THE QUEST FOR COMPOSITIONALITY -
a survey of assertion-based proof systems for concurrent programs
Part 1: Concurrency based on shared variables

Willem P. de Roever Jr.

RUU-CS-85-2
January 1985

Rijksuniversiteit Utrecht

Vakgroep informatica

Budapestlaan 6 3584 CD Utrecht
Corr. adres: Postbus 80.012 3508 TA Utrecht
Telefoon 030-53 1454
The Netherlands

THE QUEST FOR COMPOSITIONALITY -
a survey of assertion-based proof systems for concurrent programs
Part 1: Concurrency based on shared variables

Willem P. de Roever Jr.

Technical Report RUU-CS-85-2
January 1985

Department of Computer Science
University of Utrecht
P.O.Box 80.012, 3508 TA Utrecht
the Netherlands

To appear in the proceedings of the IFIP Working Conference "The Role of Abstract Models in Information Processing", Vienna, January 30th - February 1st, 1985, E.J. Neuhold (ed.), North-Holland Publ.Comp.

THE QUEST FOR COMPOSITIONALITY -
a survey of assertion-based proof systems for concurrent programs
Part 1: Concurrency based on shared variables

Willem P. de Roever Jr.
Departments of Computer Science,
University of Nijmegen[1]) / University of Utrecht[2])

3rd version - december 1984

Compositionality asserts that "the specification of a program should be verifiable in terms of the specification of its syntactic subprograms". A number of proof systems for sequential and concurrent programs is analysed from the viewpoint of this principle.

TABLE OF CONTENTS

0. Introduction

"How can one check a routine in the sense of making sure that it is right?

[1]) Address: Toernooiveld 1, 6525 ED Nijmegen, the Netherlands
[2]) Address: Budapestlaan 6, 3508 TA Utrecht, the Netherlands

In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows."
These sentences date from 1949. They are the opening sentences of the paper *Checking a large routine* by ALAN TURING, and mark the birth of so-called assertion-based proof systems for programs.
Although originally concerned with a-posteriori verification of programs, i.e., after they are written, the emphasis of these proof systems changed when it was realized that they should be part of the design process of a program itself. This resulted in the development of compositional proof systems.

Compositionality asserts that

> *"the specification of a program should be verifiable in terms of the specification of its syntactic subprograms".*

Equating compositionality with *syntax-directedness* as LAMPORT does, is useful when designing programs top-down. But it has the disadvantage that it leaves the option open that the specifications S of both the original program P and its components $C_1,...C_n$ *are the same*. I.e., verifying the combination $P + S$ is reduced via syntactic reduction of P to its components C_i only, and not via reduction of specification S; this results therefore in verifying the combinations $C_i + S$, $i=1...n$.
JONES, ackowledging the usefulness of bottom-up design in the hierarchical (de)-composition of programs, advocates that verifying the combination $P + S$ should be reduced to combinations $C_i + S_i$, where S_i is a specification of module C_i which is *independent* of any environment (such as P) in which C_i may ultimately function. Such specifications are called *modular*, and lead to the design of modular proofsystems.
In this paper, a number of proof systems for sequential and concurrent programs is analysed from the viewpoint of the compositionality principle.
For proving sequential while programs partially correct, a comparison is made between the methods of FLOYD's and HOARE's; Hoare's method is compositional whereas Floyd's is not.
For concurrent programs, compositionality is known as DIJKSTRA's requirement that *"for a concurrent program the complexity of its proof should be of the order of the sum of the complexities of the proofs for its parallel components* (and not of the order of their product)".
Then various proof systems for invariance properties of concurrent shared variable programs are discussed. First the extension of Floyd's method for shared variable concurrency is explained, and its formulation in the vein of Hoare by OWICKI & GRIES; both of these are noncompositional.
Then these are compared with LAMPORT's syntax-directed concurrent Hoare logic, and JONES' modular proof rule for parallel composition.

For Hoare's CSP similar results exist.
For proving termination and liveness properties of concurrent programs most widely known is the noncompositional temporal logic approach of PNUELI's, MANNA & PNUELI's, and LAMPORT & OWICKI's. Modular proof methods for temporal properties of concurrent programs have been presented in recent publications by BARRINGER, KUIPER and PNUELI.

This survey proposes a general strategy.
As starting points, noncompositional proofmethods à la Floyd are taken, which introduce verification conditions (§1, §5, §8).
Secondly, these verification conditions are formulated in syntax-directed fashion by introducing proofoutlines (§2, §9).
Thirdly, proofsystems are obtained by formulating the resulting proofobligations in the vein of Hoare (§3, §9).
Fourthly, it is observed that in case of concurrency the formulation of these proofobligations requires a metalevel (w.r.t. an object level), which destroys the property of syntax-directedness, as exemplified by interference freedom tests expressing properties of (sequential) proofs (§10).
Finally, various assertion languages are investigated according to their capacity to allow a formulation of these proofobligations within a unified framework, so as to enable syntax-directedness (§11) or even modularity (§12).

Acknowledgements: This article arose out of a set of notes for a survey lecture given for the dutch "Landelijk Project Concurrency", and for the opening lecture of the computing science seminar of Philips Research Laboratories. The support of Rob Gerth and comments by Leslie Lamport are gratefully acknowledged.

1. Floyd's method for partial correctness proofs of sequential programs

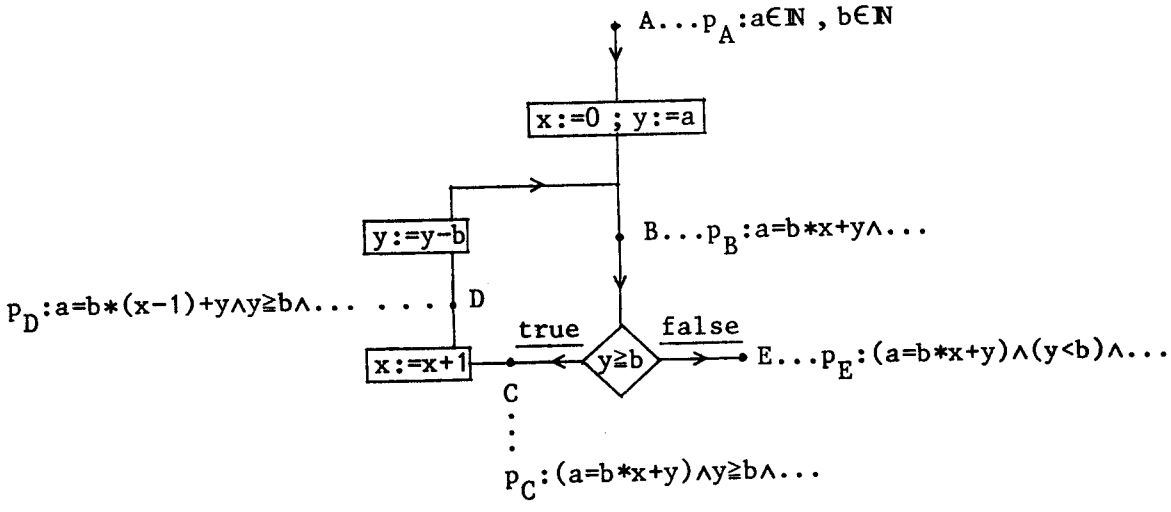
The one trait which distinguishes Floyd's method for proving partial correctness of programs from its more sophisticated successors is that it is so obviously *sound*, and therefore intuitively appealing. (Its soundness proof uses a simple induction argument on the length of execution paths.)
In a nutshell, Floyd's proof method and its rationale run as follows:

- Represent a program (composed from assignments, conditionals, sequential composition, while loops, ...) by a flowdiagram.
- Every executionpath of this diagram is a sequence of elementary paths taken from a fixed finite collection; this collection is determined by the diagram's shape only. (Observe that a flowdiagram admits infinitely many execution paths, in general.)
- Consider the collection of beginning and end points of elementary paths, called locations.
- [Verification conditions] Associate with every location a predicate s.t. every

elementary path satisfies the following *verification condition*:
If a state satisfies the predicate associated with its beginning (its precondition), and that path is taken, then the resulting output state satisfies its end-predicate (its postcondition).
(Predicates, also called assertions, are in general first order, involving propositional connectives and quantifiers.)

- If every elementary path satisfies its verification condition, then the diagram (together with its associated assertions) is called *locally correct*. Observe that checking this involves a finite number of checks.
- [Partial correctness] A flowdiagram π is called *partially correct* wrt predicates p (its precondition) and q (its postcondition) - notation: $\{p\}\pi\{q\}$ - iff if p holds for an input and the diagram terminates for that input, then q holds for the resulting output.
- [Theorem of FLOYD-NAUR] A locally correct diagram is partially correct wrt precondition p and postcondition q associated with its beginning and end points.

Example: Euclid's integer division



□

This account of Floyd's method raises the following questions:

- What is an elementary path?
- Why is every execution path a sequence of elementary paths taken from a fixed, finite, collection?
- For a given input, when is a given execution path "taken", and what is "the resulting output"? How can verification conditions be expressed (as assertion)?

The answers are provided below:

- Given a fixed collection V of locations in a diagram π, an *elementary* path of π is a nonempty path between two locations of V, possibly the same, containing no other locations of V.
- Such a fixed, finite, collection of elementary paths from which every execution

path of π can be composed is obtained by requiring that

-π's beginning and end points are contained in V,

-every loop of π contains at least one location in V.

•Defining the verification condition for execution path α, precondition p, and postcondition q, requires determining:

-when execution path α is taken, i.e. when the *path condition* $C(\alpha)(\vec{x})$ holds for inputvector $\vec{x}$,

its associated *input/output transformation* $\alpha(\vec{x})$.

Below, $C(\alpha)(\vec{x})$ and $\alpha(\vec{x})$ are defined by induction on the length of α:

-$\alpha \equiv$ empty path : $C(\alpha)(\vec{x}) = \underline{true}$, $\alpha(\vec{x}) = \vec{x}$

-$\alpha \equiv \beta \cdot test(\vec{x})$: $C(\alpha)(\vec{x}) = C(\beta)(\vec{x}) \wedge test(\beta(\vec{x}))$, $\alpha(\vec{x}) = \beta(\vec{x})$

-$\alpha \equiv \beta \cdot (\vec{x} := f(\vec{x}))$: $C(\alpha) = C(\beta)$, $\alpha(\vec{x}) = f(\beta(\vec{x}))$,

where test denotes a propositional predicate (involving only propositional connectives), and $\vec{x} := f(\vec{x})$ is a simultaneous assignment.

•Consequently, the verification condition for execution path α, precondition p, and postcondition q is expressed by $\forall \vec{x} \cdot (p(\vec{x}) \wedge C(\alpha)(\vec{x}) \rightarrow q(\alpha(\vec{x})))$, notation: $\{p\}\alpha\{q\}$.

Example (continued): the verification conditions for the flowdiagram plus associated assertions given above are:

$\{p_A\}\ \underbrace{x:=0;y:=a}_{\alpha 1}\ \{p_B\}$ ... $a \in \mathbb{N} \wedge b \in \mathbb{N} \rightarrow \underbrace{a=b*0+a}_{p_B(\alpha 1(x,y))}$,

$\{p_B\}\ \underbrace{y \geq b}_{\alpha 2}\ \{p_C\}$ ... $\underbrace{a=b*x+y}_{p_B(\vec{x})} \wedge \underbrace{y \geq b}_{test(\vec{x})} \rightarrow \underbrace{a=b*x+y \wedge y \geq b}_{p_C(\alpha 2(x,y))}$,

$\{p_C\}\ x:=x+1\ \{p_D\}$ ... $a=b*x+y \wedge y \geq b \rightarrow a=b*(x+1-1)+y \wedge y \geq b$,

$\{p_D\}\ y:=y-b\ \{p_B\}$ ... $a=b*(x-1)+y \wedge y \geq b \rightarrow a=b*x+(y-b)$,

$\{p_B\}\ \neg y \geq b\ \{p_E\}$ ... $a=b*x+y \wedge \neg y \geq b \rightarrow a=b*x+y \wedge y<b$.

□

Although presented here for sequential while programs, Floyd's method extends to programming constructs involving recursion [HPS] and concurrency (§5, §8). A more rigourous treatment of Floyd's method, in which flowdiagrams are also formalized, can be found in Livercy's exquisite book [Livercy].

2. Partial correctness proofs using proofoutlines

In general programs are not represented by flowdiagrams but as linear text in a programming language. This raises as question how to extend Floyd's method to programs written in such a language. The answer is straightforward: chose as collection of locations all beginning and end points of every syntactically contained statement of a program, associate again assertions with locations, and define verification conditions for every programming construct.

However, this strategy overlooks an important aspect of representation of programs in a programming language, which is absent from flowdiagram representation. Namely, *a program has a structure determined by the syntax of a language which shows how it is composed from its direct, so-called constituent, parts.* This structure should be present in correctness proofs, too, and leads to the following:

> [Principle of compositionality] A partial correctness proof for a program should be similarly structured as that program, showing that *a proof of a program is composed from the proofs of the constituent parts of that program.*

Obviously, this principle implies syntax-directed proofs. Technically, adopting this principle requires generalization of verification conditions for elementary paths to more complex constructs. This generalization is called *proofoutline.*

Examples:

- Assignment $x:=e:\{p\}x:=e\{q\}$ is a proofoutline for $x:=e$ iff $p\rightarrow q[e/x]$ holds, i.e., iff execution of $x:=e$ in a state satisfying p results in a state satisfying q - Floyd's verification condition for $x:=e$.
- Sequential composition $x:=e_1;y:=e_2$: Suppose you want to prove $\{p\}x:=e_1;y:=e_2\{q\}$ using proofoutlines $\{p_1\}x:=e_1\{q_1\}$ and $\{p_2\}y:=e_2\{q_2\}$ for $x:=e_1$ and $y:=e_2$. Then $\{p\}\{p_1\}x:=e_1\{q_1\};\{p_2\}y:=e_2\{q_2\}\{q\}$ is a proofoutline for proving $\{p\}x:=e_1;y:=e_2\{q\}$, provided $p\rightarrow p_1, q_1\rightarrow p_2$, and $q_2\rightarrow q$ hold. This can be understood as follows: Every execution path for $x:=e_1;y=e_2$ consists of an execution path for $x:=e_1$ followed by an execution path for $y:=e_2$. Let its input satisfy p, then $p\rightarrow p_1$ implies that p_1 holds, and therefore, since $\{p_1\}x:=e_1\{q_1\}$ is a proofoutline for $x:=e_1$, that after executing $x:=e_1$ q_1 holds. By $q_1\rightarrow p_2$ this implies that p_2 holds for the resulting intermediate state which acts as input for $y:=e_2$, and therefore, since $\{p_2\}y:=e_2\{q_2\}$ is a proofoutline for $y:=e_2$, that q_2 holds for the resulting output. By $q_2\rightarrow q$ this implies that q holds for the resulting output of the combined execution of $x:=e_1;y:=e_2$. Q.E.D.
  Observe that direct correspondence with Floyd's original method for flowdiagrams is obtained by chosing $p\equiv p_1$, $q_1\equiv p_2$ and $q_2\equiv q$ - suggesting $\{p\}x:=e_1\{q_1\}y:=e_2\{q\}$ as simplified notation.

□

Definition [Proofoutlines] Inductive definition of proofoutline pfo(p,S,q) for $\{p\}S\{q\}$ by complexity of S:

- $S\equiv x:=e$: $\{p\}x:=e\{q\}$ is a proofoutline for $\{p\}S\{q\}$ iff $p\rightarrow q[e/x]$ holds.
- $S\equiv S_1;S_2$: $\{p\}pfo(p_1,S_1,q_1);pfo(p_2,S_2,q_2)\{q\}$ is a proofoutline for $\{p\}S\{q\}$ iff $p\rightarrow p_1$, $q_1\rightarrow p_2$, and $q_2\rightarrow q$ hold.
- $S\equiv$ if t then S_1 else S_2 fi : $\{p\}$ if t then $pfo(p_1,S_1,q_1)$ else $pfo(p_2,S_2,q_2)$ fi $\{q\}$ is a proofoutline for $\{p\}S\{q\}$ iff $p\wedge t\rightarrow p_1, p\wedge\neg t\rightarrow p_2, q_1\rightarrow q$, and $q_2\rightarrow q$ hold.

- $S \equiv$ **while** b **do** S_1 **od**: $\{p\}\{I\}$ **while** b **do** $pfo(p_1,S_1,q_1)$ **od** $\{q\}$ is a proofoutline for $\{p\}S\{q\}$ iff $p \to I$, $I \wedge b \to p_1$, $q_1 \to I$, and $I \wedge \neg b \to q$ hold.

□

Theorem: $\{p\}S\{q\}$ holds iff there exists a proofoutline for $\{p\}S\{q\}$. □

Proving this theorem requires in general an assertion language stronger than first order predicate calculus, such as a language with infinite disjunctions and conjunctions, or predicate calculus over the standard model of the natural numbers.

3. Hoare's proofsystem - principles

Requested: A strategy for *proving* $\{p\}S\{q\}$.

For example, suppose you want to prove $\{p\}S_1;S_2\{q\}$. Then which proofobligations do you have to fulfill concerning S_1 and S_2, taken separately? The answer is provided by the definition of proofoutline for $\{p\}S_1;S_2\{q\}$ and the preceding theorem. Namely, find proofs for $\{p_i\}S_i\{q_i\}$ for i=1,2 , and for $p \to p_1$, $q_1 \to p_2$, $q_2 \to q$. This is schematically represented by:

$$\frac{p \to p_1 \ , \ \{p_1\}S_1\{q_1\} \ , \ q_1 \to p_2 \ , \ \{p_2\}S_2\{q_2\} \ , \ q_2 \to q}{\{p\}S_1;S_2\{q\}}$$

Soundness and completeness of this rule follow from the preceding theorem. Its soundness is proved by observing that $\{p_i\}S_i\{q_i\}$ holds iff there exist proofoutlines $pfo(p_i,S_i,q_i)$, for i=1,2. By the definition of proofoutline, $\{p\}pfo(p_1,S_1,q_1);pfo(p_2,S_2.q_2)\{q\}$ is a proofoutline for $\{p\}S_1;S_2\{q\}$, and hence, by the theorem, $\{p\}S_1;S_2\{q\}$ holds. The completeness proof is equally simple.

Similarly, rules for assignment, conditional, and while statement can be formulated. By adding to these rules a proofsystem for properties of the underlying states in order to prove the implications whose proofs are required by these rules, one obtains a proofsystem in the sense of Hoare. For more about Hoare style proofsystems, consult Apt's authoritative "Ten years of Hoare's logic" [Apt].

4. Compositionality

The main reason for first presenting Floyd's method for flowdiagrams, then proofoutlines for while statements and finally a corresponding proofsystem in the sense of Hoare, is to illustrate the notion of compositionality, and to suggest soundness, of course. This leads to the following definition.

Definition [Compositional proofsystem] A compositional proofsystem for proving $\{p\}S\{q\}$ is a proofsystem in the sense of logic in which a proof for $\{p\}S\{q\}$ is obtained by composing proofs for S's constituent parts. □

Examples involving compositionality:

•A Hoare style proofsystem for while programs is a compositional proofsystem.
•Floyd's method is not compositional since it is not syntax-directed. Although a syntax for flowdiagrams is feasible, Floyd's method is based on a proof of local correctness, whose proof obligations reduce immediately to proofs about the atomic constituents of a diagram, and not to proofs about its constituent parts.
•Proofoutlines as defined in section 2 are compositional, but represent no proofsystem.

5. Generalization of Floyd's verification method to concurrent programs

Our next goal is to generalize Floyd's verification method for sequential while programs to concurrent while programs operating upon shared variables. To reach this goal, first concurrent execution is characterized by nondeterministic sequential interleavings of atomic actions in section 6. Next, section 7 contains a critical discussion, mentioning some alternative characterizations of "true" concurrency. In section 8 Floyd's verification method is generalized to the interleaving model.

6. A characterization of concurrent execution

In this section a model is developed for characterizing concurrency in concurrent programs operating upon shared variables.
•Requirement 1: More than one process cannot have simultaneous access to the same memory location.
•Requirement 2: The execution speed of every nonterminated process is positive and arbitrary.
•Example 1: {true}P::[x:=0 || (x:=1;x:=2)]{x=0∨x=2} holds, for execution of P amounts to execution of one of the following sequences of atomic actions, as follows from the two requirements above: {x:=0;x:=1;x:=2 , x:=1;x:=0;x:=2 , x:=1;x:=2;x:=0}.
•Example 2: {x=0}Q::[Q_1::x:=x+1 || Q_2::x:=x+1]{x=2} does not hold, because execution of Q is not equivalent with the sequential execution of x:=x+1;x:=x+1 since x:=x+1 is not atomic.
Assume execution of x:=x+1 to be equivalent with execution of ti:=x;ti:=ti+1;x:=ti with ti standing for a local register of Q_i, i=1,2. Then ti:=x, ti:=ti+1 and x:=ti are considered as atomic actions, since ti is local to Q_i, and at most one access to a shared memory reference occurs per action.
Now, {x=0}Q{x=1∨x=2} holds, for execution of Q is equivalent with a nondeterministic interleaving of the atomic actions of
ti:=x;ti:=ti+1;x:=ti , for i=1,2 , such as
t1:=x;t2:=x;t2:=t2+1;x:=t2;t1:=t1+1;x:=t1 , or
t2:=x;t2:=t2+1;x:=t2;t1:=x;t1:=t1+1;x:=t1 .

This suggests as central question: *When has the concurrent execution of processes*

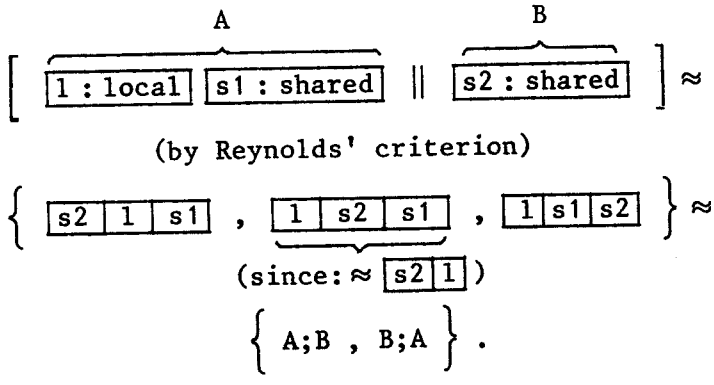
the same net effect on variables as the nondeterministic interleaved execution of the statements of those processes?

Answer: [Reynolds'criterion] Concurrent execution of processes $P_1,...P_n$ has the same net effect on program variables as the nondeterministic interleavings of their tests and assignments, provided every assignment and test contains at most one shared memory reference. □

Let a *critical reference* (of $P_1,...P_n$) denote an occurrence of a shared variable (of $P_1,...P_n$). Furthermore, let every test and assignment of P_i contain at most one critical reference. Since local actions on local variables in one process commute with every action from other processes, the following principle holds:

[Generalized criterion] Join in every process P_i statements together to groups containing at most one critical reference. Then nondeterministic interleaved execution of those groups is equivalent with (i.e., has the same net effect upon program variables as) concurrent execution of $[P_1 \| ... \| P_n]$. □

Example of the generalized criterion:



In this example "≈" denotes equivalence as far as the net effect on program variables is concerned. □

When several processors are involved in executing $[P_1 \| ... \| P_n]$, then the set of all nondeterministic interleavings of the sequences of atomic actions of $P_1,...P_n$ needs not faithfully model concurrent execution of $P_1,...P_n$, since not necessarily all of these interleavings are realistic. To remedy this situation, this set is pruned by imposing appropriate *fairness restrictions*.

To explain this point, first two additional aspects of concurrent programming must be incorporated within our model of concurrency:

(1) the semantics of boolean tests in sequential and in concurrent programming is different, and

(2) the additional use of indivisible operations to express synchronization between concurrent processes.

Ad (1): - In sequential programming boolean tests serve to decide between two

possibilities for *advancing* control flow; e.g., the value of b in if b then S fi determines whether S is executed or passed.

- In concurrent programming boolean tests serve as traffic lights for deciding *whether or not* control flow advances. (That sequential conditionals, as the one above, can still be expressed using this notion of tests can be understood by expressing them using guarded selections with simultaneous evaluation of the guards.)

Ad (2): Let <T> signify that statement T should be executed as an indivisible operation, i.e. without interleaving actions from other processes. Then synchronization between concurrent processes can be expressed using the semaphore operation P(s)::<if s>0 then s:=s-1 fi>.

Consequently, in sequential programming P(s) would always terminate, whereas in concurrent programming this is possibly not the case.

To continue our explanation, let us make the following assumptions:

- P(s) operations are executed on separate processors with similar properties.
- Concurrent execution is modeled by nondeterministic interleaving of atomic actions in which tests act as traffic lights, and execution of <T> is indivisible.

Observe that in this interleaving model both Reynolds' and the generalized criterion still hold.

Now the point raised above, that this model does not faithfully reflect "true" concurrency, is due to the fact that it admits the following unrealistic execution sequence:

> An infinite sequence in which the test s>0 in operation P(s) happens to be evaluated only when s=0 happens to be true, whereas s>0 is also infinitely often true in that sequence (due to concurrent activity).

During truly concurrent execution (as is the case when several processors are involved), this sequence should not occur in case P(s) operations test s>0 at *random* moments (as permitted by the arbitrariness of execution speeds - requirement 2 of section 6 -) since the fact that s>0 holds infinitely often implies that the probability of its occurrence is zero.

This discrepancy between the interleaving model and concurrent execution on several processors is removed by imposing fairness assumptions. An example of such an assumption is requirement 3:

Requirement 3: If an operation is infinitely often enabled in an execution sequence (i.e., its associated test is infinitely often true) then it is eventually performed.

Example: Under this fairness assumption the following program terminates:

{x=1∧s=1}[(P(s);x:=x-1;s:=s+1) || while x>0 do P(s);s:=s+1 od] . □

7. Is this characterization of concurrent execution justified?

Imagine a nondeterministic interleaving of the atomic actions of a symphony of Mozart's; obviously the result is cacophony.
Thus, if one's goal is "to yeeld to sweet Musick", then the nondeterministic interleaving of atomic actions is not justified as a model for concurrency.
Petrinet theory supports the point of view that concurrency adds an essentially new element to computation.

Lamport writes in "What good is temporal logic" [Lamport 2]:

> "Computer scientists often feel that something is lost by sequentializing a concurrent program ... and that one should use a partial ordering among actions, instead. However, as long as we consider safety and liveness properties only, there is no loss of generality in considering totally ordered sequences of atomic actions. Our model includes all possible sequences, and a partial ordering is completely equivalent with the set of all total orderings consistent with it. The real assumption implicit in the model is the existence of atomic actions.
> This assumption is made in virtually all formal models of concurrency ... "

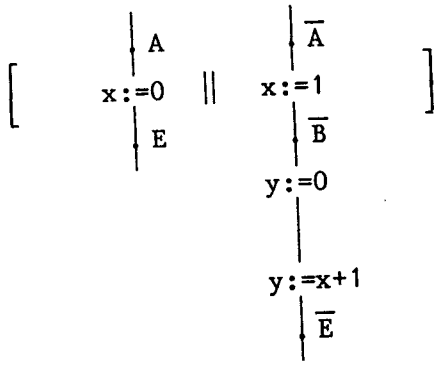
Realtime concurrent execution is modeled using Salwicki's model of maximal parallelism [S] in which the number of instructions in concurrent processes, which can be executed simultaneously without violating synchronization restrictions, is maximalized. For concurrent realtime ADA a spectrum of models, varying from pure maximal parallelism to purely nondeterministic sequential interleavings of atomic actions, is described in [S deR G K A].

8. The generalization of Floyd's verification method to nondeterministic interleavings

The essence of Floyd's method is the observation that a partial correctness proof can be reduced to checking finitely many verification conditions, i.e., for every elementary path BC : if execution arrives at B, and the associated predicate p_B holds, and if path BC is taken, then p_C holds at C. This observation is based upon the fact that every execution sequence is equivalent with a sequence of elementary paths taken from a fixed, finite, collection.
Since concurrent execution is now modeled by nondeterministic interleavings of atomic actions the incorporation of concurrency within Floyd's method dictates that every nondeterministic interleaving of atomic actions should be described using such elementary paths. According to the generalized criterion, this concerns interleavings of groups of statements containing together at most one critical reference.
Conclusion: *an elementary path should contain at most one critical reference.*
Thus, execution of $[P_1 \| \ldots \| P_n]$ is characterized by the interleaved execution of such elementary paths taken from a fixed, finite, collection.

Furthermore, during execution of an elementary path contained in P_i no execution is modeled in P_j, for $j \neq i$. This implies that an elementary path of $[P_1 \| \ldots \| P_n]$ contained in P_i is characterized by two n-triples of *composite* locations, beginning in an n-triple $X_1X_2 \ldots X_iX_{i+1} \ldots X_n$ and ending in an n-triple $X_1X_2 \ldots Y_iX_{i+1} \ldots X_n$ with $X_1, \ldots X_n$ and Y_i locations in $P_1, \ldots P_n$ and P_i, and elementary path X_iY_i in P_i containing at most one critical reference.

Example:



- composite locations: $A\bar{A}$, $A\bar{B}$, $A\bar{E}$, $E\bar{A}$, $E\bar{B}$, $E\bar{E}$
- elementary paths : $\underbrace{(A\bar{A}, E\bar{A})}_{\bar{A}\text{ fixed}}, \underbrace{(A\bar{B}, E\bar{B})}_{\bar{B}\text{ fixed}}, \underbrace{(A\bar{E}, E\bar{E})}_{\bar{E}\text{ fixed}}, \ldots$

AE contains one critical reference

□

Therefore, since elementary (and execution) paths are now described by composite locations, Floyd's method is generalized to nondeterministic interleavings by:

- associating predicates with composite locations,
- proving for elementary path α from $X_1 \ldots X_i \ldots X_n$ to $X_1 \ldots X_{i-1}Y_iX_{i+1} \ldots X_n$

$$\left\{P_{X_1 \ldots X_n}\right\} \alpha \left\{P_{X_1 \ldots X_{i-1}Y_iX_{i+1} \ldots X_n}\right\}$$

for all $i=1, \ldots n$, and all $X_1, \ldots X_n$ and Y_i.

Observe that the definitions of path transformation $\alpha(x)$ and path condition $C(\alpha)(x)$ remain the same.

This leads to the following formulation of Floyd's method generalized to concurrency:

Floyd's method of inductive assertions for verifying the concurrent composition $[P_1 \| \ldots \| P_n]$ *of sequential schemes* P_i

1. Associate every scheme P_i with a *correct**) labeling *Labels* (P_i) s.t. every 2 critical references are separated by a label.
2. Associate an assertion $p_{A_i^1 \ldots A_i^n}$ with every tuple $(A_i^1, \ldots A_i^n)$ s.t. $A_i^k \in$ *Labels* (P_k).

*) A labeling is correct if every execution path is a sequence of associated elementary paths, and the set of elementary paths is finite.

3. Prove for every elementary path

$$(A_i^1,\ldots A_i^n) \overset{\alpha}{\to} (A_j^1,\ldots A_j^n)$$

with $A_i^k=A_j^k$ excepting possibly one value of k,

that $\left\{p_{A_i^1\ldots A_i^n}\right\}\ \alpha\ \left\{p_{A_j^1\ldots A_j^n}\right\}$ holds. □

Then $\left\{p_{A^1\ldots A^n}\right\}[P_1\|\ldots\|P_n]\left\{p_{E^1\ldots E^n}\right\}$ holds, with A^i,E^i denoting, respectively, beginning and end point of P_i, i=1...n.

However, this formulation of Floyd's method is unsatisfactory for it doesn't enable one to understand the annotated schemes π_i in isolation. This leads to the following question:

Suppose every P_i *is associated with a set of assertions which is locally correct w.r.t. sequential execution of* P_i *in isolation. Then, what else must one prove to conclude partial correctness of* $[P_1\|\ldots\|P_n]$ *?*

Consider a composite location $A_1\ldots A_n$. With every single label A_i an assertion p_i has been associated in the local labeling of P_i.

The intention is to associate $p_{A_1}\wedge\ldots\wedge p_{A_n}$ with $A_1\ldots A_n$.

Consider elementary path $(A_1\ldots A_n)\overset{\alpha}{\to}(A_1\ldots A_{i-1}\overline{A}_iA_{i+1}\ldots A_n)$ with associated verification condition

$$\left\{\underset{j=1}{\overset{n}{\hat{=}}} p_{A_j}\right\}\ \alpha\ \left\{\underset{j\neq i}{\underset{j=1}{\overset{n}{\hat{=}}}} p_{A_j}\wedge p_{\overline{A_i}}\right\}\quad\ldots\quad(*)$$

Now observe that (*) holds only if p_{A_j} is invariant under state transformation $\alpha(\vec{x})$, for j≠i. Since the local correctness of the associated assertion pattern of P_i implies already that $\{p_{A_i}\}\ \alpha\ \{p_{\overline{A_i}}\}$ holds, (*) reduces to checking $\left\{\underset{j=1}{\overset{n}{\hat{=}}} p_{A_j}\right\}\alpha\left\{p_{A_k}\right\}$, for k≠i.

Owicki has shown that it is sufficient to check her famous *interference freedom test*:

> [Interference freedom test] For all A_i,A_k and α, such that A_i is a location in P_i, α an elementary path in P_i beginning in A_i, and A_k a location in P_k, for k≠i: $\left\{p_{A_i}\wedge p_{A_k}\right\}\alpha\left\{p_{A_k}\right\}$.

Since only assignments change the state, interference freedom checks need only be done for paths α containing an assignment to a shared variable.

This leads to the *second formulation of Floyd's method for verifying concurrent execution of* $[P_1\|\ldots\|P_n]$:

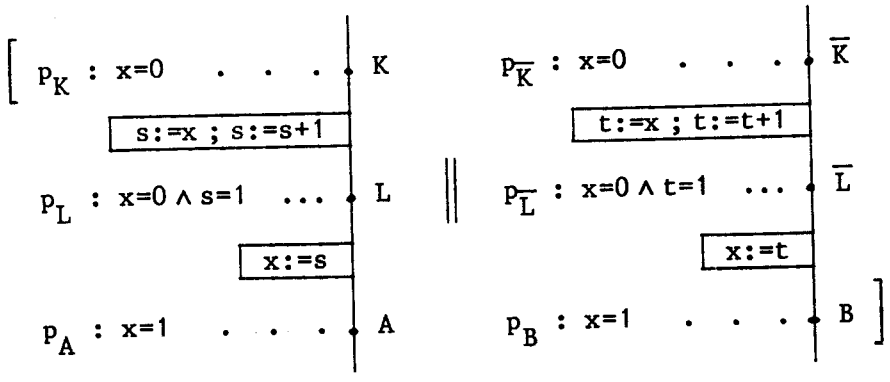
To prove $\{p\}[P_1\|\ldots\|P_n]\{q\}$,

1. Associate with every P_i an assertion pattern such that every pair of

critical references is separated.

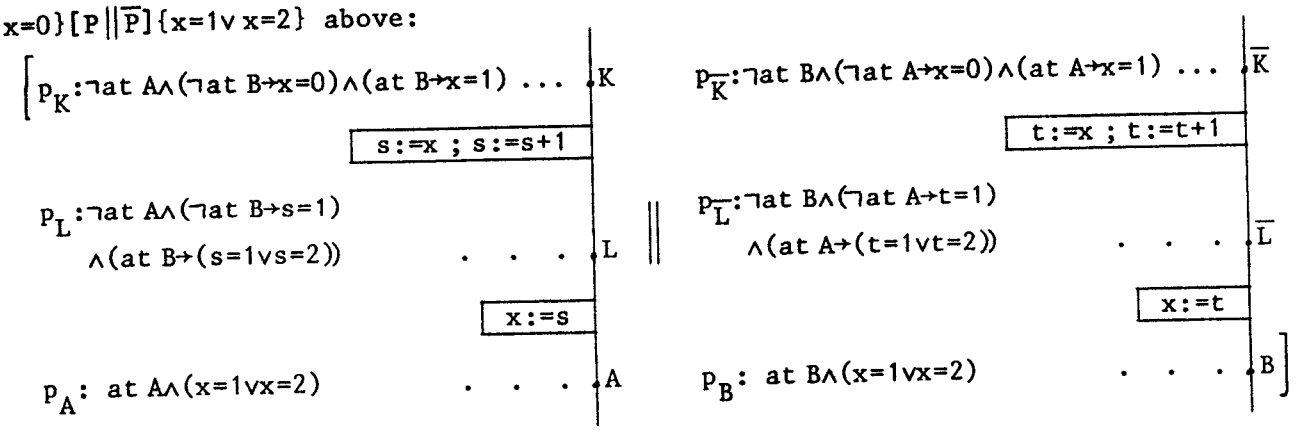
2. Prove local correctness of the assertion pattern associated with the sequential diagrams P_i.
3. Prove $p \rightarrow p_{A_1} \wedge \ldots \wedge p_{A_n}$ and $p_{E_1} \wedge \ldots \wedge p_{E_n} \rightarrow q$ with A_i, respectively, E_i denoting beginning and end points of P_i.
4. Prove, for every elementary path α in P_i containing an assignment to a shared variable whose beginning point is associated with assertion p, that for every assertion q from P_j, for $j \neq i$, holds: $\{p \wedge q\}\, \alpha\, \{q\}$ ... *interference freedom of* q ... for j=1,...n.

But ... there is still something rotten in the State of Denmark ... .
Consider, e.g., $[P::s:=x;s:=s+1;x:=s \,\|\, \overline{P}::t:=x;t:=t+1;x:=t]$ satisfying $\{x=0\}[P \,\|\, \overline{P}]\{x=1 \vee x=2\}$.
This cannot be proved using assertions containing x, s and t as free variables only.
E.g., in



the assertion patterns are locally correct but not interference free since p_K in P is invalidated by x:=t in $\overline{P}$. (A proof of this element of incompleteness is contained in Owicki's thesis.)

In order to express interference free predicates, the assertion language must be enriched with auxiliary quantities expressing that in a process control resides at a certain location, so called *location predicates*.
Use predicate constants at A, at B ... to express that control resides at labels A,B,... .
Then the following locally correct assertion pattern is interference free and proves $\{x=0\}[P \,\|\, \overline{P}]\{x=1 \vee x=2\}$ above:



9. Proofoutlines for concurrent processes - the method of Owicki & Gries

In the previous section tacit use has been made of the properties of location predicates. Now, we must formalize these properties, since, otherwise, no syntactic version of Floyd's method is possible. For without auxiliary quantities such as location predicates one cannot express interference free assertions.
Now that Floyd's method has been extended to concurrency, proofoutlines for concurrent programs must be formulated. And then the formulation of the proofobligations thus formulated must be presented. As we shall see in section 10, no compositional (syntax-directed) proofsystem for concurrency, which is syntactically based on Hoare triples only, exists.
The system presented here is that of [Owicki & Gries]. Although formulated (in their paper) as proofsystem based on Hoare triples {p}S{q}, their system turns out to be syntactically based on proofoutlines since the intermediate assertions used in proving {p}S{q} must be available syntactically in order to express the interference freedom tests.

In Owicki & Gries's proofsystem location predicates are modeled by fantomvariables. These are variables occurring in a program solely for the purpose of expressing certain assertions. Therefore they should not influence control flow.
Definition [Fantomvariables] Fantomvariables occur only in assignments to fantomvariables themselves, i.e., they do not occur in assignments and tests to variables which possibly determine flow-of-control. □
Let S contain at most one critical reference. Floyd's use of location predicates at A and at B in ... A:S ; B: ... is modeled as follows using correspondingly named local boolean fantomvariables at A and at B:

... A:S ; B: ... ⇒ ... {at A ∧¬at B ∧...} A:<at A:=false ; S ; at B:=true >; B ...

The use of notation <T> (originally suggested by Lamport), denoting indivisible execution of T, is here justified by the generalized criterion.

Our first goal is to formulate an extension of the definition of proofoutline to cover concurrency.
Example: The following, representing a proofoutline for the annotated scheme $[P \| \overline{P}]$ of the previous section, suggests what is intended:

{x=0∧¬at A∧¬at B}

[{¬at A∧(¬at B→x=0)∧(at B→x=1)}<s:=x;s:=s+1> ; {¬at A∧(¬at B→s=1)∧(at B→s=1∨s=2)}
<x:=s ; at A:=true >; A:{at A∧(x=1∨x=2)}

‖

{¬at B∧(¬at A→x=0)∧(at A→x=1)}<t:=x;t:=t+1> ; {¬at B∧(¬at A→t=1)∧(at A→t=1∨t=2)}
<x:=t ; at B:=true> ; B:{at B∧(x=1∨x=2)}

] {x=1 ∨ x=2}

with associated interference freedom tests such as

{¬at A∧(¬at B→x=0)∧(at B→x=1)∧¬at B∧(¬at A→t=1)∧(at A→t=1∨t=2)}x:=t;at B:=true

{¬at A∧(¬at B→x=0)∧(at B→x=1)}.

This test is satisfied since

- ¬at A∧(¬at B→x=0)∧(at B→x=1)∧¬at B∧(¬at A→t=1)∧(at A→t=1∨t=2) ⇒ ¬at A∧¬at B∧t=1, and
- {¬at A∧¬at B∧t=1}x:=1;at B:=true {¬at A∧(¬at B→x=0)∧(at B→x=1)} ⇔ (by Hoare's assignment axiom)

  (¬at A∧¬at B∧t=1)→(¬at A∧(¬true→1=0)∧(true→1=1)), q.e.d.

□

Proofoutlines are generalized to concurrency by extending the definition of proof-outline in section 2 with the following clauses:

- S≡<S1>:{p}<pfo(p1,S1,q1)>{q} is a proofoutline for {p}S{q} iff p→p1 and q1→q.
- S≡[S1||...||Sn] : {p}[pfo(p1,S1,q1)||...||pfo(pn,Sn,qn)]{q} is a proofoutline for {p}S{q} iff:
  - Every assignement and test in S1,...Sn, insofar not enclosed by "<" and ">", contains at most 1 critical reference of S1,...Sn.
  - p→∧pi , ∧qi→q.
  - Every proofoutline pfo(pi,Si,qi) is interference free, i.e. for every two assertions ri∈pfo(pi,Si,qi) and rj∈pfo(pj,Sj,qj) , i≠j, with rj a precondition of an assignment α to a shared variable or of a statement α of the form <α1> with α1 containing a critical reference,

    there *exists* a proofoutline for {ri∧rj}α{ri}.

    This applies to assertions ri not enclosed between "<" and ">", only.

□

Again, the following theorem can be proved for sufficiently strong assertion languages:

Theorem: {p}S{q} holds iff there exists a proofoutline for {p}S{q}. □

Next, observe that the following problem is associated with this definition:

In the example above we suggested a proofoutline for

```
{ x=0∧¬at A∧¬at B }
[ <s:=x;s:=s+1>;<x:=s;at A:=true>
|| <t:=x;t:=t+1>;<x:=t;at B:=true>
] {x=1∨x=2}
```

... (*)

However, our aim is to prove:

{x=0}[(s:=x;s:=s+1;x:=s) || (t:=x;t:=t+1;x:=t)]{x=1∨x=2}... (****)

This poses the following questions:

- How to get rid of at A,at B, and possibly "<" and ">" in (*)?
- How to justify x=0 as precondition on the basis of (*), since (x=0→x=0∧¬at A∧¬at B) does not hold.

The answer will be provided in three stages.

Question 1: How to get rid of at A and at B in (*)?

Answer (first stage):

- at A and at B are fantomvariables.
- Therefore they do not influence control flow.
- at A and at B do not occur in postcondition x=1∨x=2.
- Hence, (*) implies that (**) below holds:

$$\left.\begin{array}{l}\{x=0 \wedge \neg \text{at A} \wedge \neg \text{at B}\} \\ [\ \langle s:=x;s:=s+1\rangle;\langle x:=s\rangle \\ \|\ \langle t:=x;t:=t+1\rangle;\langle x:=t\rangle \\ ]\ \{x=1 \vee x=2\}\end{array}\right\} \ldots(**)$$

□

The general form of the rule applied above is:

[*Owicki & Gries's fantomvariable rule*]:

$$\frac{\{p\}S\{q\}}{\{p\}S'\{q\}},$$

provided q *contains no fantomvariables, and* S' *is obtained from* S *by deleting assignments to fantomvariables.*

□

Question 2: How to justify x=0 as precondition on the basis of (**)?

Answer (second stage): (**) is of the form {p}S{q} such that:

- Neither at A nor at B occur in S or q.
- Hence, execution of S is independent of the values of at A and at B, as is satisfaction of q.
- Hence, satisfaction of {p}S{q} is independent of what p expresses about the values of at A and at B.
- Therefore one can substitute arbitrary values for at A and at B in p without affecting validity of (**).
- Substitute, respectively, false and false for at A and at B in p in (**), resulting in:

  $\{x=0\}[(\langle s:=x;s:=s+1\rangle;\langle x:=s\rangle) \| (\langle t:=x;t:=t+1\rangle;\langle x:=t\rangle)]\{x=1 \vee x=2\}\ldots$ (***)

□

The general form of the rule applied above is:

[*Substitution rule II*]

$$\frac{\{p\}S\{q\}}{\{p[\vec{exp}/\vec{z}]\}S\{q\}};$$

provided $\vec{z} \cap$ *free var* $(S,q)=\emptyset$, *and* $\vec{exp}$ *denotes a sequence of expressions of the same length as* $\vec{z}$.

□

Question 3: How to get rid of "<" and ">" in (***)?

Answer (third stage):

• By adapting the generalized criterion to Hoare triples.

• By specifying which atomic actions are involved in executing an assignment.

□

Thus one may derive (****).

Finally, the diligent reader might have noticed that we extended Floyd's use of location variables only to statements S containing at most one critical reference. How is this use extended to move general constructs? The solution given below models location predicates for conditionals and iteration by a dynamic extension of the use of the "indivisibility" brackets "<" and ">".

• Conditionals:

...A: if p then Q:...;Q': else R:...;R': fi ;A': ⇒

...A:<at A:= false; if p then at Q:= true> Q:...Q': <at Q':= false
else at R:= true> R:...R': <at R':= false
fi ; at A':= true >; A':...

Here p should contain at most one critical reference, and the following constructs are regarded as indivisible:

- if p(x) is true: <at A:=false;p;at Q:=true> and <at Q':=false;at A':=true> ,
- if p(x) is false:<at A:=false;p;at R:=true> and <at R':=false;at A':=true> .

• Iteration:

...A: while p do Q:...Q':od;A':... ⇒

...A:<at A:=false;while p do at Q:=true>;Q:...Q':<at Q':=false
od;at A':=true>;A':...

Again, p should contain at most one critical reference; the following constructs are regarded as indivisible:

- upon first evaluation of p, if p(x) is true:<at A:=false;p;at Q:=true> ,
- upon later evaluation of p, if p(x) is true:<at Q':=false;p;at Q:=true> ,
- upon first evaluation of p, if p(x) is false:<at A:=false;p;at A':=true> ,
- upon later evaluation of p, if p(x) is false:<at Q':=false;p;at A':=true> .

10. There exists no syntax-directed proof rule for concurrent composition, based on Hoare triples

A compositional proof system for a language involving concurrency should contain a compositional (syntax-directed) proof rule for concurrent composition. Since up to now our proof rules have been based on Hoare triples {p}S{q}, we look for a proof rule for concurrent composition which is based on Hoare triples and which, therefore, reduces the proofobligations for $\{p\}[S_1\|\ldots\|S_n]\{q\}$ to proofobligations for $\{p_i\}S_i\{q_i\}$, for appropriate p_i and q_i, i=1...n. These proofobligations have been formulated in the preceding section, and are as follows:

(1) Every assignment and test in $S_1,\ldots S_n$, provided not enclosed in between "<"

and ">" brackets, should contain at most one critical reference.

(2) $p \rightarrow \wedge p_i, \wedge q_i \rightarrow q$.

(3) There are proofoutlines for $\{p_i\}S_i\{q_i\}$, $i=1,\ldots n$.

(4) These proofoutlines are interference free.

Proofobligations (2) and (3) suggest the following rule:

$$\frac{\{p_i\}S_i\{q_i\},\ i=1,\ldots n}{\{\wedge p_i\}[S_1 \| \ldots \| S_n]\{\wedge q_i\}}$$

This rule is sound for disjoint S_i, i.e. in case no variables are shared between any two S_i and S_j, $i \neq j$.
However, this rule is unsound when variables are shared, as indicated by clause (4). For every S_i, this clause requires a list of interference freedom proofs of the assertions occurring in the proofoutline for S_i, with respect to $S_1,\ldots S_{i-1}, S_{i+1}, \ldots S_n$. The assertions occurring in these lists concern *the way* in which $\{p_i\}S_i\{q_i\}$ has been derived, and are, as such, not derivable from validity of $\{p_i\}S_i\{q_i\}$ itself. Thus, the assertions required for formulating interference freedom tests cannot be recovered from the list $p_1,\ldots p_n, S_1,\ldots S_n, q_1,\ldots q_n$. Consequently, no syntax-directed proof rule à la Hoare exists for concurrent composition which is syntactically based on the usual notion of Hoare triples.

The only chance left for such a rule lies in reformulating the notion of program correctness as expressed by statement-assertions pairs.
This new notion should involve:

- the atomic state transitions occurring in a statement,
- the pre- and postconditions associated with these atomic transitions,
- the interference freedom tests associated with every atomic transition.

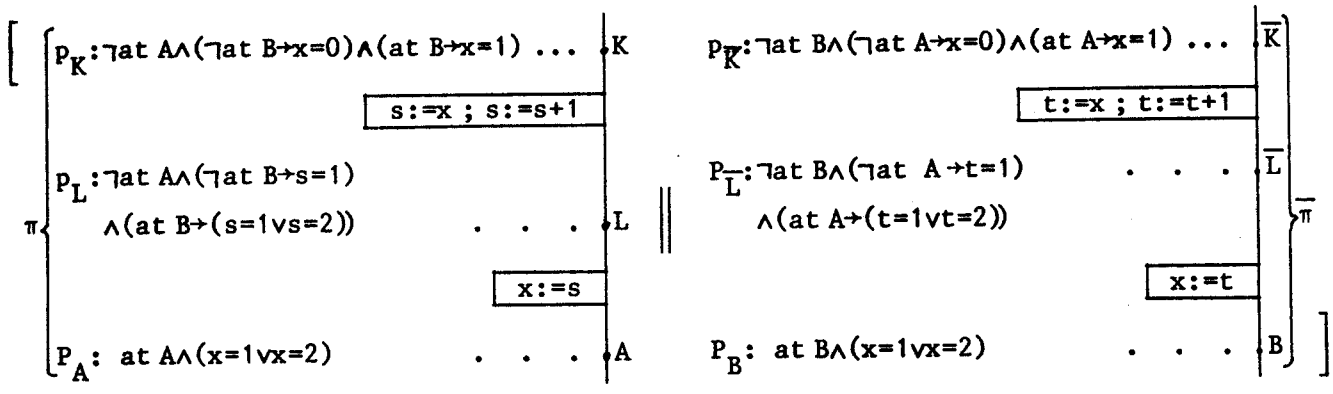
Such a notion is provided by Lamport's "Generalized Hoare Logic" [Lamport 1], whose bare essentials are discussed in section 11.

11. A syntax-directed proof rule for concurrent composition - Lamport's Generalized Hoare Logic.

The message of section 10 was that there exists no compositional proof rule for concurrent composition which is syntactically based on Hoare triples. Yet the fact, that $\{p\}[S_1 \| \ldots \| S_n]\{q\}$ holds iff there exists a proofoutline for this triple, suggests the following proof strategy for concurrent composition:
Construct for $i=1,\ldots n$ proofoutlines $pfo(p_i,S_i,q_i)$ for $\{p_i\}S_i\{q_i\}$ which are interference free w.r.t. each other, to conclude that
$\{\wedge p_i\}[pfo(p_1,S_1,q_1) \| \ldots \| pfo(p_n,S_n,q_n)]\{\wedge q_i\}$ is a proofoutline, and that this proofoutline implies $\{\wedge p_i\}[S_1 \| \ldots \| S_n]\{\wedge q_i\}$.
The point here is, that from n interference free objects (proofoutlines) for

$S_1,\ldots S_n$ one can construct a new object (again a proofoutline) for the concurrent composition of $S_1,\ldots S_n$. This suggests to look for a compositional proof rule for concurrency *which is syntactically based on proofoutlines.*

Consider again the verification à la Floyd of
$\{x=0\}[P::<s:=x;s:=s+1>;<x:=s>\,\|\,\overline{P}::<t:=x;t:=t+1>;<x:=t>]\{x=1\vee x=2\}$.
It uses the following assertion pattern:



Define I_π by $I_\pi \equiv (\text{at } K \to p_K)\wedge(\text{at } L \to p_L)\wedge(\text{at } A \to p_A)$, and $I_{\overline{\pi}}$ by $I_{\overline{\pi}} \equiv (\text{at } \overline{K} \to p_{\overline{K}})\wedge(\text{at } \overline{L} \to p_{\overline{L}})\wedge(\text{at } B \to p_B)$.
Local correctness of assertion pattern π amounts to *invariance of* I_π *under the atomic actions* $<s:=x;s:=s+1>$ *and* $x:=s$ *of* P.
Observe the fact that $\{p_K\}<s:=x;s:=s+1>\{p_L\}x:=s\{p_A\}$ is a proofoutline for P and this invariance are equivalent requirements.
Similarly, interference freedom of π w.r.t. $\overline{\pi}$, or rather,
$\{p_K\}<s:=x;s:=s+1>\{p_L\}x:=s\{p_A\}$ w.r.t. $\{p_{\overline{K}}\}<t:=x;t:=t+1>\{p_{\overline{L}}\}x:=t\{p_B\}$,
is equivalent with *invariance of* I_π *under* $\overline{P}$*'s atomic actions.*
The same holds mutatis mutandis for $I_{\overline{\pi}}$ and $\overline{P}$.
Thus, invariance of $I_\pi \wedge I_{\overline{\pi}}$ under both the atomic actions of P, and of $\overline{P}$ implies invariance of $I_\pi \wedge I_{\overline{\pi}}$ under the atomic actions of $[P\,\|\,\overline{P}]$. Or, using appropriate notation: $\dfrac{I_\pi \wedge I_{\overline{\pi}}\ \text{sat}\ P,\ I_\pi \wedge I_{\overline{\pi}}\ \text{sat}\ \overline{P}}{I_\pi \wedge I_{\overline{\pi}}\ \text{sat}\ [P\,\|\,\overline{P}]}$, which is precisely the syntax-directed pattern we were looking for!

To summarize the discussion:

- Let S sat ϕ hold iff ϕ is invariant under the atomic actions of S.
- Let S_1 sat ϕ and S_2 sat ϕ together, i.e. ϕ is invariant under the atomic actions of both S_1 and S_2, imply $[S_1\,\|\,S_2]$sat ϕ, since the atomic actions of $[S_1\,\|\,S_2]$ are those of S_1 and S_2.

  This suggests the following syntax-directed proof rule for parallel composition:

$$\frac{S_1\ \text{sat}\ \phi,\ldots S_n\ \text{sat}\ \phi}{[S_1\,\|\ldots\|\,S_n]\text{sat}\ \phi}$$

The merit of Lamport's notion of validity of S sat ϕ, and its resulting syntax