SYNVER: A SYSTEM FOR THE AUTOMATIC SYNTHESIS AND VERIFICATION OF SYNCHRONIZATION PROCESSES

Patricia Griffiths
Center for Research in Computing Technology
Harvard University

Key Words and Phrases: synchronization, assertions, automatic synthesis, verification

ABSTRACT

The automatic synthesis of systems of synchronized processes and the proof of the synchronization's correctness is discussed. A general system, SYNVER, is proposed. Its input is a problem description and its output is a set of communicating processes proven correct. A high-level assertion language is presented in which certain types of synchronization problems may be described in a natural but formal way. Heuristics are developed for SYNVER so that it may infer from the problem description what data structures and what operations are necessary to realize the described synchronization. From this information, the code pertaining to synchronization is synthesized. Finally, SYNVER applies verification techniques to prove that the assertions made about the synchronization of the processes are not only consistent, but are also realized by the code synthesized.

OVERVIEW

This paper addresses the synthesis and verification of the synchronization aspects of concurrent communicating processes. Our proposed system, SYNVER, consists of independent modules, namely a synthesizer and a verifier; users of the verification stage have not necessarily used the synthesizer. SYNVER's input is a problem description, written in a very high level assertion language. SYNVER's output is a set of communicating processes whose synchronization has been proven correct. SYNVER deals with synchronization problems where the inter-process communication is separable from the independent, asynchronous computations. Typical examples of these are mutual exclusion problems such as cigarette smokers (Parnas [16]) and dining philosophers. For these problems the synchronization portions of a program can be factored from the internal computation portions. Factoring of this nature is a goal in structured programming where procedural encapsulation is intended to minimize the assumptions one component makes about another (Zilles [23]). Our problem specification language and the code synthesized reflect this approach in that they isolate the communicating portions of a system of processes from the asynchronous portions. In this paper we do not handle message-passing (as in some producer-consumer types of synchronization problems) although we believe that it also could be made tractable by an extension of our approach.

For SYNVER we chose a set of synchronization primitives to make the code produced by the synthesizer easy to verify. Three phases of activity are involved in using SYNVER:

1) Synchronization problems are formulated in a problem description language in a manner suitable for automatic program synthesis.
2) SYNVER accepts this problem description and automatically synthesizes the programs which will be used as the synchronization portions of the problem solution.
3) The synchronization portions output by the synthesizer, or other similar portions of hand-written solutions, are proven correct.

In the following sections we discuss in detail the choice of a target vehicle for synthesis and each of the three phases of activity.

CHOICE OF TARGET VEHICLE

Before discussing the synthesizing phase of SYNVER, we must choose a set of synchronization primitives to be used in the code generated. A variety of sets of primitives are available, each with advantages and disadvantages. The criteria we used for choosing among these sets are:

(1) The semantics of the primitives must be well-specified.
(2) Typical members of the class of problems we wish to handle, such as mutual exclusion, must be easily solvable using the primitives.
(3) It should be possible to synthesize programs using the primitives.
(4) The primitives should allow a clear distinction between variables and instructions involved in communication between processes and those relevant only to the internal logic of the program.

With respect to these criteria we considered the approaches and primitives of Dijkstra [5], Habermann [8], Cerf [2], Hansen [9], Saal and Riddle [20], Fisher [6], Thomas [21], Hoare [10],

This work was supported by the Advanced Research Projects Agency under Contract F19628-74-C-0083.

and Prenner [19]. Both Prenner's control facility and Hoare's monitors satisfy these criteria. Prenner's facility, which is available at Harvard, was selected.

Prenner's CI facility was chosen as a model in which one can transact with processes explicitly in a way which clearly distinguishes communication variables and code from those which are asynchronous. Hoare's model also provides this ability. One can easily interchange the CI and monitors in the sequel without any change to the assertion language. Indeed SYNVER may be used with any model of process handling which satisfies the above criteria.

Prenner's facility permits multiple processes and multiple processors. Processes communicate and pass control by means of control primitives in an extensible language, ECL. A distinguished process, the control interpreter (CI), exists to provide mutual exclusion and explicit scheduling of processes. To communicate with each other, processes make a call on the primitive CIA, (which stands for Control Interpreter Apply), specifying a function call (which we will call a synch function) to be evaluated without interruption in the environment of the CI. Any other CIA calls are deferred until such an evaluation has terminated. Hence the CI provides for the indivisible evaluation of synch function bodies. In these bodies, queues of waiting processes may be referenced explicitly, thus allowing the user to control scheduling of processes in order to achieve synchronization. The local data structures of the CI comprise a global "state of the system" with respect to inter-process communication. The synch function modifies these structures, thus treating the CI as a sort of "control switchyard" where the entire system of processes changes state.

PROBLEM SPECIFICATION

State diagrams, Petri nets, occurrence graphs, and other graph-theoretic formulations have been used to provide precise descriptions of synchronization problems, but they are not necessarily appropriate tools for high-level problem specification. (Belpaire and Wilmotte [1], Cerf [2], Gilbert and Chandler [7], Holt [11], Lipton [14], Patil [17] [18]). There are several difficulties with these methods. For example, they are not high level, they are not intuitive, and some of them (state diagrams, Petri nets) have been proven inadequate to express certain synchronization problems.

To be a good medium for problem specification, a formalism must:

(1) be independent of any programming language or implementation,
(2) provide a natural, intuitive way of expressing the problem for humans,
(3) be sufficiently precise so that it is possible to prove that some program code solves the problem, and
(4) be suitable for input to an automatic system which synthesizes code to solve the problem.

For the kinds of problems we consider, it is appropriate to view the synchronizing primitives as changing the overall state of the system of communicating processes, as comprised by the values of the CI's local variables. For these kinds of problems there are usually only a small number of interesting (and legitimate) states that the set of communicating processes can be in. These global states are most often related to which processes are executing their critical sections. Our assertion language is one in which we can not only describe these global states but also specify how synch functions force the set of processes to make transitions among these states.

The problem description supplied to the synthesizer consists, for each type of process in the system, of a process type name and its asynchronous code intermixed with CIA calls at the points of contact between processes. We propose two kinds of assertions to describe the actions or the impact that the user expects of the synch functions.

We illustrate the two kinds of assertions with a very simple example of a mutual exclusion problem description, where at most one process at a time should be evaluating its critical section. (Note that the syntax we use is informal at this point):

```
(letting K=number of processes evaluating
    their critical sections.)

ASSERT("either K=0 or K=1");                          (S1)
CIA("enter-critical-section");                        (S2)
ASSERT("if K=0 then let K=1, otherwise wait");        (S3)
<critical section>                                    (S4)
ASSERT("K=1");                                        (S5)
CIA("leave-critical-section");
ASSERT("if K=1 then
          if a process is waiting then
               start it up and K=1,
          else K=0");
```

The first and third assertions are I-assertions, the latter one indicating what state holds throughout the critical section. The second and fourth assertions are R-assertions, dictating the actions to be performed by the respective synch functions. Notice that we use the value of the variable K to describe the possible states of the system. Here we specify (through assertions) the desired synchronization. For the purpose of describing synchronization, the asynchronous computations of a program (including the actions in the critical section) are irrelevant.

The first kind of assertion is the invariant assertion, I-assertion, which immediately precedes CIA calls and describes the state of the entire system throughout the evaluation of a particular code section. ASSERT(I1,I2,...,In) requires the Ij's to be formulas or names of formulas in quantifier-free logical expressions. These Ij's are the descriptions of the possible states of the entire system of communicating processes. Hence, we constrain that only one Ij may be true at a time. The formula I1 v I2 v ... v In is asserted to be true throughout the evaluation of a particular section code. This code begins at the last lexical occurrence of a CIA call in that process's code and terminates at the point where the assertion is placed. (e.g. throughout <critical section> above). Note that it does not matter what other CIA calls other processes may make during that evaluation. One can interpret this as specifying that throughout the evaluation of this section of code, the states of the system are restricted to be members of the set of states described by I1 ... In.

The second kind of assertion is the result assertion, R-assertion, which is placed immediately following a CIA call. Result assertions specify the transitions between the states of the system that a particular CIA call should effect. It should be clear that only CIA calls can cause a change of states. Result assertions consist of a series

of conditionals, separated by commas. These conditionals describe what the CIA call should have accomplished when control returns to the process, based on which disjunct of the invariant assertion preceding the CIA call was true at the time the CI was activated for that CIA call. The format is as follows: <possible current state> ==> <next state> AND <control actions>.

As the complete syntax of these conditionals is beyond the scope of this paper, we will discuss examples from a typical case, the second readers and writers problem (Courtois [3]). The problem states that any number of readers may access a table, or one writer may access it, but readers and writers may not access it at the same time. Writers have priority; if a writer has requested access, no further readers may be given access until that writer has finished writing. For the following discussion, let I1, I2, I3, I4 be the names of the states, which we will call invariants.
I1 means no writers, no readers in the table;
I2 means one writer, no readers;
I3 means k>0 readers, no writers, no waiting writers;
I4 means k>0 readers, no writers, writers waiting.
Clearly these are the interesting states of the readers and writers problem. The code for the reader process begins with

```
ASSERT(I1,I2,I3,I4);
CIA("startr");
```

We wish evaluation of the reader process to proceed only if I1 or I3 are true upon activation of the CI. (Note we define the terminology "Ij is true" to mean "it is the case that the state of the entire system is described by Ij"). Then the result assertion is:

```
ASSERT(I1 ==>I3 AND PROCEED,
       I2 ==>I2 AND WAIT,
       I3 ==>I3 AND PROCEED,
       I4 ==>I4 AND WAIT);
```

==> is essentially a binary conditional operator whose left operand is a state description which may be true when the CI is activated for that CIA call. The right operand consists of the state description which is to be true after the evaluation of the synch function, together with other conjuncts giving actions to be taken with respect to control. The semantics of ==>are, if the left-hand-side (lhs) is true, then change states, performing the actions specified by the right-hand-side (rhs) and exit the synch function. The result assertion specifies that these conditionals are to be considered in order of appearance, as in a LISP conditional, and as soon as a true lhs is found, no others are to be considered. A true lhs will always be found since the union of the lhs operands is required to be the set of disjuncts of the preceding invariant assertion. We permit the lhs TRUE as syntactic sugar for "all the other disjuncts of the preceding invariant assertion not explicitly found on a lhs in this result assertion." The conjuncts PROCEED or WAIT indicate whether or not control is to be transferred back to the process performing the CIA. The syntax may be read as: if I1 is true then let I3 be true and proceed.

We also permit Boolean expressions on the variables of the invariants to appear on the lhs of ==>. An example is I3 AND (K=1) ==> I1 AND PROCEED.

So far we have seen how a synch function can act like a generalized P operation, in that it may test and change state descriptions and cause the calling process to wait or proceed, but not how it may act like a generalized V operation, permitting other waiting processes to proceed. This is done by allowing result assertions to specify sets of processes, called waitsets, to which processes which are required to WAIT may belong. A process may belong to at most one waitset at a time. Then we have a conjunct of the rhs of ==>called STARTUP. The syntax is

STARTUP([ALL] <process type> OUTOF <waitset name>),

which removes (all) process(es) of type <process type> from the waitset <waitset name> and allows them to be scheduled for execution.

Corresponding to this is the conjunct WAIT IN <waitset name>, which enters that process as a member of the waitset <waitset name> and also prevents it from continuing execution. There are other operations which can be performed on members of waitsets, such as MOVE(<process type> OUTOF <waitset-1> INTO <waitset-2>). Alternatively, a waitset may be implicit; e.g., if a process of type P is specified to WAIT, then to allow a process of type P to continue later, some process has a result assertion which specifies STARTUP(P).

It is important to remember that these result assertions merely specify what things the synch function should do. They are never "executed"; they comprise the input to the synthesizer which generates code which, when evaluated, realizes the specifications given. The specifications themselves are non-procedural; they give no indication of how state descriptions are to be changed or how waiting or waitsets are to be implemented.

There is one additional facility of the assertion language. It may be the case that for one lhs condition, several alternatives may be desired with differing precedence. In the readers and writers problem, if I2 is the state description before CIA("endw"), the terminating of a writer's use of the table, then to give writers priority we wish to install I2 and startup a writer process if possible. If no writer processes are waiting, then we wish to install I3 and startup all readers if possible. Otherwise, we wish to install I1 and proceed. We write the assertions as

```
ASSERT(I2);
CIA("endw");
ASSERT(I2 ==> IF STARTUP(WRITER) POSSIBLE
                 THEN I2 AND PROCEED,
          ==> IF STARTUP(ALL READER) POSSIBLE
                 THEN I3 AND PROCEED,
          ==> I1 AND PROCEED);
```

Here the IF ... POSSIBLE brackets surround operations whose execution may not necessarily be possible. For example, if there are no waiting writers, then we cannot perform STARTUP(WRITER), and hence we then see if the second alternative may be performed. This is an extension of the LISP conditional. An omitted lhs is interpreted to be the same as the lhs of the previous conditional and

the interpretation of

```
lhs ==> rhs-1,
    ==> rhs-2,...,
    ==> rhs-n,
```

is: if lhs is true then, in order, do rhs-1 if possible; else do rhs-2 if possible; ... ; else do rhs-n if possible. We call these sets of conditionals priority sets, as there are priorities among the rhs alternatives.

We now describe the second readers and writers problem in our assertion language:
(Recall K is readers in table, L is writers in table, and M is whether writers are waiting.)

```
I1 IS K=0 AND L=0,
I2 IS K=0 AND L=1,
I3 IS K>0 AND L=0 AND M=FALSE,
I4 IS K>0 and L=0 AND M=TRUE,
READER DOES
  ASSERT(I1,I2,I3,I4);
  CIA("startr");
  ASSERT(I1 ==> I3 AND PROCEED,
         I3 ==> I3 AND PROCEED,
         TRUE ==> WAIT);
  <read>
  ASSERT(I3,I4);
  CIA("endr");
  ASSERT(I4 AND (K>1) ==> I4 AND PROCEED,
         I3 AND (K>1) ==> I3 AND PROCEED,
         I4 AND (K=1) ==> I2 AND PROCEED
                          AND STARTUP(WRITER),
         I3 AND (K=1) ==> I1 AND PROCEED);
WRITER DOES
  ASSERT(I1,I2,I3,I4);
  CIA("startw");
  ASSERT(I1 ==> I2 AND PROCEED,
         I3 ==> I4 AND WAIT,
         TRUE ==> WAIT):
  <write>
  ASSERT(I2);
  CIA("endw");
  ASSERT(I2 ==> IF STARTUP(WRITER) POSSIBLE THEN
                          I2 AND PROCEED,
             ==> IF STARTUP(ALL READER) POSSIBLE
                          THEN I3 AND PROCEED,
             ==> I1 AND PROCEED);
```

Thus, a problem specification is provided in the form of sets of assertions surrounding CIA calls in otherwise asynchronous code of a system of processes. These assertions are in a formal but readable format, natural for human use and implementation-independent. In the next section we indicate how these assertions may be manipulated by the synthesizer in order to generate code for the synch functions.

SYNTHESIS

Automatic synthesis of non-trivial programs is still in its infancy. A number of synthesizers, for example that of Manna and Waldinger [15], have employed mechanical theorem provers to verify a relationship between input and output variables and work backwards through the proof to extract a program. Our synthesizer is more powerful within its own expert problem domain. We assume that the major (independent) portions are already written and that only the synchronization (dependent) portions of the system need to be synthesized.

The synthesizer first determines the data structures for the CI, and then it examines the invariant and result assertions to determine what changes to these data structures must be performed by the synch function. From this, code for the synch functions is generated.

The synthesizer determines what data structures and variables are necessary as local variables to the CI process in order to describe the global state of the system of communicating processes. These include the variables explicitly stated in the invariant definitions together with those implied by the result assertions (for example, one process queue for every implicit and explicit waitset). The data types of these variables must also be determined by examining the values they potentially may have in the invariants. The collection of all these local variables will be called the state descriptor.

The synthesizer then examines the invariant and result assertions and uses a combination of straightforward deductions and heuristic guesses to determine how (and under what conditions) the state descriptor changes. For each synch function, we consider the state descriptor changes required. Some may be straightforward, for example, complementing a Boolean variable whenever the process performing the CIA call may proceed. The changes to integers are the most difficult (reals are not considered). To determine what changes are made to an integer, a record is made of all the values it may assume (these are kept on a list associated with each variable), and the synthesizer guesses a consistent way of changing from one value to another.

The primary heuristics the synthesizer uses are consistency and simplicity - that a synch function is consistent in its use of a variable, and that this usage is usually simple. This can be done because we are dealing with "state of the system" variables, not arbitrary ones, and with synchronization problems, where the usual uses for integers consist of counts of how many processes are in a critical section, etc. Consistency implies that it is quite unlikely that there are synch functions which specify that if I1 is true then X<-X+1, but if I2 is true then X<-X/225. Thus we can reduce the number of hypotheses about how a variable may be changed by assuming for a given synch function that the variable value is changed in the same way for all conditionals that require it to be changed at all. Thus we obtain, using consistency and simplicity, the guess that X<-X+1 from the result assertion

```
ASSERT((X=0) ==> (X=1) AND PROCEED,
       (X=1) ==> (X=2) AND PROCEED,
       (X>0) ==> (X>0) AND PROCEED);
```

rather than the more complex if X=0 then X<-1 else X<-2.

The synthesizer also uses typical arithmetic heuristics to formulate more complex guesses. One such example is that if some formula holds for a synch function's conditionals where K=0, but not when K≠0, then add to the formula a term K*X. Thus the new formula holds for at least the same set of conditionals and perhaps for some others as well.

King [12], Deutsch [4], and Wegbreit [22] discuss these kinds of heuristics in greater detail.

It is certainly possible that our heuristics will not work and SYNVER must appeal to the user at an interactive level for hints on how to change variables, or even perhaps to provide entire code sections. The ultimate hint is, of course, a complete set of all the desired synch functions, at which point SYNVER moves immediately to the verification phase and performs proof-checking. As the state of the art of synthesizing improves, this interactive level may become less necessary.

Finally the synthesizer must infer from the contents of the IF ... POSSIBLE brackets in a priority set what tests to make on the state descriptor in order to determine whether or not it is possible to do a given rhs. In most cases, this is a test of whether the appropriate waitset is empty.

Once these determinations have been made and the consistency of the guesses validated, code may be generated to realize them, and code added to do the appropriate manipulation within the CI to disallow control passage if a process is to WAIT and pass control only to those processes which are to PROCEED or STARTUP.

Here we produce in ECL, which is somewhat ALGOL-like, the (slightly hand-optimized for readability) result of SYNVER on the problem description of the readers and writers problem. LASTRUN represents the process performing the CIA call. Setting LASTRUN to NIL prevents that process from continuing execution. Performing ENTERL and REMOVEF enter and remove processes from queues, and INACTIVEQ is the queue from which the scheduler selects processes to be run.

```
Variables:  K:INT, L,M:BOOL, READER, WRITER:  process queues

STARTR <-EXPR()
     BEGIN
     NOT L AND NOT M => K<-K+1;                    if I1 or I3 then I3
     ENTERL(LASTRUN,READER);                       else wait
     LASTRUN<-NIL;
     END;


ENDR<-EXPR()
     BEGIN
     K<-K-1;                                       decrement no. of readers
     (K=0) AND M =>                                if no readers in table
           BEGIN                                   and writer waiting, then
           L<-TRUE;                                I2 and startup waiting
           ENTERL(REMOVEF(WRITER),INACTIVEQ);      writer
           END;
     END;


STARTW<-EXPR()
     BEGIN
     NOT L AND K=0 => L<-TRUE;                     if I1 then I2
     K>0 AND NOT M -> M<-TRUE;                     else set waiting
     ENTERL(LASTRUN,WRITER);                       writer flag, if necessary
     LASTRUN<-NIL;                                 and wait
     END;


ENDW -EXPR()
     BEGIN
     WRITER=NIL =>                                 if no writers waiting
           BEGIN                                   then I3
           L<-FALSE;                               or I1 and
           TILL READER=NIL DO
                 BEGIN                             startup waiting readers
                 ENTERL(REMOVEF(READER),INACTIVEQ);
                 K<-K+1;                           if any
                 END;
           END;
     ENTERL(REMOVEF(WRITER),INACTIVEQ);            else startup writer
     END;
```

VERIFICATION

As in the case of synthesis, we are concerned only with the synchronization portions of the system. Unlike proving programs correct, we have no halt box with which we may associate an output assertion. Instead we have two kinds of assertions sandwiching the points of contact between processes. The result assertions state something about the impact of this contact on the entire system. The invariant assertions indicate restrictions on the results of contacts made by other processes concurrent with the evaluation of a section of code in the given process. It is clear then, that the specification of result assertions in each process must be consistent with the invariant assertions of the other processes. In addition, the actions specified by the result assertions must be correctly realized by the synch functions.

Hence, verifying that the synchronization portions of the system are correct is equivalent to verifying that the given assertions hold when the synch functions generated are used. To do this, we propose to adapt Levitt's technique for P and V [13]. The synch functions' code can be written in terms of CHOICE and SPLIT nodes, from which verification conditions can be obtained for all paths of control between invariant assertions.

In this way, we prove

(1) that the result assertions are satisfied by the appropriate synch functions, and
(2) that the invariant assertions actually do hold throughout the asynchronous code sections with which they are associated.

(1) follows from the soundness of the synthesizer, if that phase was used.
Together (1) and (2) prove that the synchronization is correct.

In essence correctness of synchronization is established with respect to a set of user-supplied assertions. We show the equivalence between a procedural description of a problem (the code generated) and a non-procedural description (the assertions given). Other correctness proofs, involving more than correctness of synchronization, are facilitated by the functions generated by the synthesizer, because they are very straightforward. The code generated contains no procedure calls, is GOTO-less, and contains FOR loops only rarely.

CONCLUSIONS AND FUTURE WORK

We have proposed an automatic synthesis system for synchronization processes, including the areas of problem specification, synthesis, and verification. Our system consists of independent modules; users of the verification stage have not necessarily used the synthesizer. Owing to its modular nature, the system is also manageable in size; the modules are self-contained and therefore easier to understand and to use.

We have shown how automatic program verification and synthesis can be made more tractable by restricting the problem domain. Problem-specification is much simpler when we are concerned only with specifying flow of control and manipulation of a few integers and bools, rather than arbitrary data structures. In addition the choice of an appropriate target vehicle in which to express programs, Prenner's CI facility, has simplified the design of the system. Both synthesis and verification are clearer and less complex when the CI concept is used, than when other primitives are tried.

We have accomplished this without restricting ourselves to trivial problems. We permit any control configurations between concurrent processes and make no limitations on the procedural aspects of problem-solving. Our only restriction is on the type of problems to be considered, namely those of synchronization of concurrent processes in a multi-processor environment.

Finally, we have formulated for a specific problem domain, a high-level tool for problem specification which is both natural and non-procedurally oriented.

One limitation of the system is that the CI facility provides a global lock, which is not necessary in all types of synchronization and can degrade system performance. Therefore we suggest as a topic for future investigation, an optimization phase of SYNVER (to follow the synthesis and verification phases), which translates synch functions to employ more efficient (but harder to synthesize or verify) primitives such as P and V. The author has partially formulated a technique which transforms CI solutions such as Readers and Writers to P and V solutions which are comparable to the hand-coded Courtois solution. More work is necessary to adapt the techniques to all cases, and to determine if a P and V solution is always possible.

REFERENCES

1. Belpaire, G. and Wilmotte, J-P. An Approach to Concepts of and Tools for a Theory of Parallel Processes. Report 57, Institut de Mathematique Pure et Appliquee, Universite Catholique de Louvain, Dec. 1972.
2. Cerf, V.G. Multiprocessors, Semaphores, and a Graph Model of Computation. Doc. diss. UCLA, April 1972.
3. Courtois, P.J. et al. Concurrent Control with 'Readers' and 'Writers'. Comm. ACM Vol. 14, No. 10 (October 1971), pp. 667-668.
4. Deutsch, L.P. An interactive program verifier. Ph.D. Th. Dept. Computer Sci., U. of Calif., Berkeley, June 1973.
5. Dijkstra, E.W. Cooperating Sequential Processes. In Programming Languages, (F. Genuys, ed.), Academic Press, New York, 1968, pp. 43-112.
6. Fisher, D.A. Control Structures for Programming Languages. Doc. diss., Carnegie-Mellon U., June 1970.
7. Gilbert, P. and Chandler, W.J. Interference Between Communicating Parallel Processes. Comm. ACM Vol. 15, No. 6 (June 1972), pp. 427-437.

8. Habermann, A.N. Synchronization of Communicating Processes. Comm. ACM Vol. 15, No. 3 (March 1972), pp. 171-176.

9. Hansen, P. A Comparison of Two Synchronizing Concepts. Acta Informatica 1 (1972), pp. 190-199.

10. Hoare, C.A.R. Monitors: an Operating System Structuring Concept. Internal Report, The Queens Univ. of Belfast (1973).

11. Holt, A.W. et al. Final Report for the Information System Theory Project. Applied Data Research Inc., New York, 1968.

12. King, J. A program verifier. Doc. diss. Computer Sci. Dept. Carnegie-Mellon U., 1969.

13. Levitt, Karl N. The application of program-proving techniques to the verification of synchronization processes. Fall Joint Computer Conf. 1972, pp. 33-47.

14. Lipton, R.J. On Synchronization Primitive Systems. Doc. diss., Carnegie-Mellon U., June 1973.

15. Manna, Z. and Waldinger, R.J. Towards Automatic Program Synthesis. Comm. ACM Vol. 14, No. 3 (March 1971), pp. 151-165.

16. Parnas, D.L. On a Solution to the Cigarette Smoker's Problem (without conditional statements). Dept. Comp. Sci. Carnegie-Mellon U., July 1972.

17. Patil, S. Coordination of Asynchronous Events, Doc. diss., MIT, 1970.

18. Patil S. Limitations and Capabilities of Dijkstra's Semaphore Primitives for Coordination among Processes. Project MAC, Computational Structures Group Memo 57, Feb. 1971.

19. Prenner, C.J. Multi-path Control Structures for Programming Languages. Ph.D. Th., Harvard U., May 1972.

20. Saal, H. and Riddle, W. Communicating Semaphores. Comp. Sci. Dept. Stanford U. STAN-CS-71-202 (Feb. 1971).

21. Thomas, R.H. A Model for Process Representation and Synthesis. Doc. diss., MIT, 1971.

22. Wegbreit, B. The synthesis of loop predicates. Comm. ACM Vol. 17, No. 2 (Feb. 1974) pp. 102-112.

23. Zilles, S. Procedural encapsulation: A Linguistic Protection Technique. Sigplan Notices Vol. 8, No. 9 (September 1973), pp. 142-146.