The Temporal Logic of Programs

Amir PNUELI

May 1981

THE TEMPORAL LOGIC OF PROGRAMS*

Amir Pnueli
University of Pennsylvania, Pa. 19104
and
Tel-Aviv University, Tel Aviv, Israel

Summary:

A unified approach to program verification is suggested, which applies to both sequential and parallel programs. The main proof method suggested is that of temporal reasoning in which the time dependence of events is the basic concept. Two formal systems are presented for providing a basis for temporal reasoning. One forms a formalization of the method of intermittent assertions, while the other is an adaptation of the tense logic system K_b, and is particularly suitable for reasoning about concurrent programs.

0. Introduction

Due to increasing maturity in the research on program verification, and the increasing interest and understanding of the behavior of concurrent programs, it is possible to distinguish two important trends in the research concerning both these fields. The first is towards unification of the basic notions and approaches to program verification, be they sequential or concurrent programs. The second is the continuous search for proof methods which will approximate more and more the intuitive reasoning that a programmer employs in designing and implementing his programs.

As a result of the first trend, one can indeed claim today that there exist very few simple proof principles which apply equally well to both sequential and concurrent programs. Thus, the prevalent notions of what constitutes a correctness of a program can all be reduced to two main concepts:

a. The concept of invariance, i.e. a property holding continuously throughout the execution of a program. By appropriately extending the concept of an assertion to describe a relation between the values of the variables and the location at which the program is executing, it can be shown that the general notion of invariance covers the concepts of partial correctness and clean behavior for sequential programs, and in addition these of mutual exclusion, safety and deadlock freedom in concurrent programs.

b. The second and even more important concept is that of eventuality (or temporal implication). In its full generality this denotes a dependence in time in the behavior of the program. We write $\varphi \leadsto \psi$, read as: "ψ eventually follows φ" or "φ temporally implies ψ", if whenever the situation described by φ arises in the program, it is guaranteed that eventually the situation described by ψ will be attained.

The notion of eventuality covers as a special case the property of total correctness. In addition it provides the right generalization of correct behavior in time for cyclic or non-functional programs.

The classical approach to correctness of programs, such as represented in Manna [17], Hoare [6,8] and also Owicki [21,22] who addressed herself to concurrent programs, always considered functional programs only. Those are programs with distinct beginning and end, and some computational instructions in between, whose statement of correctness consists of the description of the function of the input variables computed on successful completion. This approach completely ignored an important class of operating system or real time type programs, for which halting is rather an abnormal situation. Only recently [10,11,19] have people begun investigating the concept of correctness for non-terminating cyclic programs. It seems that the notion of temporal implication is the correct one. Thus, a specification of correctness for an operating system may be that it responds correctly to any incoming request, expressible as: $\{$ Request arrival situation $\} \leadsto \{$ System grants request $\}$.

Similar to the unification of correctness basic concepts, there seems to be a unification in the basic proof methods. Thus for proving invariance the widely acclaimed method is the inductive assertion method. For proving eventuality one uses either the well founded set method or a relatively recent method which we prefer to call temporal reasoning. This method, introduced by Burstall [5] and further developed in [19] and [24] (called there the method of intermittent assertions), represents the second mentioned trend in trying to approach the intuitive natural line of reasoning one may adopt when informally justifying his program.

This paper attempts to contribute to these two trends. Two formal systems are presented which give a sound basis to the yet unformalized methodology of temporal reasoning about programs. This will on one hand enhance the particular method it formalizes, and on the other hand stress and give more insight to the important concept of eventuality.

The first of the two systems is a direct formal paraphrase of the ideas and arguments repeatedly used in [5] and [19]. Since this system seems adequate for sequential programs but too weak to accomodate the multi branching alternate reasoning needed for concurrent programs, a second system was adopted, which is richer in structure and is actually but a modification of the tense logic system K_b studied by Rescher and Urquhart in [23]. This system seems much more satisfying and able to model the more intricate reasoning involved in proving temporal correctness of concurrent programs.

The significance of temporal reasoning to concurrent programs was pointed out in [10,11]. However the tool suggested there, introduction of real time clock seems too gross and powerful for the purpose needed. We correct this situation here by formulating the system K^+_b which is tailored to have exactly the adequate power and mechanism for proving temporal dependencies of concurrent programs.

Another formalization of the intermittent assertion method using a richer tense logic has just recently appeared in [3],[14].

1. Systems and Programs

A unified approach to both sequential and concur-

*This research was supported in part by ONR under contract N00014-76-C-0416 and by NSF grant MCS 76-19466.

rent programs is provided by the general framework of a system (see also Keller[13]).

A dynamic discrete system consists of

$$< S, R, s_o >$$

where:

S - is the set of states the system may assume (possibly infinite)

R - is the transition relation holding between a state and its possible successors, $R \subseteq S \times S$

s_o - is the initial state .

An execution of the system is a sequence:

$$\sigma = s_o, s_1, \ldots$$

where for each $i \geq o, R(s_i, s_{i+1})$ holds.

Since R is nondeterministic in general, many different execution sequences are possible.

Obviously the concept of a discrete system is very general. It applies to programs manipulating digital data (conventional programs), to programs manipulating physical objects (Robot driving programs), to general engineering and even biological systems, restricted only by the requirement that their evolution in time be discrete. Consequently any proof principle that can be developed for general systems should apply to the verification of behavior of any of these systems. McCarthy and Hayes advocated in [20] such a general approach and to a certain extent this paper is a technical pursuance of some of the general ideas expressed there.

However, being chiefly motivated by problems in the programming area, all the examples and following discussions will be addressed to verification of programs. The generality provided by the system's concept is only utilized for presenting a uniform approach to both sequential and concurrent programs and their verification.

In order to particularize systems into programs further structuring of the state notion is needed.

<u>Sequential Programs</u>

The specific model of deterministic sequential programs can be obtained by structuring the general state into

$$s = <\pi , u>$$

π is the <u>control</u> component and assumes a <u>finite</u> number of values, taken to be labels or locations in the program. $L = \{ \ell_o, \ell_1, \ldots \ell_n \}$

u is the <u>data</u> component and will usually range over an infinite domain. In actual applications it can be further structured into individual variables and data structures.

The transition relation R can also be partitioned into a next-location function $N(\pi,u)$ and a data-transformation function $T(\pi,u)$. $N(\ell,u)$ will actually depend on u only if the statement at ℓ is a conditional.

We can thus express R in terms of N and T:

$$R(<\pi,u>, <\pi',u'>) <=> \pi' = N(\pi,u) \ \& \ u' = T(\pi,u)$$

The restriction to deterministic programs is not essential and is made only to simplify notation.

<u>Concurrent Programs</u>

By allowing more than one control component we get the case of parallel programs. The state is to be partitioned as:

$$s = <\pi_1, \ldots \pi_n ; u>$$

The range of each π_i may be considered as the (finite) program for the i-th processor, while u is the shared data component . We assume that the next state function $N(\pi,u)$, and the data transformation $T(\pi,u)$ are still deterministic and depend on a single control component at a time . However the scheduling choice of the next processor to be stepped is nondeterministic .

Intuitively, the model admits n programs being concurrently run by n processors. At each step of the whole system, one processor, i is selected, and the statement at the location pointed to by π_i is executed to completion (we do not allow procedures). This might seem at first glance to be restrictive, being unable to model possible interference between different phases of concurrent statements' execution. However it is up to the user to express his program in units which for his modelling purposes can be considered atomic. Thus for one user the statement

$$y \leftarrow f(y)$$

may be considered atomic, while another who may be worried about possible interference from other concurrent programs between the fetch and store phases of this instruction may write instead:

$$t \leftarrow y$$
$$y \leftarrow f(t)$$

where t is a new variable local to the particular process. Interference may now occur between those two statements. Since we will be interested in proving termination, we will require the scheduling to be <u>fair</u> i.e. no processor may be indefinitely delayed while enabled. This will be made more precise later.

Formally we can express the overall transition rule by the individual transition functions of each of the processors as:

$$R(<\pi_1, \ldots \pi_n; u>, <\pi_1', \ldots \pi_n'; u'>) \text{ iff}$$

for some i, $1 \leq i \leq n$:

$$(\pi_1', \ldots \pi_n') = (\pi_1, \ldots \pi_{i-1}, N_i(\pi_i,u), \pi_{i+1}, \ldots \pi_n)$$
$$u' = T_i(\pi_i, u)$$

2. <u>Specifications and Their Classification</u>

<u>A Time Hierarchy of Specifications</u>

To express properties of systems and their development in time we use relations on states q(s) (predicates) expressed in a suitable language. Applied to programs this will be a relation $q(\pi_1, \ldots \pi_n; u)$ between the data values and the location of all the processor pointers. The general verification problem is that of establishing facts about development of the properties $q(s)$ in time.

Introducing explicit time variables t_1, t_2, .. which in our model range over the natural numbers and may be connected by the relations $=$, $<$, and the time functional

$$H(t,q) \equiv q(s_t)$$

it is obvious that any arbitrary complex time dependency can be expressed. This approach was taken in [11] where some intricate time specifications are illustrated.

Here, however, we find it both instructive and useful to limit the expressive power of the language with respect to dependency in time, and observe the actual complexity required to express different useful properties. Thus it is possible to classify specifications according to the number of distinct time variables needed to express it in a time explicit formula.

1. Single Time Instance Specification - Invariance. Having only one time variable it may be either existentially or universally quantified. If we choose the latter we obtain the notion of Invariance - a property holding throughout all states of all possible execution sequences.

Extending the binary relation R to its transitive closure R* we define the set of accessible states

$$X = \{s \mid R^*(s_0,s)\}$$

A predicate $p(s)$ is invariant in the system if for every accessible state $s \in X$ $p(s)$ holds.

$$(\forall s \in X) p(s) \text{ i.e. } \forall t\ H(t,p)$$

Many important properties fall under the class of invariance relations:

Partial Correctness: Consider a sequential program with entry label ℓ_0 and exit label ℓ_m. To state its partial correctness with respect to $\varphi(\bar{x})$, $\psi(\bar{x},\bar{z})$ [17] we can claim the invariance of the statement:

$$(\pi = \ell_m) \supset [\varphi(\bar{x}) \supset \psi(\bar{x},\bar{z})]$$

i.e. that it is invariantly true that whenever we reach the exit, if the input satisfies its specification then so does the output.

Clean Execution[25,18] In all realistic situations it is not sufficient to prove that on termination the result is satisfactory. One should also see to it that on the way, no step is taken which will cause the program to behave illegally. Thus, attention should be paid to the host of potential mishaps such as: zero division, numerical overflow, exceeding subscript range, etc. Taking as an illustration the zero division case, let ℓ_1, ℓ_2,.. ℓ_k be all the locations at which division is executed, and y_1, y_2, y_k the respective divisors. The statement of zero division fault freedom is the invariance of the claim

$$(\pi = \ell_1 \supset y_1 \neq 0) \wedge \ \ldots \ \wedge (\pi = \ell_k \supset y_k \neq 0)$$

A variant of this (counter boundedness) can be used also to establish termination.

Mutual Exclusion Turning now to concurrent programs, let us consider establishing mutual exclusion of critical sections in two concurrent programs. Let S_1 be the critical section in the first program, i.e. a subset of the labels of the program, and S_2 the second critical section. Then the statement of mutual exclusion amounts to the claim of invariance:

$$\sim(\pi_1 \in S_1 \wedge \pi_2 \in S_2)$$

Deadlock Freedom[21,22]. Consider a set of concurrent programs which communicate via semaphores. A deadlock will be a situation in which each of the processors is waiting on a 'p' operation and none of the semaphore variables which are waited for is positive. Since each of the programs is of finite length, and only in a finite subset of their instructions are there 'p' operations, it is possible to construct a finite list of label vectors $\bar{\ell}^1$, $\bar{\ell}^2$,.. $\bar{\ell}^r$ such that each ℓ_j^i labels a 'p' instruction in the jth program. Correspondingly we can construct a list of vectors of variables $\bar{u}^1$, $\bar{u}^2$,..$\bar{u}^r$ such that $\bar{u}^i$ contains all the semaphore variables waited on in the instructions labeled by $\bar{\ell}^i$. Deadlock freedom is guaranteed by the invariance of the claim:

$$(\bar{\pi} = \bar{\ell}^1 \supset \bar{u}^1 \neq 0) \wedge \ldots \wedge (\bar{\pi} = \bar{\ell}^r \supset \bar{u}^r \neq 0)$$

2. Two Time Instances - Eventuality (Temporal Implication)

The most useful two time variables statement (by no means the only one) is that of eventuality (Temporal Implication) We write $\varphi \gtrsim \psi$ for

$$\forall t_1 \exists t_2\ (t_2 \geq t_1)\ H(t_1,\varphi) \supset H(t_2,\psi)$$

i.e. for every execution $\sigma = s_0, s_1,..$ whenever there exists an s_i such that $\varphi(s_i)$ there must exist a later s_j, $j \geq i$ such that $\psi(s_j)$.

An important instance of an eventuality is that of total correctness. For a sequential program with entry label ℓ_0 and exit label ℓ_m, the statement of total correctness with respect to predicates φ, ψ can be expressed by the eventuality:

$$(\pi = \ell_0 \wedge \varphi) \gtrsim (\pi = \ell_m \wedge \psi)$$

i.e. if we enter the program with input values satisfying φ, we will eventually reach the exit point with variables' values satisfying ψ.

In applying eventuality specifications to nondeterministic and concurrent programs we must distinguish between terminating and cyclic programs[11]. Programs of the first kind are expected to terminate and present a result of their computation. Total correctness for them involves guarantee of termination and of satisfaction of the output predicate on termination. Generalization of these to concurrent terminating programs is straightforward (in the formula above replace π, ℓ_0, ℓ_m by their vector counterparts).

Cyclic programs on the other hand are not supposed to halt and are run for providing continuous response to external stimulii. A typical example will be an operating system which runs continuously (hopefully) and is expected to respond to both external events, and requests from user programs which for modelling purposes can also be considered external stimulii. For this type of programs the notion of total correctness has to be extended. We claim that most of the reasonable extensions fall into the category of eventuality. To mention few, there is the property of accesibility. Usually in a mutual exclusion environment there is the dual property (termed liveness in [16]) of any of the processors eventually being able to access its critical section once it set its mind to it. If we denote by ℓ the location in the program where a processor decides it wishes to enter and by S the set of locations comprising the critical section, then the following eventuality expresses accessibility:

$$(\pi = \ell) \leadsto (\pi \in S)$$

A more general property is that of responsiveness which is appropriate for the operating system model. If an external stimulus such as a user program making a request for a resource is signified by setting a request variable to 1, or more generally by making φ become true, and the system response of granting this request is signalled by causing ψ to become true, then the general correct responsiveness property is expressed by $\varphi \leadsto \psi$ which guarantees that for every request, there will eventually come a correct response.

The Rest of the Hierarchy

These two, admittedely important, constructs by no means exhaust the range of interesting and even useful properties of programs. For example, continuing in the vein of stating properties of operating systems, there is the question of fairness in granting requests. This for example could state that if at t_1 user A requested a resource and then at a later t_2 user B requested the same resource then there will be a t_3 when user A will be granted his resource such that at no intermediate t_4 $t_1 \leqslant t_4 \leqslant t_3$ was B granted it, sidestepping A's prior request. This seems like a four variable statement and not too farfetched one.

Beyond the complete range of qualitative statements about one event preceding the other, lies another domain of questions relating to the quantitative relations between timely events. If the system is going to respond, will it respond within 10 μs, 10 ms or 10 seconds?, etc. In this paper we address ourselves only to the two "simple" cases of invariance and eventuality.

3. General Proof Principles

Following the description of the statements we would like to prove we present a survey of three proof principles. These will be described first in the general system framework and then applied in turn to sequential and then concurrent programs. When particularized to programs of either type they will be shown to reduce to known methods for some of the cases.

A. Invariance: The universally accepted method for establishing invariance is that of induction:

$$\frac{\begin{array}{c}\varphi(s_0)\\ \forall s,s^1\ \varphi(s)\wedge R(s,s^1)\supset\varphi(s^1)\end{array}}{(\forall s\epsilon X)\ \ \varphi(s)} \qquad (P1)$$

This is obviously the principle of computational induction. Clearly, a property which holds initially and is transferred along any legal transition (is inductive[13]) is invariant. Naturally when wishing to establish the invariance of a given property (such as correctness on exit) it will usually have to be generalized. This will correspond to the known method of inductive assertions[9].

B. Well Founded Sets. This method is one of the two that we present for establishing eventualities. We bring here only its natural number version, but its extension to other well founded sets is readily available and described[17].

Let A(s,n) be a predicate depending on the state s and a natural number $n \geqslant o$. Then

$$\frac{\begin{array}{c}\varphi(s)\supset\exists n\ A(s,n)\\ A(s,n)\wedge R(s,s^1)\supset A(s^1,n-1)\vee\psi(s)\end{array}}{\varphi\leadsto\psi} \qquad (P2)$$

The above principle incorporates both the notion of invariance realized by the family of invariants A (s,n) and the notion of well founded set. The basic idea is also due to Floyd[9], and many presentations similar to the above appear in the literature[18,16,13].

C. Reasoning About Eventualities

In this approach one derives simple eventuality relations directly from the system transition rules (R) and then use combination rules, and general logic reasoning to derive more complex eventualities. The method was first introduced by Burstall[5] and developed further, in an informal form [19,24], under the name of the Intermittent Assertions method. Two formalizations of the method are suggested below and some alternate formalizations are given in [3] and [15]

From its inception this method had several advantages over method B above:

a. It is more powerful than method B. As indicated in [19] any proof using method B can always be converted to a proof in the intermittent assertions method, and there exist some classes of programs (notably those which are obtained by translating recursive programs into iterative programs) for which a natural proof exists in method C, and any possible proof in B, will necessarily be overly cumbersome.

b. Proofs in C are inherently more intuitively appealing ("natural"). While B is essentially a proof by negation approach, showing that infinite or wrong computations are impossible, C adopts the more positive approach of establishing a chain of inevitable events, which following one another, will lead to a correct termination (or attainment of objective). Thus, similarly to any good assertions method, it not only formally proves the program's correctness, but gives the prover (and the reader) a better insight into the structure and execution of the program.

The following axiomatic system (ER) is a suggested formalization for temporal reasoning about events in a system.

Axioms

$$\forall s,s^1\ p(s)\wedge R(s,s^1)\supset q(s^1)\ \Rightarrow\ p\leadsto q \qquad (A1)$$

$$p\supset q\ \Rightarrow\ p\leadsto q \qquad (A2)$$

Inference Rules

$$p\leadsto q,\ \forall s,s^1\ r(s)\wedge R(s,s^1)\supset r(s^1)\ \Rightarrow\ (p\wedge r)\leadsto(q\wedge r) \qquad (R1)$$

$$p\leadsto q,\ q\leadsto r\ \Rightarrow\ p\leadsto r \qquad (R2)$$

$$p_1\leadsto q,\ p_2\leadsto q\ \Rightarrow\ (p_1\vee p_2)\leadsto q \qquad (R3)$$

$$p\leadsto q\ \Rightarrow\ (\exists u p)\leadsto q \qquad (R4)$$

In addition we take all theorems of the first order predicate calculus as axioms.

The axioms enable us to derive elementary eventualities. (A1) says that if for all one step transitions, p before the transition implies q after the transition, then $p \rightsquigarrow q$ is established. (A2) states that logical implication is a special case of temporal implication. The inference rules enable us to deduce complex temporal implications from simpler ones. Thus (R1) may be considered as either a Frame axiom or an invariance rule which adds an arbitrary invariant to any eventuality.

Note that once the connective $\rightsquigarrow$ is introduced, it may participate in any arbitrary logical expression using the other logical connectives, and the usual rules of logic applied to derive proofs. Thus, for example, the general integer induction scheme will yield the following induction principle as a special case:

$$\begin{array}{c} p(o) \rightsquigarrow q \\ \underline{p(n) \rightsquigarrow q \ \vdash \ p(n+1) \rightsquigarrow q} \\ p(n) \rightsquigarrow q \end{array} \qquad (I)$$

From which we may conclude $\exists n\ p(n) \rightsquigarrow q$ (by (R4))

Theorem 1 The system (ER) is sound and complete for proving any property of the form $\varphi \rightsquigarrow \psi$.

Proof's Sketch: (Completeness)

Let $\varphi \rightsquigarrow \psi$. Assuming the assertion language to be expressive, we can formulate in it the predicate $p(s,n)$:

"Every execution starting with s will reach in no more than n steps a state s^1 such that $\psi(s^1)$ holds."

If we assume that our non determinism is bounded (i.e., for each s there is at most a finite number of different s^1 such that $R(s,s^1)$ holds) then $\varphi \rightsquigarrow \psi$ must imply by Königs infinity Lemma that:

1. $\varphi(s) \supset \exists n\ p(s,n)$ is valid and hence provable in the logic.
   Similarly from the definition of $p(s,n)$ the following claim is valid and hence provable:
2. $p(s,n+1) \supset \psi(s) \vee [\forall s^1\ R(s,s^1) \supset p(s^1,n)]$
   from which
3. $p(s,n+1) \rightsquigarrow [p(s,n) \vee \psi(s)]$ is provable by (A1)
4. $p(s,o) \supset \psi(s)$ By the definition of p
   from 3., (R2) and (R3):
5. $p(s,n) \rightsquigarrow \psi(s) \Rightarrow p(s,n+1) \rightsquigarrow \psi(s)$
   By the induction principle (I) 4. and 5.
6. $p(s,n) \rightsquigarrow \psi(s)$
7. $\exists n\ p(s,n) \rightsquigarrow \psi(s)$ by rule (R4)
8. $\varphi \rightsquigarrow \psi$ by 1.,7., (R2) and (A2).

4. Application to Sequential Programs

We will now consider the application of the general principles to sequential programs showing that A., B. reduce to the known Floyd's methods[9] while C. forms a formalization of the Intermittent Assertions method[5] 19.

Invariance Consider a general assertion on a deterministic sequential program $q(\pi,u)$. By considering that π may assume only a finite number of values $\pi \in \{\ell_o, .. \ell_m\}$ we can always rewrite

$$\begin{aligned} q(\pi,u) \equiv\ & (\pi = \ell_o) \supset q(\ell_o, u) \\ \wedge\ & (\pi = \ell_1) \supset q(\ell_1, u) \\ & \vdots \\ \wedge\ & (\pi = \ell_m) \supset q(\ell_m, u) \end{aligned}$$

Consequently, we can express any global assertion $q(\pi,u)$ as a set of local assertions $q_i(u) = q(\ell_i,u)$ attached at each program location ℓ_i , $i=o,..\ m$ (full annotation). We call this rewriting attachment. Conversely any network of local assertions $\{q_i \mid i=o,..m\}$ can be grouped to form a global assertion.

$$q(\pi,u) = \bigwedge_i [(\pi = \ell_i) \supset q_i(u)]$$

If we examine the proof principle (P1) substituting the attachment form of $q(\pi,u)$ we get the following conditions:

$$q_o(u_o)$$

For each ℓ: $\quad q_\ell(u) \supset q_{N_\ell}(T_\ell(u))$

i.e the initial values u_o should satisfy q_o, and then considering any location ℓ in the program, let $N_\ell(u)$ denote its successor location (if ℓ labels a conditional N_ℓ will depend on u) and $T_\ell(u)$ the transformation $u \leftarrow T_\ell(u)$ affecting u on going from ℓ to N_ℓ. We require that if $q_\ell(u)$ is true at ℓ then q_{N_ℓ} should be true at N_ℓ for the transformed values. These are exactly the verification conditions for Floyd's method in the full annotation case. As a result the principle ensures that $q(\pi,u)$ is invariant throughout the execution, in particular if execution reaches the exit point ℓ_m then $q_m(u)$ holds. Thus partial correctness with respect to q_o, q_m has been established.

Eventuality (Total Correctness) In an identical way, method B for the sequential case can be shown to be equivalent to Floyd's well founded sets method.

Consider now the method of temporal reasoning (C). When we study the informal intermittent assertions method, as exemplified in [19], we find that the basic statement is:

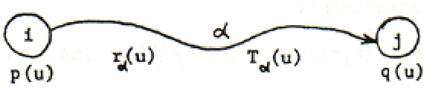
"if sometime $p(u)$ at ℓ_1 then sometime (later) $q(u)$ at ℓ_2"

ℓ_1, ℓ_2 being program locations (labels).

This can obviously be formulated as the temporal implication:

$$[\pi = \ell_1 \wedge p(u)] \rightsquigarrow [\pi = \ell_2 \wedge q(u)]$$

In order to complete the formalization we should clarify the form that axiom (A1) and rule (R1) will assume in the sequential program case. In its most useful form we will consider an arbitrary finite path in the program:



Let $r_\alpha(u)$ denote the condition on u at i such that the path α will be traversed. Let $T_\alpha(u)$ describe the transformation applied to u along α . Then (A1) for the path α will be:

$$\forall u\ p(u) \supset r_\alpha(u) \wedge q\ (T_\alpha(u)) \Rightarrow$$
$$[\pi = i \wedge p(u)] \rightsquigarrow [\pi = j \wedge q(u)]$$

For the more formally minded we should restrict the path to a single statement and consider the system (ER) augmented by a finite number of axioms which are instances of (A1), considering any of the possible types of statements.

It is now an exercise in formalization to take any of the proofs in [19], justify the basic lemmas by instances of (A1) and transitivity (R2) and work out the higher level lemmas and theorems using the induction principle (I).

Consequently (ER) is not only formally complete as proved in theorem 1, but as just shown is a natural formalization (describing the formal machinery required for a system implementing the intermittent assertion method) of a method distinguished for its intuitive appeal.

5. Concurrent Programs

Besides offering some additional insight into known methods for sequential programs, the main justification for the uniform approach suggested here is the strong guidelines it provides for verification methods for concurrent programs.

Invariance Using the next location function N and the next transformation function T it is straightforward to rewrite the general invariance principle for concurrent programs:

$$q(\bar{\pi}_0;\ u_0)$$

For each $i=1,..n$

$$q(\pi_1,..\pi_n,\ u)\supset q(\pi_1,..\pi_{i-1},N(\pi_i,u),..\pi_n;\ T(\pi_i,u))$$

$$\overline{\qquad\qquad\qquad}$$

$$q(\bar{\pi};u)\ \text{is invariant.}$$

The main problem and rationale for the different variations of this general principle is the complexity of $q(\pi;u)$ and of the set of verification conditions.

The most straightforward and inefficient approach is that of full attachment[1]. Similar to the sequential case we rewrite for the two program case:

$$q(\pi_1,\ \pi_2;\ u)\ = \bigwedge_{i,j}[(\pi_1=i\wedge\pi_2 = j)\supset q_{ij}(u)]$$

This gives rise to a number of local assertions which is proportional to the product of the sizes of the participating programs, and a corresponding number of verification conditions.

An improvement on the above is the idea of using only partial attachment:

$$q(\pi_1,\ \pi_2;\ u)\ = \bigwedge_{i}[(\pi_1=i)\supset p_i(\pi_2,u)]\wedge\bigwedge_{j}[(\pi_2=j)\supset q_j(\pi_1,u)]$$

i.e. at each point in each of the programs we attach a local assertion which might still depend on the location of the other process. This dependence is sometimes implicit and is expressed by use of additional control or shadow variables. Formally the number of assertions is now proportional to the sum of the sizes of the individual programs. However, if the interaction between the programs is high we may have to consider in the verification conditions all possible values of the opposite processor, thus regaining the exponential complexity.

On the other hand if the interaction is loose (as is very often the case) we do get an appreciable improvement and approach linear complexity (sum of sizes). All the advanced methods suggested in [2], [21], [22] and [16] may be roughly classified as partial attachment methods.

Another promising approach does no attachment at all [13,10,12], but works directly in terms of global invariants, and the verification conditions presented at the beginning of this section. The dependence on location is usually expressed in more uniform way, sometimes arithmetic, than that of case enumeration. When successful, this will also yield linear complexity Since this method is less familiar we enclose a correctness proof of the producer-consumer problem taken out of [10].

Example 1 (Producer-Consumer)

Consider the producer - consumer concurrent program in Fig. 2. The producer places an item in the buffer after its production while the consumer removes it from there. These operations are represented by respective incrementation and decrementation of n - the buffer's current load.

We wish to prove:

a. The producer and consumer are never simultaneously at their respective critical sections (mutual exclusion)

b. $0 \leqslant n \leqslant N$ i.e. the buffer capacity is never exceeded.

c. There is no deadlock.

To prove these three properties we prove first the invariance of the following three global assertions. Note that the dependence on the processor's pointer value is expressed in terms of the three characteristic functions m_i, r_i, s_i, $i = 1,2$ which assume the value 1 on some locations and 0 on the rest.

Invariants:

(1) $m_1 + m_2 + \text{MUTEX} = 1$

(2) $r_1 + r_2 + \text{IS\_EMPTY} + \text{IS\_FULL} = N$

(3) $s_1 + s_2 + \text{IS\_EMPTY} = n$

To establish each of these, check that they hold in initial state and then consider each possible single transition of each of the processors. We will use (1)-(3) now in order to prove a-c.

a. Assume that both processors are in their critical sections. We have then $m_1 = m_2 = 1$ which by (1) implies MUTEX = -1 in contradiction to MUTEX being a semaphore.

b. From 3, since IS_EMPTY is semaphore and s_1, $s_2 \geqslant 0$ we get $n \geqslant 0$. By observing that $s_i \leqslant r_i$ $i=1,2$

substitute (3) and bound it by (2) to get $n = s_1 + s_2 + \text{IS\_EMPTY} \leqslant r_1 + r_2 + \text{IS\_EMPTY} = N - \text{IS\_FULL} \leqslant N$

c. A deadlock can occur only if the two processors are waiting on a p operation. None can wait on a p(MUTEX) since then, assuming, say, that π_1 is waiting we get $m_1 = 0$, MUTEX = 0 which by (1) implies

$m_2 = 1$ which means that π_2 is in its critical section and cannot be waiting on a p. The remaining possibility is that the producer is waiting on p(IS_EMPTY) and the consumer on p(IS_FULL) but that means that $r_1 = r_2 = $ IS_EMPTY = IS_FULL = 0 which by (2) leads to N=0 in contradiction to the buffer having positive capacity.

Many other cases of program synchronized by semaphores can be handled in a similarly efficient way employing global assertions and arithmetized location dependence[12].

To summarize the issue of the complexity of concurrent program verification, it seems always possible to contrive an example which will defeat any proposed method by causing it to become exponentially complex. On the other hand we may bring once more the metaphysical argument advanced in [2], namely, that after all it was a human programmer who wrote the program and believes it to be correct. He could not have possibly considered an exponential number of cases and must have had some very few guiding reasons for writing it the way he did. It is the role of the proof method designer to come up with a method and language which will let him make these reasons more rigorous (and more conscious) and generate an efficient natural proof.

Eventuality and Tense Logic

The method of well founded sets for termination or other eventualities can also be similarly considered with either full, partial or no attachment 13,16. However the dissatisfaction at its indirectness is even more intense then in the sequential case.

Consider next application of temporal reasoning to concurrent programs. A first attempt at formalization was done in [10] and reported in [11] by the explicit introduction of a real (or integer) valued time parameter for each event. Thus, we write H(t,p) for the statement that the assertion p is realized (holds) at the time instance t. Obviously any kind of dependency on time can be expressed by this powerful device. On the other hand it might be too powerful and obscure the question of which properties of time are really essential in order to establish simple properties such as temporal implication.

The system (ER), on the other hand, seems too weak. This is somewhat surprising in view of its completeness. But this proves to be the case in the sense that we find it difficult to express natural intuitive arguments for the behavior of concurrent programs in (ER).

Obviously, we are not the first ones to face the problem of finding a minimal basis for temporal reasoning without taking the brute force approach of installing an explicit real time clock variable. Rescher and Urquhart in their book "Temporal Logic"[23] give a survey of different logical systems which increasingly capture more and more of the properties of time. Out of this selection we adopted a fragment of the tense logic K_b, which we would like to offer here as a verification tool for temporal reasoning about concurrent programs.

We introduce two basic tense operators, F and G. Denoting the present by n we can describe semantically

F(p) – It will be that p $\cdot \exists t[t \geq n \wedge H(t,p)]$
G(p) – Henceforth always p – $\forall t[t \geq n \supset H(t,p)]$

F and G are unary operators which may be used in constructing arbitrary tense well formed formulas (tWFF's), using also the conventional logical connectives and quantifiers. The temporal interpretation of a formula W involving no tense operators is that it holds in the present.

In our study of systems the absolute present is identified with s_o the initial state. For clarification let us consider some tense formulas and their system interpretation:

p – p holds at s_o

$p \supset Fq$ – if p holds at so then at a future instance q will hold.

$p \supset Gq$ – if p holds at s_o then q is invariably true for all states.

$G(p \supset Fq)$ – Whenever p is true, it will eventually be followed by a state in which q will be true (note that this matches our notion of eventuality)

$G(p \supset Gq)$ – Whenever p is true, q will be true thereafter.

Our formal system contains the following axioms:

$G(A \supset B) \supset (GA \supset GB)$ (G1)

$GA \supset A$ (The future includes the present) (G2)

$GA \supset GGA$ (G3)

Where A and B are arbitrary tWFF's.
By defining $FA \equiv \neg G(\neg A)$ we can derive the following counterparts to G2, G3:

$A \supset FA$ (F2)
$FFA \supset FA$ (F3)

The following are the inference rules:

If A is a classical tautology then $\vdash A$ (RT)
$\vdash A \Rightarrow \vdash GA$ (Generalization) (RG)
$\vdash A,\ \vdash A \supset B \Rightarrow \vdash B$ (MP)

Rule (RG) deserves special attention. It is based on the assumption of homogeneous development and that every statement which is provable for the present must be equally true in all possible futures. As long as the only way to prove basic facts about the present is through rule (RT) this assumption is justified. However if other means of deriving facts about the present are introduced, the use of rule (RG) has to be restricted.

The K_b fragment introduced here differs from the one presented in [23] by several aspects:

1. In our presentation we consider the present as part of the future.

2. While the original K_b contains primitives for events both in the future and in the past, we find it convenient and adequate to work only in terms of the future operators. Therefore, only these operators are introduced and discussed.

3. To the pure tense logic we have to add "domain dependent" axioms, restricting the future to only

these developments which are consistent with the transition mechanism of the system. These will be discussed later.

The keen observer would have realized by now that the system presented is completely isomorphic to the modal logic system S4[27,23]. Indeed one way of arriving at it is to give a temporal interpretation to the basic notion of modality, regarding "possible worlds" as "worlds developable in the future starting from the present world". In this isomorphism G stands for $\Box$ and F for $\Diamond$. We resist full identification of the two not only because of typographic reasons but because we believe that the full K_b and even more powerful tense systems will have to be used for proving properties stronger than eventualities. Once one introduces possible worlds both in the past and in the future the correspondence between G and $\Box$ fails. On the other hand in our discussion we will fully utilize this isomorphism as exemplified in the following:

Theorem 2 The system given above (pure, propositional future restricted K_b fragment) is complete (in the absolute sense) and decidable.

For completeness we may modify the proof in [23] showing the completeness of the full K_b. For decidability (which subsumes completeness) we may turn to known decidability results of S4[27,30]. We even have some results on the complexity of the decidability procedure[29].

Quantifiers: From the universal character of G and the existential character of F the following axioms make sense:

$$G(\forall x p) \equiv \forall x G(p) \qquad (Q1)$$
$$F(\exists x p) \equiv \exists x F(p) \qquad (Q2)$$
$$F(\forall x p) \supset \forall x F(p) \qquad (Q3)$$
$$\exists x\, G(p) \supset G(\exists x p) \qquad (Q4)$$

Non Pure Axioms

These are additional axioms which restrict the future to be consistent with the system, and tie the reasoning to the particular system or program we wish to study.

Invariance Axiom:

The first invariance axiom is identical to the invariance principle (P1):

$$\frac{\begin{array}{c} p(s_o) \\ p(s) \wedge R(s,s^1) \supset p(s^1) \end{array}}{\vdash Gp} \qquad (I1)$$

The second invaraince axiom is more general and it allows us to prove invariance of q not necessarily starting from the beginning but from the first time that p becomes true, i.e. from a certain moment on.

$$\frac{\begin{array}{c} p \supset q \\ q(s) \wedge R(s,s^1) \supset q(s^1) \end{array}}{\vdash p \supset Gq} \qquad (I2)$$

In fact, the more appropriate form for the consequence of (I2) is $\vdash G(p \supset Gq)$, however in view of (RG) and (G2) the two forms are equivalent.

Eventuality Axiom:

$$\frac{p(s) \wedge R(s,s^1) \supset q(s^1)}{p \supset Fq} \qquad (E)$$

This enables us to derive the most elementary eventualities, those holding across a single transition of the system.

Inevitability Axiom:

If we intend to prove termination or accesibility we must give expression to our assumption of fair scheduling, which assures in a concurrent process that every processor will ultimately be scheduled to take a step. In order to capture this notion within the system framework we partition $R = \bigvee_i A_i$ into a finite number of actions: $\mathcal{A} = \{A_i\}$. To the usual definition of execution sequence we add the restriction:

For no $A \in \mathcal{A}$ is there an i such that
$$\forall j\ (j \geq i) \supset \neg A(s_j, s_{j+1}) \qquad (R)$$

i.e. no action can be indefinitely delayed. In our model of concurrent programs, each of the actions is one of the processors taking a step. With this notation we have the following axiom reflecting the weak inevitability property:

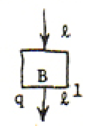
$$\frac{\begin{array}{c} p(s) \wedge R(s,s^1) \wedge \neg A(s,s^1) \supset p(s^1) \\ p(s) \wedge A(s,s^1) \supset q(s^1) \end{array}}{p \supset Fq} \qquad (N)$$

i.e. if p is invariant as long as A is not executed, and if execution of A when p is true causes q to become true, then once p is true q is inevitable (since A must eventually be executed).

A scheme of a proof in our system will consist of two separate phases. In the first phase we reason about states, immediate successors and their properties, proving all the required premises for the use of the axioms (I1),(I2),(E),(N). This phase culminates in deriving a set of basic tense formulas using the domain dependent axioms. Its role is to translate all the relevant properties of the program into basic tense-logic statements. The next phase is purely tense logical (domain independent), uses only the pure rules and manipulate the basic tense logical statements into the final result.

Consider examples of utilization of the axioms (I),(E),(N) under the concurrent programs context. Axiom (I1) may be used to derive global invariants. Example 1 is a case in point. To verify the antecedents of (I1) one has to assume that p currently hold and consider all possible one step effects of each of the processors, showing that p is preserved. A similar verification is performed in order to establish the antecedents of (E). In fact (E) is only infrequently used. This is because in analyzing a concurrent program we are either able to show invariance independently of which processor moves, or to indicate development because of the action of one specific processor. It is only rarely that we can trace development (going from p to q) independently of who moves next.

An example of the use of (N) is given by the following situation:

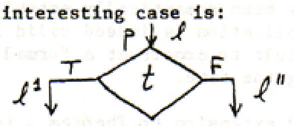


i.e. one of the processors is currently at location ℓ and is about to execute B which will cause q to become true. We can then use (N) to establish

$$(\pi=\ell) \supset F(\pi=\ell^1 \wedge q)$$

A more intriguing case is when B is a statement depending on some right hand side variables which in general can be altered by the other processors thus preventing q from becoming true. In some cases the only one who may alter these variables is π itself and then we use the fact that as long as π remain at ℓ it cannot perform any alteration and hence once it moves q will be true.

Another interesting case is:



It might be the case that $p \supset t$, and as long as π does not move p remains invariant. We use then (N) to derive that $\pi=\ell^1$ is inevitable.

<u>Theorem 3</u> K_b fragment is at least as strong as (ER)

<u>Proof</u> Express $p \rightsquigarrow q$ as $p \supset Fq$. It is then possible to show that all the axioms of (ER) are theorems of K_b fragment.

<u>Corollary</u> K_b fragment is relatively complete for proving temporal implications of the form $p \supset Fq$.

While this theoretical result does not show any advantage of K_b over (ER), the following example may serve to show how a relatively informal proof of eventual correctness of a concurrent program is naturally formalized in K_b.

<u>Example 2</u>: Consider the example of the Mutual Exclusion problem presented in Fig. 1. For simplification in notation we use the following abbreviations:

α_i for $\pi_1 = \alpha_i$ $\qquad$ i=1,..8

β_j for $\pi_2 = \beta_j$ $\qquad$ j=1,..8

c_i for $c_i = 1$, $\bar{c}_i$ for $c_i = 0$ $\qquad$ i=1,2

t for t=1, $\bar{t}$ for t=2

$\bar{p}$ for $\neg p$ where p is any of the above.

$p_o = \alpha_1 \wedge \beta_1 \wedge c_1 \wedge c_2 \wedge t$

The theorem we would like to derive (accessibility) is: $\alpha_2 \supset F\alpha_5$

We start by deriving the following invariants:

I_1: $c_1 \equiv \alpha_1 \vee \alpha_2 \vee \alpha_8$

I_2: $c_2 \equiv \beta_1 \vee \beta_2 \vee \beta_8$

I_3: $\bar{\alpha}_5 \wedge \bar{\alpha}_7 \vee \bar{\beta}_5 \wedge \bar{\beta}_7$

(Their actual form should be GI_1, etc) All these are direct consequences of (I1). In particular GI_3 proves mutual exclusion.

In the sequel we will use stronger versions of (I2) and (N) which can be derived from them

$$\frac{\begin{array}{c} p_1 \supset q \\ q(s) \wedge p_2(s^1) \wedge R(s,s^1) \supset q(s^1) \end{array}}{p_1 \wedge Gp_2 \supset Gq} \quad \text{(DI)}$$

$$\frac{\begin{array}{c} p_1(s) \wedge R(s,s^1) \wedge \neg A(s,s^1) \wedge p_2(s^1) \supset p_1(s^1) \\ p_1(s) \wedge A(s,s^1) \wedge p_2(s^1) \supset q(s^1) \end{array}}{p_1 \wedge Gp_2 \supset Fq} \quad \text{(DN)}$$

<u>Lemma A</u> $\alpha_4 \wedge t \supset F\alpha_5$

We use (DI) to establish
$(\alpha_4 \wedge t) \wedge G\bar{\alpha}_5 \supset G((\alpha_3 \vee \alpha_4) \wedge t)$

Consider now all possible locations of π_2. Consider first $\pi_2 = \beta_8$. By I_2

$$\beta_8 \supset c_2$$

Using (DI) again we get:

$\beta_8 \wedge G((\alpha_3 \vee \alpha_4) \wedge t) \supset G\beta_8$

Also $G\beta_8 \supset Gc_2$

Summarizing the above we get

$$(\alpha_4 \wedge t) \wedge G\bar{\alpha}_5 \wedge \beta_8 \supset G((\alpha_3 \vee \alpha_4) \wedge t \wedge c_2)$$

By (DN):

$$\alpha_3 \wedge G((\alpha_3 \vee \alpha_4) \wedge t \wedge c_2) \supset F\alpha_5$$

Similarly

$$\alpha_4 \wedge G((\alpha_3 \vee \alpha_4) \wedge t \wedge c_2) \supset F\alpha_3$$

Hence we can join these two together to get:

$$\beta_8 \wedge G_1 \supset F\alpha_5$$

where we denote $G_1 = G((\alpha_3 \vee \alpha_4) \wedge t)$

Similarly we can get

$G_1 \wedge \beta_6 \supset F(G_1 \wedge \beta_8)$

$G_1 \wedge \beta_4 \supset F(G_1 \wedge \beta_6)$

And can further produce under G_1 the chains of temporal implications

$$\beta_1 \rightsquigarrow \beta_2 \rightsquigarrow \beta_3 \rightsquigarrow \beta_4 \rightsquigarrow \beta_6 \rightsquigarrow \beta_8 \rightsquigarrow \alpha_5$$

and

$$\beta_5 \rightsquigarrow \beta_7 \rightsquigarrow \beta_1$$

Thus regardless of where π_2 is we derived

$$(\alpha_4 \wedge t) \wedge G\bar{\alpha}_5 \supset F\alpha_5$$

By the Lemma in the Appendix this implies:

$$\alpha_4 \wedge t \supset F\alpha_5$$

<u>Lemma B</u> $G[(\alpha_4 \vee \alpha_6 \vee \alpha_8) \wedge \bar{t} \supset \bar{\beta}_1]$

Informally: When π_1 first enters α_4, $c_2 = 0$ and hence $\pi_2 \not\to \beta_1$. The only exit to β_1 is by making t=1.

<u>Lemma C</u> $\alpha_8 \supset F\alpha_5$

Informally: Consider the next test of t by π_1 at α_8 (inevitable). If t=1 we can follow events to α_2, α_3. We then either enter α_5 or get to α_4 with t=1.

Henceforth by Lemma A, $F\alpha_5$.

If $t=2$ at α_8 then $\pi_2 \not\equiv \beta_1$, $c_1 = 1$.

If $\pi_2 = \beta_7$ then later $\pi_2 = \beta_1$, $t=1$ and remains so.

Otherwise we can follow

$$\beta_6 \rightsquigarrow \beta_8 \rightsquigarrow \beta_2 \rightsquigarrow \beta_3 \rightsquigarrow \beta_5 \rightsquigarrow \beta_7 \rightsquigarrow \alpha_5$$

$$\beta_4 \rightsquigarrow \beta_3.$$

Theorem $\alpha_2 \supset F\alpha_5$.

Follow π_1 to α_3. If we do not arrive at α_5 we get to α_4 and eventually test t. If $t=1$ then by Lemma A we get to α_5. If $t=2$ we get to α_8 and lemma C ensures the same.

6. Finite State Systems

In conclusion we will consider the special case of finite state systems. For finite state systems the validity of eventualities (and other tense formulas) is decidable. Furthermore many difficult synchronization and other concurrent programs happen to be finite state, or are usually presented in a simplified finite state form (including example 2 above).

Consider the case of a system whose state set is finite. For such a system we can consider all properties of the states as temporal propositions p(s) (a proposition possibly varying with time or state). The values of these propositions can be evaluated for each of the states and presented in a finite table. Thus the tense formula to be proved will be a propositional tense formula.

Let $\Sigma = \langle S, R, s_o \rangle$ be a finite state system, where $R = \bigvee_i A_i$, $|S| < \infty$. We can represent Σ as a finite directed edge labeled graph $G = \langle S, E \rangle$ whose nodes are the states of Σ, and there is an edge $s_1 \xrightarrow{A_i} s_2$ iff $A_i(s_1, s_2)$ holds. A proper execution of Σ will be a path in G, starting at s_o, and such that if it is infinite it passes infinitely often through edges labeled A_i for each of the A_i. For simplicity let us assume that there are no halting states or deadlocks in the system so that only infinite execution paths have to be considered.

Theorem 4 The validity of an arbitrary eventuality: $G(A \supset FB)$ is decidable for any finite state system Σ.

A and B here stand for arbitrary propositional expressions, but since they will always be evaluated on states we may as well consider each to be just a single proposition, hence checking $G(p \supset Fq)$ for validity.

We sketch below a semantic decision procedure: Obviously, it is sufficient to verify that $p \supset Fq$ holds at each state in the graph G representing Σ. Also it is sufficient to consider only states s at which p(s) = true. If also q(s) is true, the checking at s is concluded. Otherwise denote by $G_q = \langle S_q, E_q \rangle$ the subgraph, defined by deleting all states which satisfy q. By our assumption $s \in S_q$. $p \supset Fq$ will be valid at s iff G_q contains no infinite proper execution sequence starting at s, because then every s execution sequence in G must run into one of the missing states, i.e. a state satisfying q.

To check for the existence of a proper path, decompose S_q into strongly connected components $C_1, .. C_k$ where we assume that $s \in C_1$. We can construct a derived graph whose nodes are the C_i such that $C_i \to C_j$ iff there are $s_i \in C_i$, $s_j \in C_j$ and $s_i \to s_j$ in G_q. Label each of the nodes C_i by all the actions labeling edges of nodes comprising C_i.

It is not difficult to see that G_q contains an infinite proper path starting at s if and only if in the derived graph there is a path from C_1 to one of the components C_m which is labeled by all the actions.

Once it has been semantically established that the temporal implication is indeed valid in the system it is not difficult to construct a formal proof in K_b fragment proving the same.

The natural extension to Theorem 4 is whether the validity of any arbitrary tense formula is also decidable on finite state systems. The answer is indeed positive. However two extensions are needed to the logical system to be able to express the proof for a general tense formula.

a. The Initial State Axiom:

$$\frac{p(s_o)}{\vdash p} \qquad \text{(P)}$$

This enables us to derive properties which are initially true.

b. In view of (P) the generalization rule (RG) fails to be universally valid. Obviously any p which holds only initially does not necessarily hold thereafter. We thus have to modify (RG) into:

"if $\vdash A$ then $\vdash GA$ provided the proof of $\vdash A$ did not involve any use of axiom (P)." (MRG)

Thus the extension of theorem 4 is:

Theorem 5 The validity of an arbitrary tense formula on a finite state system is decidable, and the extended system K_b is adequate for proving all valid (propositional) tense formulas.

Discussion of possible proofs appears in Appendix B.

7. Discussion and Criticism

Justifying the special system introduced here by the minimality principle (use the simplest system that will work - but no simpler), we should be the first to ask: Is the notion of external time or temporality really needed in order to discuss intelligently and usefully the behavior of programs? We hope that the exposition made it clear that it is not needed in order to reason about invariance properties of program. How about properties of the eventuality type? It seems clear that for deterministic, sequential structured programs, temporality is not essential. This is so because for these programs we have an internal clock, namely the execution itself. By knowing the location in the program and the values of several loop counters we can pinpoint exactly where we are in the execution.

Therefore for these programs the simple temporal notions of "before" and "after" the execution of a program segment, implicit in all the deductive systems such as Hoare's and more recent ones[26,28] are completely adequate. It is not surprising therefore that for such programs, also the intermittent assertions method has no advantage. On the other hand when we attack programs which are cyclic, and hence being in a location we cannot identify whether this is the first or second time we are there, or nondeterministic, or concurrent, in which execution consist of intermixing operations for different processors, or even unstructured in which there exists a relation between the "where" and "when" but may be very complex, in all of these cases we must distinguish between the "where" and "when" and maintain an external time scale independent of the execution. Thus, our answer to the query above, is that as soon as we get to discuss eventuality for these more intricate type of programs, some temporal device is necessary.

Another point that is worth mentioning is that the approach taken here can be classified together with Floyd's[9], Burstall's[5] (also [4] which is very close in spirit to our work). Manna and Waldinger's and McCarthy's[20] as being Endogenous approaches. By that we mean that we immerse ourselves in a single program which we regard as the universe, and concentrate on possible developments within that universe. Characteristic of this approach is the first phase which translates the programming features into general rules of behavior which we later logically analyze. This is in contrast with Exogenous approaches such as Hoare's, Pratt's, Constables' and other deductive systems. These suggest a uniform formalism which deals in formulas whose constituents are both logical assertions and program segments, and can express very rich relations between programs and assertions. We will be the first to admit the many advantages of Exogenous systems over Endogenous systems. These include among others:

a. The uniform formalism is more elegant and universal, richer in expressibility, no need for the two phase process of Endogenous systems.
b. Endogenous systems live within a single program. There is no way to compare two programs such as proving equivalence or inclusion.
c. Endogenous systems assume the program to be rigidly given, Exogenous systems provide tools and guidance for constructing a correct system rather than just analyse an existent one.

Against these advantages endogenous system can offer the following single line of defense: When the going is tough, and we are interested in proving a single intricate and difficult program, we do not care about generality,uniformity or equivalence. It is then advantageous to work with a fixed context rather than carry a varying context with each statement. Under these conditions, endogenous systems attempt to equip the prover with the strongest possible tools to formalize his intuitive thinking and ease his way to a rigorous proof.

References:

1. - Aschroft E.A. and Manna Z (1970): "Formalization of Properties of Parallel Programs," Machine Intelligence 6, Edinburgh University Press.

2. - Aschroft E.A. (1975): "Proving Assertions About Parallel Programs," JCSS 10(1) 11

3. - Aschroft E.A. and Wadge, W.W: "Intermittent Assertion Proofs in Lucid," IFIP, Toronto 1977.

4. - Burstall, R.M.: "Formal Description of Program Structure and Semantics of First Order Logic" in B. Meltzer & D. Michie (eds.) Machine Intelligence 5 (1970) 79-98, Edinburgh.

5. - Burstall, R.M. (1974): "Program Proving as Hand Simulation With A Little Induction," Information Processing, 1974, North Holland Publishing Company, Amsterdam, 308-312.

6. - Hoare, C.A.R. (1969): "An Axiomatic Basis Of Computer Programming", CACM 12(10).

7. - Hoare, C.A.R. (1970): "Procedures and Parameters: An Axiomatic Approach", in Engeler (Ed.) Lecture Notes in Mathematics 188, Springer Verlag.

8. - Hoare, C.A.R. (1972): "Towards a Theory of Parallel Programming", in Hoare, C.A.R, Perrot, R.H. (Eds.): Operating Systems Techniques, New York Academic Press.

9. - Floyd, R.W.: "Assigning Meanings to Programs," Proc. Symp. Appl. Math. 19, in J.T. Schwartz (ed.) Mathematical Aspects of Computer Science, pp. 19-32, 1967.

10. - Francez, N.: "The Analysis of Cyclic Programs," Ph.D. thesis, Weizmann Institute of Science, Rehovot, Israel 1976.

11. - Francez, N. and Pnueli, A: "A Proof Method For Cyclic Programs," Proceedings of the 1976 Conference on Parallel Processing, 235-245.

12. - Francez, N. and Pnueli, A.: "Proving Properties of Parallel Programs by Global Invariants," to appear.

13. - Keller, R.M.: "Formal Verification of Parallel Programs," CACM 19(7) 1976.

14. - Kröger, F.: "Logical Rules of Natural Reasoning About Programs," Third Intern. Symposium on Automata, Languages and Programming, Edinburgh, Edinburgh University Press, 1976, 87-98.

15. - Kröger, F: "A Uniform Logical Basis For The Description, Specification and Correctness Proof Techniques of Programs". Institute fur Informatik der Technischen Universitat Munchen.

16. - Lamport, L (1976): "Proving the Correctness of Multiprocess Program," Massachusetts Computer Associates, Inc. Mass. 01880.

17. - Manna Z: "Mathematical Theory of Computation," McGraw-Hill (1974).

18. - Manna Z. and Pnueli, A: "Axiomatic Approach to Total Correctness," Acta Informatica 3, 243-263.

19. - Manna Z. and Waldinger, R: "Is "sometime" sometimes better than "always"? Intermittent assertions in proving Program Correctness. Proc. 2nd International Conference on Software Engineering, San Francisco (Calif.) 1976, 32-39.

20. - McCarthy, J., Hayes, P.J: "Some Philosophic Problems from the Standpoint of Artificial Intelligence" in B. Meltzer and D. Michie (eds.) Machine Intelligence 4 (1969) 463-502, Edinburgh.

21. - Owicki, S. and Gries, D.: "An Axiomatic Proof Technique for Parallel Programs I", Acta Informatica 6, 319-339.

22. - Owicki, S. and Gries, D: "Verifying Properties of Parallel Programs: An Axiomatic Approach", CACM 19(5) 1976, 279-284.

23. - Rescher, N. and Urquhart, A: "Temporal Logic," Springer Verlag 1971.

24. - Schwarz, J: "Event Based Reasoning - A System for Proving Correct Termination of Programs". Research Report No. 12, Dept. of Artificial Intelligence, University of Edinburgh, Edinburgh, Scotland.

25. - Sites, R.L.: "Proving that Computer Programs Terminate Cleanly," Stanford University, Technical Report, May 1974.

26. - Constable, R.L: "On the Theory of Programming Logic," Proc. of the 9th Annual Symposium on Theory of Computing, Boulder, Colorado, May 1977.

27. - Hughes, G.E. and Creswell, M.J: "An Introduction to Modal Logic," London: Methuen and Co. Ltd, 1972.

28. - Harel, D., Meyer, A.R. and Pratt, V.R: "Computability and Completeness in Logics of Programs," Proc. of the 9th Annual Symp. on Theory of Computing, Boulder, Colorado, May 1977.

29. - Fischer, M.J. and Ladner, R.E.: "Propositional Modal Logic of Programs," Proc. of the 9th Annual Symp. on Theory of Computing, Boulder, Colorado, May 1977.

30. - Kripke, S.A.: "Semantical Analysis of Modal Logic I: Normal Modal Propositional Calculi," Zeitschr. f. Math. Logik und Grundlagen d. Math. 9 (1963) pp. 67-96.

Appendix A

Derived Rules and Theorems of K_b:

The following are theorems proved in [23]:

T2 $\quad Gp \wedge Fq \supset F(p \wedge q)$

Corollary $\quad Gp \wedge F_q \supset F(Gp \wedge q)$

T3 $\quad F(p \vee q) \supset Fp \vee Fq$

Corollary $\quad Gp \wedge Gq \supset G(p \wedge q)$

Lemma: $\quad p \wedge G(\neg q) \supset Fq \Rightarrow p \supset Fq$

Proof:

1. $\vdash p \wedge G(\neg q) \supset Fq$ Ass.
2. $\vdash Fq \vee \neg Fq$ Tau.
3. $\vdash Fq \vee G(\neg q)$ by F's definition.
4. $\vdash p \wedge Fq \supset Fq$ Tau.
5. $\vdash p \wedge (Fq \vee G(\neg q)) \supset Fq$ 1, 4
6. $p \supset Fq$ 3,5

Appendix B

Discussion of the proof of Theorem 5 :

Theorem 5 may be proved by reduction of the problem of validity of propositional tense formula on a finite state system to that of the validity of a formula in the Monadic Second Order Theory of Successor . This is done by reintroducing explicit time variables . Referring to the definitions and results of [31] , there is a decision algorithm for the validity of formulas in this theory .

Alternately , it is possible to reconstruct the proof for our special case :

We first observe that for a given finite state system Σ it is possible to construct a finite state automaton A_Σ which will accept exactly those infinite sequences $s_0, s_1, \ldots$ which form a proper execution sequences of Σ. Denote the language of infinite words defined by A_Σ by $L(A_\Sigma) \subseteq S^\omega$. We then show that for each propositional tense formula W , we can construct an ω-regular[32] language L(W) which describes all those S^ω sequences on which W is true . This construction is defined inductively by the rules:

$$L(p)=(s_1 + \ldots + s_m)\ S^\omega$$

where $s_1, \ldots s_m$ are these states out of S on which p is true .

$$L(\neg W)= S^\omega - L(W)$$
$$L(W_1 \wedge W_2)=L(W_1) \cap L(W_2)$$
$$L(W_1 \vee W_2)=L(W_1) \cup L(W_2)$$
$$L(FW)=S^* L(W)$$
$$L(GW)=L(\neg F \neg W)$$

Since the family of ω-regular languages is closed under all the operations used above , this gives an effective way to construct L(W) . Our decision problem reduces then to the question:

is $L(A_\Sigma) \subseteq L(W)$?

i.e. do all proper execution sequences of Σ satisfy W ? . This problem is known to be decidable for ω-regular languages[33] .

31. - Büchi, J.R. :"On a Decision Method in Restricted Second Order Arithmetic" , International Congress on Logic Methodology and Philosophy of Science, Stanford , California (1960) .

32. - McNaughton, R. : "Testing and Generating Infinite Sequences by a finite Automaton" , Information and Control 9 (1966) 521-530 .

33. - Landweber , L.H. :"Decision Problems for ω-Automata " , Mathematical Systems Theory 3 (1969) 376-384 .