. One of these is a "test" vector, the other a "replacement" vector. The tree construction, and finiteness result, immediately carry over to vector replacement systems, and this allows one to have a correspondence between vector replacement systems and generalized Petri nets without equivalent transitions giving analogous results to those we have discussed; see Keller [74] and Miller [99].

Today: Computation Graphs

We now switch to discussing a different g... ...ical model of parallel computation called the computation graph. This ... s introduced in [69] and studied and extended in a number of furthe... ...udies; e.g., [1, 96, 120].

Basic Definitions

Definition 1: A computation graph G is a fi... directed graph consisting of:

(i) nodes $n_1, n_2, \ldots, n_\ell$.

(ii) edges $d_1, d_2, \ldots, d_t$, where any given ... d_p is directed from a specified node n_i to a specified ... n_j.

(iii) four nonnegative integers A_p, U_p, W_p, T_p associated with each edge d_p, where $T_p \geq W_p$.

In a computation graph each node n_i is us... to represent an operation 0_i and each edge is used to represent a first-... first-out queue of data. Thus an edge d_p directed from n_i to n_j rep...ents a queue of data flowing from n_i to n_j. Results of operation ... $_i$ represented by n_i are placed in the queue and may later be used as op...ands for operation 0_j represented by n_j. The four parameters on edge ...$_p$ are interpreted as follows:

(1) A_p is the number of items initially in the ...ueue from n_i to n_j.

(2) U_p is the number of items added to the que... each time operation 0_i terminates.

(3) W_p is the number of items removed from th... ...ueue each time operation

0_j initiates.

(4) T_p is a threshold giving the minimum number of items required in the queue before operation 0_j can initiate.

Computations are represented in a computation graph as sequences of operation performances. An operation 0_j, associated with node n_j is said to be eligible for initiation if and only if each branch d_p directed into n_j contains at least T_p items in its queue . It is assumed that no two performances of a given operation 0_j can be initiated simultaneously. When 0_j is initiated W_p items are removed from the queue of edge d_p for each such edge directed into n_j. When 0_j terminates each edge d_q directed out of n_j has U_q items added to its queue .

These definitions of operation initiation and termination describe how computations of the computation graph are sequenced. Note that the actual times required for operation performance are not specified. They are, in essence, asynchronous. The possible sequences of initiations for computation graphs are called executions. An execution is represented as a sequence of sets $E = S_1, S_2, \ldots, S_n, \ldots$ such that each S_n is a subset of $\{1,2,\ldots,\ell\}$, the set of node indices. If $j \in S_n$ then this means that 0_j is initiated at step n in execution E. To be more precise we define $x(j,n)$ for $j \in \{1,2,\ldots,\ell\}$ and $n = 0,1,2,\ldots$ as:

$$x(j,0) = 0$$

$$x(j,n) = \text{the number of sets } S_m,\ 1 \le m \le n, \text{ for which } j \text{ is an element.}$$

That is, $x(j,n)$ is the number of initiations of operation j in the prefix $S_1, S_2, \ldots, S_n$ of execution E. With this notation we can define executions more precisely.

Definition 2: The sequence $E = S_1, S_2, \ldots, S_n, \ldots$ is an execution of the computation graph G if and only if, for all n, the following conditions hold:

(i) if $j \in S_{n+1}$ and G has an edge from n_i to n_j, then

$$A_p + U_p x(i,n) - W_p x(j,n) \geq T_p$$

(ii) if E is finite and of length r, then for each n_j there exists an n_i such that d_p is an edge from n_i to n_j and $A_p + U_p x(i,r) - W_p x(j,r) < T_p$.

Definition 3: An execution E is called proper if the following implication holds:

(iii) if, for all n_i and every edge d_p directed from n_i to n_j

$$A_p + U_p x(i,n) - W_p x(j,n) \geq T_p,$$

then $j \in S_r$ for some $r > n$.

In an execution the occurrence of a set S_n in the sequence denotes the simultaneous initiation of O_j for all $j \in S_n$. This model is one of the few that formally (rather than just informally) allows for simultaneous initiation of operations.

Thus, an execution E is viewed as a sequence of sets of events, not necessarily equally spaced in time, where an event is the initiation of an operation of G. As performances of operations in G proceed they generate an execution prefix. Each time an event, or set of simultaneous events, occurs an new element of the execution is generated.

The linear forms

$$A_p + U_p x(i,n) - W_p x(j,n)$$

associated with each edge d_p of G and each S_n of an execution gives the number of items in the queue associated with d_p at this point in the execution if we assume that all of the operations up to this point in E

have actually terminated. Thus, part (i) of the definition for executions insures that sufficient items are in the queues for 0_j to initiate. Condition (ii) insures that an execution will terminate only when no further operations are eligible for initiation. Part (iii), for proper executions, insures that if an operation becomes eligible for initiation at a certain step, then it will actually be initiated after some finite number of steps. This property, often called the "finite delay property," occurs in various forms in different models of parallel asynchronous computation and was apparently first introduced via asynchronous logic circuits by D.E. Muller.

In an execution E terminations are not explicitly mentioned. This does not mean, however, that an execution physically is a set S_n of operations that all initiate simultaneously and then all terminate before any further initiations. For example, if the inequality (stronger than that of (i) in Definition 2)

$$A_p + U_p(x(i,n)-1) - W_p x(j,n) \geq T_p,$$

holds then it is possible that the $x(j,n+1)$st initiation of 0_j may actually occur before the $x(i,n)$th termination of 0_i. No violation of the execution definitions would result.

Any computation graph G may have a large set of executions, and this corresponds to the parallel and asynchronous nature of the model. This set of executions is, thus, the object to study since in some way it represents the behavior of G.

We now consider some simple examples of computation graphs, shown in Figure 1, to illustrate our definitions.

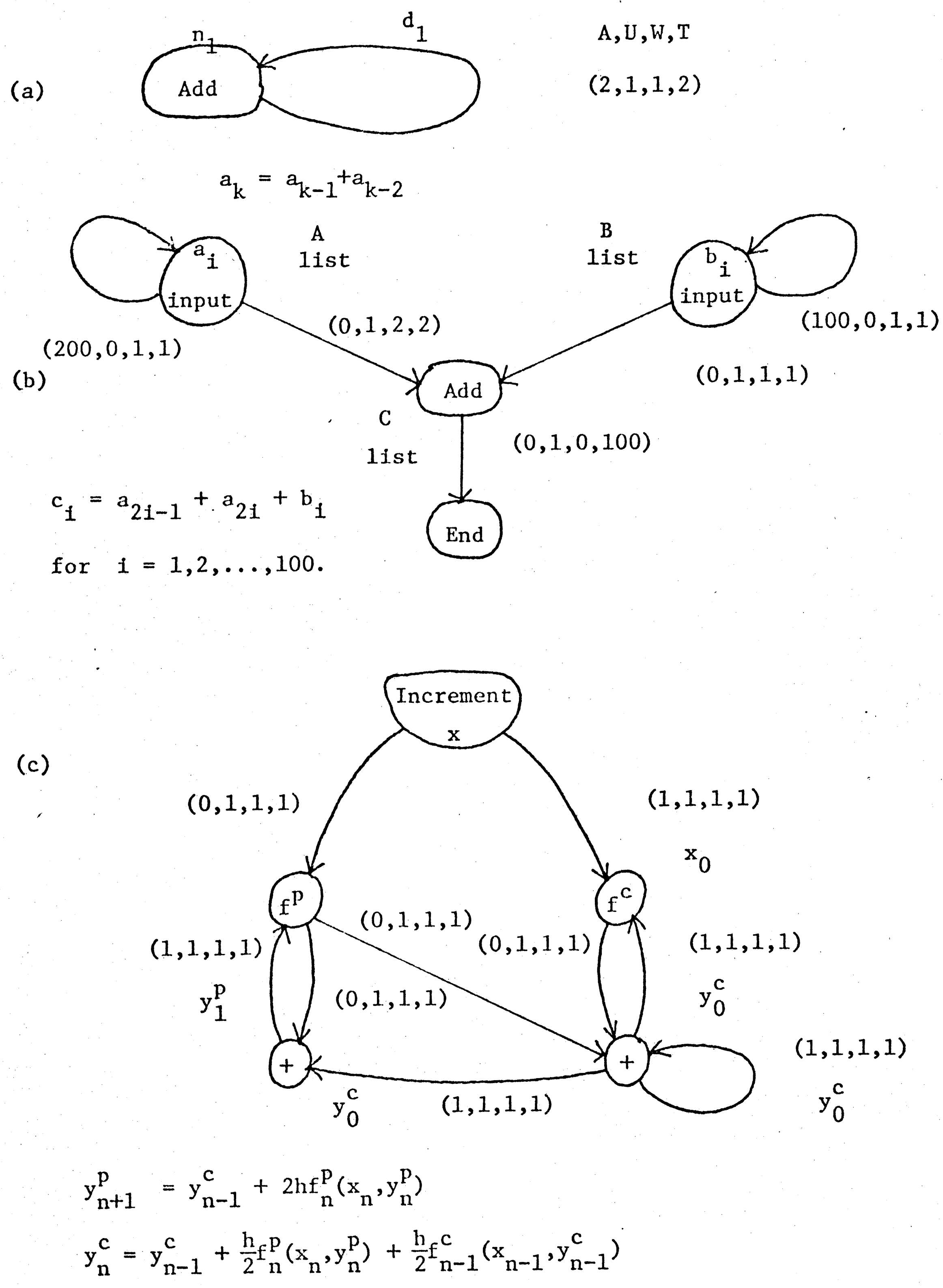


Figure 1: Example Computation Graphs

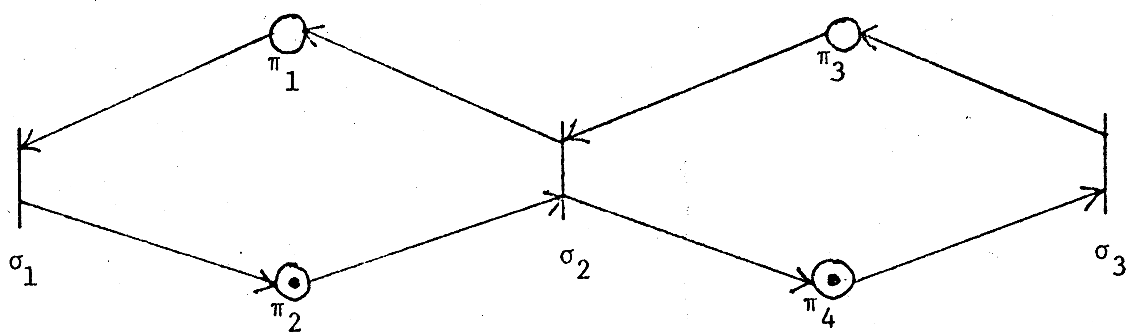
In Figure 1 we have indicated within the graphs, and by equations, a particular interpretation of the computation graph of interest. Of course, the computation graph model does not include any particular interpretation of operations, it models only the sequencing of the operations.

Figure 1(a) shows a single node single edge computation graph with initially two data items in the queue. Each performance of the operation removes one item and places one item on the queue, and two items are required as the threshold for operation initiation. Here we get only a single execution $E = \{1\},\{1\},\ldots,\{1\},\ldots$. If we assume 0_1 to be an add, and the two initial items each to be the integer 1, then E computes the Fibonacci sequence. In part (b) of the figure we can view the operation as adding two lists together (see equation) in which the A list has 200 items, the B list has 100 items and the C list, which is formed on the edge entering the end node, has 100 items. Note that many different sequences of execution exist for this graph.

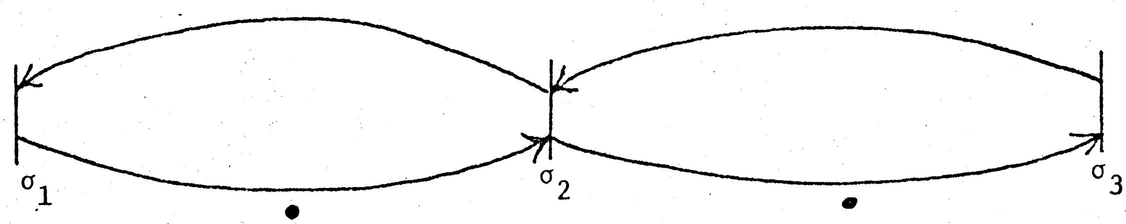
Part (c) of Figure 1 depicts a parallel predictor-corrector scheme of computation for an ordinary differential equation devised by Miranker [101]. The computation graph can be analyzed to determine the amount of parallelism possible in this computation.

Relationship Between Marked Graphs and Computation Graphs

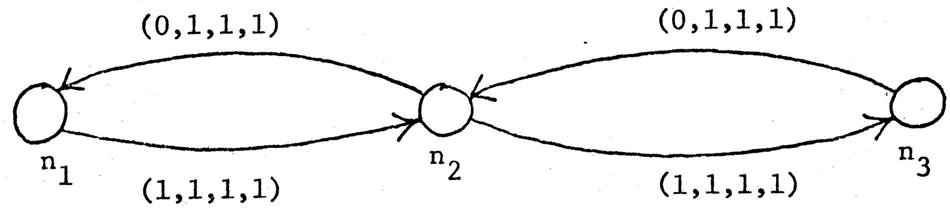
Previously (lecture 5) we defined a marked graph to be a special type of Petri net in which each place $\pi \in \Pi$ has exactly one input transition and one output transition. Thus, the places in a marked graph can be absorbed into edges from transition to transition where tokens are then thought of lying on the edges. Our example marked graph



then becomes:



This graph can now be considered to be a computation graph, of the same node and edge structure. The number of tokens on an edge become the number of items in the queue associated with the edge, and the transition firing rules directly transform into the restriction that for any edge d_p of the computation graph $U_p = W_p = T_p = 1$. The A_p values correspond to the initial marking M_0. A formal correspondence, which should be obvious from this informal discussion, thus could be given. Thereby each firing sequence of the marked graph would correspond to an execution of the computation graph. Executions of the computation graphs having sets S_n with $|S_n| > 1$ would not, however, correspond directly to a single firing sequence but rather a subset of firing sequences where each such S_n would result in an arbitrary ordering of firings. Our example marked graph then becomes the following computation graph,



where n_i corresponds to σ_i, $i = 1,2,3$.

Through this correspondence results of computation graphs can be directly applied to marked graphs. See [96] for an example; we will not amplify this here.

It should be noted that marked graphs for generalized Petri nets (i.e., same restriction on the graphical structure, but using the Δ_I and Δ_0 functions) would also provide a correspondence with computation graphs. Let P represent such a marked graph and G represent the analogous computation graph. Here each transition σ_j of P corresponds to a node n_j of G. Each place π_p of P corresponds to an edge d_p of G where d_p is directed from n_i to n_j if and only if $(\sigma_i,\pi_p)\in R$ and $(\pi_p,\sigma_j)\in R$. The restriction to marked graphs makes this well defined. The edge parameters on d_p are now defined by, $A_p = M_0(\pi_p)$, $U_p = \Delta_0(\sigma_i,\pi_p)$, $W_p = T_p = \Delta_I(\pi_p,\sigma_j)$. Thus marked graphs of generalized Petri nets correspond directly to computation graphs in which $W_p = T_p$ for each edge d_p.

<u>Today</u>: Computation Graphs, continued.

Determinacy of Computation Graphs

We now consider computation graphs with the added constraint that any operation O_j corresponding to a node n_j of G is assumed to create a unique ordered set of data values on any edge leaving n_j for any unique ordered set of T_p values on each edge entering n_j. That is O_j is a function from input values to output values. (The formidable indexing required to be precise is omitted here.) Under these conditions one can ask how the sequence of values appearing in each of the queues is effected by which execution, from the set of executions, is chosen. We will show that the sequence of data items is the same for all proper executions of G with given initial data. We call this property <u>determinacy</u> of computation graphs.

We use the following terminology. Σ_j is defined as the set of ordered pairs (i,p) such that d_p is an edge from n_i to n_j. Operation O_j associated with n_j is assumed specified as a function. Also, we let d_{pv} denote the vth data item placed on edge d_p in execution E, where $d_{p1}, d_{p2}, \ldots, d_{pA_p}$ are the initial values on d_p.

<u>Theorem 1</u>: Let $E = S_1, S_2, \ldots, S_n, \ldots$ and $E' = S_1', S_2', \ldots, S_n', \ldots$ be two proper executions of a computation graph G. Let $x(j,n)$ and $x'(j,n)$ be the number of occurrences of j in $\{S_1, S_2, \ldots, S_n\}$ and $\{S_1', S_2', \ldots, S_n'\}$ respectively. Then for all $j \in \{1,2,\ldots,\ell\}$ and all S_n there exists an S_r' such that $x(j,n) = x'(j,r)$.

Proof: By contradiction. Let n_0+1 ... ce the first positive intèger such that for some j (call it j_0) $x(j_0,n$ ...$) > x'(j_0,r)$ for all r. That is, the first point in *E* where the number ... initiations of some operation 0_{j0} exceeds the number of initiations ... 0_{j0} anywhere in *E'*. Since *E* and *E'* are arbitrarily named we can ... ne this occurs in *E*. Now note that since $j_0 \epsilon S_{n_0+1}$ we have $A_p+U_px($ ...$)-W_px(j_0,n_0) \geq T_p$ for all (i,p) $\epsilon\Sigma_{j_0}$. Since n_0+1 is minimal there ... s an n such that for all (i,p) $\epsilon\Sigma_{j_0}$ $x'(i,n) \geq x(i,n_0)$ and $x'(j_0,n$ ... $x(j_0,n_0)$. By assumption, however, $x(j_0,n_0+1) = x(j_0,n_0)+1$ so $x(j_0,n_0)+$ ... $x'(j_0,r)$ for all r giving $x(j_0,n_0)+1 > x'(j_0,n)$. Using these tw... equalities on $x'(j_0,n)$ we obtain $x(j_0,n_0) = x'(j_0,n)$. We substitute in ... T_p inequality both $x'(i,n)$ which is $\geq x(i,n_0)$ and $x'(j_0,n)$ whic... uals $x(j_0,n_0)$ and obtain

$$A_p + U_px'(i,n) - W_px'(j_0,n) \geq T_p$$

for all $(i,p)\epsilon\Sigma_{j_0}$. Therefore, since ... is a proper execution there must exist an $r>n$ such that $j_0 \epsilon S_r'$. But ... gives $x'(j_0,r) \geq x(j_0,n_0+1)$ contradicting our assumption and provi... ie theorem.

This theorem proves that the numb... f performances of an operation 0_j is the same for all proper executi...

Theorem 2: Let $E = S_1,S_2,\ldots,S_n,\ldots$ ... $E' = S_1',S_2',\ldots,S_n',\ldots$ be two proper executions of a computation ... ph G. If for all p, $d_{pv} = d'_{pv}$, $1 \leq v \leq A_p$, then, for all p, $d_{pv} = d'_{pv}$ ... any value for which d_{pv} is defined.

Proof: Let n_0 be the least n wit... following property: for some $i\epsilon S_n$ the $x(i,n)$th performance 0_i ... uces an output d_{pv} such that

$d_{pv} \neq d'_{pv}$. By minimality of this n_0 all arguments d_{qw} of this performance of 0_i satisfy $d_{qw} = d'_{qw}$, since they are either original data or were formed at an earlier step of E. Since 0_i is assumed to be a function the value determined by a given set of arguments is unique. This contradiction proves the theorem.

From Theorems 1 and 2 we have that for each edge a unique sequence of values occurs in the queue associated with the edge, no matter what proper execution occurs. Thus, the computation graph G is determinate. We can conclude that the asynchronous nature of the sequencing in a computation graph has no effect on the values computed. Thus, whenever a computational process can be represented by a computation graph it is automatically known to be determinate.

These determinacy results for computation graphs are somewhat expected when we consider the restrictions that the computation graphs impose. First, there is no ability within the model to represent conditional branching, and second all memory is "private" to pairs of operations in a result to operand relation as imposed by the queues. Thus no sharing or conflicts can occur in memory utilization.

Relationship between Computation Graphs and Vector Addition Systems

We restrict our attention here to computation graphs which are:

(i) productive, i.e. for each edge d_i $U_i>0$ and $W_i>0$.

(ii) irreflexive, i.e. no edge is a self loop.

(iii) conservative, i.e. for each edge d_i $T_i = W_i$.

(iv) nonequivalent edge: two edges d_m and d_n are called equivalent if $U_m=U_n$, $W_m=W_n$ and they are both directed between the same pair of nodes n_i to n_j.

For any productive, irreflexive, conservative, nonequivalent edge computation graph G with ℓ nodes and t edges there exist a corresponding t-dimensional vector addition system $\mathcal{W}(G)$ defined as follows:

$\mathcal{W}(G) = (d,W)$ where:

$d = (A_1,A_2,\ldots,A_t)$

W is a set of ℓ t-dimensional vectors $w_1,w_2,\ldots,w_\ell$, where

$$(w_i)_j = \begin{cases} -W_j & \text{if } d_j \text{ is directed into } n_i \\ U_j & \text{if } d_j \text{ is directed out of } n_i \\ 0 & \text{otherwise.} \end{cases}$$

From this correspondence one readily sees that points in $R(\mathcal{W}(G))$ correspond to simultaneously achievable queue length values. Thus, from the $T(\mathcal{W}(G))$ results we immediately determine which queues are bounded, what their upper bound is, and which subsets of queues are simultaneously unbounded. By adding an extra coordinate to $\mathcal{W}(G)$ for each node n_j to "count" the number of performances of 0_j one can also determine from the tree on this modifed vector addition system whether any operation or set of operations necessarily terminate, and for any 0_j that terminates the number of performances of 0_j.

In [69] other algorithms for determining termination and queue length are described which, although their complexities have not been analyzed, are probably simpler than the general tree constructions for vector addition systems. Of course, the class of vector addition systems obtained from computation graphs is quite restricted, and the tree construction for this class of vector addition systems may itself be rather simple to construct. The restriction in the class of vector addition systems is that in any coordinate the set of W vectors has exactly one vector with a strictly positive value and exactly one vector with a negative value. This comes from the fact that a coordinate corresponds to an edge of the computation graph which is directed into exactly one node and is directed out of exactly one node.

Producer-Consumer Systems and their Relationship to Semaphores, Computation Graphs and Generalized Petri Nets [See 99]

Producer-consumer problems are an important class of synchronization problems that arise when one considers the interconnection of a set of processes. Essentially, the idea of a producer-consumer system is that a given process of the system produces results that are used (consumed) by some other process. We first define a restricted system we call unshared.

Definition 1: An unshared producer-consumer system $\mathcal{S}$ consists of:

(i) a finite set $B = \{p_1,p_2,\ldots,p_\ell\}$ of processes,

(ii) a finite set $S = \{s_1,s_2,\ldots,s_t\}$ of semaphores,

(iii) a finite set α: $S \to B\times B$ which associates an ordered pair of processes with each semaphore,

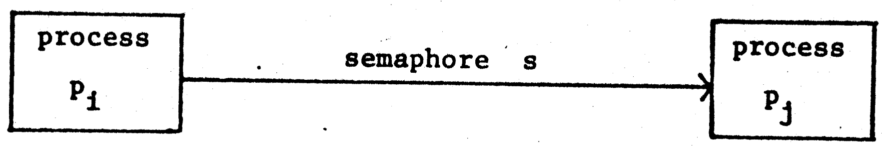
(iv) three functions μ: $S \to N$

π: $S \to N$

ν: $S \to N$

where for a semaphore s with $\alpha(s) = (p_i,p_j)$, $\pi(s)$ is the number of $P(s)$ operations in the beginning of p_j, $\nu(s)$ is the number of $V(s)$ operations at the ending of p_i, and $\mu(s)$ is the initial value assigned to s.

In an unshared producer-consumer system each semaphore is associated with a pair of processes as shown below:



Here the process p_i is thought of as the "producer" of results for "consumer" p_j, where P and V operations are used to indicate to the consumer when sufficient items have been produced for the consumer to start.

This unshared producer-consumer system is a very restricted usage of semaphores. The semaphore is "private" to the producer-consumer pair rather than being shared by several producers or several consumers. However a process may be considered to be a producer (or consumer) for several processes, just as long as one semaphore is used for each producer-consumer pair.

A fairly direct representation of unshared producer-consumer systems by computation graphs should be evident. For an unshared producer-consumer system $\mathcal{S}$ of ℓ processes and t semaphores we can construct a computation graph $G_{\mathcal{S}}$ with ℓ nodes and t edges.

Each process p_i of $\mathcal{S}$ is represented by a node n_i of $G_{\mathcal{S}}$, and each semaphore s_k is represented by an edge d_k of $G_{\mathcal{S}}$ directed from n_i to n_j if $\alpha(s_k) = (p_i,p_j)$. The parameters A_k, U_k, W_k, and T_k are defined as:

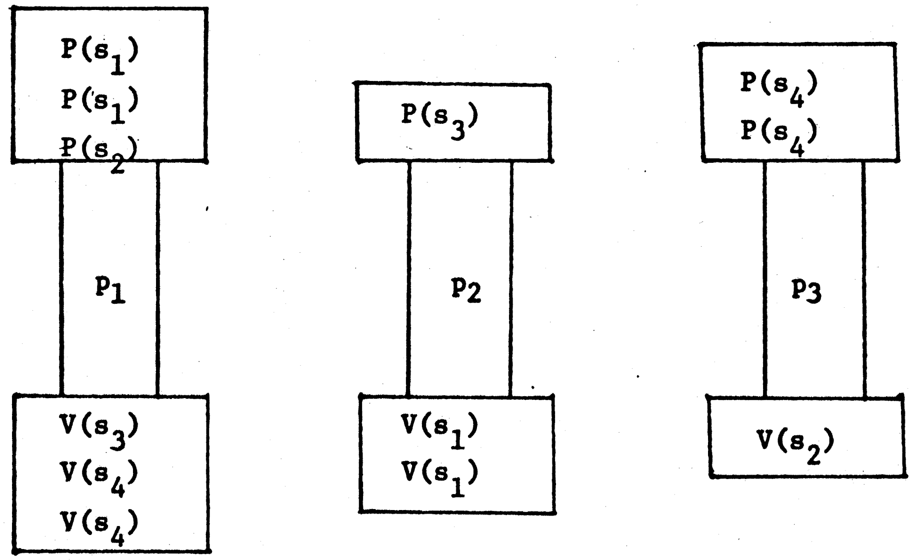
$$A_k = \mu(s_k)$$
$$U_k = \nu(s_k)$$
$$W_k = T_k = \pi(s_k).$$

With this representation, the performance of an operation O_j associated with n_j of G_S corresponds to the performance of process p_j of S. An execution of G_S corresponds to an allowed sequence of process performances in S, where termination properties of the two systems correspond, and where queue length of d_k corresponds to attained semaphore value of s_k.

This correspondence also shows why the generalized P and V operations of the form P(n,s), V(n,s) and P(T,W,s) (see Lecture #4) are natural extensions of P's and V's to consider.

An example of the computation graph G_S for an unshared producer-consumer system is shown in Figure 1.

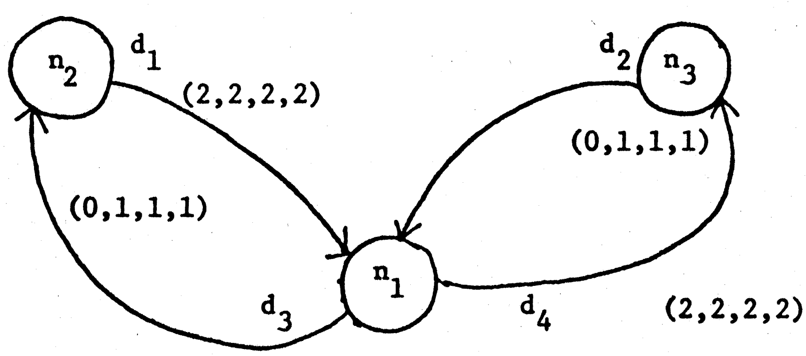
(a) S



$\mu(s_1) = \mu(s_4) = 2$

$\mu(s_2) = \mu(s_3) = 0$

(b) G_S



<u>Figure 1</u>: Unshared Producer-Consumer System S and Corresponding Computation Graph G_S.

This system S, by definition 1 is:

$B = \{p_1,p_2,p_3\}$	$S = \{s_1,s_2,s_3,s_4\}$	
$\alpha(s_1) = (p_2,p_1)$	$\pi(s_1) = 2$	$\nu(s_1) = 2$
$\alpha(s_2) = (p_3,p_1)$	$\pi(s_2) = 1$	$\nu(s_2) = 1$
$\alpha(s_3) = (p_1,p_2)$	$\pi(s_3) = 1$	$\nu(s_3) = 1$
$\alpha(s_4) = (p_1,p_3)$	$\pi(s_4) = 2$	$\nu(s_4) = 2$

We see that in this example initially only process p_3 can start. When p_3 terminates and updates s_2 then p_1 can start. When p_1 finishes and updates s_3 and s_4 then both p_2 and p_3 can start. Process p_1 can initiate again only when both p_2 and p_3 have finished.

The "unshared" aspect of the systems we have just defined is quite restrictive. We generalize.

Definition 2: A producer-consumer system S consists of:

(i) a finite set $B = \{p_1,p_2,\ldots,p_\ell\}$ of processes,

(ii) a finite set $S = \{s_1,s_2,\ldots,s_t\}$ of semaphores,

(iii) three functions μ: $S \to N$

π': $S\times B \to N$

ν': $S\times B \to N$

where for any $s\in S$ and $p\in B$, $\mu(s)$ is the initial value of s, $\pi'(s,p)$ is the number of $P(s)$ operations at the beginning of p, and $\nu'(s,p)$ is the number of $V(s)$ operations at the end of p.

Here the π' and ν' functions let a semaphore be used by any process.

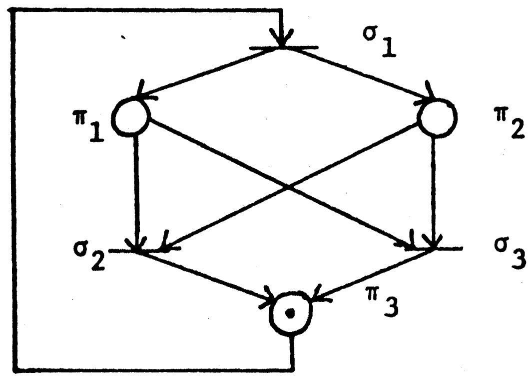
As before, however, we assume all P operations to occur at the start of a process and all V operations to occur at the end of a process. A formal correspondence between producer-consumer systems and generalized Petri nets is depicted below:

Producer-Consumer System S with $B = \{p_1, p_2, \ldots, p_\ell\}$ $S = \{s_1, s_2, \ldots, s_t\}$		Generalized Petri net P with $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_\ell\}$ $\Pi = \{\pi_1, \pi_2, \ldots, \pi_t\}$
p_j	~	σ_j
s_i	~	π_i
$\pi'(s_i, p_j) \neq 0$	~	$(\pi_i, \sigma_j) \in R$
$\nu'(s_i, p_j) \neq 0$	~	$(\sigma_j, \pi_i) \in R$
$\mu(s_i)$	~	$M_0(\pi_i)$
$\pi'(s_i, p_j)$	~	$\Delta_I(\pi_i, \sigma_j)$
$\nu'(s_i, p_j)$	~	$\Delta_0(\sigma_j, \pi_i)$

Although this correspondence between producer-consumer systems and generalized Petri nets gives an isomorphism between the two models, we will show that it does not automatically provide an isomorphism between behaviors. This is shown by the next sample. Consider the three process producer-consumer system with $\mu(s_1) = \mu(s_2) = 0$ and $\mu(s_3) = 1$:

p_1:	p_2:	p_3:
	$P(s_1)$	$P(s_2)$
$P(s_3)$	$P(s_2)$	$P(s_1)$
$V(s_1)$	$V(s_3)$	$V(s_3)$
$V(s_2)$		

This corresponds to the Petri net:



This producer-consumer system has a deadlock. Note that after process p_1 is performed both s_1 and s_2 change to a value of 1. Then p_2 can execute $P(s_1)$ and p_2 can execute $P(s_2)$ which deadlocks the system. No deadlock occurs in the corresponding Petri net, however. Rather, after σ_1 fires then both σ_2 and σ_3 become active. There is a conflict between σ_2 and σ_3, but the global rules for firing transitions do not allow both σ_2 and σ_3 to fire. Thus, the conflict situation in the Petri net is related to the deadlock in the producer-consumer system. More complex examples, like the Cigarette Smokers Problem of Patil show that even a rearrangement of $P(s)$ operations in the processes cannot always circumvent the deadlocking problem. The simultaneous taking of tokens from several places by a transition firing, which prevents the firing of conflicting transitions, is what gives rise to the desire to generalize P operations to operate simultaneously (or in an indivisible manner) on arbitrary subsets of semaphores.

This example should amply demonstrate that one needs to carefully analyze correspondence between models to be sure that the desired properties carry over in the correspondence from one model to the other. Here we see they did not. A weak relationship between conflicts and deadlocks was noted but this has not been precisely described.

Today: Bernstein Analysis of Parallel Processing [15]

Next Time: October 19: Richard Lipton

Complexity of the VAS Tree Construction

October 21: Larry Snyder

Linear Asynchronous Structures

October 26: Fred Sayward

Parallelism Ideas in Operating Systems

October 28: Parallel Program Schemata

(first of several lectures)

In the Bernstein approach we assume a semi-formal model of programs and machines. A program is broken up into blocks (tasks, procedures) which in a sequential program have sequencing as specified by a flow-chart like control. For example, a simple case might be three sequential blocks P_1, P_2, and P_3 depicted in Figure 1.

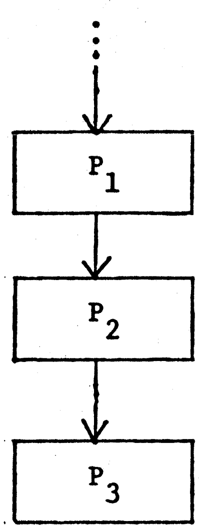


Figure 1

Here P_2 is to follow P_1, and P_3, the remainder of the program, is to follow P_2. How can one tell if either the order of executing P_1 and P_2 can be interchanged, or whether P_1 and P_2 could be executed in parallel? These situations are depicted in Figures 2 and 3 respectively.

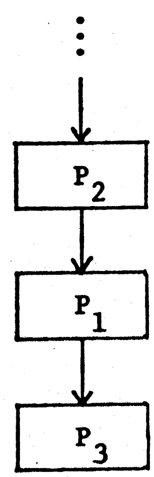


Figure 2: Commuting P_1 and P_2

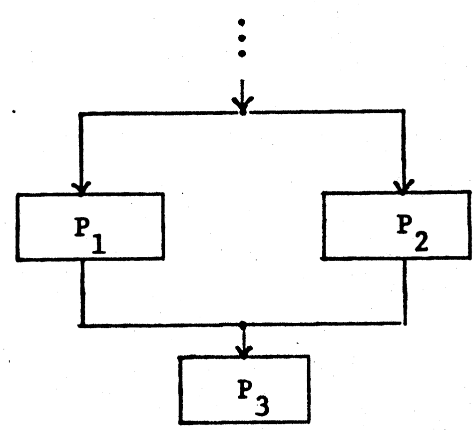


Figure 3: Parallel Operation of P_1 and P_2

Note that the commutativity of P_1 and P_2, as shown from Figure 1 to Figure 2, is not exactly the same as parallel operation of P_1 and P_2 shown in Figure 3. For example, considering P_1, P_2, and P_3 as computing functions f_1, f_2, and f_3 on a variable x it could be that $f_3(f_2(f_1(x)))$ was, by commutativity of f_1 and f_2, equal to $f_3(f_1(f_2(x)))$. This would not imply that parallel operation would be allowed, however, since in parallel operation both P_1 and P_2 would take x as input computing $f_1(x)$ and $f_2(x)$ but then, depending on whether P_1 or P_2 finished last, P_3 would operate either on $f_1(x)$ or $f_2(x)$ giving $f_3(f_1(x))$ or $f_3(f_2(x))$ and neither of these would necessarily equal $f_3(f_2(f_1(x))) = f_3(f_1(f_2(x)))$.

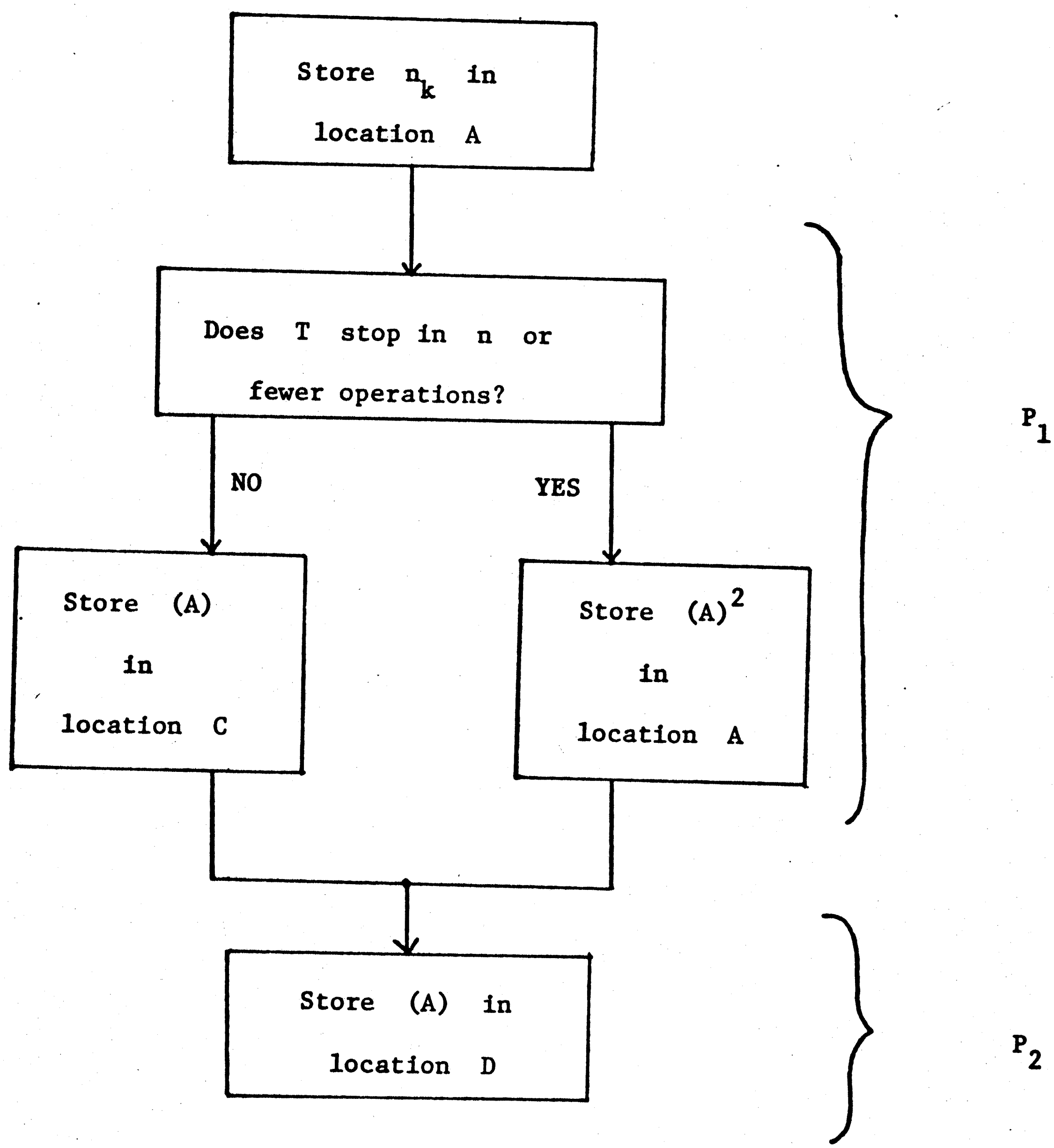
Undecidability of Parallelism Detection

One would like, given any program with a program block structure, to be able to have an algorithm which would answer the question: For P_i and P_j blocks of the program can P_i and P_j be done in parallel? Or similarly can P_i and P_j be commuted? Unfortunately, no such algorithms exist. We show this for the parallelism question.

Theorem 1: The parallelism of two program blocks is undecidable.

Proof: If we had an algorithm to decide block parallelism we show this implies that the halting problem for Turing machines is decidable (i.e. there exists an algorithm for it). Since this is impossible it follows, then, that our parallelism question is undecidable.

Consider the program of Figure 4.



<u>Figure 4</u>: Undecidability of Parallelism Detection

Here n is assumed to be an arbitrary integer stored on an input tape, n_k is assumed to be the k most significant digits of n, locations A, C, and D are assumed to be three different locations used only as mentioned in Figure 4, and (A) means the value in location A.

We are assuming we have an algorithm to test for parallel blocks and apply this to P_1 and P_2 of Figure 4. We also assume that T is an arbitrary Turing machine read into the program as input. Now, if T never halts, then for all input data n, P_1 takes the NO branch. In this case P_1 and P_2 can be done in parallel, storing (A) in locations C and D. If T eventually stops, however, for some n, then P_1 takes the YES branch for these n. That is, T could sometimes stop, and sometimes not stop. In this case P_1 and P_2 must be performed serially since the YES branch in P_1 causes the value in location A to change, and this changed value is needed by P_2 to place in location D. Thus, P_1 and P_2 can be done in parallel if and only if T never halts. Since the halting problem is undecidable so is the parallel block problem. Q.E.D.

Similar reasoning shows that commutativity of blocks is also undecidable.

Sufficient Conditions for Parallelism Detection

Theorem 1 shows that we cannot obtain necessary and sufficient conditions, which are decidable, for parallelism detection, thus we look for rather simple decidable conditions on blocks that are sufficient, when satisfied, to enable the blocks to be done in parallel. These conditions will be based on what memory locations each block uses in various ways.

For our analysis we assume that each program block fetches and stores into a common memory of the machine. We assume that these effected memory locations can be predetermined, and are fixed for each block. Thus, for each block P_i we distinguish four different sets of locations.

1) W_i is the set of memory locations that are only fetched during execution of P_i.

2) X_i is the set of memory locations that are only stored into during execution of P_i.

3) Y_i is the set of memory locations which have first a fetch reference and then some succeeding store reference during the execution of P_i.

4) Z_i is the set of memory locations which have first a store reference followed by some fetch reference during execution of P_i.

Note that execution of P_i does not modify values in locations of the set W_i. Modification can occur only in locations which are in X_i, Y_i or Z_i and not elsewhere in memory. We do not actually know, nor will we attempt to determine, whether each element fetched by P_i is actually used in the execution of P_i or whether each location that P_i stores into actually has its value changed or not. Nevertheless, we assume that since these values may actually be used or be changed, that any transformation from sequential to a parallel or commuted form will have to insure that there was no way these changes could effect the final outcome of the program. Thus, if we wish to go from sequential to parallel form, i.e. from Figure 1 to Figure 3, for blocks P_1 and P_2 we require:

$$(W_1 \cup Y_1) \cap (X_2 \cup Y_2 \cup Z_2) = \phi. \tag{1}$$

That is, that what P_1 uses as input values $(W_1 \cup Y_1)$ cannot be changed by values stored into by P_2 $(X_2 \cup Y_2 \cup Z_2)$. Similarly, P_2 should not destroy results of P_1 which may be needed later during the performance of P_1; thus:

$$Z_1 \cap (X_2 \cup Y_2 \cup Z_2) = \phi. \tag{2}$$

These two conditions (1) and (2) combine into the requirement:

$$(W_1 \cup Y_1 \cup Z_1) \cap (X_2 \cup Y_2 \cup Z_2) = \phi. \tag{3}$$

In the parallel form of operation P_2 no longer necessarily follows P_1, and thus P_2 should not require results of P_1 as input data. Thus:

$$(X_1 \cup Y_1 \cup Z_1) \cap (W_2 \cup Y_2) = \phi. \tag{4}$$

Also P_1 should not overwrite a result that P_2 has written and later needs to use. Thus:

$$(X_1 \cup Y_1 \cup Z_1) \cap Z_2 = \phi. \tag{5}$$

Combining (4) and (5) we obtain:

$$(X_1 \cup Y_1 \cup Z_1) \cap (W_2 \cup Y_2 \cup Z_2) = \phi. \tag{6}$$

Finally, we must insure that when P_3 is entered the values it requires as input; namely, $(W_3 \cup Y_3)$ are not affected by the order in which P_1 and P_2 were executed. The locations that are so affected are those common locations into which both P_1 and P_2 write; namely,

$(X_1 \cup Y_1 \cup Z_1) \cap (X_2 \cup Y_2 \cup Z_2)$.

This leads to the requirement:

$$(X_1 \cup Y_1 \cup Z_1) \cap (X_2 \cup Y_2 \cup Z_2) \cap (W_3 \cup Y_3) = \phi. \qquad (7)$$

Thus, conditions (3), (6) and (7) should be sufficient for transforming P_1 and P_2 from sequential to parallel form. Condition (7) can be simplified, however. Note that from (3) we require $(Y_1 \cup Z_1) \cap (X_2 \cup Y_2 \cup Z_2) = \phi$ and from (6) that $(X_1 \cup Y_1 \cup Z_1) \cap (Y_2 \cup Z_2) = \phi$. Thus,

$$(X_1 \cup Y_1 \cup Z_1) \cap (X_2 \cup Y_2 \cup Z_2) = X_1 \cap X_2, \qquad (8)$$

and applying (8) to (7) we get:

$$X_1 \cap X_2 \cap (W_3 \cup Y_3) = \phi \qquad (9).$$

We conclude that (3), (6) and (9) are sufficient to allow P_1 and P_2 to be done in parallel. Summarizing, we say that P_1 and P_2 can be done in parallel if system I holds:

$$\mathrm{I} \begin{cases} (W_1 \cup Y_1 \cup Z_1) \cap (X_2 \cup Y_2 \cup Z_2) = \phi & (3) \\ (X_1 \cup Y_1 \cup Z_1) \cap (W_2 \cup Y_2 \cup Z_2) = \phi & (6) \\ X_1 \cap X_2 \cap (W_3 \cup Y_3) = \phi & (9). \end{cases}$$

We now turn to whether P_1 and P_2 can commute (Figure 1 to Figure 2) with no change in the program. The conditions we have just considered are again useful. Since the input to P_1 should not be changed by what P_2 computes condition (1) is still required. Condition (2) is not required since for commutativity the executions of P_1 and P_2 are not arbitrarily interleaved. Continuing this reasoning we see that only (1), (4) and (7) are required for commutativity. Similar to our previous simplification (1) and (4) can be seen to give:

$$(X_1 \cup Y_1 \cup Z_1) \cap (X_2 \cup Y_2 \cup Z_2) = (X_1 \cup Z_1) \cap (X_2 \cup Z_2).$$

This then simplifies (7) to

$$(X_1 \cup Z_1) \cap (X_2 \cup Z_2) \cap (W_3 \cup Y_3) = \phi.$$

The final set of sufficient conditions to insure commutativity are then:

$$\text{II} \quad \begin{cases} (W_1 \cup Y_1) \cap (X_2 \cup Y_2 \cup Z_2) = \phi \\ (W_2 \cup Y_2) \cap (X_1 \cup Y_1 \cup Z_1) = \phi \\ (X_1 \cup Z_1) \cap (X_2 \cup Z_2) \cap (W_3 \cup Y_3) = \phi \end{cases}$$

Systems I and II give simple sets of sufficient conditions for allowed transformation from sequential to parallel or commuted form respectively, when the flow structure is originally as depicted in Figure 1.

We note that in both I and II we are testing for "overlap" of memory utilization between two processes P_1 and P_2. Our approach was made simple by assuming that we could determine, with no ambiguity, the sets W_i, X_i, Y_i and Z_i. This, of course, may not always be possible. For example a block P_i may contain conditional branching for which memory utilization is different on each branch. In this situation all paths in P_i must be analyzed. If one branch only fetches from some location, and another branch only stores into that location then, to be safe, one needs to have that location as elements of both W_i and X_i.

Using similar reasoning this sort of approach can be used to derive conditions for other forms of branching and looping structures. Bernstein [15] does some of this and also treats another memory organization using private slave memories for temporary results. The points of interest here are the "domain" and "range" locations in memory that are read or written by the processes, and the order in which processes do this reading and writing. The lack of conflicting uses of common memory locations determine whether these simple transformations to parallel or commuted forms are possible. We will see similar, but more formal, treatment of this via parallel program schemata in subsequent lectures.

(Fred Sayward)

Today: Parallelism in Operating Systems

0. Introduction

Among the various reasons for studying parallelism is the fact that some computer applications are more easily viewed, designed, and implemented as parallel algorithms. This is most evident in operating systems where the underlying computer actually consists of parallel hardware. For example, a CPU and data channels as on the IBM/360 or a CPU and ten peripheral processes as on the CDC 6600.

In today's lecture we will argue why operating system organization is best viewed as cooperating sequential processes and then examine the merits of three forms of interprocess comunication: semaphores, conditional critical regions, and monitors.

1. Why Cooperating Sequential Processes?

Cooperating sequential processes are a system consisting of concurrently executing processes, sharable resources, and primitives for interprocess comunication. Each process is a sequential program which is always executing at some unknown non-zero rate. A process may at any time access a resource. Harmonious accessing of resources is accomplished via interprocess communication (i.e., cooperation).

Most commercial operating systems for second and third generation computers have been designed as a system of *interrupt driven processes*: processes are started, stopped, and re-started as a result of interrupts generated from both within and external to the system. The *indeterminacy* and irreproducability of interrrupts makes system testing and debugging at best a very difficult task and at worst an impossible task. Moreover, when programming at the level of interrupts, the added complication of speed dependent errors arises. A classic example of this was the presence of bugs in OS/360 running on the IBM 360/65 which didn't appear when OS/360 was running on the slower IBM 360/40.

The major advantages of viewing an operating system as cooperating sequential processes as opposed to interrupt driven processes are thus:

(1) it is easier to express the natural synchronizations which take place in the operating system (e.g., no information is used before it is created)

(2) the operating system is easier to prove correct (debug).

2. Hypothetical Four-Level System

The vast majority of current computer systems consist of one (or a few) interruptable CPU and several peripheral devices which generate interrupts. The question is: How can the benefits of cooperating sequential processes be realized? Dijkstra (3) first addressed this question. He views an operating system as levels of abstraction, the hardware being level 0, with each level creating a more attractive system to those levels above it. At some level the system is the cooperating sequential processes and interrupts have been abstracted away. It is at this level that the vast majority of the user service routines are found. Obviously, given the system hardware constraints, we cannot completely ignore the interrupt; however, we have isolated it and put it in its proper perspective.

For the purpose of this lecture we will consider a hypothetical four-level system. At level 0 is the system hardware, with which we will have little concern. At level 1 is the so-called "implementation of cooperating sequential processes." This level controls the system hardware, responds to interrupts, and does I/O, again things with which we will have little concern. We will be concerned with two aspects of level 1 which create the cooperating sequential processes system: the scheduling of processes to be executed on the CPU and the implementation of interprocess communication primitives. At level 2 is the cooperating sequential processes system. This level controls the processing of user jobs, the details of which will not concern us. We will be concerned with the choice of communication primitives and the affects this has with respect to correctness and efficiency at levels 1 and 2. Level 3 is the job stream. This hypothetical four-level system is summarized in figure 1.

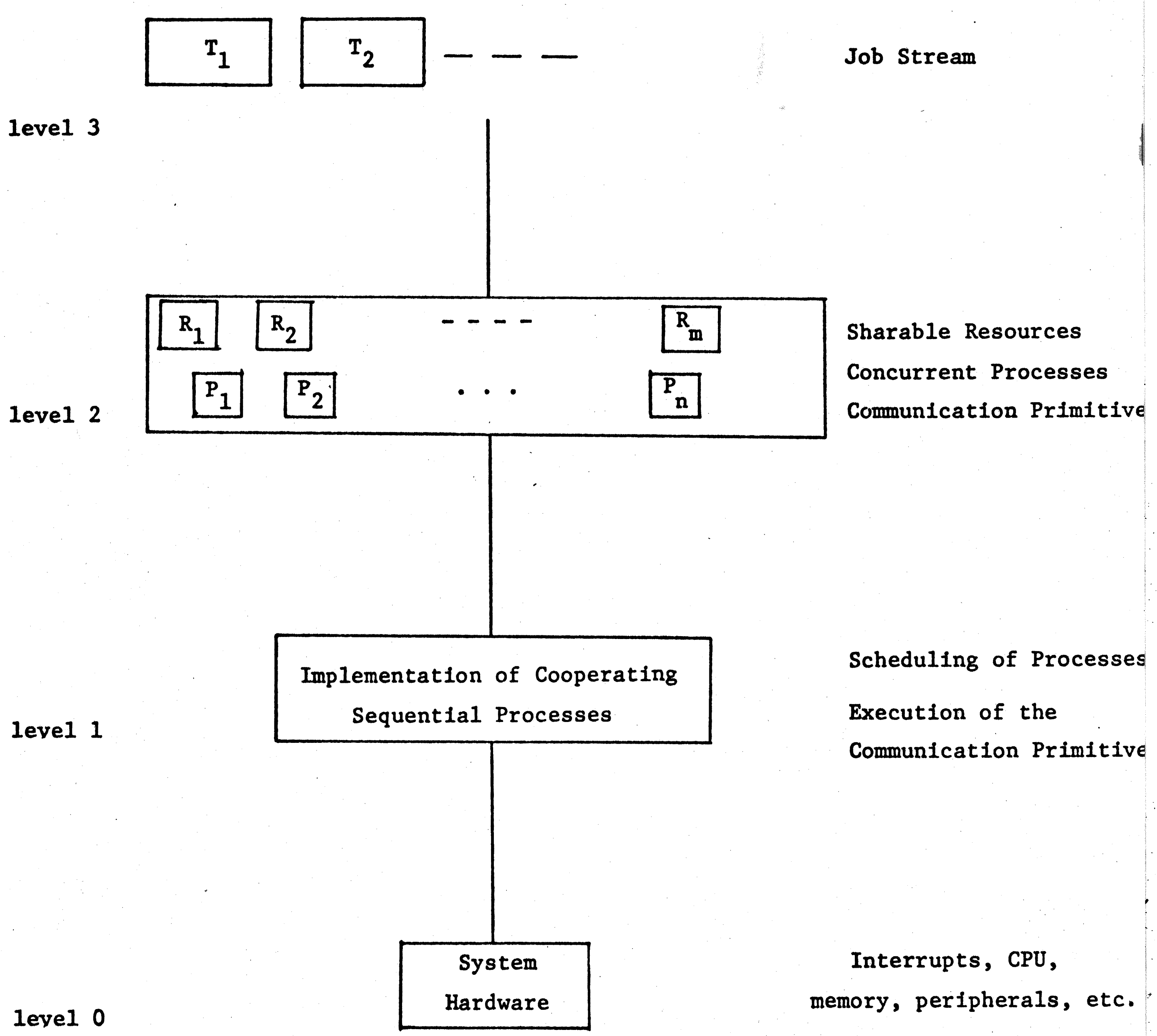


Figure 1: Hypothetical Four-Level System

3. The Key Issues

As alluded to above, our major purpose in this lecture is to evaluate the affect of the choice of interprocess communication primitives with respect to the ease of showing system correctness and the system's efficiency. More specifically, we will treat the following issues:

Level (2)

(1) Mutual Exclusion - although, by definition, all processes may be simultaneously accessing a given resource R, a common type of synchronization is that at most one process accesses R at any given time.

(2) Deadlock - mutual exclusion implies that a process P_1 might have to wait to access a resource R if another process P_2 is currently accessing R. Deadlock is when this wait never ends. Note that this includes deadly-embrace as well as other types of level 2 infinite waits.

(3) Self-Imposed Priorities - aside from mutually exclusive, normally processes must access the resources in some given order (or set of orders). For example, when a card reader process and a disk writer process use a single memory buffer to do spooling, the order is alternation with the reader process first.

(4) Correct Use of Resources - apart from synchronization, how hard it is to prove that the resources, treated as data objects, are operated on correctly.

Level (1)

(5) Implementation of Communication Primitives - how difficult are they to implement and how difficult is it to prove the implementation correct.

(6) Scheduler Fairness - the scheduling of processes is fair if a process

which is eligible to access a resource eventually does so. Although this resembles deadlock, we note that deadlock freeness at level 2 does not imply scheduler fairness at level 1 and vice versa.

(7) Busy Wait - by definition, all processes are always executing in their level 2 environment, even when they are waiting to access a resource which is currently in use. Busy wait avoidance is seeing that the scheduler never assigns the CPU to a process which is in such a waiting state.

4. Semaphore-Based Communication

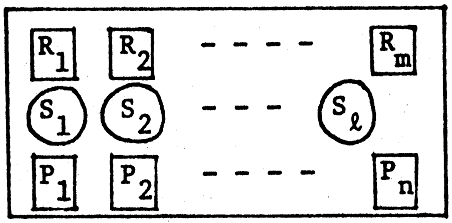
Semaphore-based interprocess communication was first designed and implemented by Dijkstra (3) in the THE operating system. Semaphore variables and operations on them are added at level 2. These primitives have been defined in a previous lecture but for completeness we briefly review them here. Semaphore variables are non-negative integer variables which can only be operated on by two operations: P and V. P and V are non-interruptable and at any time at most one process may be operating on a given semaphore. At level 2 the P and V operations appear as follows:

P(S) L: <u>if</u> $S>0$ <u>then</u> $S \leftarrow S-1$ <u>else</u> <u>go</u> <u>to</u> L

V(S) $S \leftarrow S+1$

Level 2 under semaphore-based communication is summarized in figure 2.

level 2



Sharable Resources
Semaphore Variables
P and V Operations
Sequential Processes

Figure 2: Level 2 with Semaphore Communication

We now give a typical implementation of semaphores; in fact, the implementation described in (3). Each process is always in one of two states: active or sleeping. At level 1 there are $\ell+1$ queues: a sleeping queue, denote q_{S_i}, associated with each semaphore variable and an active queue which we denote by q_a. Processes in q_a are eligible to be executed on the CPU, others are not. Initially all processes are active. At level 1 the P and V operations are as follows:

$P(S_i)$	1: $S_i \leftarrow S_i-1$
	2: <u>if</u> $S_i<0$ <u>then</u> put P_j on q_{S_i} <u>else</u> P_j remains active
$V(S_i)$	1: $S_i \leftarrow S_i+1$
	2: <u>if</u> $S_i\le 0$ <u>then</u> choose some P_k in q_{S_i} to activate

where P_j is the process executing the operation.

In evaluating the communication primitives we will give a list of pros and cons with respect to our key issues.

Pros

1. Ease of Implementation - as seen above the code for the semaphore operations is short and efficient. If there is but one CPU then that one process accesses a given semaphore at any time is trivial. Indivisibility of the operations can be done by inhibiting all interrupts.

2. Scheduler Fairness - this is most easily accomplished by making all queues FIFO.

3. No Busy Wait - although there is conceptual busy wait at level 2, at level 1 only active processes may get the CPU.

Cons

1. Mutual Exclusion - this is hard to prove under semaphores. It is easy to make programming errors and the compiler can be of little help since there is no relationship between resources and semaphores.

2. Absence of Deadlock - again difficult to prove and easy to make programming errors.

3. Self-Imposed Priorities - very difficult to program semaphores to realize complicated synchronizations. Consequently, very hard to prove this aspect.

4. Correct Use of Resources - since the accessing of a given resource R may be scattered throughout any number of processes, this aspect is difficult to formalize and hard to prove.

5. Conditional Critical Region-Based Communication

Hoare (5) proposed the conditional critical regions as a way of eliminating some of the cons of semaphores and their use in operating systems has been described by Brinch Hansen (2). They have been defined in a previous lecture and are summarized below.

Under conditional critical region communication the structure of level 2 is as described in figure 1, there are no special common variables. Processes access the shared resources with either the critical region or the conditional critical region statement, the former being a subcase of the latter. Their syntax and semantics are as follows:

CRITICAL REGION

region R_i do statement end

Semantically, a critical region says: "With exclusive access to R_i execute 'statement' (which accesses R_i) and then release the exclusive access to R_i."

CONDITIONAL CRITICAL REGION

region R_i *when* B *do* statement *end*

Semantically, a conditional critical region says: "With exclusive access to R_i evaluate the predicate B (which accesses R_i). If B is true, then execute 'statement' (which accesses R_i) and then release the exclusive access to R_i. If B is false then release the exclusive access to R_i and re-execute the conditional critical region."

Note that instances of the conditional critical region may be nested.

A typical implementation of conditional critical regions is not much different than one for semaphores. Suppose we have semaphores at level 2 as was illustrated in figure 2. Our conditional critical region level will be 2'. At level 2 we associate a semaphore S_i with each resource R_i. Initially each S_i=1. In terms of level 2, the level 2' interprocess communication primitives become:

CRITICAL REGION

1: $P(S_i)$

2: statement

3: $V(S_i)$

CONDITIONAL CRITICAL REGION

1: $P(S_i)$

2: *if* B *then* statement;$V(S_i)$

else $V(S_i)$; *goto* 1

Pros

1. Ease of Implementation - only slightly more difficult than semaphores.
2. Scheduler Fairness - same as for semaphores.
3. Mutual Exclusion A Priori - the implementation guarantees that

syntactically correct processes (can be checked by the compiler) have the mutual exclusion property.

4. Deadlock Avoidance - although the problem has not been completely solved, the compiler can detect potential deadlocks and the programmer is less likely to commit errors which lead to deadlocks.

Cons

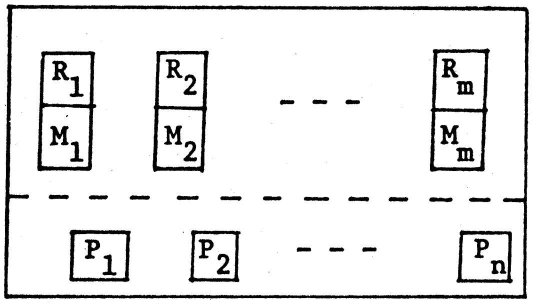
1. Self-Imposed Priorities - the programming of complicated synchronizations is only slightly easier than with semaphores.

2. Correct Use of Resources - the accessing of a given resource is still scattered among the processes.

3. Inherent Busy Wait - in our typical implementation of conditional critical regions in terms of semaphores there is a potentially very inefficient busy wait. At level 2 (part of the implementation) when statement 2 of the conditional critical region is executed with B resulting in false, we have wasted CPU time. Furthermore, this waste is potentially unbounded since we known nothing about the speed of the process which will eventually alter R_i to make B true. One could argue that this is a product of our naive typical implementation. However, although it has been shown (10) how to alleviate the problem, it is in general impossible to completely remove it.

6. Monitor-Based Communication

The use of monitors for interprocess communication which we now describe was suggested by Dijkstra (4), formalized by Hoare (6), and first implemented by Brinch Hansen (1). In monitor communication processes cannot directly access the shared resources; rather, they access resources via "monitor calls." At level 2 there is a monitor process associated with each shared resource (the shared resource is local to the monitor) and the cooperating sequential processes system is visualized as in figure 3.



level 2

↑ indirect accessing of resources

Figure 3: Cooperating Sequential Processes with Monitors

A monitor process consists of three parts: data, procedure calls (monitor calls), and code for data initialization. The data also consists of three parts: the shared resource, local variables, and queue variables. A queue variable (call it q) may only be accessed by a monitor procedure (not the initialization section) via the operations "q.wait" and "q.signal," the semantics of which will be defined below. Figure 4 illustrates this organization. For a monitor with a name of "monitorname" a process accesses the associated resource via an ALGOL-like procedure call:

monitorname.procedurename (actual parameters)

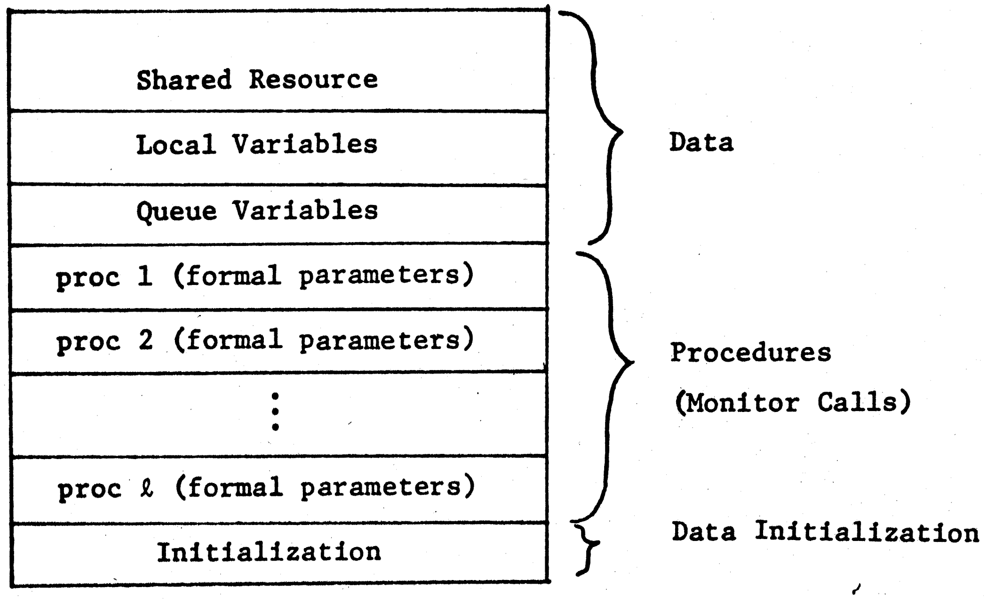


Figure 4: Monitor Organization

The rules under which the monitor processes, the sequential processes, and the implementation operate are now summarized:

(1) The implementation maintains $m+1$ queues: an active queue "AQ" and for each monitor M_i a waiting to enter monitor queue "$WEMQ_i$." As in previously described implementations, only processes in AQ are eligible to be executed on the CPU.

(2) The implementation guarantees that for a given monitor at most one process is executing a monitor call at a time.

(3) Before any process is activated each monitor executes its initialization section and is thereafter considered inactive.

(4) A process executing a monitor call is considered to be active. It is actively executing a procedure in the traditional sense.

(5) The implementation maintains a queue for each queue variable declared in a monitor. A monitor may only access these queues via the signal and wait operations alluded to above. To see the semantics of these operations, let's assume that process "proc" is executing a monitor

call in which one of these queue operations is executed on "q." The affect is:

(i) q.wait - put "proc" on "q" and suspend this monitor call at the point of the q.wait (a la coroutines) thus freeing the monitor to be called by other processes.

(ii) q.signal - (a) terminate this monitor call, and (b) if q is non-empty (i.e. some processes were put there as a result of q.waits) choose some process on q and complete its monitor call.

Hence, q.signal gives priority to processes in level 2 queues over those processes in the level 1 queue WEMQ. Note also that it is impossible for a monitor to signal without terminating.

To solidify these complex definitions we now give a simple example of using monitors: a single buffer producer/consumer. The producer is a card reader process which deposits card images in a common memory buffer and the consumer is a disk writing process which takes card images from the buffer and writes them on a disk. This situation, commonly found in spooling systems, is illustrated in figure 5.

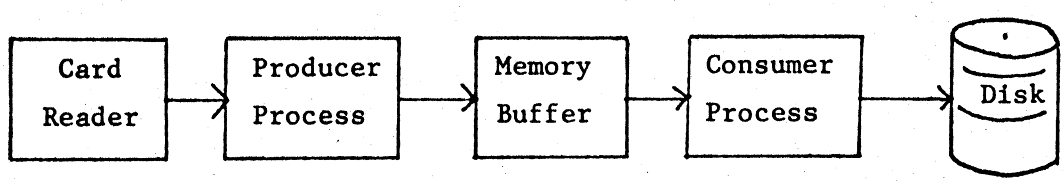


Figure 5: Single Buffer Producer/Consumer

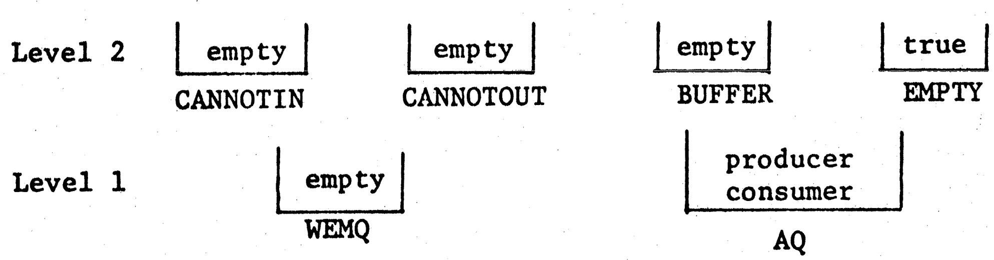
The monitor process which controls the single memory buffer has two monitor calls: an "inbuf" for depositing a card image into the buffer and an "outbuf" for taking a card image from the buffer. Clearly, the monitor must be constructed so that (1) the producer and consumer alternate in the buffer accessing, (2) an inbuf monitor call must wait if the buffer is full, (3) an outbuf monitor call must wait if the buffer is empty, and (4) the buffer is initially empty. Parts (2) and (3) will be done via two queue variables, "cannotin" and "cannotout," and the state of the buffer by a local variable "empty." Satisfying points 2-4 will satisfy point 1. The monitor process is given in figure 6.

The producer and consumer processes have the following structure:

```
PRODUCER    1:  read card reader
            2:  singlebuffer.inbuf (card image)
            3:  goto 1

CONSUMER    4:  singlebuffer.outbuf (card image)
            5:  store on disk
            6:  goto 4
```

In the implementation there are four queues: the cannotin and cannotout queues which are conceptually at level 2, the waiting to enter the single-buffer monitor queue WEMQ at level 1, and the active queue AQ. To see how they interact let's assume that there are two inbuf monitor calls followed by two outbuf calls: ... inbuf 1 ... inbuf 2 ... outbuf 1 ... outbuf 2 ... Initially we have the following situation:



After the execution of inbuf 1 we have only changed the state of the buffer

```
singlebuffer:  monitor
               begin  character  buffer (80);          {shared resource}
                      boolean  empty;                  {local data}
                      queue  cannotout, cannotin;

                      procedure  outbuf (card)
                         begin  character  card (80);
                                if empty then cannotout.wait;
                                card: = buffer;
                                empty: = true;
                                cannotin.signal;
                         end  outbuf;

                      procedure  inbuf (card)
                         begin  character  card (80);
                                if ¬ empty then cannotin.wait;
                                buffer: = card;
                                empty: = false;
                                cannotout.signal;
                         end  inbuf;

                      empty: = true;                   {initializing of local data}

               end  singlebuffer;
```

Figure 6: Monitor Process for Accessing the Buffer

and the local variable empty:

card 1	false
BUFFER	EMPTY

When inbuf 2 gets processed, since the buffer is full, the monitor call must be suspended and we arrive at the following:

producer (inbuf 2)	empty	card 1	false
CANNOTIN	CANNOTOUT	BUFFER	EMPTY
empty		consumer	
WEMQ		AQ	

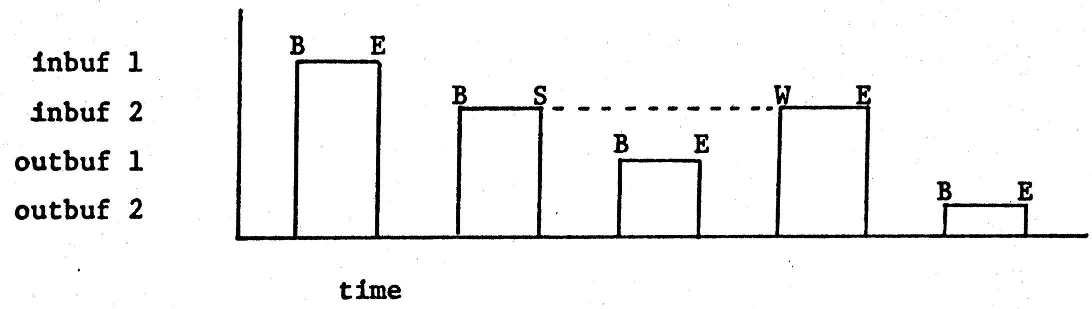
Next, outbuf 1 gets processed and it terminates via a cannotin.signal. The state is as follows:

producer (inbuf 2)	empty	empty	true
CANNOTIN	CANNOTOUT	BUFFER	EMPTY
empty		consumer	
WEMQ		AQ	

The monitor signal rules say that the pending ibuf 2 monitor call <u>must</u> now be resumed. Even if the WEMQ were non-empty this would still be the case. This also prevents the monitor call outbuf 2 from beginning. After inbuf 2 finishes we get:

empty	empty	card 2	false
CANNOTIN	CANNOTOUT	BUFFER	EMPTY
empty		producer consumer	
WEMQ		AQ	

Only now can the monitor call outbuf 2 begin. If we look at the processing of the monitor calls by the implementation with respect to time we get the following:



where B represents begin, E end, S suspend, W wakeup, solid lines the processing of the monitor call, and broken lines a suspended monitor call.

In evaluating monitors with regard to our key issues we note that monitors may call other monitors and hence deadlock is possible. The PROS list now contains all seven criteria. In (9) it is described how monitors may be efficiently implemented. The only apparent drawback is in actually proving the implementation and monitors themselves correct. This is illustrated by the complex proof of fairness given in (8).

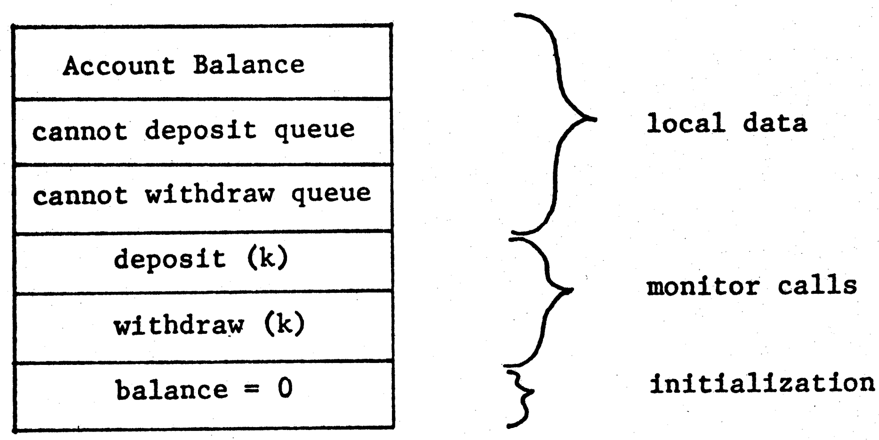
7. A Weakness of Hoare/Brinch Hansen Monitors

When Hoare (6) formalized the monitor concept he questioned his own definition of the monitor signal operation: "The question whether signal should always be the last operation of a monitor procedure is still open." The answer to this question has recently been shown to be no as the following problem of Howard (7) indicates:

JOINT CHECKING ACCOUNT PROBLEM

A husband and wife share a joint checking account. The only operations they may make are (i) deposit (k) which increases the account balance by k dollars and (ii) withdraw (k) which decreases the account balance by k dollars. Furthermore, the account balance should never exceed M dollars for fear of bank failure. However, deposit and withdrawal of k>M dollars are allowed. In this case, deposits and withdrawals are to alternately fill and empty the account, signaling as they go.

The problem is to write a monitor program to represent this action where the monitor will have the following structure:



bank monitor

Note that the only allowed bank operations by the husband and wife are monitor calls. A bank.deposit (k) is a single operation in their domain. If the monitor cannot do the transaction it must suspend the monitor call by using the cannotdeposit queue. The situation is similar for withdrawals.

Now consider the following state. The account balance is 0, the husband is in the cannotwithdraw queue, and the wife executes a bank.deposit (M+1) monitor call. Then the monitor deposit procedure must do the following three things: (1) increase the balance to M, (2) cannotwithdraw.signal and (3) cannotdeposit.wait (with one dollar pending). (1) can easily be done.

However, if (2) is done next then by the Hoare/Brinch Hansen signaling rule the monitor call terminates, (3) never gets done and the wife thinks that the entire deposit of M+1 dollars has taken place. Alternatively, if (3) is done first then both the husband and the wife are waiting and the system deadlocks. Hence the problem cannot be solved.

Howard (7) had proposed and studied several possible extensions to the Hoare/Brinch Hansen signaling rule which allow the joint checking account problem to be solved. Note that the problem is a bit contrived. For example, how do you prevent the deadlock situation of both the husband and wife trying to withdraw when the account balance is zero without giving out information which allows a solution to joint checking account problem? Nevertheless, it indicates that there are situations when following a signal the monitor procedure would prefer to have the calling process go into a monitor queue rather than terminate the monitor call.

References:

(1) Brinch Hansen, P., "The Programming Language Concurrent Pascal," IEEE Transactions on Software Engineering SE-1 (2), June 1975, 199-207.

(2) Brinch Hansen, P., Operating System Principles, Prentice-Hall, 1973.

(3) Dijkstra, E.M., "The Structure of 'THE' Multiprogramming System," Comm. of the ACM 11 (5), May 1968, 341-346.

(4) Dijkstra, E.M., "Hierarchical Ordering of Sequential Processes," in Operating Systems Techniques, Academic Press, 1971, 72-93.

(5) Hoare, C.A.R., "Towards a Theory of Parallel Programming," ibid., 61-71.

(6) Hoare, C.A.R., "Monitors: An Operating System Structuring Concept," Comm. of the ACM 17 (10), October 1974, 549-557.

(7) Howard, J.H., "Signaling in Monitors," presented at the Second International Conference on Software Engineering, San Francisco, October 1976.

(8) Karp, R.A. and Luckham, D.C., "Verification of Fairness in an Implementation of Monitors," ibid.

(9) Saxena, A.R., "An Efficient Implementation of Monitors and Condition Variables," Stanford University Digital Systems Laboratory Technical Note No. 72, August 1975.

(10) Schmid, H.A., "On the Efficient Implementation of Conditional Critical Regions and the Construction of Monitors," _ACTA Informatica_ 6, 1976, 227-249.

Today: Introduction to Parallel Program Schemata

The parallel program schema model [72] is an abstract formulation of parallel programs. It is a complex model, incorporating a very general form of operation sequencing and operations which fetch operands and store results in a common memory. Various types of program schema models have been proposed [19, 37, 72, 75, 80, 81, 82, 90, 94, 97, 129, 130, 135, 136] this being the most general one which includes parallel operation. The term "schema" is used to indicate that the model is an abstraction which concentrates on the sequencing aspects of the program and leaves unspecified certain functional aspects of the program. By doing this, certain properties of the program which are invariant over the functional specification can be more readily studied.

We will first introduce the formal schema definitions, then later prove some theorems.

Definition 1: A parallel program schema $\mathcal{S} = (M,A,\mathcal{T})$ consist of:

(i) M, a set of memory locations,

(ii) A, a finite set of operations $A = \{a,b,c,\dots\}$ where for each $a \in A$ we have:

(a) a positive integer $K(a)$ called the number of outcomes of a,

(b) $D(a) \subseteq M$, a specified set of domain locations,

(c) $R(a) \subseteq M$, a specified set of range locations.

(iii) $\mathcal{T} = (Q,q_0,\Sigma,\tau)$, the control, where:

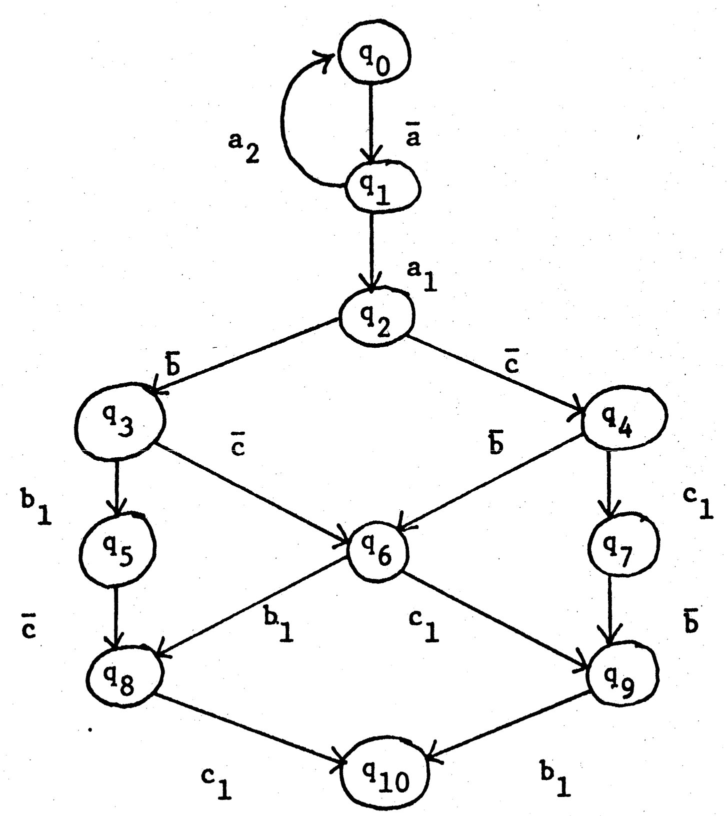
Q is a set of states,

$q_0 \in Q$ is the initial state,

$\Sigma = \Sigma_i \cup \Sigma_t$, the event alphabet with $\Sigma_i = \bigcup_{a \in A} \{\bar{a}\}$, the initiation

<u>symbols</u>, and $\Sigma_t = \bigcup_{a \in A} \{a_1, a_2, \ldots a_{K(a)}\}$, the <u>termination symbols</u>.
τ is the <u>transition function</u> which is a partial function from $Q \times \Sigma$ into Q which is total on $Q \times \Sigma_t$.

A parallel program schema is thought to operate as follows. A computation is a sequence of events, where events are initiations and terminations of operations. When an operation a initiates it reads its operands from its domain locations $D(a)$. The initiation of operation a in a computation is indicated by the symbol $\bar{a}$. Sometime after initiation operation a may terminate. This is indicated by one of its termination symbols $a_1, a_2, \ldots, a_{K(a)}$, which also indicates the conditional branch outcome as well. Upon termination the operation stores the results of its performance in its range locations $R(a)$. The example of a schema control shown in Figure 1 is worth considering.



<u>Figure 1</u>: The control structure for a simple schema.

This schema has three operations $\{a,b,c\}$. All initiation symbol transitions are shown but only those termination symbol transitions that are possible in computations are shown. For our discussion we do not need to specify M or the domain and range locations for the operations. Here, starting in state q_0 the only operation that can initiate is a, where $\bar{a}$ takes us to q_1. In q_1 only terminations of a can occur. If a_2 occurs it takes us back to q_0 and we are in a loop with repeated performances of operation a. If a_1 occurs we "branch" to q_2. In q_2 either b or c can initiate. Essentially this is a FORK situation with b and c being the next two operations which can be done in parallel. From the figure we see that b and c can be performed in parallel (their initiations and terminations being interspersed in any order). When q_{10} is finally reached both b and c have been performed, and q_{10} indicates a JOIN. From q_{10} other events might procede. Some of the possible computations for this example are

(1) $\bar{a}a_2\bar{a}a_1\bar{b}b_1\bar{c}c_1$

(2) $\bar{a}a_1\bar{b}\bar{c}b_1c_1$

(3) $\bar{a}a_1\bar{c}\bar{b}b_1c_1$

Note that in (2) and (3) after the fourth symbol both b and c are in concurrent performance. In (1) the sequence of events was such that no parallel performance occurred.

We now continue with more definitions so that we can be more precise about what computations are.

<u>Definition 2</u>: An <u>interpretation</u> I of a schema S consists of:

(i) a function C associating a set $C(i)$ with each $i \in M$, specifying the set of values allowed in location i.

(ii) the initial memory contents $c_0 \in \underset{i \in M}{X} C(i)$.

(iii) For each $a \in A$, two functions:

$$F_a : \underset{i \in D(a)}{X} C(i) \rightarrow \underset{i \in R(a)}{X} C(i)$$

$$G_a : \underset{i \in D(a)}{X} C(i) \rightarrow \{a_1, a_2, \ldots, a_{K(a)}\}.$$

The F_a function specifies the value that operation a computes for its range values $R(a)$ with given domain values from $D(a)$. The G_a function determines the outcome, or conditional branch, as a function of the domain values in $D(a)$. Now, when we talk about a computation of a schema under a given interpretation we wish the sequence of events to be consistent with both the schema control and with the G_a function in the interpretation.

As is common in formal automata theory models, the complete state or "instantaneous description" of the model is useful in determining the step-wise action of the model.

<u>Definition 3</u>: An <u>*I*-instantaneous description</u> α is a triple (c,q,μ) in which:

(i) $c \in X_{i \in M} C(i)$ is the current memory contents, and $c(i)$ designates the contents of location i.

(ii) q is the current state of the schema control.

(iii) μ is a function associated with each $a \in A$ a finite sequence of elements from $\underset{i \in D(a)}{X} C(i)$. For each a, this is a queue of domain values for each initiated but not yet terminated performance of a.

The initial I-instantaneous description $\alpha_0 = (c_0, q_0, \mu_0)$ where c_0 is the initial memory contents of interpretation I, q_0 is the initial state, and μ_0 is a set of empty queues.

The precise sequencing of operations is now defined in terms of a $\cdot$ operation which, given any event $\sigma \epsilon \Sigma$ and instantaneous description α, produces a new instantaneous description.

Definition 4: A partial function $\alpha \cdot \sigma$, for an I-instantaneous description α and $\sigma \epsilon \Sigma$ is defined by:

(1) (Initiation symbol case) $\sigma = \bar{a}$, $\alpha = (c,q,\mu)$: $\alpha \cdot \bar{a}$ is defined iff $\tau(q,\bar{a})$ is defined. If so: $\alpha \cdot \bar{a} = (c', q', \mu')$ where

$$c' = c$$

$$q' = \tau(q,\bar{a}) \text{ and}$$

for $b \neq a$ $\mu'(b) = \mu(b)$ and $\mu'(a)$ is the queue $\mu(a)$ with $\underset{i \epsilon D(a)}{X} c(i)$ added to the end.

(2) (Termination symbol case) $\sigma = a_j$, $\alpha = (c,q,\mu)$:

$\alpha \cdot a_j$ is defined iff $\mu(a)$ is nonempty and $G_a(\xi) = a_j$, where ξ denotes the first element in the $\mu(a)$ queue.

In this case $\alpha \cdot a_j = (c', q', \mu')$ where:

(i) for $i \notin R(a)$ $c'(i) = c(i)$

(ii) for $i \epsilon R(a)$ $c'(i)$ is the component of $F_a(\xi)$ corresponding to location i

(iii) $q' = \tau(q, a_j)$

(iv) $\mu'(b) = \mu(b)$ for $b \neq a$ and $\mu(a) = \xi \mu'(a)$; that is, $\mu'(a)$ is the tail of $\mu(a)$ after the first element ξ is deleted.

For $y \in \Sigma^*$, $\alpha \cdot y$ is defined in the normal way by letting $\alpha \cdot x\sigma = (\alpha \cdot x) \cdot \sigma$.

<u>Last Time</u>: Parallel Program Schemata
defn., examples
interpretation I
I-instantaneous description α $\alpha' = \alpha\cdot\sigma$ function

<u>Today</u>: I-computation, history, determinacy,equivalence, boundedness

With the $\alpha\cdot\sigma$ definition from last time we are now ready to give a precise definition of an I-computation for a schema.

<u>Definition 5</u>: A finite or infinite word x over the alphabet Σ is an I-computation for schema S iff:

(i) for every prefix y of x, $\alpha_0\cdot y$ is defined;

(ii) if x is finite, then for all $\sigma\epsilon\Sigma$, $\alpha_0\cdot x\sigma$ is undefined;

(iii) (Finite delay property) If y is a prefix of x and $\sigma\epsilon\Sigma$ with the property that for every z such that yz is a prefix of x, $\alpha_0\cdot yz\sigma$ is defined, then for some z' yz'σ is a prefix of x.

Part (i) of this definition insures that an I-computation is consistent with the $\alpha\cdot x$ definition; that is, that it is consistent with both the schema control and with the interpretation. Part (ii) indicates that a computation can end only if no other event could occur, and (iii) says that if after some point in the computation an event can "constantly" occur, then it eventually does occur after some finite delay.

If x is an I-computation it gives rise to a sequence of instantaneous descriptions called the <u>history</u> of x, namely

$$\psi(x) = \alpha_0,\alpha_0\cdot{}_1x,\alpha_0\cdot{}_2x,\ldots,\alpha_0\cdot{}_kx,\ldots$$

where ${}_ix$ is the prefix of length i of x. Also we let $\psi_i(x)$ denote the subsequence of $\psi(x)$ starting with α_0 containing the successive values of $\psi(x)$ that "store values" in location i. That is, $\alpha_0 \cdot {}_kx$ is in $\psi_i(x)$ iff $x_k = a_j$ where a is an operation such that $i \epsilon R(a)$. For $\alpha = (c,q,\mu)$ we also let $\Pi_i(\alpha) = c(i)$, $i\epsilon M$; $\Pi_a(\alpha) = \mu(a)$, $a\epsilon A$, etc. where these Π's are projection operators. The projection operators also apply to sequences of instantaneous descriptions, for example $\Pi_i(\alpha_0,\alpha_1,\alpha_2,\ldots) = \Pi_i(\alpha_0),\Pi_i(\alpha_1),\Pi_i(\alpha_2),\ldots$. In particular we denote by $\Omega_i(x) = \Pi_i(\psi_i(x))$ for an I-computation or prefix of an I-computation x. We call $\Omega_i(x)$ the <u>contents sequence of cell i</u> for x. Note that $\Omega_i(x)$ gives the successive values that appear in location i during the computation x. With these definitions we are now ready to define some of the basic properties of schemata.

<u>Definition 6</u>: A schema S is <u>determinate</u> if whenever x and y are I-computations for the same interpretation I, then:

$$\forall i\epsilon M[\Omega_i(x) = \Omega_i(y)].$$

Determinacy establishes that the entire sequence of values stored in any single location is determined by the interpretation and is not dependent upon which particular I-computation occurs. It provides a rather strong form of "proper behavior."

<u>Definition 7</u>: Two schemata $S = (M,A,T)$ and $S' = (M,A,T')$ are called <u>equivalent</u> if, for each $i\epsilon M$ and each interpretation I;

$$\{\Omega_i(x) \mid x \text{ is an } I\text{-computation for } S\}$$
$$= \{\Omega_i(y) \mid y \text{ is an } I\text{-computation for } S'\}.$$

That is, the schemata are equivalent if they produce equal sets of cell content sequences, cell by cell.

<u>Definition 8</u>: A schema S is called <u>bounded</u> if there is a constant K such that, for every I, and every I-instantaneous description (c,q,μ) which occurs in the history of an I-computation, the sum of the lengths of all the queues $\mu(a)$ is bounded by K. If K can be taken equal to 1 then S is called <u>serial</u>.

A bounded schema has a limit of K on the parallelism in computations. The serial property corresponds to the safeness property of Petri nets.

In what follows we will give some necessary and sufficient conditions for determinacy and then investigate the decidability of these various schemata properties.

Necessary and Sufficient Conditions for Determinacy

We have defined a strong form of determinacy. In effect it means that no matter what the interpretation, when one focuses on any particular memory location, any computations for that interpretation will provide identical sequences of values to occur in that memory location. Of course, this insures that two weaker forms of determinacy hold; namely (1) that for terminating computations the final memory contents will be equal, and (2) that for any specified subset of memory (e.g. a set which might be called the output or result locations) the final values are identical.

We will stick with this stronger form of determinacy on sequences of values. No work on parallel program schemata has been done using the weaker, but possibly practically interesting, forms of determinacy.

Our aim in this section will be to prove the following theorem.

Theorem 1: Let S be a persistent, commutative, lossless schema. Then S is determinate if and only if, for all interpretations I, condition (A) holds:

(A) If $\alpha_0 \cdot u\sigma\pi$ and $\alpha_0 \cdot u\pi\sigma$ are both defined, then $\alpha_0 \cdot u\sigma\pi = \alpha_0 \cdot u\pi\sigma$, where $\pi, \sigma \in \Sigma$ and $u \in \Sigma^*$.

We will presently define the properties persistent, commutative and lossless, but first we consider what this theorem says. It says that if we reach some point in a computation that two events (σ and π) can occur in either order, or intuitively simultaneously, then the results of this race as appearing in memory cells is independent of which event occurs first.

The properties listed in the hypothesis of the theorem are defined as follows:

Definition 9: A schema S is persistent if and only if whenever σ and π are distinct elements of Σ and $\tau(q,\sigma)$ and $\tau(q,\pi)$ are both defined, then $\tau(q,\sigma\pi)$ and $\tau(q,\pi\sigma)$ are also defined.

Definition 10: A schema S is commutative if and only if whenever $\tau(q,\sigma\pi)$ and $\tau(q,\pi\sigma)$ are both defined then $\tau(q,\pi\sigma) = \tau(q,\sigma\pi)$.

Definition 11: A schema S is lossless if for all $a \in A$, $R(a) \neq \phi$.

A number of lemmas are required to prove theorem 1. The first is to introduce the concept of a one-one interpretation. Essentially, a one-one interpretation is one that records in each memory cell the complete history of events that effect that cell. It can be shown that for any interpretation I, one can obtain a one-one interpretation that has the same set of computations as I. This then leads to the lemma:

Lemma 1: Condition (A) of Theorem 1 holds for every interpretation if and only if it holds for every one-one interpretation.

This result, see [72] for details, then allows us to consider only one-one interpretations for the rest of our consideration in proving Theorem 1.

Today: Proof of Theorem on necessary and sufficient condition for determinacy of schemata.

We repeat the statement of Theorem 1.

Theorem 1: Let S be a persistent, commutative, lossless schema. Then S is determinate if and only if, for all interpretations I, condition (A) holds:

(A) If $\alpha_0 \cdot u\sigma\pi$ and $\alpha_0 \cdot u\pi\sigma$ are both defined, then $\alpha_0 \cdot u\sigma\pi = \alpha_0 \cdot u\pi\sigma$, where $\pi \in \Sigma$, $\sigma \in \Sigma$, and $u \in \Sigma^*$.

The proof of this theorem proceeds by proving a sequence of lemmas. Last time we mentioned one-one interpretations and the lemma that allows us to consider henceforth only one-one interpretations. We now prove some properties of the behavior of the $\cdot$ relation for initiation and termination symbols.

Lemma 2: Let S be a persistent, commutative, lossless schema, I a one-one interpretation, and $\alpha = (c,q,\mu)$ an I-instantaneous description. Then for each pair of operations a and b:

(a) If $\alpha \cdot \bar{a}\bar{b}$ and $\alpha \cdot \bar{b}\bar{a}$ are defined then $\alpha \cdot \bar{a}\bar{b} = \alpha \cdot \bar{b}\bar{a}$;

(b) If $\alpha \cdot \bar{a}b_\ell$ and $\alpha \cdot b_\ell \bar{a}$ are defined then $\alpha \cdot \bar{a}b_\ell = \alpha \cdot b_\ell \bar{a}$ if and only if (i) $R(b) \cap D(a) = \phi$ or (ii) b_ℓ is a repetition; i.e. $\Pi_{R(b)}(\alpha) = \Pi_{R(b)}(\alpha \cdot b_\ell)$;

(c) If $\alpha \cdot a_j b_\ell$ and $\alpha \cdot b_\ell a_j$ are defined then: (i) for $a \neq b$, $\alpha \cdot a_j b_\ell = \alpha \cdot b_\ell a_j$ if and only if $R(a) \cap R(b) = \phi$, (ii) for $a=b$, $j=\ell$ and $\alpha \cdot a_\ell a_\ell = \alpha \cdot a_\ell a_\ell$.

<u>Proof</u>: The proof is by cases:

(a) If $a=b$ then $\alpha\cdot\overline{a}\overline{a} = \alpha\cdot\overline{a}\overline{a}$ is obvious. In both cases $\mu(a)$ has two equal tuples of $D(a)$ values added to the queue. If $a\neq b$ then

$$(c,q,\mu)\cdot\overline{a}\overline{b} = (c,\tau(q,\overline{a}\overline{b}),\mu') \quad\text{and}\quad (c,q,\mu)\cdot\overline{b}\overline{a} = (c,\tau(q,\overline{b}\overline{a}),\mu'').$$

Now c is unchanged since initiations do not change memory. $\tau(q,\overline{a}\overline{b}) = \tau(q,\overline{b}\overline{a})$ by commutativity, and all that remains is to show that $\mu'=\mu''$. For $d\neq a,b$ $\mu'(d)=\mu''(d)=\mu(d)$. $\mu'(a)=\mu(a),\Pi_{D(a)}(c)$ and $\mu''(a)=\mu(a),\Pi_{D(a)}(c)$ since the b initiation does not change memory, thus $\mu''(a)=\mu'(a)$. Similarly $\mu'(b)=\mu''(b)$ so $\alpha\cdot\overline{a}\overline{b}=\alpha\cdot\overline{b}\overline{a}$ completing part (a) of the lemma.

(b) Let $a\neq b$. We have

$$(c,q,\mu)\cdot\overline{a}b_\ell = (c',\tau(q,\overline{a}b_\ell),\mu') \quad\text{and}$$
$$(c,q,\mu)\cdot b_\ell\overline{a} = (c'',\tau(q,b_\ell\overline{a}),\mu'').$$

Here $c'=c''$ since a does not change memory so the b_ℓ termination is the only thing that causes a change both times acting with the first element of the $\mu(b)$ queue. By commutativity $\tau(q,\overline{a}b_\ell) = \tau(q,b_\ell\overline{a})$, and again only the μ lists need to be checked.

For $d\neq a,b$ $\mu'(d)=\mu''(d)=\mu(d)$. By definition $\overline{a}$ does not change the $\mu(b)$ list so $\mu'(b)$ is $\mu(b)$ with the first element of $\mu(b)$ deleted. Thus, clearly, $\mu'(b)=\mu''(b)$. Now $\mu'(a) = \mu(a)\Pi_{D(a)}(c)$ and $\mu''(a) = \mu(a)\Pi_{D(a)}(c'')$. These are equal if and only if $\Pi_{D(a)}(c) = \Pi_{D(a)}(c'')$ and since the interpretation is one-one this only happens if c'' is equal to c on $D(a)$. That is, only when $D(a)\cap R(b) = \phi$ or b_ℓ is a repetition. Next consider part (b) for $a=b$. Here, $(c,q,\mu)\cdot\overline{a}a_\ell = (c',\tau(q,\overline{a}a_\ell),\mu')$

and $(c,q,\mu)\cdot a_\ell \bar{a} = (c'',\tau(q,a_\ell \bar{a}),\mu'')$. Now $c'=c''$ since a_ℓ is the only termination. By commutativity $\tau(q,\bar{a}a_\ell) = \tau(q,a_\ell \bar{a})$, and by an argument essentially the same as above, $\mu'=\mu''$, concluding part (b) of the lemma.

(c) (ii) If $a=b$ then $\alpha\cdot a_j b_\ell$ and $\alpha\cdot b_\ell a_j$ are both defined only if $\ell=j$, since the first outcome is uniquely determined by G_a. Thus we have $\alpha\cdot a_j a_j$ and obviously this equals $\alpha\cdot a_j a_j$.

(c) (i) If $a\neq b$, let

$$(c,q,\mu)\cdot a_j b_\ell = (c',\tau(q,a_j b_\ell),\mu') \text{ and}$$
$$(c,q,\mu)\cdot b_\ell a_j = (c'',\tau(q,b_\ell a_j),\mu'').$$

Using the one-one interpretation we see that $c'=c''$ if and only if $R(a)\cap R(b) = \phi$. $\tau(q,a_j b_\ell) = \tau(q,b_\ell a_j)$ by commutativity, and $\mu'=\mu''$ since in each case the only changes from μ are a simple removal from the $\mu(a)$ and $\mu(b)$ lists in both cases.

This completes the proof of Lemma 2.

Lemma 3: Let S be a persistent, commutative, lossless schema, I a one-one interpretation, and α_0 the initial instantaneous description. Let $\nu\in\Sigma^*$, and $\sigma,\pi\in\Sigma$ such that $\alpha_0\cdot\nu\sigma\pi = \alpha_0\cdot\nu\pi\sigma$. Then, for any $w\in\Sigma^*\cup\Sigma^\omega$:

(a) $\nu\sigma\pi w$ is an I-computation if and only if $\nu\pi\sigma w$ is;

(b) for any $i\in M$, $\Omega_i(\nu\sigma\rho w) = \Omega_i(\nu\pi\sigma w)$.

The proof of part (a) of this lemma follows from the $\cdot$ relation definition, persistence and commutativity. By checking the cases of Lemma 2 whenever $\alpha_0\cdot\nu\sigma\pi = \alpha_0\cdot\nu\pi\sigma$ part (b) follows.

Lemma 4: Let S be a persistent schema, I an interpretation and α_0

the initial instantaneous description. Let $u,v\epsilon\Sigma^*$, $w\epsilon\Sigma^*\cup\Sigma^\omega$ and $\sigma\epsilon\Sigma$.

(a) If $\alpha_0\cdot u\sigma$ is defined, $\sigma\notin v$ and $\alpha_0\cdot uv$ is defined, then $\alpha_0\cdot uv\sigma$ is defined.

(b) If $\alpha_0\cdot u\sigma$ is defined and uw is an I-computation then $\sigma\epsilon w$.

The proof of part (a) of this Lemma follows from persistence, and part (b) follows from persistence and the finite delay property.

We now are ready to prove the theorem. Suppose (A) holds, then we wish to prove that S is determinate. Assume S is not determinate. That is, that there exists an interpretation I such that x and y are I-computations and for some location $i\epsilon M$, $\Omega_i(x) \neq \Omega_i(y)$. We shall prove that for any $n\leq\ell(x)$ † there is an I-computation $z(n)$ such that:

(1) $z(n)$ has the same cell sequences as y. That is, $\Omega_i(z(n)) = \Omega_i(y)$ for all $i\epsilon M$.

(2) ${}_n(z(n)) = {}_nx$

Since this is true for all $n\leq\ell(x)$ we obtain ${}_n(\Omega_i(z(n))) = {}_n(\Omega_i(x)) = {}_n(\Omega_i(y))$ so for no n can $\Omega_i(x)$ differ from $\Omega_i(y)$. This provides a contradiction proving that

$$\forall i\epsilon M \quad \Omega_i(x) = \Omega_i(y)$$

so condition (A) implies determinacy.

The proof of properties (1) and (2) for $z(n)$ is done inductively on n.

Basis: Assume $z(0)=y$. Then (1) and (2) hold for $n=0$.

Inductive Assumption: Assume (1) and (2) hold for $n=k$ and $\ell(x)\geq k+1$.

† $\ell(x)$ is the length of x.

Then $\alpha_0 \cdot ({}_k x) x_{k+1}$ is defined so $\alpha_0 \cdot t x_{k+1}$ is defined, where $t = {}_k(z(k))$, and $\alpha_0 \cdot ({}_k x) x_{k+1} = \alpha_0 \cdot t x_{k+1}$ since by the inductive assumption (2) holds for n=k. By Lemma 4 $z(k) = t v x_{k+1} u$, $x_{k+1} \epsilon v$. That is, x_{k+1} appears somewhere in the sequence since $\alpha_0 \cdot t x_{k+1}$ is defined. If v is null then let $z(k+1) = z(k)$ and (1) and (2) hold. If v is not null then $v = w\pi$, $w \epsilon \Sigma^*$, $\pi \epsilon \Sigma$. Since $\alpha_0 \cdot t x_{k+1}$ is defined it follows from (a) of Lemma 4 that $\alpha_0 \cdot t w x_{k\ 1}$ and $\alpha_0 \cdot t w \pi x_{k+1}$ are defined. Also, since $\alpha_0 \cdot t w \pi$ is defined, $\alpha_0 \cdot t w x_{k+1} \pi$ is defined. But by assumption (A) holds, so

$$\alpha_0 \cdot t w \pi x_{k+1} = \alpha_0 \cdot t w x_{k+1} \pi .$$

Thus, by lemma 3, $t w x_{k+1} \pi u$ is an I-computation, and has the same cell contents sequences as $z(k)$. That is, for all $i \epsilon M$, $\Omega_i(z(k)) = \Omega_i(t w x_{k+1} \pi u)$. We have thus succeeded in moving x_{k+1} one place to the left in the sequence. By identical reasoning we can continue to "slide" x_{k+1} to the left until we obtain $t x_{k+1} v u$ which is an I-computation and has the same cell contents sequences as $z(k)$, and therefore the same as y also. We set $z(k+1) = t x_{k+1} v u$ and note that it satisfies (1) and (2). This completes the inductive step so we have that condition (A) implies determinacy. The "sliding argument" used here is a technique used in other schema proofs also, as well as in other Church-Rossen type theorems.

To complete the proof we must show that determinacy implies condition (A). Assume determinacy but that (A) does not hold. That is, that $\alpha_0 \cdot u \sigma \pi \neq \alpha_0 \cdot u \pi \sigma$. Then by the cases of Lemma 2 it can be seen that a difference in $\Omega_i(u \sigma \pi)$ and $\Omega_i(u \pi \sigma)$ must exist for some $i \epsilon M$. This contradicts determinacy, however, and completes the proof of the theorem.

This theorem shows how determinacy, a property on cell contents sequences, is equivalent to a type of commutativity of events in I-computations. That is, when a "racing" of several operation performances does not create a change in behavior.

An immediate corollary of this theorem and the preceeding lemmas is:

Corollary: Let S be a persistent, commutative, lossless schema. Then S is determinate if and only if, for each interpretation I with initial instantaneous description α_0:

(i) if $\alpha_0 \cdot u\bar{a}b_\ell$ and $\alpha_0 \cdot ub_\ell\bar{a}$ are defined, then $R(b) \cap D(a) = \phi$ or $\pi_{R(b)}(\alpha_0 \cdot ub_\ell) = \pi_{R(b)}(\alpha_0 \cdot u)$, and

(ii) if $\alpha_0 \cdot ua_j b_\ell$ and $\alpha_0 \cdot ub_\ell a_j$ are defined, and $a \neq b$, then $R(b) \cap R(a) = \phi$.

We say that S is repetition-free if whenever $v\bar{a}w\bar{a}x$ is an I-computation for some I, then w contains some termination symbol c_j such that $R(c) \cap D(a) \neq \phi$. This allows us to reduce determinacy to a non-memory conflict situation. For this we introduce a relation $\rho \subseteq A \times A$ defined as follows:

$$a\rho b \iff R(a) \neq \phi,\ R(b) \neq \phi \text{ and } [D(a) \cap R(b)] \cup [R(a) \cap D(b)] \cup [R(a) \cap R(b)] \neq \phi.$$

With these definitions we can state another corollary for determinacy.

Corollary: Let S be a schema which is repetition-free, lossless, persistent, commutative, and permutable. Then S is not determinate if and only if for some interpretation I, with initial instantaneous description α_0, there exists $w \in \Sigma^*$, $a \in A$ and $b \in A$ such that $a\rho b$ and $\alpha_0 \cdot w\bar{a}$ and $\alpha_0 \cdot w\bar{b}$ are both defined.

It is striking to note the similarity of the ρ relation and its relation to determinacy, and the Bernstein conditions on memory conflict which are sufficient to have two processes operate in parallel. They are very closely related. In essence, the ρ relation is the schema equivalent to the Bernstein conditions.

<u>Last Time</u>: Necessary and Sufficient Conditions for Determinacy in Schemata

<u>Today</u>: Decidability of Determinacy

Today we are aiming at proving the theorem:

<u>Theorem 1</u>: It is decidable whether a repetition-free, lossless, persistent, commutative, counter schema is determinate.

Our proof will be based on encoding the problem into vector addition systems and then appealing to the finite tree construction to provide decidability. First, we must define counter schema.

<u>Definition</u>: A schema is <u>repetition-free</u> if whenever an $\mathcal{I}$-computation contains two initiation symbols of the same operation, as in $v\bar{a}w\bar{a}x$ then w contains a termination symbol of some operation c for which $R(C) \cap D(a) \neq \phi$.

<u>Definition</u>: A <u>counter schema</u> $S = (M,A,\mathcal{T})$ has $\mathcal{T}$ defined by:

(1) a nonnegative integer k, the number of counters,

(2) a finite set Σ,

(3) a finite set S with distinguished element s_0,

(4) a vector $\pi \epsilon N^k$,

(5) a function v from Σ into N^k such that if $\sigma \epsilon \Sigma_t$ then $v(\sigma) \geq 0$.

(6) a partial function $\theta: S \times \Sigma \to S$ which is total on $S \times \Sigma_t$.

Here $\mathcal{T} = (Q, q_0, \Sigma, \tau)$ where:

$Q = S \times N^k$, $q_0 = (s_0, \pi)$, $\tau((s,x),\sigma)$ is defined if $\theta(s,\sigma)$ is defined and $x+\nu(\sigma) \geq 0$, and in that case $\tau((s,x),\sigma) = (\theta(s,\sigma), x+\nu(\sigma))$.

Thus a counter schema is a parallel program schema with a control specified in a particular way. The state part of the schema control is a pair, the first element being an element of a finite set S and the second element being a set of k counter values. Each initiation and termination causes a change of state in the S part and an incrementing or decrementing of counter values.

We now construct a vector addition system to simulate the counter schema. For a given counter schema $\mathcal{S}$ we construct a vector addition system $\mathcal{W}_{\mathcal{S}} = (d,W)$ as follows:

$\mathcal{W}_{\mathcal{S}}$ has $|S| + k + |A|$ coordinates. The $|S|$ coordinates represent the state behavior of S, the k coordinates directly represent the counter values, and the $|A|$ coordinates represent the μ list lengths for each $a \in A$. Thus, each coordinate represents a particular state, counter, or operation. We define d as follows:

$$d(s_0) = 1,$$
$$d(s) = 0 \text{ for } s \in S \text{ and } s \neq s_0,$$
$$d(i) = \pi_i,\ i=1,2,\ldots,k$$
$$d(a) = 0,\ a \in A.$$

The vectors in W are described by looking separately at the form of the vectors for each part $|S|$, k, and $|A|$.

We concentrate on a transition from a state (s_j, π) to a state

(s_i,π') under an event σ.

The $|S|$ part of a reachable point has a 1 in the coordinate representing the current state s_j and zeros elsewhere. Thus, for an element of W the $|S|$ part has a form

0....010...0-10...0

where the +1 (say in coordinate i) indicates the state s_i that is being entered, the -1 (say in coordinate j) indicates the state s_j that is being left under the transition $\theta(s_j,\sigma) = s_i$.

The k part of the reachable point contains the current counter values π and $\nu(\sigma)$ is entered in the appropriate coordinates of the k part of the element of W to show the change of counter values caused by event σ.

For the $|A|$ part of the element of W: if $\sigma = \bar{a}$ then +1 is entered in the coordinate for operation a; if $\sigma = a_k$ then -1 is entered into the coordinate for operation a.

A vector is placed into W for each pair (s,σ) $s \in S$ and $\sigma \in \Sigma$ for which $\theta(s,\sigma)$ is defined. Since both S and Σ are finite sets we see that W is also a finite set as required. Thus W_S is clearly a vector addition system.

To summarize, the a vector in $R(W_S)$ can be seen to represent:

(1) the current state s of the schema by the position of a 1 in the $|S|$ part of the vector;

(2) the current counter values by the k part coordinate values;

(3) for the coordinate representing $a \in A$ in the $|A|$ part of the number of performances of a that are currently in progress.

Now, starting from d in W_S and applying successive elements of W a path of reachable points is formed and this corresponds directly to a computation for S. The nonnegativity condition for vector addition systems insures first, that for the $|S|$ part only vectors can be added which correspond to the current S state, second, that for the k part the counter values will always remain nonnegative, and third, that for the $|A|$ part a termination will be allowed only if there is a currently outstanding performance of the operation to terminate.

We are now ready to show decidability of determinacy, we will do this informally.

<u>Theorem 2</u>: It is decidable whether a repetition-free, lossless, persistent, counter schema S is determinate.

<u>Proof</u>: Given S we can construct W_S and the tree $T(W_S)$. From the necessary and sufficient conditions for determinacy, in particularly Corollary 1, we see that S is not determinate if and only if there is a pair of operations a and b such that:

(1) $R(b) \cap D(a) \neq \phi$ and there exists a u such that $\alpha_0 \cdot u\bar{a}b_\ell$ and $\alpha_0 \cdot ub_\ell\bar{a}$ are both defined, or

(2) $a \neq b$, $R(b) \cap R(a) \neq \phi$ and there exists a u such that $\alpha_0 \cdot ua_jb_\ell$ and $\alpha_0 \cdot ub_\ell a_j$ are both defined.

Now, there is only a finite number of such conflicting pairs of operations and $T(W_S)$ can be inspected to see if for any such pair (a,b) μ lists for a and b are simultaneously greater than 1. The schema S will be determinate if and only if no such pair exists.

In [72] many other properties of schemata are shown to be decidable through this encoding to vector addition systems. For example, the following should be clear:

Theorem 3: It is decidable whether a given repetition-free counter schema is bounded or serial.

Today: Undecidability Results for Parallel Program Schemata

We now turn to proving undecidability of schema equivalence, determinacy and other properties (see [94]). We use a construction based on the Post correspondence problem, showing that if the schema property being considered were decidable then the Post correspondence problem would also be decidable. Of course, the Post correspondence problem is one of the basic undecidable problems so through such a construction it follows that the schema property must also be undecidable.

The form of the Post correspondence problem that we use is as follows:

Given two n-tuples of words

$$X = x_1, x_2, \ldots, x_n$$

$$Y = y_1, y_2, \ldots, y_n$$

over the alphabet $\{b_1 b_2\}$ it is undecidable whether there exists a sequence of indices $i_1, i_2, \ldots, i_p$ such that:

$$x_{i_1} x_{i_2} \ldots x_{i_p} = y_{i_1} y_{i_2} \ldots y_{i_p} .$$

As a simple example of such a Post correspondence problem let $n=4$ where:

$$X \begin{cases} x_1 = b_1 b_2 b_2 \\ x_2 = b_1 b_2 \\ x_3 = b_2 b_1 b_2 \\ x_4 = b_1 b_1 \end{cases} \qquad Y \begin{cases} y_1 = b_2 b_1 \\ y_2 = b_1 \\ y_3 = b_2 b_2 b_1 \\ y_4 = b_2 b_2 b_2 \end{cases}$$

Now consider the sequence of indices 2,3,4. Here $x_2x_3x_4 = b_1b_2b_2b_1b_2b_1b_1$ and $y_2y_3y_4 = b_1b_2b_2b_1b_2b_2$. Thus $x_2x_3x_4 \neq y_2y_3y_4$. The two words differ in their last two symbols. But is there any other sequence of indices that gives equal words? The decision problem is to give a uniform procedure for deciding this for any Post correspondence problem over $\{b_1,b_2\}$. To use the undecidability of this problem we start with "encoding" the problem into schema terms.

For a Post correspondence problem $P(X,Y)$ we construct schemata $S(X)$ and $S(Y)$ in which $M = \{1,2\}$, $A = \{a,b\}$, $D(a) = R(a) = 1$, $D(b) = R(b) = 2$, and $K(a) = K(b) = 3$.

Since neither operation affects the domain location of the other, the sequence of outcome of a and b depend only on the interpretation and not on how the performances of a and b are interspersed. Since $S(X)$ and $S(Y)$ are constructed in an identical manner we describe the construction for $S(X)$ only. We say that an interpretation I is <u>consistent</u> with $(X,i_1,i_2,\ldots,i_p)$ iff:

(1) if a could be executed repeatedly beginning in q_0 and starting with the initial memory contents as specified by I in location 1, then the sequence of outcome of a would have as a prefix:

$$a_1^{i_1-1}a_2a_1^{i_2-1}a_2\ldots a_1^{i_p-1}a_2a_3,$$

and,

(2) if b could be executed repeatedly starting in q_0 and with initial memory contents in location 2, then the sequence of outcomes would have a prefix:

$$x_{i_1}x_{i_2}\ldots x_{i_p}b_3.$$

Thus, $S(X)$ is constructed so that for a consistent interpretation the outcomes of a generate a sequence of indices $i_1, i_2, \ldots, i_p$ and the outcomes of b generate the word of X indicated by the $i_1, i_2, \ldots, i_p$ sequence. The actual I-computation starts in q_0 and ends in q_e if and only if it is a consistent interpretation, and it takes on the form

$$\underbrace{a_1^{i_1-1} a_2}_{i_1 \text{ a's}} x_{i_1} \underbrace{a_1^{i_2-1} a_2}_{i_2 \text{ a's}} x_{i_2} \ldots x_{i_p} a_3 b_3 .$$

The control of $S(X)$ which generates such a sequence is given by example for $X = x_1, x_2$ where $x_1 = b_2 b_1$ and $x_2 = b_2 b_2 b_1$. This is shown in Figure 1.

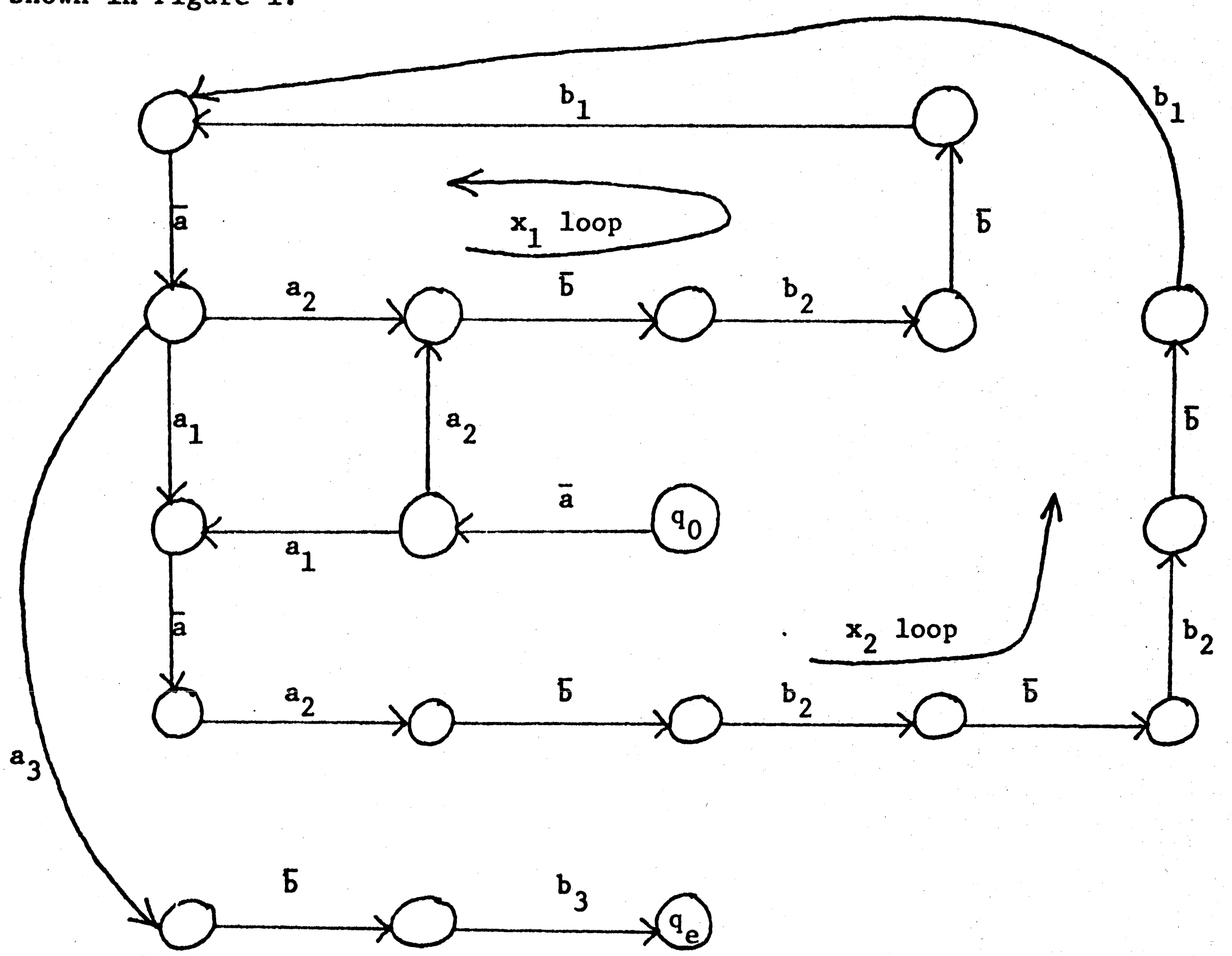


Figure 1: $S(X)$ construction

In general $S(X)$ is constructed to have a "loop" for each x_i. The outcomes of a choose the appropriate loop and then the loop is completed only if the b outcomes are consistent. The $a_3\bar{b}b_3$ exit indicates the end of the sequence. It should be clear that this construction generalizes to any X whatever. Figure 1 shows all transitions for the consistent part of the schema. If any termination which is not consistent occurs, it takes a transition to a "sink" region that has the form of Figure 2.

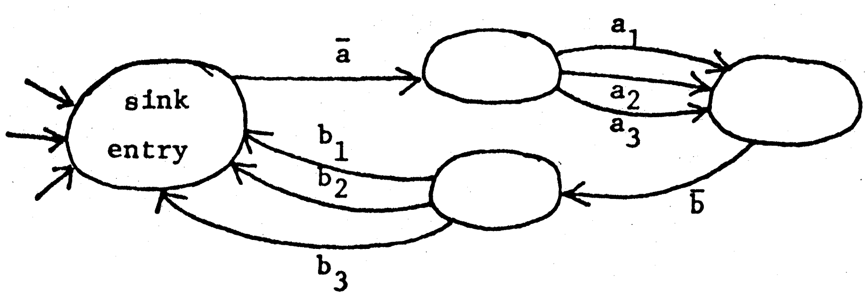


Figure 2: Sink construction for $S(X)$.

From this construction it should be evident that $S(X)$ is serial, and for any interpretation there is only one computation. If the interpretation is consistent that computation ends in q_e, otherwise it goes to the sink region and is infinite in length.

We are now ready to assume for $P(XY)$ we have constructed both $S(X)$ and $S(Y)$. From these we construct a schema $S(XY)$ as shown in Figure 3.

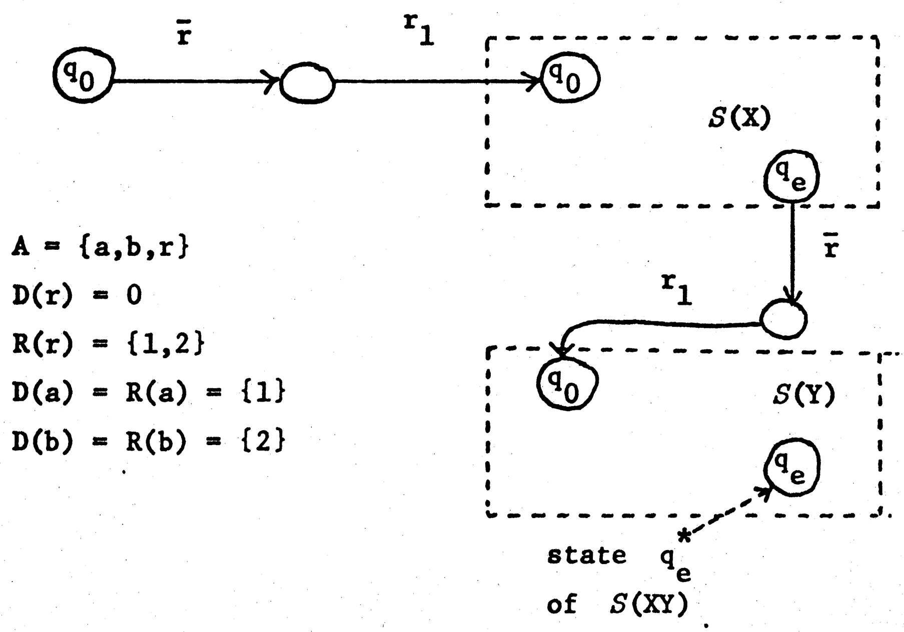


Figure 3: Structure of $\mathcal{S}$(XY)

Note that $\mathcal{S}$(XY), like $\mathcal{S}$(X), has exactly one computation for each interpretation (this property is called one-valued) so obviously it is serial. Also, it is a finite state schema. The computation is finite if and only if state q_e^* is reached and this happens if and only if there is a solution to the Post correspondence problem P(XY). Thus we have encoded the P(XY) into the schema $\mathcal{S}$(XY). We are now ready to use this construction to prove a number of undecidability properties of schemata.

Theorem 4: It is undecidable whether two serial finite-state determinate schemata are equivalent.

Proof: Construct $\mathcal{S}$(XY) and $\mathcal{S}'$(XY) where $\mathcal{S}'$(XY) is identical $\mathcal{S}$(XY) except it has a loop structure like Figure 2 at q_e^*. Now $\mathcal{S}$(X,Y) is not equivalent to $\mathcal{S}'$(X,Y) if and only if q_e^* is entered for some interpretation. That is, if and only if there is a solution to P(X,Y).

Thus, if we had an algorithm to decide equivalence we would have an algorithm to decide $P(X,Y)$. Thus equivalence is undecidable.

In [72] two undecidability theorems for equivalence are given using a somewhat different approach. These theorems show undecidability of equivalence for (1) persistent finite-state schemata, and (2) serial finite-state schemata, but in both cases the schemata are nondeterminate.

The most basic result obtained from the $\mathcal{S}(XY)$ construction is the following.

<u>Theorem 5</u>: It is undecidable for one-valued finite-state schemata whether a state is reachable.

A state q is called reachable if there exists an instantaneous description α in some I-computation where $\Pi_q(\alpha) = q$.

<u>Proof</u>: q_e^* of $\mathcal{S}(XY)$ is reached if and only if there is a solution to $P(XY)$.

We now use $\mathcal{S}(XY)$ and other variants of $\mathcal{S}(XY)$ which attach additional control structure leaving q_e^* to prove a series of other schema properties to be undecidable.

<u>Theorem 6</u>: It is undecidable whether a given finite state schema is computationally commutative, one-valued, or serial.

A schema S is called <u>computationally commutative</u> if whenever, for some interpretation I, $x\pi\sigma$ and $x\sigma\pi$ are prefixes of I-computations, then $\tau(q_0,x\pi\sigma) = \tau(q_0,x\sigma\pi)$. This is a slight weakening of the commutative property which can replace the commutative property in any hypothesis of schema theorems we have mentioned so far.

Proof: Construct $S^{ii}(X,Y)$ which is identical to $S(XY)$ except that it has the structure of Figure 4 leaving q_e^*.

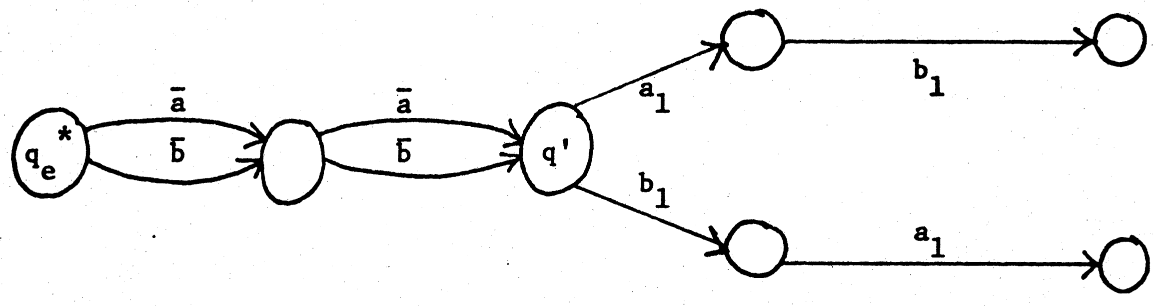


Figure 4: A noncomputationally-commutative schema structure.

Now, this added structure to $S^{ii}(XY)$ is finite-state so $S^{ii}(XY)$ is finite state. If x is a prefix of an I-computation that reaches q_e^* then $y = x\bar{a}\bar{b}$ is a prefix of this I-computation and ya_1b_1 and yb_1a_1 are both I-computations. But, $\tau(q_0,yb_1a_1) \neq \tau(q_0,ya_1b_1)$ so $S^{ii}(XY)$ is not computationally commutative if and only if q_e^* is reachable by an I-computation. Thus computational commutativity is undecidable. Similarly, since by this construction $S^{ii}(XY)$ is also not one-valued or serial if and only if q_e^* is reached, the theorem follows.

We should remark that by construction of $S^{ii}(XY)$ it is determinate, permutable, persistent, lossless, and since it is finite state, it is also a counter schema, and thus this undecidability theorem holds for schemata so restricted. Similar restrictions should be evident for the following theorems.

Theorem 7: It is undecidable whether a given finite-state schema is bounded.

Proof: Construct $S^{iii}(XY)$ from $S(XY)$ by adding to q_e^* an unbounded behavior. Such a construction is shown in Figure 5. The proof is then immediate.

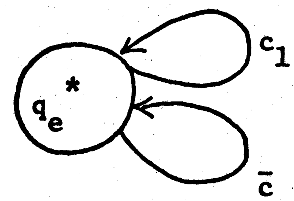


Figure 5: A simple unbounded behavior construction.

Theorem 8: It is undecidable for a finite-state schema whether a given operation $a \epsilon A$ is terminating.

Proof: Modify $S(XY)$ to form $S^{iv}(XY)$ as follows. Replace the sink construction of Figure 2 with the construction shown in Figure 6, and then let the sink construction of Figure 2 be attached to q_e^*.

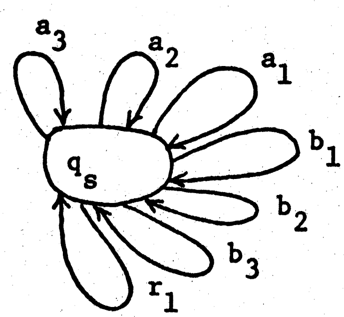


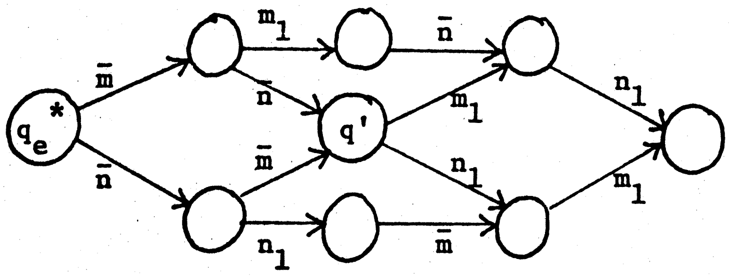
Figure 6: Terminating sink construction

Clearly, operations a and b terminate if and only if q_e^* is not reached.

From Theorems 5 and 8 we see that both the questions of the existence of finite and infinite computations are undecidable for these types of schemata.

Theorem 9: It is undecidable whether a finite-state schema is determinate.

Proof: Construct $S^{v}(XY)$ from $S(XY)$ by adding the nondeterminate part shown in Figure 7 to q_e^*. Then clearly undecidability results.



$D(m) = D(n) = \{0\}$

$R(m) = R(n) = \{1,2\}$

Figure 7: A nondeterminate construction.

It is now worth repeating Theorem 2.

Theorem 2: It is decidable whether a repetition-free, lossless, persistent, commutative, counter schema is determinate.

This can be contrasted with an immediate corollary of Theorem 9.

Corollary 9.1: It is undecidable whether a lossless, persistent, commutative, counter schema is determinate.

Here we see the crucial nature of the repetition-free property; it being a boundary between decidability and undecidability for this as well as other properties. In fact, in the $S(XY)$ and related constructions the only repetition possible is that of operation r and this can be repeated at most once, when following the transitions from $S(X)$ to $S(Y)$. So, in a sense, these constructions are "minimally" repetitive but nevertheless lead to undecidabilities.

By using the $S^{iii}(XY)$ construction or by having a construction for which from q_e^* we allow c to be performed exactly once, the following theorem is also immediate.

<u>Theorem 10</u>: It is undecidable for a given finite-state schema and operation c, whether any computation exists containing $\bar{c}$.

Today: Parallel Flowcharts and Flow Graph Schemata

Up to this point we have discussed the basic decidable and undecidable results of various types of parallel program schemata. The schemata have been precisely, but abstractly, defined, and it may be somewhat difficult to see how to use schemata to represent any more-or-less practical parallel processing problems. To clarify this we introduce several variants of schemata, and give some examples of their use on particular parallelism and synchronization problems.

Parallel Flowcharts

Flowcharts have traditionally been a convenient graphical tool in depicting the flow of control for sequential programs. We define a restricted class of counter schemata that can readily be graphically represented in a "parallel" flowchart form.

Definition 1: A parallel flowchart is a counter schema in which:

(1) $S = \{s_0\}$;

(2) $\theta(s_0,\sigma)$ is defined for all $\sigma\epsilon\Sigma$;

(3) If σ is a termination symbol, then each component of $\upsilon(\sigma)$ is either 0 or 1.

(4) If σ is an initiation symbol, then each component of $\upsilon(\sigma)$ is either 0 or -1.

(5) For initiation symbols σ and σ', $\sigma\neq\sigma'$, if $(\upsilon(\sigma))_i = -1$ then $(\upsilon(\sigma'))_i = 0$.

This restriction of counter schemata first, by reducing S to a single state, says that all the control is via the counters. Second, that terminations only increment counters (by 1) and that initiations only decrement counters (by 1). And finally, that if a counter is decremented by the initiation of some operation a, then it is not decremented by any other operation. With these comments the following theorem should be clear.

Theorem 1: Every parallel flowchart is persistent, commutative and permutable.

We have previously omitted providing a formal definition for permutable. We give it now.

Definition 2: A schema is permutable if, whenever σ and π are initiation symbols and $\tau(q,\sigma\pi)$ is defined, then $\tau(q,\pi)$ is also defined.

Proof of Theorem 1: Commutativity of parallel flowcharts follows directly from the commutativity of vector addition, since in parallel flowcharts states are counter value vectors and transitions under events amount to adding a "change" vector to the state vector. Persistence and permutability follow directly from the fact that initiations of different operations do not decrement the same counter.

From this theorem we see that the restrictions on the control structures of schemata that are required for the various decidability theorems, are automatically satisfied for all parallel flowcharts.

For any parallel flowchart S we represent S by a graph $G(S)$ as follows:

(1) Each operation in $A = \{a,b,c,\ldots\}$ is represented by a node labelled $0_a, 0_b, 0_c, \ldots$.

(2) Each counter is represented by a node, and these are labelled $d_1, d_2, \ldots d_k$. Where d_i represents counter i.

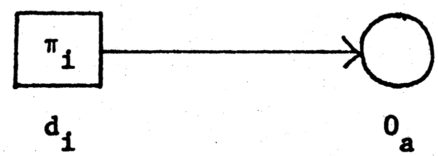
(3) The initial value π_i is added as a label of d_i, $i=1,2,\ldots,k$.

(4) For each $a \epsilon A$, if $(\nu(\bar{a}))_i = -1$ then an edge is directed from d_i to 0_a.

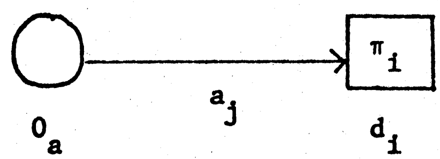
(5) For each $a_j \epsilon \Sigma_t$ if $(\nu(a_j))_i = 1$ then an edge is directed from 0_a to d_i, and the edge is labelled a_j.

Using circles to represent operation nodes and squares to represent counter nodes we obtain the following graphical form for (4) and (5) respectively.

$(\nu(\bar{a}))_i = -1$



$(\nu(a_j))_i = 1$



As an example of a parallel flowchart so depicted we can represent a schema control (similar to that shown in Figure 1 of Lecture 14) with three operations a, b, and c as shown here in Figure 1.

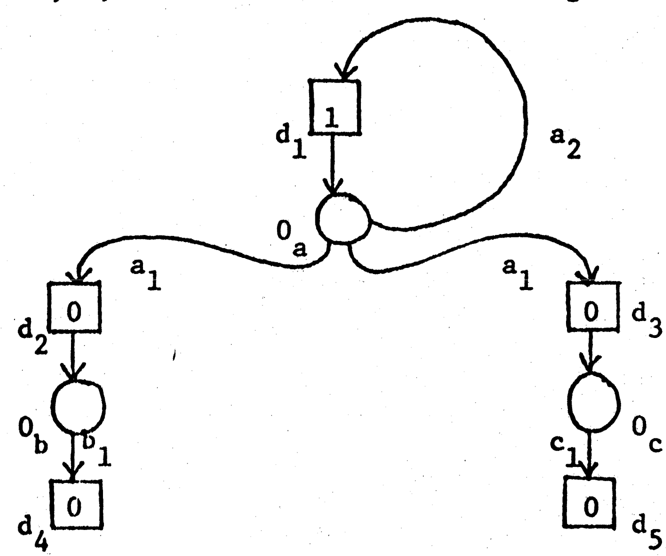


Figure 1: A graph for a simple parallel flowchart.

In this flowchart operation a is the only one that can initially initiate. Performances of a repeat with outcome a_2 until an outcome a_1 is obtained. The a_1 outcome increments counters 1 and 2, and this represents a FORK upon the a_1 termination for operations b and c to initiate.

A somewhat more concrete example, with an interpretation given to each operation, is shown in Figure 2.

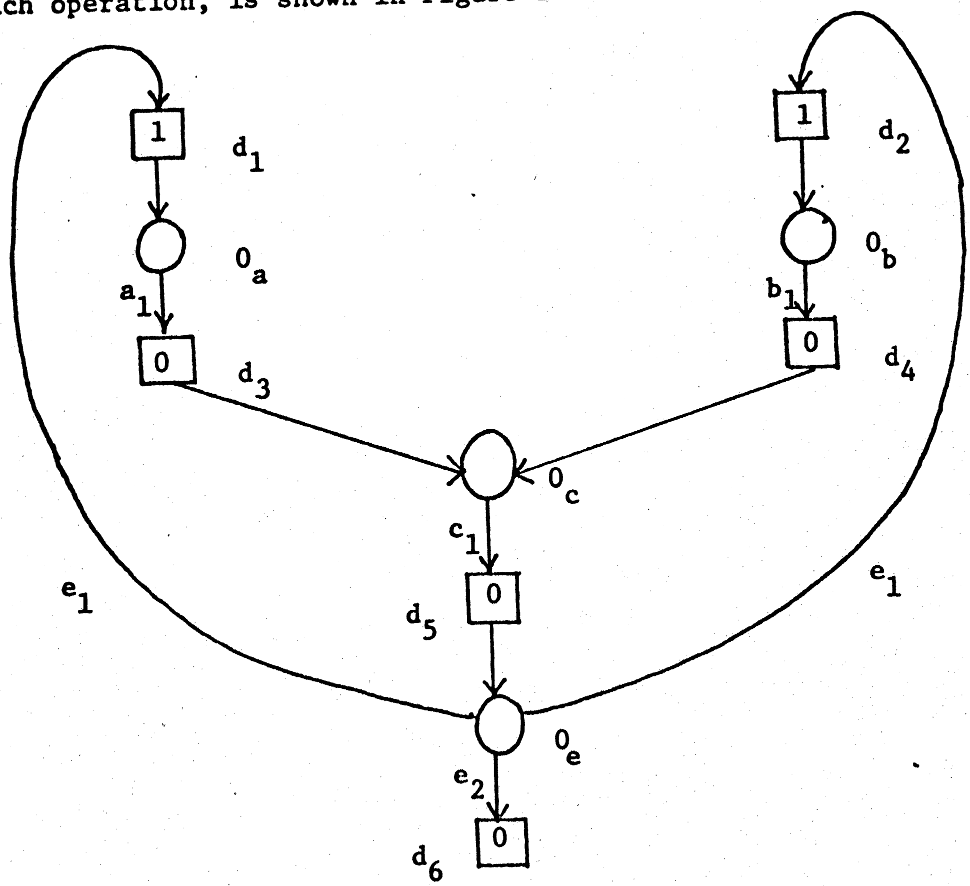


Figure 2: Five-point relaxation parallel flowchart.

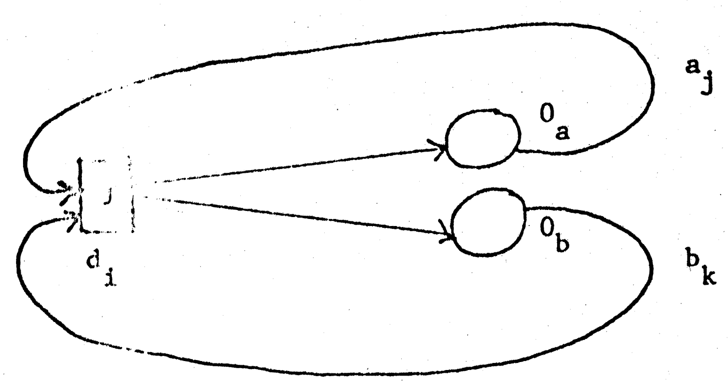
Here we are modelling the standard five point relaxation $P \leftarrow P+\frac{1}{4}(N+S+E+W)$. Operation a performs $T_1 \leftarrow N+S$, operation b performs $T_2 \leftarrow E+W$, operation c performs $T_3 \leftarrow T_1+T_2$, operation e performs $P \leftarrow P+\frac{1}{4}T_3$. Also, e has two outcomes. Outcome e_1 indicates repeating (until convergence) and outcome e_2 indicates stopping when a convergence criterion is satisfied. The domain and range locations could be assigned as:

$D(a) = \{1,2\}$	$R(a) = \{3\}$
$D(b) = \{4,5\}$	$R(b) = \{6\}$
$D(c) = \{3,6\}$	$R(c) = \{7\}$
$D(e) = \{7,8\}$	$R(e) = \{8\}$.

In this example we see a FORK construction for outcome e_1 to allow the starting of both a and b. Operation c initiation is the implementation of a JOIN, where both counters d_3 and d_4 must become 1 before c initiates. Outcome e_2 and counter d_6 represent a QUIT operation since d_6 feeds no other operation.

In [72] a more complicated parallel flowchart example is given. Some of the sequencing problems which have been discussed via semaphore implementations can also be represented by parallel flowcharts simply by letting a semaphore be represented by a counter, and the semaphore value be the counter value. The constraint that a counter is not decremented by more than one operation initiation, however, means that P operations for any semaphore can only be associated with the starting of a single operation (or process, in semaphore terms).

For "mutual exclusion" of operations a and b, for example, one would like a structure of the form:



but this sharing of a counter for initiation of more than one operation was specifically disallowed in parallel flowcharts so that persistence would hold.

Another type of difficulty is depicted by the example in Figure 3. (Note we have simplified the labels here in an obvious fashion.)

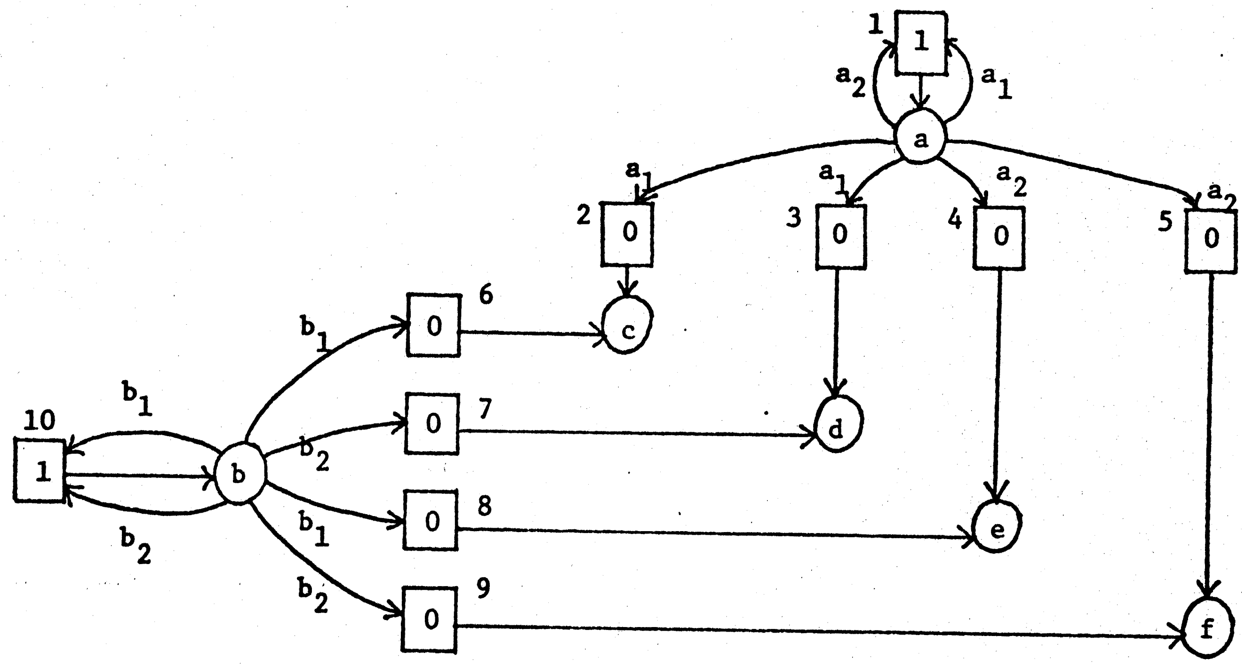


Figure 3: Another parallel flowchart.

The idea here is that operations a and b are to be performed in parallel and repeatedly. The (a_1,b_1) outcome pair is to select operation c to be performed, the (a_1,b_2) outcome is to select the operation d, etc. Here, however, if a and b have first outcomes a_1, b_1 and second outcomes a_2, b_2, then in addition to c and f being selected d and e may also initiate due to the "spurious" ones in counters 3, 4, 7 and 8. The sharing of counters by initiations again would lead to a possible solution as shown in Figure 4.

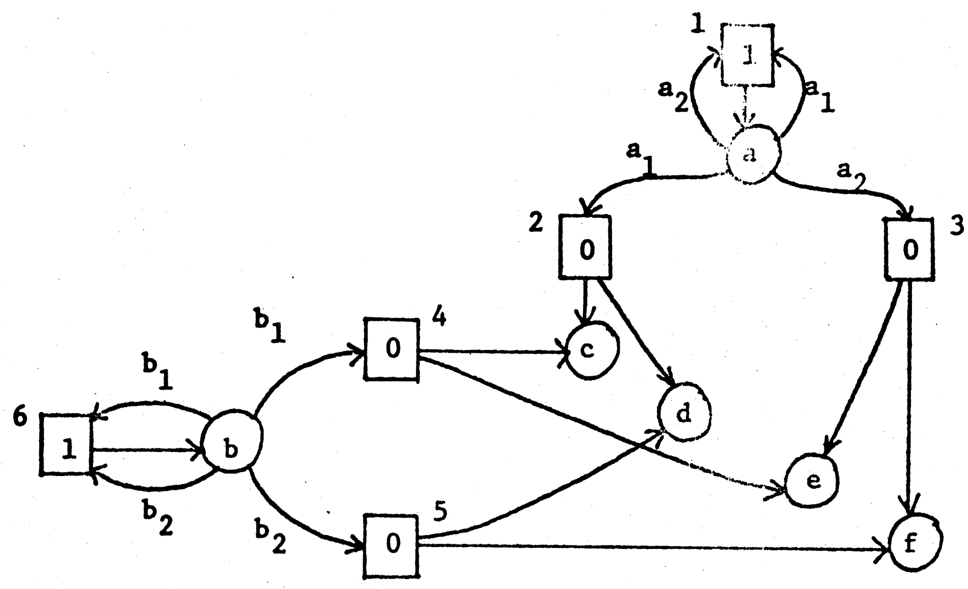


Figure 4: Counter "sharing" solution.

Rather than modifying parallel flowcharts in this form we discuss the flow graph schemata model of D. R. Slutz [135, 136] which handles this problem in a somewhat different fashion.

Flow Graph Schemata [135, 136]

Definition 3: A flow graph schema S is a schema $S = (M,A,T)$, where:

(1) M is a finite set;

(2) A is a finite set of operations, where for each $a \in A$, D(a) and R(a) are the domain and range locations for a, and K(a) is the number of outcomes for a;

(3) $T = (Q,q_0,\Sigma,\tau)$, the control, is specified over vectors of p coordinates by a function $V: \Sigma \to (\{-1\} \cup N)^p$ as follows:

(i) $Q = N^p$ is the set of control states.

(ii) $q_0 \in Q$ is the initial control state.

For each $a \in A$ there is a coordinate j_a such that $(q_0)_{j_a} = 0$.

(iii) Σ is the set of initiation and termination symbols.

(iv) The transition function τ is a partial function $\tau: Q \times \Sigma \to Q$. $\tau(q,\sigma)$ is defined if $q+V(\sigma) \geq 0$ and in this case $\tau(q,\sigma) = q+V(\sigma)$. The function V is constrained as follows for all $a \in A$:

(a) $(V(a_i))_k = -1$ implies that $(V(a_j))_k = -1$ for $i,j=1,2,\ldots,K(a)$.

(b) for all $b \in \Sigma$ where $b \neq \bar{a}, a_1, \ldots a_{K(a)}$ $(V(\bar{a}))_{j_a} = 1$, $(V(a_1))_{j_a} = -1$ and $(V(b))_{j_a} = 0$.

Flow graph schemata are a generalization of parallel flowcharts. The control is represented via p counters. Part (3)(ii) and (3)(iv)(b) set up a separate counter for each operation $a \in A$ that keeps a record of the number of performances of a currently in progress. Counters can only be decremented by 1 through the function V but can be incremented by more than one. Another generalization over parallel flowcharts is that the same counter can be decremented by several initiations. This type of counter sharing was prohibited in parallel flowcharts and was one of the main features of parallel flowcharts that limited the type of permissable parallel control.

A graphical representation for flow graph schemata, somewhat different than the parallel flowchart representation, will now be given. Given a flow graph schema F we define its graph $G(F)$ as follows: $G(F)$ has two parts, a data flow graph and a control graph. In the data flow graph each $m \in M$ is represented by a shaded rectangular node and each operation is represented by a circular node. For $a \in A$ and $m \in M$ an edge is directed from the a node to the m node iff $m \in R(a)$. Similarly an edge is directed from an m node to an a node iff $m \in D(a)$. This completes the construction of the data flow graph. It shows what data locations are affected by the various operation performances. Although not done previously, a data flow graph could be constructed for any parallel program schema. The control graph also contains two types of nodes: a rectangular node for each counter, labelled with the initial value of the counter, and for each operation $a \epsilon A$ two circular nodes, an initiation node labelled $\bar{a}$ and a termination node labelled a_t. Let $\sigma \epsilon \Sigma$ and $n(\sigma)$ be the node for σ; if $\sigma = \bar{a}$ then $n(\sigma)$ is the node labelled $\bar{a}$, if $\sigma = a_j$, $j=1,2,\ldots,K(a)$, then $n(\sigma)$ is the node labelled a_t. Edges in the control graph are constructed by inspecting $V(\sigma)$ for each $\sigma \epsilon \Sigma$. For all $\sigma \epsilon \Sigma$, and $i=1,2,\ldots,p$, if $(V(\sigma))_i = -1$ then an edge is directed from counter i to $n(\sigma)$. Counter i is then called an <u>input counter</u> to $n(\sigma)$. If $(V(\sigma))_i = k>0$ then an edge is directed from $n(\sigma)$ to counter i and label k is attached to the edge. If in addition σ is a termination symbol then σ is also attached to the edge as a label. Counter i is called an <u>output counter</u> of $n(\sigma)$ if there is an edge from $n(\sigma)$ to i.*

We now present a flow graph schema example for a producer-consumer problem in which the producer (called operation p) produces items placing them in a buffer (called operation b) and a consumer (called operation c) that takes items from the buffer. The graph for this schema is shown in Figure 5.

---

* Slutz constructs a slightly different control graph in which input counters are connected in a chain to the event node. We omit this simplification here.

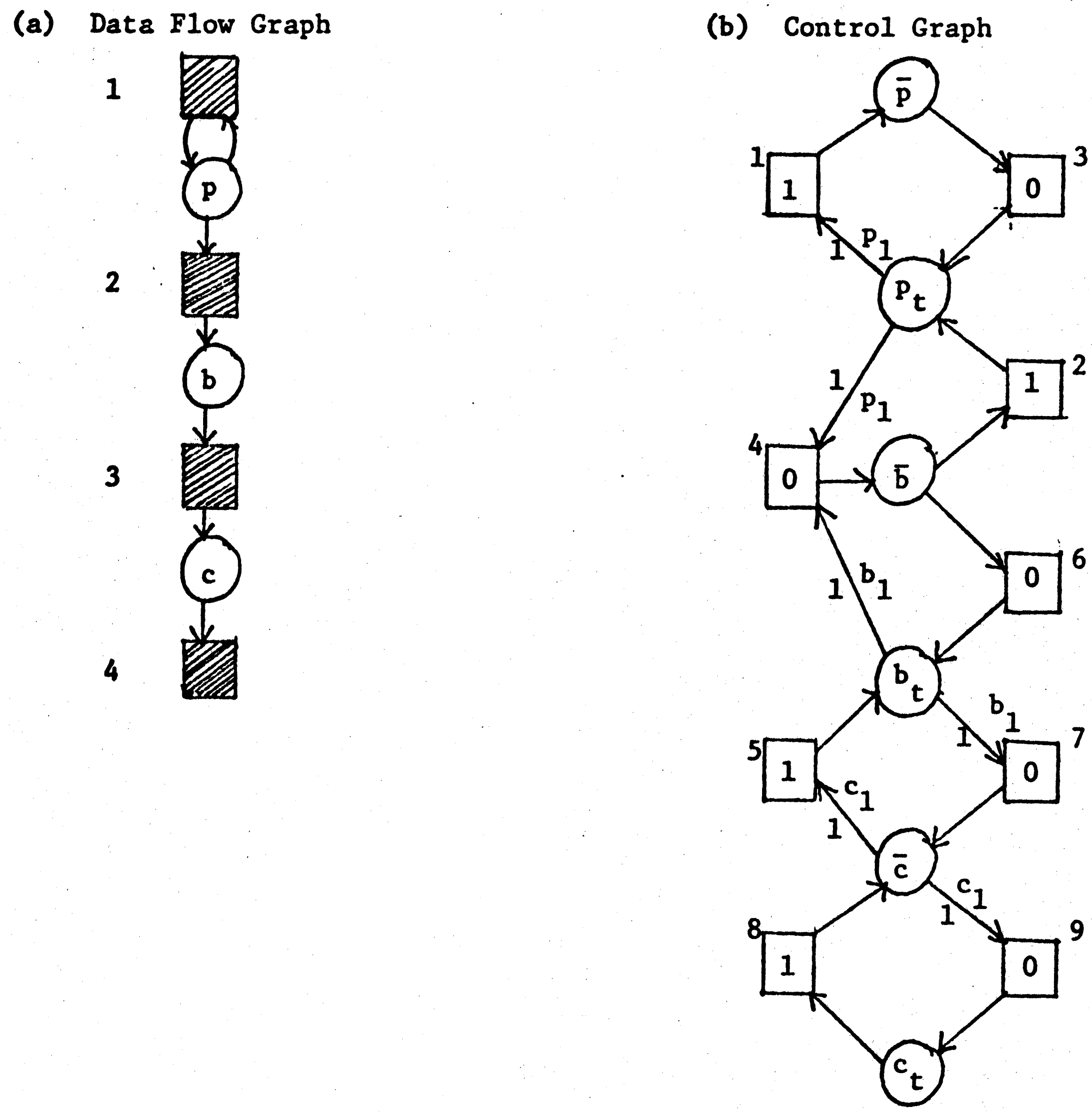


Figure 5: A Flow Graph Schema

In this example it is clear that the only event that can occur initially is $\bar{p}$ since this is the only event for which all it's input counters are positive. After $\bar{p}$ occurs counter 3 becomes 1 so that p can then terminate. When p_1 occurs both $\bar{p}$ and $\bar{b}$ can occur. Note that here the second $\bar{p}$ can occur before the first $\bar{b}$ ($\bar{b}$ corresponds to the buffer reading the output of the producer) but since counter 2 is zero event p_t cannot occur a second time until $\bar{b}$ occurs. Thus the second value of the producer cannot be written into location 2 until the buffer has read the first value. As the computation proceeds the value of counter 6 represents the number of items in the buffer. Also, since counter 6 is an input counter to $\bar{c}$ the consumer cannot initiate unless an item is in the buffer.

Figure 6 shows a modification of the control graph, by adding counters 10 and 11. Counter 10 limits the size of the buffer to n or less items, and counter 11 provides for mutually exclusive manipulation of the buffer by the producer or consumer. (We omit labels on edges since we are incrementing only by 1 and each operation has only a single outcome in this example.)

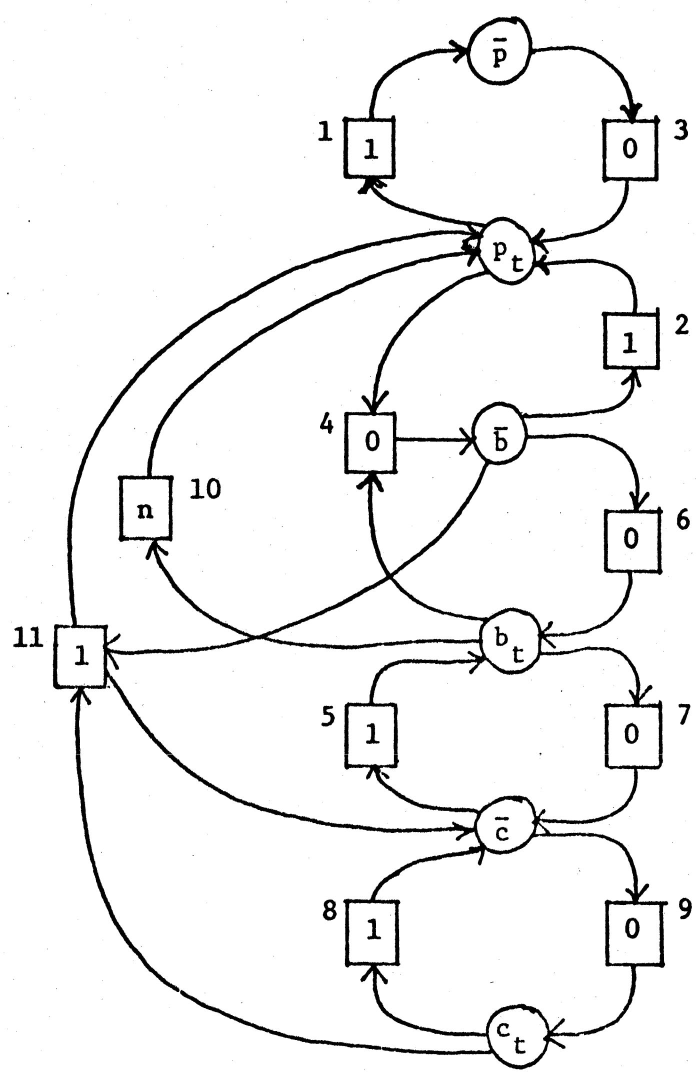


Figure 6: Modified example.

These two schemata are determinate and equivalent.

Using instantaneous description and $\cdot$ operation definitions as in parallel program schemata, a sequence of events is defined to be a computation as follows.

Definition 4: For a flow graph schema and an interpretation, a finite or infinite string over Σ is called a computation if:

(1) for all y such that $x=yz$, $\alpha_0 \cdot y$ is defined.

(2) if x is finite $\alpha_0 \cdot x\sigma$ is undefined for all $\sigma \epsilon \Sigma$.

(3) if $x=yz_1z_2\ldots$ there does not exist an infinite set of instantaneous descriptions $H \subseteq \{\alpha_i | \alpha_i = \alpha_0 \cdot yz_1z_2\ldots z_i\}$ such that for all $\alpha_i \epsilon H$ either of the following holds:

(a) there exists a $\sigma \epsilon \Sigma$ such that $\alpha_i \cdot \sigma$ is defined, unless for some j, $z_j = \sigma$ (finite delay property).

(b) there exists a $q \epsilon Q$ and $\sigma \epsilon \Sigma$ such that $\alpha_i = (c_i, q, \mu_i)$ and $\alpha_i \cdot \sigma$ is defined, unless for some j, $z_j = \sigma$ (finite response property).

Condition (a) is similar to the finite delay property for parallel program schemata, but here is required only for an infinite sequence of α_i following $\alpha_o \cdot y$ not for all instantaneous descriptions following $\alpha_0 \cdot y$. Condition (b) says that if an event can occur for some state, and this state recurs infinitely often then the event must occur after some finite number of occurrences. For example, consider the control graph shown in Figure 7.

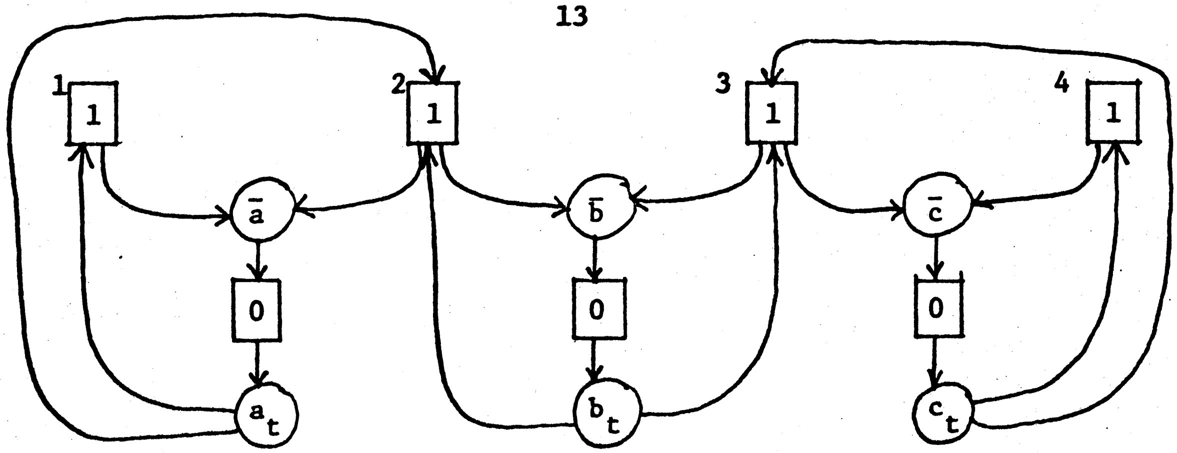


Figure 7: Example to demonstrate finite-response property.

Here $x = \bar{a}(\bar{c}a_1\bar{a}c_1)^{\infty}$ satisfies the control constraints for a sequence of events but is not a computation since operation b violates the finite response property. Event $\bar{b}$ is allowed to occur in the initial state, but because of the particular sequencing is never allowed to occur thereafter. Finite response disallows such anamolies in sequencing.

Although we will not go into theoretical results here, flow graph schemata have been shown to have many decidable properties.

Today: Schema Composition and Renaming

In this lecture we briefly describe the work of references [19, 89, 90, 97]. The idea of schema composition goes back to the original notion that a parallel program schema is a model for a parallel program. If one wishes to develop a large program, then it is often convenient to first develop identifiable subtasks as subprograms and then later put these subprograms together in a suitable fashion to make the complete program. Similarly, to model the program it might be desirable to first model certain subprograms by schemata then later "compose" these schemata together suitably to represent the complete program. This could be viewed as a simple structured approach to modelling the program. The work on schema composition is aimed at defining some basic types of interconnections for schemata and then proving theorems that say, essentially, that the proper behavior of the subparts is carried over to the complete schema when composition is done in the prescribed way.

In [19, 97] special types of schemata, called finishing schemata and exit schemata, are defined which are suitable for defining composition. We omit the details of these schema definitions here, but just point out some of their essential features. In both cases these schemata are assumed to have a finite subset of "begin" states and a finite subset of "end" states. The begin and end states are useful in composition, as we shall see. For end states, one assumes that no transitions are defined out of end states, and that whenever an end state is reached a finite computation for the schema has been completed. Additional constraints upon finishing schemata result in the fact that no more than one performance of any operation can be going on simultaneously. This restricts μ lists to be either of length zero or one and simplifies the analysis. From this point on, for composition, when we say schemata we mean finishing schemata.

We will now define four types of composition and state some results about the composed forms. We use terminology that subscripts symbols with the schema symbol to avoid confusion. Thus, for example, if we have two schemata $\mathcal{A}$ and $\mathcal{B}$ we let $\mathcal{A} = (M_{\mathcal{A}}, A_{\mathcal{A}}, \mathcal{T}_{\mathcal{A}})$ and $\mathcal{B} = (M_{\mathcal{B}}, A_{\mathcal{B}}, \mathcal{T}_{\mathcal{B}})$, etc.

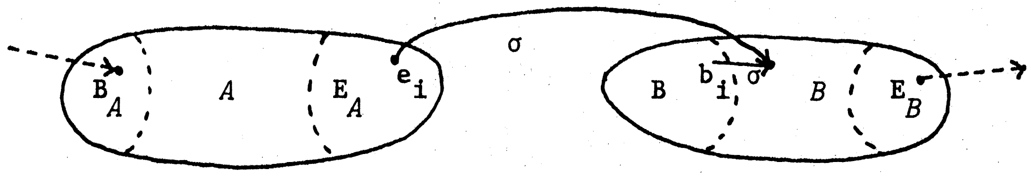
The <u>serial composition</u> of two schemata $\mathcal{A}$ and $\mathcal{B}$ is designated by $O(\mathcal{A},\mathcal{B})$. For this composition we require $Q_{\mathcal{A}} \cap Q_{\mathcal{B}} = \phi$ and that the number of end states of $\mathcal{A}$ equal the number of begin states of $\mathcal{B}$. Essentially $O(\mathcal{A},\mathcal{B})$ is a serial linking from end states of $\mathcal{A}$ to destinations of begin states of $\mathcal{B}$. Let $\mathcal{A} = (M_{\mathcal{A}}, A_{\mathcal{A}}, \mathcal{T}_{\mathcal{A}})$ and $\mathcal{B} = (M_{\mathcal{B}}, A_{\mathcal{B}}, \mathcal{T}_{\mathcal{B}})$ with the end states of $\mathcal{A}$ being $E_{\mathcal{A}} = \{e_1, e_2, \ldots, e_n\}$ and the begin states of $\mathcal{B}$ being $B_{\mathcal{B}} = \{b_1, b_2, \ldots, b_n\}$. Then we define $O(\mathcal{A},\mathcal{B}) = (M, A, \mathcal{T})$ where:

$$M = M_{\mathcal{A}} \cup M_{\mathcal{B}}$$
$$A = A_{\mathcal{A}} \cup A_{\mathcal{B}},$$
$$\Sigma = \Sigma_{\mathcal{A}} \cup \Sigma_{\mathcal{B}}$$
$$Q = Q_{\mathcal{A}} \cup Q_{\mathcal{B}}$$
$$E = E_{\mathcal{B}}$$
$$B = B_{\mathcal{A}}.$$

The transition function τ is defined by:

$$\tau(q,\sigma) = \begin{cases} \tau_{\mathcal{A}}(q,\sigma) & \text{if } \tau_{\mathcal{A}}(q,\sigma) \text{ is defined.} \\ \tau_{\mathcal{B}}(q,\sigma) & \text{if } \tau_{\mathcal{B}}(q,\sigma) \text{ is defined.} \\ \tau_{\mathcal{B}}(b_i,\sigma) & \text{if } \tau_{\mathcal{B}}(b_i,\sigma) \text{ is defined and } q = e_i. \end{cases}$$

Pictorially the serial composition can be shown as follows:



Intuitively this composition makes end state e_i of A act like begin state b_i of B for a computation of A ending in e_i, and then a computation of B, starting in b_i could follow.

The <u>concurrent composition</u> of A and B, denoted by $X(A,B)$ is defined when $Q_A \cap Q_B = \phi$ and $A_A \cap A_B = \phi$, as $X(A,B) = (M,A,T)$ where:

$$M = M_A \cup M_B$$

$$A = A_A \cup A_B$$

$$\Sigma = \Sigma_A \cup \Sigma_B$$

states are pairs, $Q = Q_A \times Q_B$

$$B = B_A \times B_B$$

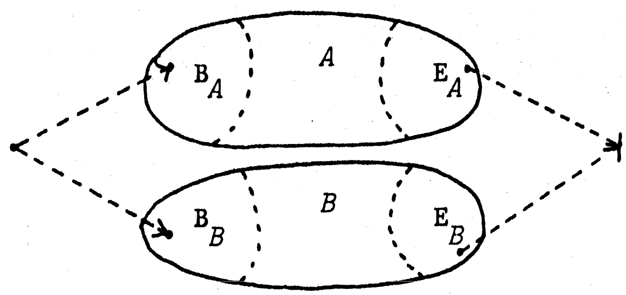
$$E = E_A \times E_B$$

The transition function is defined only in the following cases (let $q_{ij} = (q_i)_A \times (q_j)_B$)

If $\sigma \in \Sigma_A$ then $\tau(q_{ij},\sigma) = (\tau_A(q_i,\sigma),q_j)$

If $\sigma \in \Sigma_B$ then $\tau(q_{ij},\sigma) = (q_i,\tau_B(q_j,\sigma))$

The concurrent composition of two schemata allows parallel operation of the two schemata in a simple FORK-JOIN manner. Pictorially it can be viewed as:



The first type of result needed, is that under these compositions the type of object we get is still within the class of objects we wish to study. This is so.

<u>Theorem</u>: If A and B are schemata, then $O(A,B)$ and $X(A,B)$ are

schemata.

This is a basic closure result. The next types of results deal with the form of I-computations for the composed schemata. Without going into details, interpretations for the composed form are formed from "compatible" interpretations from the constituent schemata. Compatible here means that if the two interpretations are defining something for the same element, then the definitions are the same.

Theorem: Let A and B be schemata with $O(A,B)$ defined. Any computation z of $O(A,B)$ is of one of the following two forms:

(1) $z = x$, where x is a nonterminating computation of A.

(2) $z = xy$, where x is a terminating computation of A and y is a computation of B.

To state a similar result for concurrent composition we first need to define a "memory conflict" relation between A and B. Let $D_A = \bigcup_{a \in A_A} D(a)$ and $R_A = \bigcup_{a \in A_A} R(a)$, and let D_B and R_B be similarly defined. Then we say

$$A \rho B \Longleftrightarrow [D_A \cap R_B] \cup [D_B \cap R_A] \cup [R_A \cap R_B] \neq \phi.$$

Theorem: If A and B are schemata such that $X(A,B)$ is defined and $A \not\rho B$, then any computation of $X(A,B)$ is a shuffle of a computation of A and a computation of B.

The "shuffle" is essentially combining two strings into a single string, with the order of each original string maintained in the combined string, but having no other restrictions on the number of contiguous symbols (other than it being finite) taken from one string, before one or more symbols are taken from the other string.

Another form of composition consists of connecting an end state e of a schema to a begin state b to form a loop. We define such a composition, which we call an <u>iterate</u>, and designate this for a schema $\mathcal{A} = (M_{\mathcal{A}}, A_{\mathcal{A}}, \mathcal{T}_{\mathcal{A}})$ as $+(\mathcal{A},e,b) = (M,A,\mathcal{T})$ where $M = M_{\mathcal{A}}$, $A = A_{\mathcal{A}}$, $\Sigma = \Sigma_{\mathcal{A}}$, $Q = Q_{\mathcal{A}}$, and $B = B_{\mathcal{A}}$. $E = E_{\mathcal{A}} - \{e\}$ and τ for $+(\mathcal{A},e,b)$ is defined as follows:

$$\tau(q,\sigma) = \begin{cases} \tau_{\mathcal{A}}(b,\sigma) & \text{if } q = e \text{ and } \tau_{\mathcal{A}}(b,\sigma) \text{ is defined} \\ \tau_{\mathcal{A}}(q,\sigma) & \text{whenever } \tau_{\mathcal{A}}(q,\sigma) \text{ is defined.} \end{cases}$$

Clearly $+(\mathcal{A},e,b)$ is a schema, so we also get the desired closure result for this type of composition. The computations of $+(\mathcal{A},e,b)$ are characterized by the next theorem.

<u>Theorem</u>: Let $\mathcal{A}$ be a schema. If z is a computation for $+(\mathcal{A},e,b)$ then

(1) $z = x^1x^2\ldots$, where x^i are computations for $\mathcal{A}$ ending in state e, or

(2) $z = x^1x^2\ldots x^ky$ where $k \geq 0$ and for all $i = 1,2,\ldots k$ x^i are computations for $\mathcal{A}$ ending in e and y is a computation for $\mathcal{A}$ not ending in e.

The final type of composition we consider is more complex. We give only an intuitive idea of the form of this composition ([97] contains details). The idea here is that we would like to replace an operation as defined in one schema by a more detailed description (i.e., a schema) of the operation. This type of composition we call <u>insertion</u>. It allows one to hierarchically define a program. Naturally, a number of consistency requirements must be satisfied, such as the number of outcomes of the operation matching, in some sense, the number of end states of the schema replacing the operation. In [97] closure and computation characterization theorems are given for insertion.

Another class of theorems is given in [19, 97] for composition. These theorems show under what conditions determinacy carries over from the constituent schemata to the composed schemata. Briefly, this usually involves some constraints upon the operations and their effect on memory locations.

We now discuss "renaming" in schemata. For our schemata which we have defined and studied so far we have always specified the memory locations D(a) and R(a) that an operation a effects when it is performed. However, it may be advantageous to respecify the D(a) or R(a) locations of some operations. For example, some operation might be storing a "temporary result" in some location which is later used as an operand for another operation. Because this location is used during this period, however, it may restrict the use of this location by other operations, and only due to this memory conflict it may mean that these other operations must wait until the first pair of operations have finished using the location in question. This situation illustrates that by a relocation of memory, or a "renaming," we could attain more parallelism. Secondly, a renaming might decrease the number of memory locations needed to perform a computation, thus providing an economy of memory usage. This then is the subject of "renaming" in schemata. How can the assignment of memory locations be changed in a consistent and advantageous way? We will illustrate this notion of renaming (as done in [89, 90]) only by two simple examples. The first example considers only a simple sequence of function evaluations -- or a computation. The sequence is:

$$(f(2,3) \rightarrow 0),\ (g(3) \rightarrow 1),\ (h(0,1) \rightarrow 0,3),\ (m(3) \rightarrow 3)$$

That is, we start by performing a function evaluation f which uses locations 2 and 3 for operands and places a result in location 0, then follow this by a g using 3 and putting its result in location 1, etc.

Here we note that the result of the f calculation (in location 0) is used by h. That is, location 0 is busy for this usage over the segment indicated below:

$$(f(2,3) \rightarrow \underbrace{0),\ (g(3) \rightarrow 1),\ (h(0,1) \rightarrow}_{\text{0-busy}} 0, \underbrace{3),\ (m(3) \rightarrow}_{\text{3-busy}} 3)$$

Similarly 3 is in use for a certain purpose where shown.

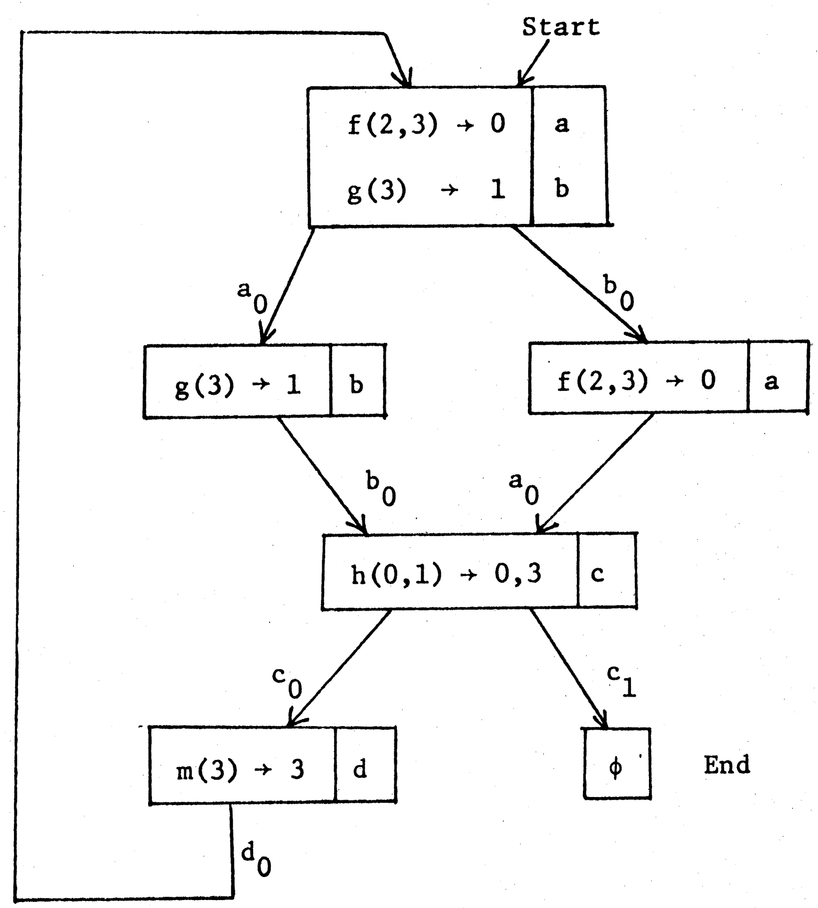
Now this use of location 0 could be moved to a different location. For example, no change in the final results in locations 0, 1, 2 or 3 would result if we changed this use to location 4 rather than 0, i.e. giving

$$(f(2,3) \rightarrow 4),\ (g(3) \rightarrow 1),\ (h(4,1) \rightarrow 0,3),\ (m(3) \rightarrow 3).$$

This renaming works but seems to be of no great use. However, if location 2 were of no interest beyond the use in the first f calculation we could rename the use of 0 to a use of 2 giving:

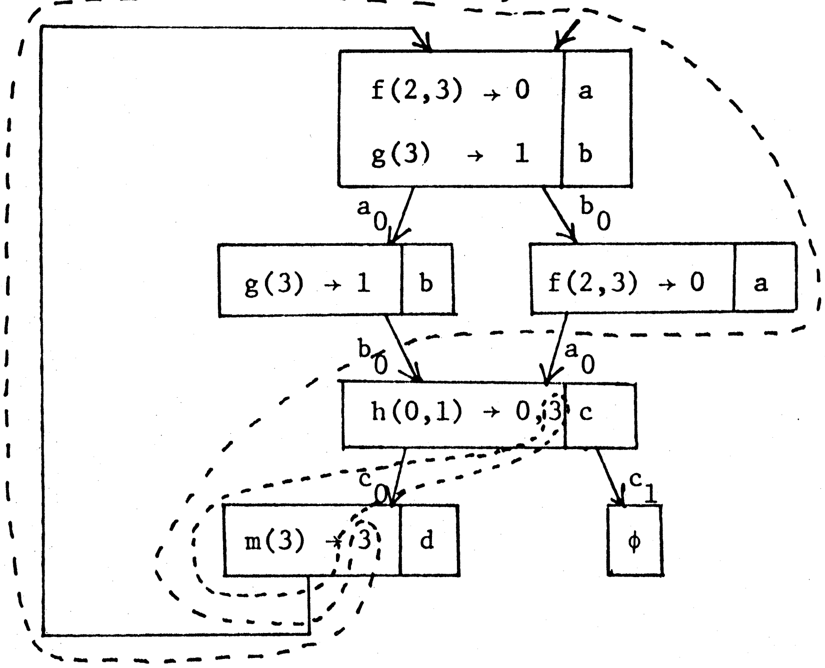
$$(f(2,3) \rightarrow 2),\ (g(3) \rightarrow 1),\ (h(2,1) \rightarrow 0,3),\ (m(3) \rightarrow 3)$$

and this would "free" location 0 for a longer period of time. The reader may see that the use of location 3 indicated above could also be changed by a renaming. Thus, for computation sequences, we are interested in contiguous segments of location usage starting with the point a value is stored in the location, and ending with the last usage of that value. This is one of the ideas developed by Logrippo in [89, 90]. This idea of renamings can now be extended to schemata. We use here a different representation of schemata which looks more like a flowchart. This form of schema is like Keller's schema [75].



In this case each box of the flowchart specifies function operations on memory. Symbol a refers to operation a which is f(2,3) → 0, b refers to g(3) → 1, c to h(0,1) → 0,3, and d to m(3) → 3. More than one operation in a box indicates concurrent performance is possible. Arrows are labelled with operation outcomes controlling the flow through the schema flowchart. Note here, that the computation that we looked at earlier is the start of one possible computation in this schema.

Now, we are again interested in renamings which do not change the overall behavior. Rather than simple segments, we now get "regions" of usage for each location. The regions for location 3 are shown in the next figure.



Clearly, for consistent renamings the renaming must be done consistently through a complete region so outlined.

The properties of such regions for renaming, and how they can be used advantageously for maximizing parallelism or for conserving memory are studied in [89, 90].

Today: Start "System of Processes" model, slices, etc.

References:

(1) R.J. Lipton, "On Synchronization Primitive Systems," Yale Computer Science Research Report #22, October 1973.

(2) R.J. Lipton, "Limitations of Synchronization Primitives with Conditional Branching and Global Variables," Proceedings of the 6th Annual Symposium on Theory of Computing, April 1974, pp. 230-241.

(3) R.J. Lipton, L. Snyder, and Y. Zalcstein, "A Comparative Study of Models of Parallel Computation," Conference Record Fifteenth Annual IEEE Symposium on Switching and Automata Theory, October 1974, pp. 145-155.

(4) D. Dolev, "Abstract Characterization of Slices of Various Synchronization Primitives," Report, Dept. of Applied Mathematics, Weizmann Institute of Science, September 1976.

What we hope to do in the next several lectures is introduce yet another model of synchronization and parallelism called a "system of processes." Using this model we will investigate how to represent some of the models of parallelism and problems of synchronization that we have discussed previously and then study how these models can be compared within the system of processes model. Initially our discussion follows reference (3) rather closely, but there are some differences in our approach.

Informally a system of processes consists of a set of sequential processes that can execute in parallel but are controlled, or synchronized, in some way. Each process consists of a sequential set of actions. An action could be viewed as an instruction, a block of instructions, a subroutine, etc. Branching or multiple exits could be modelled as well.

The control of the model is accomplished by specifying "states" of the system, where the system is started in some initial state. We define this as follows:

Definition 1: A system of processes $P = (A,D,w)$ consists of :

(1) A finite set A of actions, where each $f \in A$ is a function from D to D, and where A is partitioned into disjoint sets $A_1, A_2, \ldots, A_n$ such that $A_1 \cup A_2 \cup \ldots \cup A_n = A$. Actions in A_i are said to be actions of process i;

(2) A state set $D = D_1 \times D_2 \times \ldots \times D_{n+1}$ where $w \in D$ is called the initial state; for $i=1,\ldots,n$ D_i corresponds to the domain of instruction addresses for process i, and D_{n+1} corresponds to global program and synchronization variables.

(3) Two functions called address and program-counter

$$\text{address: } A \rightarrow D_1 \times D_2 \times \ldots \times D_n$$

$$\text{program-counter: } A \rightarrow \{1,2,\ldots,n\}$$

such that:

(a) Using $V = (L_1, L_2, \ldots, L_n, G)$ as variables over $D_1 \times D_2 \times \ldots \times D_{n+1}$, $f \in A$ is a function from D to D of the form

$$\underline{\text{when}}\ L_k = \text{address}(f) \wedge p(G)$$

$$\underline{\text{do}}\ L_k \leftarrow s(L_k, G);\ G \leftarrow t(G),$$

where $k=$ program-counter(f). Here p is a predicate and s and t are functions.

(b) For $f \in A$ and $g \in A$, if address(f) = address(g) and program-counter(f) = program-counter(g), then $f=g$.

Informally $P = (A,D,\omega)$ is a system of n processes where A is the collective set of actions of all the processes. The function "program-counter" partitions the set of actions into the actions for each process, and the function "address(f)," $f \in A$, gives a program counter value. Thus if program-counter(f) = k, then f is an action of process k, and action f will be enabled if the current value of L_k is equal to address(f) and predicate $p(G)$ is true. The predicate $p(G)$ is usually thought of as a predicate requiring certain synchronization variables to have certain values (e.g., like a semaphore being positive). Assuming action f occurs, functions s and t are performed concurrently. Function s computes a new value of L_k, specifying the address of the next action of process k; and function t computes new values for the program and synchronization variables (for example, updating semaphores and performing the desired computation). The functions address and program-counter are the only two functions used to distinguish actions. Thus condition (3)(b) of the definition insures that when f and g are the same action of the same process then f must equal g.

Example: We show how this definition can be used to model a PV system of processes; i.e., a system using semaphores. For $P = (A,D,\omega)$ let $D = D_1 \times D_2 \times \ldots \times D_n \times Z^m \times E$ and let $V = (L_1, L_2, \ldots, L_n, S_1, S_2, \ldots, S_m, G)$ be a variables over D. Here we are representing a system of n processes with m semaphores. The D_{n+1} component of D has been further decomposed into $Z^m \times E$ to represent the m semaphore and the data domains. Since semaphores values are integral Z^m is suitable for representing the m semaphore values. Finally, G represents the program variables which range over some set E. Each action $a \in A$ has one of the following forms:

(1) <u>when</u> $L_i = \text{address}(a) \wedge S_j > 0$

<u>do</u> $L_i \leftarrow \text{address}(a'); S_j \leftarrow S_j-1;$

(2) <u>when</u> $L_i = \text{address}(a)$

<u>do</u> $L_i \leftarrow \text{address}(a'); S_j \leftarrow S_j+1;$

(3) <u>when</u> $L_i = \text{address}(a)$

<u>do</u> $L_i \leftarrow s(L_i,G); G \leftarrow t(G)$

where for all values of L_i,G: $s(L_i,G) \neq L_i$.

Note here that the actions of form (1) are commonly written $P(S_j)$, those of form (2) as $V(S_j)$, and those of form (3) are the computing or <u>nonsynchronizing</u> actions. In both forms (2) and (3) the $p(G)$ predicate is left out since it is identically true and not needed as a constraint.

Definition 1 gives the static structure of this model for process interaction, but, of course, we are interested in discussing the dynamics of such systems, This is done by describing allowed sequences of actions. To do this we introduce further terminology and definitions.

Today and Next Time: Continue System of Processes model.

Dec. 7: Chee Yap - Term project report:

- A model for parallelism and synchronization

Dec. 9: Sharon Laskowski - Term project report:

- Vector Addition Systems; their results and applications

Last time we defined the system of processes model $P = (A,D,\omega)$, and showed how PV systems could be represented in this model. Today we will discuss the notions of computations in this model.

Definition 2: A timing for a system of processes $P = (A,D,\omega)$ is any finite sequence of actions $\alpha \in A^*$.

Definition 3: The value_p of a system of processes P is defined inductively as

(1) $\text{value}_p(\Lambda) = \omega$

(2) $\text{value}_p(\alpha f) = f(\text{value}_p(\alpha))$ for $\alpha \in A^*$ and $f \in A$.

Thus value_p is a function that maps a timing into the state reached by applying the timing to the system started in the initial state.

We say that f and g of A are members of the same process, and denote this by process(f,g) if and only if program-counter(f) = program-counter(g). Clearly, the relation "process" is an equivalence relation A, and A_i designates that set of actions that are in process i. For $f \in A$ and $\alpha \in A^*$ we say $f \in$ pointer-set p(α) if and only if address(f) = $\Pi_{L_k}(\text{value}(\alpha))$,† where k = program-counter(f). That is, f belongs to

† Here $\Pi_{L_k}(x)$ denotes the projection operation on the state x to obtain the value in L_k, i.e., the current address value in process k.

the pointer-set of α for P if, in the state reached after applying α, the address part of the state for the process that f is in, is equal to address(f). Also, we say that f <u>ready-set</u>(α), where k=program-counter(f), if and only if $f\in$pointer-set(α) and $p(\Pi_G(\text{value}(\alpha)))$ is true. Thus, the ready set designates those actions that could actually occur at the next step.

The following properties follow easily:

<u>Lemma 1</u>: For a system of processes $P = (A,D,\omega)$ and timings α and β:

<u>Property I</u>: ready-set(α) $\subseteq$ pointer-set(α).

<u>Property II</u>: $|\text{pointer-set}(\alpha) \cap A_i| \leq 1$.

<u>Property III</u>: $A_i \cap \text{pointer-set}(\alpha) = A_i \cap \text{pointer-set}(\alpha\beta)$ for timings α and β if no element in β is a member of A_i.

It is clear that not all timings $\alpha\in A^*$ correspond to allowed sequences of actions for a system of processes P. Indeed, if they were, the system of processes would be quite uninteresting. The restrictions imposed by P on an $\alpha\in A^*$, for it to be allowed, come from both the address function and the $p(G)$ predicate. We designate such timings as "active" as follows:

<u>Definition 3:</u> A timing $\alpha = \alpha_1\alpha_2\ldots$ of $P = (A,D,\omega)$ is called <u>active</u>, if and only if, for all $i = 1,2,\ldots,\text{length}(\alpha)$, $\alpha_i\in$ ready-set$(\alpha_1\ldots\alpha_{i-1})$.

Thus active timings is the term used for "computations" within this model. We will also assume within this model that $f(\text{value}(\alpha)) \neq \text{value}(\alpha)$ whenever αf is a timing. That is to say, that within the model there is a way of distinguishing between whether an action has already occurred or not. In [3] ready-set is defined in this way as $f(\text{value}(\alpha)) \neq \text{value}(\alpha)$, but the definition seems more direct on the structure of P the way we did it.

Definition 4: A system of processes P is called commutative if whenever $f, g \in A$ and αfg and αgf are active, then value(αfg) = value(αgf).

This definition is much like the computationally commutative definition for program schemata, i.e., a requirement on states of the system when transitions can occur in either order. We will see that commutativity enters as a hypothesis in some of the results.

In the system of processes model we wish to study how to represent various synchronization problems within the model and how to compare different problems. Earlier we showed how PV systems could be represented within the model, references [1, 2, 3] give numerous other examples. Before giving further examples here we wish to make precise the notions of "realization" and "simulate" in the model. Thus, if two synchronization systems are realized within the model and one system can simulate the other, then this provides a way of comparing the two systems.

Definition 5: Let $P = (A,D,\omega)$ and $P' = (A',D',\omega')$ be two systems of processes. A realization from P' to P is a function $r\colon A' \to A \cup \{\Lambda\}$, where Λ is the null sequence. We extend r from actions to timings as $r(\alpha_1\alpha_2\ldots\alpha_n) = r(\alpha_1)r(\alpha_2)\ldots r(\alpha_n)$ where the α_i are elements of A'.

We call an action α observable if $r(\alpha) \neq \Lambda$, and α is called a bookkeeping action if $r(\alpha) = \Lambda$.

Note here that we have two systems defined in terms of this formal model. Precise comparison can only be made within the model, not between two word statements of synchronization problems. The function r thus maps timings of P' to timings of P. Certain actions of P' may be necessary as extra, or bookkeeping steps in the realization so in r these are deleted via the mapping to Λ. Naturally, for the realization of P by P' to make any sense there must be some correspondence between the active timings of the two systems. This is formalized in the next definition of "simulate," by putting conditions on the realization function r.

Definition 6: Let P and P' be systems of processes. P' simulates P provided there is a realization r from P' to P such that:

(1) $\{r(\alpha)|\alpha \text{ active in } P'\} = \{\beta|\beta \text{ active in } P\}$,

(2) if $\text{ready-set}_{P'}(\alpha)\cap A'_i = \phi$ then $\text{ready-set}_P(r(\alpha))\cap r(A'_i) = \phi$,

(3) There is a constant $c>0$ such that for each active timing α of P' $1+\text{length}(r(\alpha)) \geq c\cdot\text{length}(\alpha)$,

(4) for any observable actions $f, g\in A'$

(a) if $\text{process}_{P'}(f,g)$ then $\text{process}_P(r(f),r(g))$,

(b) if $r(f) = r(g)$ then $\text{process}_{P'}(f,g)$.

The restrictions (1)-(4) on r in the simulate definition are all quite natural but do warrant some discussion. Condition (1) says that the two systems (under the r-mapping) should have the same set of active timings. That is, if P' can make a change then P can make a corresponding change. This condition is called onto safe since it is close to the notion of "safeness" given by Dijkstra. Condition (2) says that if after α P' causes the i^{th} process to halt, then this must also be a characteristic of P. That is, P' should not cause a process to deadlock if P does not have an analogous deadlock. This condition is called deadlock-free on processes.

Condition (3) limits the length of an active timing of P' with respect to the corresponding active timing in P. That is, the percentage of bookkeeping steps that P' can use in an active timing is bounded. This condition is called busy-wait free since any waiting due to synchronization constraints must be bounded, which is not normally the case in busy-waits. Condition (4) restricts r in such a way that in a sense maintains the process structure between P and P'. Condition (4)(a) restricts a single process in P' to be within a single process in P. That is the P' process structure is a refinement of the P process structure. Condition (4)(b) restricts the identification of any two actions in P' to be within a process of P'. This condition of maintaining the process structure between models is called faithful.

We now aim at showing that when P' simulates P this simulation is, in some sense, an efficient simulation. Condition (3) of the simulate definition is an essential feature, as one can imagine since otherwise P' could become hopelessly embroiled in bookkeeping operations. However, we want to claim somewhat more than what condition (3) directly implies, and this is based on the notion that this model allows parallel operation of the various processes within the model. Our formulation of active timings, however, obscures the notion of concurrency since a timing is a simple sequence of actions. Thus, for our efficiency result we need definitions of concurrent and of cost.

Definition 7: Let P be a system of processes and let α and β be timings for P. Then β is concurrent after α if and only if for any permutation β' of β:

(1) $\alpha\beta'$ is an active timing,

(2) value($\alpha\beta$) = value($\alpha\beta'$).

Thus, we model concurrency or parallelism by allowing within ihe set of active timings all possible permutations of β following α. Also, the state reached is to be idependent of what order the actions take place. Thus, as far as P is concerned, there is no difference within the system in the order in which the actions of β are performed once the state value(α) is reached. Thus should imply that each of these actions could be performed concurrently on separate processors, and if each action took one unit step and we had sufficient processors then all of β could be performed in a single unit step. This notion of "number of steps" is amplified in the next definition.

Definition 8: The cost $T_k(\alpha)$ of executing α on k processors is the least n such that $\alpha = \alpha^1 \ldots \alpha^n$ where α^i is concurrent after $\alpha^1 \ldots \alpha^{i-1}$ and length(α^i) $\leq k$, $i = 1,2,\ldots,n$.

The cost $T_k(\alpha)$ is thus the minimum number of steps required to execute α on k processors, assuming that each action can be performed in a unit step.

We now state the efficiency theorem.

Theorem 1 (Efficiency Theorem): Let P and P' be systems of processes such that P' simulates P and P' is commutative. Then there is a positive constant C such that for every positive integer k and every active timing α of P' $T_k(\alpha) \leq C \cdot T_k(r(\alpha))$. Moreover, C is the maximum number of bookkeeping steps which may occur consecutively in an active timing.

See [3] for a proof of this theorem.

The simulate definition and the efficiency theorem for systems of processes that we discussed last time provide a way of comparing synchronization problems. What we introduce today is the notion of "slices" which provides a more local means of comparing the behavior of synchronization problems.

<u>Definition 9</u>: Let Σ be a finite set. The set Π is a <u>Σ-slice</u> if and only if:

(1) Each element of Π is a finite sequence of distinct elements of Σ.

(2) If $\sigma\in\Sigma$ then $\sigma\in\Pi$.

(3) If $\alpha\beta$ is in Π, then α is is Π.

Property (3) of slices is the <u>prefix inclusive</u> property.

<u>Definition 10</u>: The system of processes $P = (A,D,\omega)$ <u>defines</u> the Σ-slice Π if and only if there is a one-one correspondence $d: A \to \Sigma$ such that d, extended to $A^* \to \Sigma^*$ is $\{d(\alpha)|\alpha \text{ active in } P\} = \Pi$.

<u>Definition 11</u>: The system of processes $P = (A,D,\omega)$ <u>implicitly defines</u> the Σ-slice Π if and only if there is a subset A' of A and an active timing α in P such that $(A',D, \text{value}(\alpha))$ defines Π.

Thus allows us to state how slices connect with the simulate property.

<u>Theorem 2</u>: (Invariance Theorem) If P' simulates P and P implicitly defines a Σ-slice Π, then P' implicitly defines Π.

Some properties of Σ-slices are of interest.

Property 1: Every Σ-slice is finite.

This is immediate from the definition. Σ is finite and each element of Π is made up of distinct elements of Σ.

Property 2: Let P define slice Π, then no two actions of P belong to the same process.

If f and g are any two actions of P then $d(F)$ and $d(G)$ are elements in Π. Thus f and g are both active timings of P (from definition 10), and since each process is sequential property 2 must be true.

Property 3: Let P define Π and $f,g \epsilon A$.

Then $f \epsilon$pointer-set(g) and $g \epsilon$pointer-set(f). This follows from property 2 and the earlier stated property III.

These properties clarify some of the intuitive notions of slices and their relation to system of processes. In some sense a slice corresponds to the set of active timings (see definition 10). Yet any action that can occur must be possible (i.e., in ready-set(Λ)) at the beginning. That is, the slice gives a cut (or slice) across the processes at the initial time, providing a view of the next actions that can occur in each of the processes. The implicitly defines definition (definition 11) provides a similar view across the processes at an arbitrary point in time; that is, at the state value(α) for any active timing α. Thus, the set of slices implicitly defined by a system of processes captures, in a local way, the particular synchronization of the system at that point.

In the references [1-4] quite a few different classes of slices are defined and given names such as exclusion, commutative, symmetric, permutable, Abelian, etc. Then particular classes of slices are shown to characterize the various types of synchronization and parallel models.

Before we discuss a sampling of these characterizations we note a difference in the definition of slices between references [3] and [4]. We have given the definition used in [3]. The three conditions on a Σ-slice Π are: (1) elements of Π use distinct elements of Σ; and this means Π is a finite set,

(2) $\sigma \in \Sigma \implies \sigma \in \Pi$, and

(3) slices are prefix inclusive.

In [4] condition (1) is not stated, so conceiveably slices could be infinite sets, and indeed that seems to be what is used since his relationships between slices and vector addition systems and vector replacement systems define a slice as a set of sequences in which each sequence can be viewed as a reachable path in the VAS or VRS. Of course, the set of reachable paths can be infinite, and in most interesting cases is infinite. Although I don't currently understand this essential difference between these two references, it may be that [4] introduces the notion of a slice being partitioned up into a sequence of subslices, and the subslices may give appropriate "local behavior" notions. Because of this difference, we continue our discussion following mainly reference [3].

If we are to compare various parallel models and synchronization systems we would like to compare over classes of instances of such systems rather than just between particular instances. Thus we broaden our notion from a given P defining a slice Π to a class of P defining a slice.

Definition 12: The class C of systems of processes defines the Σ-slice Π if and only if there exists a $P \in C$ such that P defines Π. We also say that in this case Π is C-definable.

Definition 13: A class C of systems of processes is called closed if and only if whenever $P = (A,D,w) \in C$, $A' \subseteq A$, and α is an active timing, then $(A',D, \text{value}(\alpha))$ is in C.

We now see how classes of systems are compared.

Definition 14: Let C and C' be two classes of systems of processes. Then $C \to C'$ if there is a P in C such that for no P' in C' does P' simulate P.

Intuitively $C \to C'$ means that from a synchronization point of view C is more powerful than C'. If C and C' are classes such that both $C \not\to C'$ and $C' \not\to C$ then neither is more powerful than the other and we denote this by $C \equiv C'$.

A corollary of these definitions and the invariance theorem is:

Corollary: Let C and C' be classes of systems of processes, and let C' be closed. Then if C defines Π and C' does not define Π then $C \to C'$.

This corollary provides the central comparative technique in this theory.

In contrast to the concepts of concurrent actions that are discussed to define the cost $T_k(\alpha)$ of an active timing α, much of the interest in the various synchronization techniques is the facility of the technique (here to be encoded as a class C of system of processes) to "inhibit" or "stop" actions. That is, when does one action have the ability to stop another action from occurring. Of course, this sort of behavior is the essence of the semaphore solution to the mutual exclusion problem as well as many of the other toy synchronization problems. To help capture this notion we introduce the concept of stopping via "exclusion slices."

Definition 15: Let R be a reflexive relation over the finite set Σ. Exclusion(R) is the set of all finite sequences $f_1, f_2, \ldots, f_n$ in Σ^* such that for $1 \le i \le j \le n$, not $f_i R f_j$. Then a Σ-slice Π is called an exclusion slice if and only if Π = exclusion(R) for some reflexive relation R on Σ.

Thus, an exclusion slice defines constraints of the type f stops g.

Another restriction on slices is:

Definition 16: A Σ-slice Π is symmetric if and only if $ab \in \Pi$ implies $ba \in \Pi$ for all $a, b \in \Sigma$.

We wish to show, as our sample of results in this area, two results for VAS systems. They are:

Lemma 2: Let Π be a Σ-slice which is VAS-definable, then Π is symmetric.

Lemma 3: Let Π be a symmetric exclusion slice, then Π is VAS-definable.

We first must define the class of VAS systems of processes. Here again we view the general form of a state of such a system of processes to be of the form:

$$(L_1,L_2,\ldots,L_n,S_1,\ldots,S_m,E).$$

Here, however, we are only interested in the part $S = (S_1,\ldots,S_m)$ of the state. Our alphabet of actions will consist of one element, say w_i, for each element $w_i \epsilon W$, where w_i is an m-dimensional vector from Z^m. The initial state is just the vector d and the action w_i takes the form:

$$\underline{\text{when}} \quad S + w_i \geq 0$$
$$\underline{\text{do}} \quad S \leftarrow S + w_i.$$

Now Lemma 2 follows from the simple fact that if

$$(S + w_i) + w_j \geq 0$$

then $(S + w_j) + w_i$ cannot be less than zero.

Lemma 3 is somewhat more complex and we do not prove it here (see [3]).

It is also interesting to note that through this approach it has been shown that irreflexive Petri nets and vector addition systems are both characterized by the same kind of slices, and are, in that sense, equivalent. This ties into our earlier isomorphism comparison in which such Petri nets without equivalent transitions were shown to be isomorphic to vector addition systems (see Theorem 1 of Lecture #7). A complete comparison between the isomorphic approach and the slice approach has not yet been investigated.