MODAL LOGIC OF CONCURRENT NONDETERMINISTIC PROGRAMS*

Karl Abrahamson
Department of Computer Science
University of Washington
Seattle, Wa., 98195/USA

Abstract

This paper describes a logic, L, for reasoning about concurrent programs. Programs are similar to those of dynamic logic, with a shuffle operator included. L is a modal logic including the modalities [], meaning "throughout the future," and $\langle\rangle^+$, meaning "sometime in the future." These modalities are extended by constraints, so that they can be used to express assertions such as "p holds as long as q does." Programs contain labels. Using labels, it is possible to isolate the behavior of a single process or segment of a process, while at the same time keeping the segment in the context of the whole parallel system. A certain subset of the propositional case of L is known to be decidable.

1. Introduction.

After writing many a bug-ridden program, many computer scientists and programmers have come to the conclusion that some formal verification method for programs is needed. Consequently, a number of logics of (sequential) programs have been developed [4,6,8,12]. When programs run concurrently, they are enormously more complex, and formal verification is proportionately more desirable. A number of concurrent program logics have been proposed [1,7,9,10,11]. I will assume familiarity with some of them. Some desirable properties of concurrent program logics are described below.

1) The fundamental properties of sequential programs are comparatively simple. Basicly, we wish to know about program A:

Does A always halt?

When A halts, will P be true?

The situation is not so cut and dried when programs are running concurrently. Con-

---

*This Research was supported by the National Science Foundation under Grant No. MCS77-02474.

current processes can interact with each other in complex ways. In order to describe their interaction we must be able to make statements about what a program does while it is running. Desirable properties of programs may be quite complex. For example, we may require A to eventually set condition P true, provided request Q is held true by B until acknowledgement R becomes true.

2) For the purposes of this paper, I will assume that sequential proofs are relatively well understood. It is therefore desirable to apply sequential program proof techniques to concurrent programs. Owicki [9] gives a method for doing this. In her method, one must show that two proofs are "interference-free." But the property to be proved, namely the non-interference of two proofs, cannot itself be stated within the logic! This is more than a matter of elegance. Because of it, the shortest proof of any statement about any n line concurrent program has $O(n^2)$ steps. A reasonable logic of concurrent programs should be powerful enough that the intermediate steps in proofs, as well as the ultimate goals, are expressible.

3) As noted, we wish to apply sequential proof techniques to concurrent programs. Therefore a sequential logic should be a subset of our concurrent program logic. Many sequential program logics,for example those of Hoare [6], Pratt [12], Manna and Waldinger [8] have programs as syntactic entities. A concurrent logic containing these must also have syntactic programs. This is in contrast to the logics of [7,10,11].

This paper describes a logic, denoted L, which at least approaches the goals of 1 - 3. L is a temporal logic of programs, as described by Lamport [14]. L is described in detail in sectios 2 and 3. Section 4 briefly considers proofs in L. Section 5 gives some theorems concerning decidability and expressiveness in L.

2. Programs of L.

Programs are built from some set of basic programs. These could include assignment statements, synchronization primitives, or just about anything else. Their exact nature does not concern us. There is one restriction on the semantics of basic programs; they are indivisible, in both the sense that they cannot be interrupted by other programs, and that intermediate steps are invisible when considering such properties as global invariance. This is not an unreasonable restriction. Assignment statements can be thought of as indivisible provided they contain at most one instance of any shared variable. At any rate, interleaving must take place at some levle of detail. It is simplest to make that level the level of basic programs.

Programs are built from basic programs using the operations ?, ;, $\cup$, * of dynamic logic [12], plus the parallel operator //. Dynamic logic-type programs have some advantages over the usual Algol-like programs.

1) Programs are nondeterministic. Dijkstra [2] has shown that nondeterminism is a useful concept, even in sequential programming. Also, since concurrent programs are inherently nondeterministic, we have a symmetry which can be exploited.

2) The concepts of sequencing, choice, looping and testing are separated. The familiar if-then-else construct does both a test and a choice (based on the outcome of the test.) Both the formal semantics and proofs can be simpler if we deal with only one type of action at a time.

I will briefly describe the operations of sequential (regular) dynamic logic programs before going on to parallel programs.

Tests. If p is a formula then p? is a program. p? acts as a no-op when p is true, and may not be executed when p is false. p? is an indivisible program. For example, $(p \vee q)$? tests the values of both p and q instantaneously. Tests which are not indivisible can be written, if desired, using $p?\cup q?$ for $(p \vee q)$? and p?;q? for $(p \wedge q)$? Examples of programs with tests are given under "choice" and "looping" below. I should point out that any formula can be tested (actually any closed formula.) p could, for instance, be a partial correctness assertion about a program. My reason for allowing arbitrary tests is not so much a practical one as a matter of elegance. I don't need to define separate "testable" and "writable" formulas.

Sequencing. A;B means simply run A then run B.

Choice. Since programs are nondeterministic, this operation is simple. $A \cup B$ means "nondeterministically choose to execute either A or B." The familiar construct "if p then A else B" is simulated by the program $(p?;A)\cup(\sim p?;B)$.

Looping. The program A* means "repeat A zero or more times, the choice being made nondeterministically." The familiar construct "while p do A" is simulated by the program $(p?;A)^*;\sim p?$.

Concurrent programs differ in syntax from sequential programs in two respects.

1) There is one more operator, //. A//B denotes the interleaving of execution sequences of A with those of B. The interleaving is not assumed to have the

finite delay property.

2) Every program in given a unique label. This both facilitates the formal semantics definition, and gives us a means of identifying points and regions of a program. Non-essential labels are omitted for readability.

Syntax of concurrent programs.

If ℓ is a label, α a basic program, p a closed formula, A and B programs, then the following are also programs.

1. $\ell:\alpha$
2. ℓ:p?
3. $\ell:(A \cup B)$
4. $\ell:(A;B)$
5. $\ell:(A^*)$
6. $\ell:(A//B)$

provided no label appears twice in the same program.

Semantics of concurrent programs.

A sequential program can be completely described by a set of transitions between states which the program can make. Essentially, because there are no other processes, it does no harm to consider the entire program as one indivisible step. The relational semantics of [3] exploit that. Relational semantics will not work for concurrent programs. Pratt [13] has described a semantics of processes based on trajectories. A trajectory is a finite or infinite sequence of states, through which a program can travel. However, as Pratt has noted, even trajectories are inadequate for concurrent programs, if programs are to be built using an operator such as //. There is not enough information in a trajectory to tell how it interleaves with other trajectories. The semantics of L gets around this problem by letting a program describe a set of sequences of moves. A move is an indivisible transition between two states. The move sequence ...(u,v)(w,z)... contains a move from state u to state v, followed by a move from state w to state z. For a sequential program this sequence would make sense only if v = w. But we must take into account the fact that the "phantom" move from v to w may have been made by some other process. For example, suppose A has semantics {(u,v)(w,z)}, and B has semantics {(v,w)}. Then the semantics of A//B is {(v,w)(u,v)(w,z), (u,v)(v,w)(w,z), (u,v)(w,z)(v,w)}. There is only one legal sequence in A//B, namely (u,v)(v,w)(w,z). Illegal sequences are ignored when the semantics of formulas are defined.

3. Formulas.

There are two kinds of formula, open and closed. An open formula makes a statement about a single program, G. Open formulas can make complex statements about G, without ever having to state explicitly what G is. The same formula may be true for many different programs. Open formulas are described in detail below.

A closed formula simply applies an open formula to an explicitly mentioned program. Pnueli [11] describes program logics as either endogenous, where sentences apply to a single, known, program, or exogenous, where programs are explicitly mentioned. He discusses the merits of both. Basicly, in an endogenous logic, it is easier to make complex statements about the program in question. Exogenous logics allow for statements about equivalence of programs. Programs are more a part of the logic, rather than special outside objects. Since most sequential program logics are exogenous (e.g. those of Hoare [6], Pratt [12,13]), an exogenous concurrent program logic can include a sequential one. In a sense, the open formulas form an endogenous logic, and the closed formulas form a powerful exogenous logic.

Open formulas.

An open formula describes how the state evolves with time, starting in the current state, during the execution of program G. In addition to such important information as variable values, the state includes program counter(s) values. Program counters change in the obvious way as G executes. Since G is generally nondeterministic, from any current state, there may be several different paths which G can take, each a different evolution of the state with time. By a future, I mean a sequence of states through which G could possibly travel, starting in the present state. When an open formula makes an assertion about the future behavior of G, it always considers all possible futures. The open formulas of L are as follows. Let P be a closed formula, E and F open formulas, and let ℓ be a label.

1. P is an open formula. G is ignored.

2. $\sim$E, E $\vee$ F, E $\supset$ F, etc. are open formulas.

3. a) before(ℓ) is an open formula, meaning "some program counter is at the point labeled by ℓ in G."

   b) in(ℓ) is an open formula, meaning "some program counter is within the region labeled by ℓ in G." If G contains ℓ:(ℓ_1:a; ℓ_2:b), in(ℓ) would mean before ℓ_1 or before ℓ_2.

c) after(ℓ) is an open formula, meaning "some program counter is at the point immediately following region ℓ."

4. (Special case of rule 5.)
   a) []E is an open formula meaning "E is true now and in every state of every possible future."

   b) $<>^{+}$E is an open formula meaning "Every possible future contains some state where E holds."

   c) The duals of [] and $<>^{+}$ are defined as <>E = ~[]~E, and $[]^{+}$E = $~<>^{+}~$E.

Closed formulas

We assume a base logic, such as propositional or predicate calculus. Closed formulas are just sentences of the base logic, augmented with the "closures" of open formulas. If q is an open formula and a is a program, then A.q is a closed formula meaning "apply q to program A." A.q can be combined with other formulas in the usual ways of the base logic.

Constraints

The modalities [] and $<>^{+}$, and their duals, allow one to proceed blindly into the future. Constraints give a means of "watching the states as they go by." More precisely, constraints restrict the allowable futures of G. A constraint restricts not just the individual states in the future, but the sequence of states as a whole.

5. If c is a constraint, then
   a) [c]E is an open formula, meaning "in every possible future, E will remain true as long as c does."

   b) $<c>^{+}$E is an open formula, meaning "in every possible future, E becomes true before c becomes false."

We still have left unanswered the question of just what a constraint is. I will give three kinds of constraints below. There may be others which are useful,

and for that reason constraints are left loosely defined.

No constraints.

Many interesting statements about programs can be made without any constraints at all. When there are no constraints, boxes and diamonds are left empty.

1. (Global invariance.) A.[]p means "p is true throughout the execution of A."

2. (Partial correctness.) (ℓ:A).[](after(ℓ)$\supset$ p) means "whenever A halts, p is true."

3. (A preserves p.) A.[](p$\supset$[]p) means "once p becomes true, it remains true during execution of A."

4.(no divergences.) (ℓ:A.)<>$^{+}$after(ℓ) says that all paths of A terminate.

5. We can state that, in A//B, whenever A halts, p is true, regardless of what B has done. This is written ((ℓ:A)//B).[](after(ℓ)$\supset$ p). This sort of explicit label referencing is handy in isolating the behavior of a single process. Another method of isolating a process is discussed under label constraints.

Label constraints.

Constraints apply to prefixes of paths. The constraint "ℓ" means "every move within the path is made by some program within the region labeled by ℓ." The constraint "$\ell_1, \ell_2, \ldots, \ell_n$" allows moves to be made by programs within ℓ_1 or ℓ_2 or ... or ℓ_n. Label constraints are particularly useful when the label applies to a single component of a parallel program, for then the moves made by that component can be isolated. Another use of label constraints is discussed under proofs. Notice that [ℓ]p is not the same as [in(ℓ)]p. The latter states only that one process must remain within region ℓ. [ℓ]p states that every move is made by that process which is in region ℓ. Other processes are suspended.

Example. Suppose program A works according to specifications provided concurrent programs preserve the truth of Q. A itself does not preserve Q. We can state thet B preserves Q in A//B by (A//(ℓ:B)).[](Q$\supset$[ℓ]Q). This says "after running A//B for any number of steps, if Q is found true, then running B any further will leave Q true."

The section on single step constraints gives another example using label

constraints.

Formula Constraints.

Owicki [10] has developed a logic which includes the statement "p while q", meaning "p remains true as long as q remains true." If we say that constraint q on paths means that every point on the path satisfies q, then p while q can be written [q]p.

Single step constraints.

Let the constraint ss mean "the path is of length 2" (involving one state transition.) Some interesting formulas are

1. A cannot deadlock: $A.[]\langle ss\rangle true$. This says "no matter how A runs, it can always go one more step."

2. In A//B, A never waits: $((\ell:A)//B).[]\langle ss,\ell\rangle true$.

3. In A//B, A cannot starve: $((\ell_A:A)//(\ell_B:B)).[]\langle\rangle^{+}(\langle ss,\ell_A\rangle true \wedge \sim\langle ss,\ell_B\rangle true)$

4. We might say that a program inherently deadlocks if it can never reach a state where it is free from deadlock. "A inherently deadlocks" can be stated as $ID = A.[]\langle\rangle\sim\langle ss\rangle true$. ID involves alternation of quantifiers of paths. In Owicki's logic [10], paths are always implicitly universally quantified. Hence neither ID nor its negation can be stated in her logic.

4. Proofs.

I have as yet no proof system for L. This section will briefly describe how partial correctness assertions might be proved. The basic idea is to modify sequential proof methods to make them work for parallel programs. This is the approach taken by Owicki [9], and others [1,7]. In Owicki's method, a sequential proof is done for each process of a parallel system, and then each step of each proof is shown not to be invalidated by other processes. In L, it is possible to integrate the non-interference proof with the sequential proofs. Suppose, for example, I wish to use the Hoare-style rule

(1) $$\frac{P\{A\}Q \ , \ Q\{B\}R}{P\{A;B\}R}$$

Suppose A;B occurs in process C of C//D. Assuming that the proofs of $P\{A\}Q$ and $Q\{B\}R$ have been carried out with due consideration to the action of process D, rule (1) can be applied provided it is proved that D preserves the truth of Q. Thus, tentatively, $D.[](Q \supset []Q)$ becomes another condition for rule (1). But it is not just D which must preserve Q, it is D, running concurrently with C, which must preserve Q. The additional condition should be $(C//\ell_D:D).[](Q \supset [\ell_D]Q)$. Above it was assumed that $P\{A\}Q$ was proved with due consideration for process D. That assumption should not be part of the proof rule. Rather, it should be part of the statement of what has been proved, namely $P\{A\}Q$. Ordinarily $P\{A\}Q$ is written in L as $(\ell_A:A).((\text{before}(\ell_A) \wedge P) \supset [](\text{after}(\ell_A) \supset Q))$. To show that D was accounted for, we could change the program to $(\ell_A:A//D)$. But the actions of D can be affected by the rest of C. The correct way to state $P\{A\}Q$ is $((\underbrace{\ldots \ell_A:A \ldots}_{C})//$ $\ell_D:D).((\text{before}(\ell_A) \wedge P) \supset [\ell_A,\ell_D](\text{after}(\ell_A) \supset Q)$. The labels in the box prevent C from exitting A and then looping back through A. Rule (2), related to rule (1), can now be stated. Let $E = (\ldots \ell:(\ell_A:A;\ell_B:B)\ldots)//\ell_D:D$.

(2)
$$\begin{array}{rl} \text{prove:} & E.((\text{before}(\ell_A) \wedge P) \supset [\ell_A,\ell_D](\text{after}(\ell_A) \supset Q), \\ & E.((\text{before}(\ell_B) \wedge Q) \supset [\ell_B,\ell_D](\text{after}(\ell_B) \supset R), \\ & E.\ [](Q \supset [\ell_D]Q) \\ \text{conclude:} & E.((\text{before}(\ell) \wedge P) \supset [\ell,\ell_D](\text{after}(\ell) \supset R). \end{array}$$

Rule (2) is certainly not simple. In fact, the statements of seemingly simple ideas, such as $P\{A\}Q$, are rather long. Other proof rules for L are bound to be at least as complex. I offer the following defense of these rules.

1. Abbreviations can be used to shorten the rules and simplify statements.

2. Consider rule (2). $E.[](Q \supset [\ell_D]Q)$ needs to be proven only once, even though it may be used many times, for instance to prove $P\{A;B\}R$, $P\{S;T\}R$, and so on. Short proofs are possible in some cases.

Label Constraints in proofs.

Rule (2) makes use of label constraints. There is good reason for this. An elegant way to do sequential proofs is inductively on program structure. Rule (1) is such an inductive rule. First we prove a statement about A, then one about B, and we combine them to get a statement about A;B. Concurrent programs are more difficult to handle inductively. Unless explicit reference is made to how the

proofs of P(C) and Q(D) are carried out, we can conclude nothing from them about a statement R(C//D). We can't conclude anything about R((A;B)//D) from P(A//D) and Q(B//D), since it may be the interaction of A and B which causes disaster in D. A solution is to use label constraints. Statements are made about E = C//D, and, as was the case with rule (2), programs are referred to by their labels. The "label structure" can be built up inductively without constantly changing programs. In rule 2, a statement about ℓ_A is combined with one about ℓ_B to obtain a statement about ℓ, which covers both ℓ_A and ℓ_B. ℓ represents the program A;B. It appears that labels can play a useful role in proofs.

5. Decidability and Expressiveness.

This section states some theorems concerning the propositional case of L, denoted L_0. Proofs and other results will appear later. The appendix contains a formal semantics of L_0. For these theorems, all of the constraint types mentioned so far can be in L_0. The base logic for L_0 is propositional calculus. Basic programs are uninterpreted program letters, with no inputs or outputs. A program modifies a "global environment" in some unknown way. A formula of L_0 is valid if it holds under all interpretations and propositional variable values. Let L_0^- be L_0 without $\langle\rangle^+$ and $[]^+$.

Theorem 1. The validity problem for L_0^- is decidable, and is in Co-NTIME($2^{2^{cn}}$) for some c.

PDL is the propositional case of (sequential regular) dynamic logic [3,13]. It appears on the surface that much more can be said in L than in PDL. While that is true if formulas are to be kept short, every formula of L_0^- is equivalent to a (possibly very long) formula of PDL.

Theorem 2. Every length n formula of L_0^- is equivalent to some PDL formula Q of length at most $2^{2^{cn}}$ for some c.

Theorem 3. Let PDL^+ be PDL augmented with the formula loop(A) = "A can diverge," for every program A (see [5]). Theorem 2 holds with L_0^- replaced by L_0 and PDL replaced by PDL^+.

Acknowledgement. I would like to thank Michael J. Fischer for many helpful suggestions. It was he who suggested the general form of constraints.

Appendix - Formal Semantics of L_0.

Preliminaries

Σ_0 = basic programs

Φ_0 = basic formulas (propositional variables)

Γ = labels

A structure (or interpretation) is a triple (W,π_0,ρ_0) consisting of a set of "worlds", or states, W, a function $\pi_0:\Sigma_0 \to \mathcal{P}(W)$ assigning to each basic formula the worlds where it holds, and a function $\rho_0:\Phi_0 \to \mathcal{P}(W\times W)$, assigning to each basic program a set of transitions between worlds.

$M = W\times \mathcal{P}(\Gamma)\times W$ (the set of labeled moves);

$H = M^{\infty}$ = the set of paths, finite and infinite sequences of labeled moves;

$H_r = \{h\in H : h= \ldots(u_1,t_1,v_1)(u_2,t_2,v_2)\ldots \Rightarrow v_1=u_2\}$

= the set of legal paths.

The paths in H are "discontinuous." The path X= (u,s,v)(w,t,z) is in H, even if $v \neq w$. X denotes a move from state u to state v, followed by a move from state w to state z. Only continuous paths, members of H_r, are of ultimate interest. However, since interleaving discontinuous paths can result in a continuous path, discontinuous paths cannot be ignored. The label set s in (u,s,v) denotes that this move is made by a program with all of the labels in s. A program can have several labels. For example, A is labeled by both ℓ_1 and ℓ_2 in $\ell_1:(\ell_2:A\cup \ell_3:B)$.

If X is a set of paths, then $[X]_\ell$ is defined as the set of paths in X, with every label set s of every move replaced by $s\cup\{\ell\}$.

Programs.

The set Σ of programs, and semantics $\rho:\Sigma \to \mathcal{P}(H)$, assigning to each program a set of paths, are given inductively below. Let $\alpha\in\Sigma_0$, $A,B\in\Sigma$, $p\in\Phi$, $\ell\in\Gamma$. Then

1. $\ell:\alpha\in\Sigma$, $\rho(\ell:\alpha) = \{(u,\ell,v) : (u,v)\in\rho_0(\alpha)\}$
2. $\ell:p?\in\Sigma$, $\rho(\ell:p?) = \{(u,\ell,u) : u\in\pi(p)\}$
3. $\ell:(A\cup B)\in\Sigma$, $\rho(\ell:(A\cup B)) = [\rho(A)\cup\rho(B)]_\ell$
4. $\ell:(A;B)\in\Sigma$, $\rho(\ell:(A;B)) = [\rho(A)\cdot\rho(B)]_\ell$ (concatenation of sequences. $a\cdot b=a$ if $|a|=\infty$)
5. $\ell:(A^*)\in\Sigma$, $\rho(\ell:A^*) = [\lambda\cup\rho(A)\cdot\rho(A^*)]_\ell$, and is the least solution which is closed under least upper bound of prefix chains.[1]
6. $\ell:(a//B)\in\Sigma$, $\rho(\ell:(A//B)) = [\text{shuffle}(\rho(A),\rho(B))]_\ell$, where shuffle interleaves

---

[1] This says that a program which can make arbitrarily much progress can make infinitely much progress. A prefix chain is an infinite chain of sequences $s_1 \leq s_2 \leq \ldots$, where $\leq$ denotes the prefix relation.

sequences.

Open Formulas.

Ω is the set of open formulas. For every $G \in \mathcal{P}(H)$, there is a function $\delta_G : \Omega \to \mathcal{P}(W \times H_r)$ assigning to each open formula the complete states where it holds. A complete state consists of a world and a prefix computation, which essentially encodes all program counter values. Let $p \in \tilde{\Phi}$, $q, r \in \Omega$, $\ell \in \Gamma$, and let c be a constraint. The semantics of constraints is omitted. Then

1. $p \in \Omega$, $\delta_G(p) = \{(w,h) \in W \times H_r : w \in \pi(p)\}$.
2. $\sim q \in \Omega$, $\delta_G(\sim q) = W \times H_r - \delta_G(q)$.
3. $q \vee r \in \Omega$, $\delta_G(q \vee r) = \delta_G(q) \cup \delta_G(r)$.
4. a) $in(\ell) \in \Omega$, $\delta_G(in(\ell)) = \{(w,h) \in W \times H_r : \exists u, r \in W,\ t \in \mathcal{P}(\Gamma), h' \in H.$
$[h(u,t,v)h' \in G, \ell \in t]\}$.

Note that $h(u,t,v)$ does not have to be in H_r. The process at ℓ may be blocked.

b) $before(\ell) \in \Omega$, $\delta_G(before(\ell)) =$
$\{(w,h) \in W \times H_r : \exists u_1, u_2, v_1, v_2 \in W,\ t_1, t_2 \in \mathcal{P}(\Gamma),\ h', h'' \in H.$
$[((h = h''(u_1, t_1, v_1) \wedge \ell \notin t_1) \vee h = \lambda)$
$\wedge\ h(u_2, t_2, v_2)h' \in G \wedge \ell \in t_2]\}$

c) $after(\ell) \in \Omega$, δ_G is similar to δ_G for $before(\ell)$.

5. a) $[c]q \in \Omega$, $\delta_G([c]q) =$
$\{(w,h) \in W \times H_r : (\forall h_2 .\ hh_2 \in G)(\forall h_1$ prefix of h_2, $hh_1 \in H_r$, h_1 satisfies c)
$[(w', hh_1) \in \delta_G(q)$, where w' is the second world of the last move
of hh_1 ($w' = w$ if $hh_1 = \lambda$) $]\}$.

b) $\langle c \rangle^{+} q \in \Omega$, semantics same as for $[c]q$, but replace $\forall h_1$ by $\exists h_1$.

Closed formulas.

The closed formulas $\tilde{\Phi}$, and their semantics $\pi : \tilde{\Phi} \to \mathcal{P}(W)$, are given below. Let $A \in \Sigma$, $r \in \Omega$, $p, q \in \tilde{\Phi}$, $P \in \tilde{\Phi}_0$. Then

1. $P \in \tilde{\Phi}$, $\pi(P) = \pi_0(P)$.
2. $\sim p \in \tilde{\Phi}$, $\pi(\sim p) = W - \pi(p)$.
3. $p \vee q \in \tilde{\Phi}$, $\pi(p \vee q) = \pi(p) \cup \pi(q)$.
4. $A.r \in \tilde{\Phi}$, $\pi(A.r) = \{w \in W : (w, \lambda) \in \delta_{\rho(A)}(r)\}$.

References.

1. Aschcroft, E. A. and Z. Manna. "Formalization of Properties of Parallel Programs." Machine Intelligence 6, Edinburgh University Press.
2. Dijkstra, E. W. "Guarded Commands, Nondeterminacy and Formal Derivation of Programs," C.A.C.M. 18,8, 1975.
3. Fischer, M. J. and R. E. Ladner. "Propositional Modal Logic of Programs," Proc. 9th ann. ACM Symp. on Theory of Computing, 286-294, Boulder, Col., May, 1977.
4. Floyd, R. W. "Assigning Meaning to Programs," Proc. AMS Symp. Appl. Math. 19, 1967, 19-32.
5. Harel, D. and V. R. Pratt. "Nondeterminism in Logics of Programs," Proc. 5th ann ACM Symp. on Principles of Prog. Lang., 203-213, Tuscon, Arizona, Jan., 1978.
6. Hoare, C. A. R. "An Axiomatic Basis for Computer Programming," C.A.C.M. 12,10, 1969, 576-580.
7. Lamport, L. "Proving the Correctness of Multiprocess Programs," Mass. Computer Associates, Inc. Mass. 01880.
8. Manna, Z. and R. Waldinger. "Is 'Sometime' Sometimes Better than 'Always'?," C.A.C.M. 21,2, 1978.
9. Owicki, S. and D. Gries. "An Axiomatic Proof Technique for Parallel Programs I," Acta Informatica 6, 319-339.
10. Owicki, S. Colloquium presentation, Dept. of Comp. Sci., University of Washington, Nov. 16, 1978.
11. Pnueli, A. "The Temporal Logic of Programs," 18th IEEE Symp. on Foundations of Computer Science, 46-57, Oct. 1977.
12. Pratt V. R. "Semantical Considerations on Floyd-Hoare Logic," 17th IEEE Symp. on Foundations of Computer Science, 109-121, 1976.
13. Pratt V. R. "A Practical Decision Method for Propositional Dynamic Logic," Proc. 10th ACM Symp. on Theory of Computing, 326-337, 1978.
14. Lamport, L. "'Sometime' is Sometimes 'Not Never'," S.R.I. International Report, Menlo Park, California, January, 1979.