VERIFICATION OF CONCURRENT PROGRAMS, PART I. THE TEMPORAL FRAME--ETC(U)

JUN 81 Z MANNA; A PNUELI

UNCLASSIFIED

STAN-CS-81-836
Verification of Concurrent Programs,
Part I. The Temporal Framework

by

Zohar Manna
Amir Pnueli

Research sponsored by
Office of Naval Research

Department of Computer Science
Stanford University
Stanford, CA 94305

This document has been approved for public release and sale; its distribution is unlimited.
VERIFICATION OF CONCURRENT PROGRAMS:
THE TEMPORAL FRAMEWORK

by

ZOIIAR MANNA
Computer Science Department
Stanford University
Stanford, CA
and
Applied Mathematics Department
The Weizmann Institute
Rehovot, Israel

AMIR PNUELI
Applied Mathematics Department
The Weizmann Institute
Rehovot, Israel

ABSTRACT

This is the first in a series of reports describing the application of temporal logic to the
specification and verification of concurrent programs.

We first introduce temporal logic as a tool for reasoning about sequences of states. Models
of concurrent programs based both on transition graphs and on linear-text representations are
presented and the notions of concurrent and fair executions are defined.

The general temporal language is then specialized to reason about those execution sequences
that are fair computations of a concurrent program. Subsequently, the language is used to describe
properties of concurrent programs.

The set of interesting properties is classified into invariance (safety), eventual (liveness),
and precedence (until) properties. Among the properties studied are: partial correctness, global
invariance, clean behavior, mutual exclusion, absence of deadlock, termination, total correctness,
intermittent assertions, accessibility, responsiveness, safe liveness, absence of unsolicited response,
fair responsiveness, and precedence.

In the following reports of this series, we will use the temporal formalism to develop proof
methodologies for proving the properties discussed here.

A preliminary version of this paper appears in The Correctness Problem in Computer Science (R.
S. Boyer and J. S. Moore, eds.), International Lecture Series in Computer Science, Academic Press,

This research was supported in part by the National Science Foundation under grants MCS79-
00495 and MCS80-06030, by the Office of Naval Research under Contract N00014-76-C-0687, and
by the United States Air Force Office of Scientific Research under Grant AFOSR-81-0014.
INTRODUCTION

Temporal logic is a special branch of logic that deals with the development of situations in time. Whereas ordinary logic is adequate for describing a static situation, temporal logic enables us to discuss how a situation changes due to the passage of time. An execution of a program is precisely a chain of situations, called execution states, that undergo a series of transformations determined by the program's instructions. This suggests that temporal logic is an appropriate tool for reasoning about the execution of programs. The special advantage of this approach is that it enables us to formalize the entire execution of a program and not just the function or relation it computes.

The temporal logic approach offers special advantages for the formalization and analysis of the behavior of concurrent programs. Concurrent programs have long been a difficult subject to formalize and have often defied generalization of methods that worked perfectly for sequential programs.

One inherent difficulty in analysing a concurrent program is that when combining two processes to be run in parallel, we cannot infer the input-output relation computed by the combined program from just the input-output relations computed by each of the individual component processes. The obvious reason for this is that, running in parallel, the processes may interfere with one another, altering the behavior each would have when run alone. Consequently, in order for any approach to stand a chance of success, it must deal with more than the input-output relation computed by a program. It should be concerned with execution sequences in one form or another, as well as be able to discuss mid-execution events.

Another inherent difficulty is the discontinuity associated with the simulation of concurrency by multiprogramming. A very convenient and widely used model of real concurrency is to regard the participating events as composed of many atomic basic steps. Then instead of requiring that these basic steps occur concurrently, we consider sequences in which these steps are interleaved in all possible ways. The problem with modelling concurrency by multiprogramming (interleaving) is that without further restrictions a certain process can be discriminated against by having its execution continually delayed. Disallowing this discrimination introduces a discontinuity into the set of interleaved execution sequences.

Consequently, any approach which is based strongly on the concept of continuity, such as the denotational approach or equivalent relational ones, is bound to face severe difficulties when extended to deal with concurrency.

Temporal logic avoids both these difficulties by (a) being geared from the start to analyse and formalize properties in terms of execution sequences, and (b) not being based on limits and assumptions of continuity. In fact, it can very easily and naturally express such concepts as "eventually" which describes an event arbitrarily ahead in the future, but still a finite duration away.

In this report we introduce the framework and language of temporal logic and demonstrate its appropriateness for describing properties of programs.

We start with an exposition of modal logic whose domain of interpretation is a set of states and (general) accessibility relations connecting these states. We then specialize to temporal logic which requires that the states form a linear discrete sequence. Linear discrete sequences can be
used to describe a dynamic process that goes through changes at discrete instants. Consequently, temporal logic is suitable for reasoning about such dynamic processes and their behavior in time.

Next, we present a model of concurrent programs. The basic model is based on several concurrent processes, each of which is given in the form of a transition graph or a linear-text program. Executions of concurrent programs are defined to be an interleaving of execution steps, each taken from one of the processes. We discuss the conditions under which an interleaved execution faithfully represents real concurrency. One of these conditions calls for the interleaving to be fair in that no process is neglected for too long.

We then show how the language of temporal logic can be further specialized to reason about execution sequences of programs. In this way, properties of programs which are expressible as properties of their execution sequences are readily formalisable.

The rest of the report overviews in a systematic manner the different properties of interest. They are classified into:

- **Invariance properties**, stating that some condition holds continuously throughout the computation.
- **Eventuality properties**, stating that under some initial conditions, a certain event (such as the program's termination) must eventually be realized.
- **Precedence properties**, stating that a certain event always precedes another.

For each class of properties, we present several typical and useful properties together with sample programs illustrating these properties.
1. THE GENERAL CONCEPTS OF TEMPORAL LOGIC

In the development of logic as a formalisation tool, we can observe an increasing ability to express change and variability. *Propositional Calculus* was developed to express constant or absolute truth, stating basic facts about the universe of discourse. The propositional framework mainly deals with the question of how the truth of a composite sentence depends on the truth of its constituents. In *Predicate Calculus* we deal with variable or relative truth by distinguishing the statement (the predicate) from its arguments. It is understood that the statement may be true or false according to the particular individuals it is applied to. Thus we may regard predicates as parameterized propositions. The *Modal Calculus* adds another dimension of variability to this description by predicates. If we contemplate a major transition in which not only individuals, but also the meaning of functions and predicates are changed, then the modal calculus provides a special notation for this major change. For instance, any chain of reasoning which is valid on Earth may become invalid on Mars because some of the basic concepts naturally used on Earth may assume completely different meanings (or become meaningless) on Mars. Conceptually, this calls for a partition of the universe of discourse into worlds of similar structure but different contents. Variability within a world is handled by changing the arguments of predicates, while changes between worlds are expressed by the special modal formalism.

Consider for example the statement: “It rains today”. Obviously, the truth of such a statement depends on at least two parameters: The date and the location at which it is stated. Given a specific date \( t_0 \) and location \( \ell_0 \), the specific statement: “It rains at \( \ell_0 \) on \( t_0 \)” has propositional character, i.e., it is fully specified and must either be true or false. We may also consider the fully variable predicate \( rain(\ell, t) \): “It rains at \( \ell \) on \( t \)” which gives equal priority to both parameters. The modal approach distinguishes two levels of variability. In this example, we may choose time to be the major varying factor, and the universe to consist of worlds which are days. Within each day we consider the predicate \( rain(\ell) \) which, given the date, depends only on the location. Alternatively, we can choose the location to be the major parameter and regard the raining history of each location as a distinct world.

As is seen from this example, the transition from predicate logic to modal logic is not as sharp as the transition from propositional logic to predicate logic. For one thing it is not absolutely essential. We could manage quite reasonably with our two parameter predicate. Second, the decision as to which parameter is chosen to be the major one may seem arbitrary. It is strongly influenced by our intuitive view of the situation.

In spite of these reservations there are some obvious advantages to the introduction and use of modal formalisms. It allows us to explicitly make one parameter more significant than all the others, and makes the dependence on that parameter implicit. Nowadays, when increasing attention is being paid to the clear correspondence between the syntactical structure of a program and its functional decomposition (as is repeatedly stressed by the discipline of structured programming), it seems only appropriate to introduce extra structure into the description of varying situations. Thus a clear distinction is made between variation within a world, which we express using predicates and quantifiers, and variation from one world to another, which we express using the *modal operators*.

Another way to view the generalization offered by modal logic is to claim that predicate calculus is appropriate for describing static situations. It gives statements about basic objects and their interrelation. The additional dimension provided by the modal logic is that of dynamic change from one situation into the other. One of the characteristics of changes due to time transitions is the fact that the same basic objects and entities exist in each of the static situations but that their
attributes and interrelations may change. Thus modal logic faithfully and conveniently portrays for us a dynamic situation consisting of a set of static situations and rules of change between them.

THE MODAL FRAMEWORK

The general modal framework ([HCH]) considers a universe that consists of many similar states (or worlds) and a basic accessibility relation between the states, $R(s, s')$, which specifies the possibility of getting from one state $s$ to another state $s'$.

Consider again the example of rainy days, with time taken to be the major parameter. There, each state in the universe is a day. A possible accessibility relation might hold between two days $s$ and $s'$ if $s'$ is in the future of $s$.

The main notational idea is to avoid any explicit mention of either the state parameter (date in our example) or the accessibility relation. Instead we introduce two special operators that describe properties of states which are accessible from a given state in a universe.

The two modal operators introduced are $\Box$ (called the necessity operator) and $\Diamond$ (called the possibility operator). Their meaning is given by the following rules of interpretation in which we denote by $|w|_s$ the truth value of the formula $w$ in a state $s$:

\[
|\Box w|_s = \forall s'[R(s, s') \supset |w|_{s'}]
\]
\[
|\Diamond w|_s = \exists s'[R(s, s') \land |w|_{s'}].
\]

Thus, $\Box w$ is true at a state $s$ if the formula $w$ is true at all states $R$-accessible from $s$. Similarly, $\Diamond w$ is true at a state $s$ if $w$ is true in at least one state $R$-accessible from $s$. Usually, $R$ is taken to be reflexive, so that every state is $R$-accessible from itself and thus $R(s, s)$ always holds.

A modal formula is a formula constructed from proposition symbols, predicate symbols (including equality), function symbols, individual constants and individual variables, the classical operators and quantifiers, and the modal operators. A formula without any modal operators is called a static formula. A fully modal (dynamic) formula is conveniently viewed as consisting of static subformulas to which modal and classical operators are applied. The truth value of a modal formula at some state of a given universe is found by a repeated use of the rules above for the modal operators and evaluation of any static subformula on the state itself. It is assumed that every state contains a full interpretation of all the classical symbols in the formula, which fully determines the truth value of every static formula.

For example, the formula

\[
\text{rain}(\ell) \supset \Diamond \neg \text{rain}(\ell)
\]

is interpreted in our model of rainy days as stating: For a given day and a given location $\ell$, if it rains on that day at $\ell$ then there exists another day in the future on which it will not rain at $\ell$; thus any rain will eventually stop. Similarly,

\[
\text{rain}(\ell) \supset \Box \neg \text{rain}(\ell)
\]

claims that if it rains on that day it will rain everafter. Note that any modal formula is always considered with respect to some fixed reference state, which may be chosen arbitrarily. In our example, it has the meaning of “today”.

Consider the general formula
\[ \Box \sim w \equiv \sim \Diamond w. \]

As we can see from the definitions this claims that all R-accessible states satisfy \( \sim w \) if and only if there does not exist an R-accessible state satisfying \( w \). This formula is true in any state for any universe with an arbitrary \( R \).

We now give a more precise definition. A universe \( U \) for a modal formula \( w \) consists of a nonempty domain \( D \), a set of states (or worlds) \( S \), and a binary relation \( R \) on \( S \), called the accessibility relation. Each state \( s \) provides a first-order interpretation over the domain \( D \) for all the proposition symbols, predicate symbols, function symbols, individual constants, and (free) individual variables in \( w \). A model \( (U, s_0) \) is a universe \( U \) with one of the states of \( U \), \( s_0 \in S \), designated as the initial or reference state. In short,

\[
\begin{align*}
\text{universe of } w &= \begin{cases} 
\text{domain} = D \\
\text{set of states} = S \\
\text{accessibility relation between states} = R 
\end{cases} \\
\text{where} \\
\text{state} &= \text{assignment to symbols of } w \text{ over } D
\end{align*}
\]

We define the truth value of a modal formula \( w \) at a state \( s \) (denoted by \( |w|_s \)) in a given universe \( U \) inductively:

1. If \( w \) is static, i.e., contains no modal operators, then its truth value \( |w|_s \) is found by interpreting \( w \) in \( s \).
2. \( |\Box w|_s \) is \( \forall s' [R(s, s') \supset |w|_{s'}] \).
3. \( |\Diamond w|_s \) is \( \exists s' [R(s, s') \land |w|_{s'}] \).
4. \( |w_1 \lor w_2|_s \) is true iff either \( |w_1|_s \) is true or \( |w_2|_s \) is true.
5. \( |\sim w|_s \) is true iff \( |w|_s \) is false.

Note that by our rules of interpretation
- \( |\Diamond(\Box w)|_s \) means that \( |\Box w|_{s'} \) is true at some state \( s' \), R-accessible from \( s \). That is,
  \[ \Diamond \Box w \]

stands for: we can get to a point where \( w \) is true everafter; i.e., there is a state \( s' \) R-accessible from \( s \) such that \( s' \) itself and all of its R-descendants satisfying \( w \).
- \( |\Box(\Diamond w)|_s \) means that \( |\Diamond w|_{s'} \) is true for all states \( s' \), R-accessible from \( s \). That is,
  \[ \Box \Diamond w \]

stands for: wherever we go \( w \) is still realizable; i.e., for every state \( s' \) accessible from \( s \) it is possible to find an R-descendant of \( s' \) which satisfies \( w \).
• \(\Box(w \supset \Box w)\), means that \(w \supset \Box w\) is true for all states \(s'\), \(R\)-accessible from \(s\). That is,
\[
\Box(w \supset \Box w)
\]
stands for: if \(w\) ever becomes true in some \(s'\) accessible from \(s\), it remains true for all descendants of \(s'\).

If a formula \(w\) is true in a state \(s_0\) in a universe \(U\) we say that \((U, s_0)\) is a \textit{(satisfying) model} for that formula, or that the formula is \textit{satisfied} in \((U, s_0)\).

A formula \(w\) which is true in all states of every universe is called \textit{valid}; that is, for every universe \(U\) of \(w\) and for every state \(s\) in \(U\), \(|w|_s\) is true. For example, the formula
\[
\Box \sim w \equiv \sim \Diamond w
\]
is a valid formula. This formula establishes the connection between "necessity" and "possibility".

Another valid formula is
\[
\Box(w_1 \supset w_2) \supset (\Box w_1 \supset \Box w_2),
\]
i.e., if in all accessible states \(w_1 \supset w_2\) holds and if \(w_1\) is true in all accessible states, then \(w_2\) must also be true in all of those states.

Both formulas are valid for any accessibility relation. If we agree to place further general restrictions on the relation \(R\), we obtain additional valid formulas which are true for any model with a relation satisfying these restrictions. According to the different restrictions we may impose on \(R\), we obtain different modal systems. \textit{In our discussion we stipulate that \(R\) is always reflexive and transitive}; i.e., we consider a formula to be \textit{valid} iff it is true in all states of every universe with a reflexive and transitive accessibility relation.

For example, the formula
\[
\Box w \supset w
\]
is valid since it is true for every reflexive model. It claims for a state \(s\) that if all states accessible from \(s\) satisfy \(w\), then \(w\) is satisfied by \(s\) itself. This is obvious since \(s\) is accessible from itself (by reflexivity).

The formula
\[
\Diamond \Diamond w \supset \Diamond w,
\]
which stands for \((\Diamond(\Diamond w)) \supset (\Diamond w)\), is valid since it is true for all transitive models. It claims for a state \(s_0\): if there exists an \(s_2\) accessible from \(s_1\) which is accessible from \(s_0\) such that \(s_2\) satisfies \(w\), then there exists an \(s_3\) accessible from \(s_0\) which satisfies \(w\). This always holds in a transitive model since by transitivity, \(s_2\) is also accessible from \(s_0\) and we may take \(s_3 = s_2\).

THE TEMPORAL FRAMEWORK

The framework of temporal logic is a modal framework in which we impose further restrictions on the models of interpretation ([PRI], [RU]). The interpretation given by temporal logic to the
basic accessibility relation is that of the passage of time. A world \( s' \) is accessible from a world \( s \) if through development in time, \( s \) can change into \( s' \). We concentrate on histories of development which are linear and discrete. Thus, the models of temporal logic consist of \( \omega \)-sequences, i.e., infinite sequences of the form \( \sigma = s_0, s_1, \ldots \). In such a sequence, \( s_j \) is accessible from \( s_i \) if \( i \leq j \). Due to the discreteness of the sequences we can refer not only to states that lie in the future of a given state, but also to the (unique) immediate future state or next state. This leads to the introduction of an additional operator, the next instant operator denoted by \( \Box \).

Relating these concepts to the general modal framework, a universe for temporal logic consists again of a collection of states (worlds). On these states we define an immediate accessibility relation \( \rho \) which is required to be a function. That means that every world \( s \) has exactly one other world \( s' \) such that \( \rho(s, s') \). This corresponds to our intuition that in a discrete time model each instant has exactly one immediate successor. \( R = \rho^* \), the transitive reflexive closure of \( \rho \), is the accessibility relation discussed under the general modal framework and is indeed both reflexive and transitive. Intuitively \( R(s, s') \) holds when \( s' \) is either identical to \( s \) or lies in the future of \( s \).

Given the restrictions imposed on \( R \), the resulting model \((U, s_0)\) can be represented as an infinite sequence of states,

\[ \sigma = s_0, s_1, s_2, \ldots \]

where \( \rho(s_i, s_{i+1}) \) is true for \( i \geq 0 \). This intuitively corresponds to the temporal development of a process observed at a sequence of discrete points in time.

\*

We will now give a more complete definition of the language we are going to use. Note that this language is designed specially for the application we have in mind, namely reasoning about programs, and is not necessarily the most general temporal language possible.

**Symbols.** The language uses a set of basic symbols consisting of individual variables and constants, and proposition, function and predicate symbols. The set is partitioned into two subsets: global and local symbols. The global symbols have a uniform interpretation over the complete universe and do not change their value or meaning from one state to another. The local symbols, on the other hand, may assume different meanings and values in different states of the universe. For our purpose, the only local symbols that interest us are local individual variables and local propositions. We will have global symbols of all types.

Our symbols are further partitioned into different sorts. Each sort corresponds to a different domain, and the interpretation will associate a domain with every sort. Corresponding to a sort we may have individual constants that are interpreted over the associated domain, individual variables that assume values from that domain, function symbols that represent functions over the domain, and predicate symbols that represent predicates over the domain. The symbols used for individual constants, functions and predicates will be typical of the first-order theory of the domain we wish to formalize. For example, in dealing with the theory of natural numbers we use the conventional symbols:

\{0, 1, \ldots, +, -, \times, \div, \ldots, >, \ge, \ldots \}.
Note that some functions and predicates may have a non-homogenous signature, i.e., they may have different sorts associated with different argument positions. A typical example is the if-then-else function which accepts one boolean argument and two arguments of possibly another sort.

**Operators and quantifiers.** We use the regular set of boolean connectives: \( \land, \lor, \lnot, \equiv, \), and \( \sim \) together with the equality operator \( = \) and the first-order quantifiers \( \forall \) and \( \exists \). This set is referred to as the classical operators. The modal operators are:

\( \Box, \Diamond, \O, \) and \( \mathcal{U} \);

they are called respectively the always, sometime, next and until operators. The first three operators are unary while the \( \mathcal{U} \) operator is binary.

The quantifiers \( \forall \) and \( \exists \) are applied only to global individual variables.

**Terms.** Terms are constructed from individual constants and individual variables to which we apply functions. The application must conform with the arity and sort signature restrictions associated with each symbol. An additional rule is that if \( t \) is a term so is \( \mathcal{O} t \) - referred to as the next (value of) \( t \). Note that we use the next operator \( \mathcal{O} \) in two different ways - as a temporal operator applied to formulas and as a temporal operator applied to terms.

**Formulas (sentences).** Formulas are constructed from atomic formulas to which we apply the boolean connectives, the modal operators and quantification over global individual variables. Atomic formulas consist of propositions and predicates (including the \( '=' \) operator) applied to terms of the appropriate sorts.

Recall that a formula is said to be classical (static) if it involves no modal operators.

We will sometimes regard propositions and (closed) formulas as integer-valued functions yielding 1 for true and 0 for false. These functions can then be combined arithmetically in order to provide a compact representation for equivalent but longer propositional formulas. For example, for propositions \( p_1, \ldots, p_n \), the statement

\[
p_1 + \cdots + p_n = 1 \quad \text{or} \quad \sum_{i=1}^{n} p_i = 1
\]

states that exactly one of the \( p_i \)'s is true. This is of course equivalent to the formula

\[
\forall_{1 \leq i \leq n} \, p_i \land \bigwedge_{1 \leq i < j \leq n} \lnot(p_i \land p_j).
\]

**MODELS (ENVIRONMENTS)**

A model \((I, \alpha, \sigma)\) for our language consists of an (global) interpretation \( I \), a (global) assignment \( \alpha \) and a sequence of states \( \sigma \).

The interpretation \( I \) specifies a nonempty domain \( D \), corresponding to each sort, and assigns concrete elements, functions and predicates to the (global) individual constants, function and predicate symbols.
The assignment $\alpha$ assigns a value over the appropriate domain to each of the global free individual variables.

The sequence $\sigma = s_0, s_1, \ldots$ is an infinite sequence of states. Each state $s_i$ assigns values to the local free individual variables and propositions.

For a sequence

$$\sigma = s_0, s_1, \ldots$$

we denote by

$$\sigma^{(i)} = s_i, s_{i+1}, \ldots$$

the $i$-truncated suffix of $\sigma$.

Given a temporal formula $w$, we present below an inductive definition of the truth value of $w$ in a model $(I, \alpha, \sigma)$. The value of a subformula or term $\tau$ under $(I, \alpha, \sigma)$ is denoted by $\tau|_{\sigma}^{I}$, $I$ being implicitly assumed.

Consider first the evaluation of terms:

- For a local individual variable or local proposition $y$:
  $$y|_{\sigma}^{I} = y_{s_0},$$
  \text{i.e., the value assigned to $y$ in $s_0$, the first state of $\sigma$.}

- For a global individual variable or global proposition $u$:
  $$u|_{\sigma}^{I} = \alpha[u],$$
  \text{i.e., the value assigned to $u$ by $\alpha$.}

- For an individual constant the evaluation is given by $I$:
  $$c|_{\sigma}^{I} = I[c].$$

- For a $k$-ary function $f$:
  $$f(t_1, \ldots, t_k)|_{\sigma}^{I} = I[f](t_1|_{\sigma}^{I}, \ldots, t_k|_{\sigma}^{I}),$$
  \text{i.e., the value is given by the application of the interpreted function $I[f]$ to the values of $t_1, \ldots, t_k$ evaluated in the environment $(I, \alpha, \sigma)$.}

- For a term $t$:
  $$t|_{\sigma}^{I} = t|_{\sigma^{(i)}}^{I},$$
  \text{i.e., the value of $\triangledown t$ in $\sigma = s_0, s_1, \ldots$ is given by the value of $t$ in the shifted sequence $\sigma^{(i)} = s_i, s_{i+1}, \ldots$.}
Consider now the evaluation of sentences:

- For a k-ary predicate \( p \) (including equality):
  \[
  p(t_1, \ldots, t_k)_{\sigma} = I[p](t_1)_{\sigma}, \ldots, t_k)_{\sigma}.
  \]
  Here again, we evaluate the arguments in the environment and then test \( I[p] \) on them.

- For a disjunction:
  \[
  (w_1 \lor w_2)_{\sigma} = true \quad \text{iff} \quad w_1_{\sigma} = true \text{ or } w_2_{\sigma} = true.
  \]

- For a negation:
  \[
  (\neg w)_{\sigma} = true \quad \text{iff} \quad w_{\sigma} = false.
  \]

- For a next-time application:
  \[
  \bigcirc w_{\sigma} = w_{\sigma(1)}.
  \]
  Thus \( \bigcirc w \) means: \( w \) will be true in the next instant – read "next \( w \)".

- For an all-times application:
  \[
  \square w_{\sigma} = true \quad \text{iff} \quad \text{for every } k \geq 0, \ w_{\sigma(k)} = true,
  \]
  i.e., \( w \) is true for all suffix sequences of \( \sigma \). Thus \( \square w \) means: \( w \) is true for all future instants (including the present) – read "always \( w \)" or "henceforth \( w \)".

- For a some-time application:
  \[
  \Diamond w_{\sigma} = true \quad \text{iff} \quad \text{there exists a } k \geq 0 \text{ such that } w_{\sigma(k)} = true,
  \]
  i.e., \( w \) is true on at least one suffix of \( \sigma \). Thus \( \Diamond w \) means: \( w \) will be true for some future instant (possibly the present) – read "sometimes \( w \)" or "eventually \( w \)".

- For an until application:
  \[
  w_1 \mathcal{U} w_2_{\sigma} = true \quad \text{iff} \quad \text{for some } k \geq 0, \ w_2_{\sigma(k)} = true \text{ and }
  \text{for all } i, 0 \leq i < k, \ w_1_{\sigma(i)} = true.
  \]
  Thus \( w_1 \mathcal{U} w_2 \) means: there is a future instant in which \( w_2 \) holds, and such that until that instant \( w_1 \) continuously holds – read "\( w_1 \) until \( w_2 \)" ([KAM], [GPSS]).

- For a universal quantification:
  \[
  (\forall u.w)_{\sigma} = true \quad \text{iff} \quad \text{for every } d \in D, \ w_{\sigma'} = true,
  \]
  where \( \sigma' = \sigma [u \leftarrow d] \) is the assignment obtained from \( \sigma \) by assigning \( d \) to \( u \). \( D \) is the domain corresponding to the sort of \( u \).
For an existential quantification:

\[(\exists u. w)|_{\sigma} = \text{true} \text{ if for some } d \in D, \ w|_{\sigma'} = \text{true},\]

where \(\sigma' = \sigma \circ [u \leftarrow d].\)

Following are some examples of temporal expressions and their intuitive interpretations:

- \(u \sqsupset \Diamond v\) — If \(u\) is presently true, \(v\) will eventually become true.
- \(\Box (u \sqsupset \Diamond v)\) — Whenever \(u\) becomes true it will eventually be followed by \(v\).
- \(\Diamond \Box w\) — At some future instant \(w\) will become permanently true.
- \(\Diamond (w \land \Box \sim w)\) — There will be a future instant such that \(w\) is true at that instant and false at the next.
- \(\Box \Diamond w\) — Every future instant is followed by a later one in which \(w\) is true, thus \(w\) is true infinitely often.
- \(\Box (u \supset \Box v)\) — If \(u\) ever becomes true, then \(v\) is true at that instant and ever after.
- \(\Box u \lor (u \mathcal{U} v)\) — Either \(u\) holds continuously or it holds until an occurrence of \(v\). This is the weak form of the until operator that states that \(u\) will hold continuously until the first occurrence of \(v\) if \(v\) ever happens or indefinitely otherwise.
- \(\Diamond v \supset (\sim v \mathcal{U} u)\) — If \(v\) ever happens, its first occurrence is preceded by (or coincides with) \(u\).

If \(w\) is true under the model \((I, \alpha, \sigma)\) we say that \((I, \alpha, \sigma)\) satisfies \(w\) or that \((I, \alpha, \sigma)\) is a satisfying model for \(w\). We denote this by

\[(I, \alpha, \sigma) \models w.\]

A formula \(w\) is satisfiable if there exists a satisfying model for it.

A formula \(w\) is valid if it is true in every model, and we write

\(\models w.\)

Sometimes we are interested in a restricted class of models \(C\). A formula \(w\) which is true for every model in \(C\) is said to be \(C\)-valid, denoted by

\(C \models w.\)

Example:

The formula \(\Diamond (w_1 \land w_2) \supset (\Diamond w_1 \land \Diamond w_2)\) is valid, i.e.,

\(\models \Diamond (w_1 \land w_2) \supset (\Diamond w_1 \land \Diamond w_2)\).
It says that if there exists an instant in which both \( w_1 \) and \( w_2 \) are true then there exists an instant in which \( w_1 \) is true and there exists an instant in which \( w_2 \) is true.

The \( \Box \)-converse of this formula is not valid, i.e.,

\[ \forall \ (\Box w_1 \land \Box w_2) \not\supset \Box(w_1 \land w_2). \]

For, consider an interpretation in which \( w_1 \) is true and \( w_2 \) is false at state \( s_1 \), and in which \( w_1 \) is false and \( w_2 \) is true at state \( s_2 \), and \( s_2 \) is accessible from \( s_1 \) (also clearly \( s_1 \) from \( s_1 \) and \( s_2 \) from \( s_2 \))

\[ s_1 \quad s_2 \]

\[ w_1 : \text{true} \quad w_1 : \text{false} \]
\[ w_2 : \text{false} \quad w_2 : \text{true} \]

Then at state \( s_1 \):

\[ \Box w_1 \text{ is true (since } w_1 \text{ is true at } s_1) \]
\[ \Box w_2 \text{ is true (since } w_2 \text{ is true at } s_2) \]
\[ \Box (w_1 \land w_2) \text{ is false (since } w_1 \land w_2 \text{ is false at } s_1 \text{ and at } s_2). \]

Therefore, the formula is false under this interpretation. \( \square \)

A REPERTOIRE OF VALID TEMPORAL STATEMENTS

In this section we present a list of valid temporal statements (schemata) which we justify by semantic considerations. There are two reasons for presenting them here. First we would like to illustrate the type of temporal reasoning we will later use. Second, the statements presented here will later be taken to be established valid statements and used freely in proofs. When, in a later part of this work, we present a formal deductive system for temporal reasoning, we will take some of the valid statements listed here as axioms and deduce the others as theorems.

In the following list, whenever we write a valid temporal statement in form \( \models A \supset B \) and not \( \models A \equiv B \), it implies that its \( \supset \)-inverse is not valid, i.e., \( \nexists \ B \supset A \). That is, a model can be found under which an instance of \( B \supset A \) will be false.

1. \( \models \square \neg w \equiv \neg \Diamond w \)
2. \( \models \Diamond \neg w \equiv \neg \square w \)
3. \( \models \neg \Diamond w \equiv \neg \Diamond w \)

These statements point out the duality between the operators.

Statement 1 says that \( w \) is false in all states (instants) of a sequence iff there is no state in which \( w \) is true.
Statement 2 says that there is a state in which $w$ is false iff it is not the case that $w$ is true in all states.

Statement 3 says that $w$ is false in the next state iff it is not the case that $w$ is true in the next state. This statement restricts each state to have a single successor.

4. $\vdash w \supset \diamondsuit w$

5. $\vdash \Box w \supset w$

6. $\vdash \Box w \supset \diamondsuit w$

7. $\vdash \Box w \supset \Box w$

8. $\vdash \Box w \supset \diamondsuit w$

9. $\vdash \Box w \supset \Box \Box w$

10. $\vdash w_1 \cup w_2 \supset \diamondsuit w_2$

11. $\vdash \diamondsuit \Box w \supset \Box \diamondsuit w$.

Statement 4 says that if $w$ is true now, then it will be true sometime in the future. This is an immediate consequence of the fact that the present is considered to be part of the future.

Statement 5, a dual of 4, says that if $w$ is true in all future instants it is also presently true.

Statement 6 says that if $w$ is true at the next instant it will sometime be true. This is because the next instant is also a part of the future.

Statement 7, a dual of 6, says that if $w$ is true in all future instants it is also true for the next instant.

Statement 8 says that if $w$ is always true then it is sometimes true.

Statement 9 says that if $w$ is true in all future instants it is also true for all future instants of the next instant, i.e., all future instants excluding the present.

Statement 10 says that if $w_1$ is true until $w_2$ will happen then $w_2$ will eventually happen.

Statement 11 says that if $w_1$ is true until $w_2$ will happen then $w_2$ will eventually happen.

Statement 11 says that if $w$ is permanently true beyond a certain instant then $w$ is true infinitely often.

12. $\vdash \Box w \equiv \Box \Box w$

13. $\vdash \diamondsuit w \equiv \diamondsuit \diamondsuit w$.

The statements 12 and 13 say that both $\Box$ and $\diamondsuit$ are idempotent. Intuitively speaking both imply that the future is equivalent to the future of the future. Note that a corresponding statement does not hold for $\Box$, since both $\not\vdash \Box \diamondsuit w \supset \diamondsuit \Box w$ and $\not\vdash \Box \Box w \supset \Box w$.
16. \[ ((\diamond w_1) \sqcup (\diamond w_2)) \equiv (\diamond (w_1 \sqcup w_2)). \]

Statements 14 to 16 indicate the *commutativity* of the *next* operator \( \diamond \) with each of the others. It amounts to a shift of our reference point from the present to the immediately next instant.

Statement 14 says that \( w \) holds for the instant next to every future instant *iff* \( w \) holds for all future instants, barring the present.

Statement 15 says that \( w \) is realized in an instant next to some future instant *iff* it is realized sometimes in the future, excluding the present.

Statement 16 says that \( \diamond w_1 \) holds until an instance of \( \diamond w_2 \) *iff* \( w_1 \) holds until \( w_2 \) starting from the next instant.

17. \[ ((\Box w_1) \land (\Box w_2)) \equiv (\Box (w_1 \land w_2)) \]

18. \[ ((\Diamond w_1) \lor (\Diamond w_2)) \equiv (\Diamond (w_1 \lor w_2)) \]

19. \[ ((\Box w_1) \land (\Box w_2)) \equiv (\Box (w_1 \land w_2)) \]

20. \[ ((\Diamond w_1) \lor (\Diamond w_2)) \equiv (\Diamond (w_1 \lor w_2)) \]

21. \[ ((\Box w_1) \supset (\Box w_2)) \equiv (\Box (w_1 \supset w_2)) \]

22. \[ ((\Box w_1) \equiv (\Box w_2)) \equiv (\Box (w_1 \equiv w_2)) \]

23. \[ ((w_1 \land w_2) \sqcup w_3) \equiv ((w_1 \sqcup w_3) \land (w_2 \sqcup w_3)) \]

24. \[ ((w_1 \sqcup (w_2 \lor w_3)) \equiv ((w_1 \sqcup w_2) \lor (w_1 \sqcup w_3)) \]

Statements 17 to 24 indicate *distributivity* relations between the temporal operators and the boolean connectives.

The \( \Box \) operator has a *universal* character – stating \( w \) for all future instants, and the \( \Diamond \) operator has an *existential* character – stating \( w \) for some future instant. Consequently \( \Box \) distributes with \( \land \) (17) stating that both \( w_1 \) and \( w_2 \) hold in every future instant *iff* \( w_1 \) holds for all future instants and so does \( w_2 \). The \( \Diamond \) operator distributes with \( \lor \) (18) stating that there will be an instant in which either \( w_1 \) or \( w_2 \) hold *iff* there either will be an instant in which \( w_1 \) holds or there will be an instant in which \( w_2 \) holds.

The \( \Diamond \) operator has both universal and existential character because it refers to a unique instant – the next one. Therefore it distributes with both \( \land \) and \( \lor \), as is shown by statements 19 and 20.

Since the \( \Box \) operator has been shown to distribute with the basic boolean connectives \( \sim \), \( \land \), \( \lor \), it will also distribute over any other boolean connective such as \( \supset \) and \( \equiv \). For example, Statement 21 says that if in the next instant \( w_1 \) implies \( w_2 \) and \( w_1 \) is known to hold at the next instant then so does \( w_2 \).

The *until* operator has a different character with respect to its two arguments. It is universal with respect to its first argument which appears in the semantic definition under a \( \forall i (0 \leq i < k) \) quantification. It is existential with respect to its second argument which appears in the semantic definition under a \( \exists k (k \geq 0) \) quantification.
Statement 23 says that \( w_1 \) and \( w_2 \) both hold until an instance of \( w_3 \) iff \( w_1 \) holds until an instance of \( w_3 \) and \( w_2 \) holds until an instance of \( w_3 \). To justify the implication from right to left, we are guaranteed of having a \( t_1 \) such that \( w_3 \) is true at \( t_1 \) and \( w_1 \) holds until then, and a \( t_2 \) such that \( w_3 \) is true at \( t_2 \) and \( w_2 \) holds until then. By considering the earliest of these two instants \( t = \min(t_1, t_2) \) we know that \( w_3 \) is true at \( t \) and both \( w_1 \) and \( w_2 \) hold until then.

Statement 24 says that \( w_1 \) holds until an instance of either \( w_2 \) or \( w_3 \) iff either \( w_1 \) holds until an instance of \( w_2 \) or \( w_1 \) holds until an instance of \( w_3 \).

25. \( \vdash (\Box w_1 \lor \Box w_2) \supset (\Box(\Box w_1 \lor \Box w_2)) \)
26. \( \vdash \Diamond (w_1 \land w_2) \supset (\Diamond w_1 \land \Diamond w_2) \)
27. \( \vdash ((w_1 \lor w_3) \lor (w_2 \lor w_3)) \supset (w_1 \lor w_2) \lor w_3 \)
28. \( \vdash (w_1 \lor (w_2 \lor w_3)) \supset ((w_1 \lor w_2) \lor (w_1 \lor w_3)) \).

Statements 25 to 28 indicate implications that hold when we interchange the order between temporal operators and the boolean connectives. They are not equivalences and only the direction of the given implication is true.

Statement 25 says that if either \( w_1 \) is true for all future instants or \( w_2 \) is true for all future instants then in every future instant either \( w_1 \) or \( w_2 \) holds.

Statement 26 says that if there exists an instant in which both \( w_1 \) and \( w_2 \) are true then there exists an instant in which \( w_1 \) is true and there exists an instant in which \( w_2 \) is true.

Statement 27 says that if either \( w_1 \) holds until \( w_3 \) or \( w_2 \) holds until \( w_3 \) then there is an instance of \( w_3 \) such that until then either \( w_1 \) or \( w_2 \) holds.

Statement 28 says that if \( w_1 \) holds until an instant \( t \) in which both \( w_2 \) and \( w_3 \) are true then both \( w_1 \) holds until \( w_2 \) at \( t \) and \( w_1 \) holds until \( w_3 \) at \( t \) implying the conjunction.

29. \( \vdash (\Box w_1 \lor \Box w_2) \supset (\Box w_1 \lor \Box w_2) \)
30. \( \vdash (\Box w_1 \lor \Box w_2) \supset (\Diamond w_1 \lor \Diamond w_2) \)
31. \( \vdash (\Box w_1 \lor \Box w_2) \supset (\Box w_1 \lor \Box w_2) \)
32. \( \vdash (\Box w_1 \lor \Box w_2) \supset ((w_1 \lor w_3) \lor (w_2 \lor w_3)) \)
33. \( \vdash (\Box w_1 \lor \Box w_2) \supset ((w_1 \lor w_2) \lor (w_2 \lor w_2)) \).

Statements 29 to 33 indicate the monotonicity of each of the temporal operators; that is, if its application to a formula \( w_1 \) is true and \( w_1 \) universally implies \( w_2 \) (for all instants) then its application to \( w_2 \) is also true.

This property is stated respectively for \( \Box \) in 29, \( \Diamond \) in 30, \( \Diamond \) in 31 and the two positions of \( \lor \) in 32 and 33.

34. \( \vdash (\Box w_1 \land \Box w_2) \supset \Box (w_1 \land w_2) \)
35. \( \vdash (\Box w_1 \land \Diamond w_2) \supset \Diamond (w_1 \land w_2) \)
36. $\vdash (\Box w_1 \land (w_2 \cup w_3)) \supset (w_1 \land w_2) \cup (w_1 \land w_3)$.  

Statements 34 to 36 are frame rules. They say that if $w_1$ is known to hold for all states then $w_1$ may be added as a conjunct under any other temporal operator. This is respectively stated for $\Box$ in 34, for $\Diamond$ in 35 and for both argument positions of $\cup$ in 36.

37. $\vdash (w \land \Box (w \supset \Box w) ) \supset \Box w$

38. $\vdash (w \land \Diamond \sim w) \supset \Diamond (w \land \Box \sim w)$.  

39. $\vdash (\Diamond w_1 \land \Diamond w_2) \supset [\Diamond (w_1 \land \Diamond w_2) \lor \Diamond (w_2 \land \Diamond w_1)]$.  

Statements 37 and 38 are induction rules and Statement 39 describes the linearity property.

Statement 37 (corresponding to computational induction) says that if the fact that $w$ holds at any instant implies that it also holds at the next instant, and $w$ holds in the present, then $w$ holds at all future instants.

Statement 38 (corresponding to the least number principle) is the dual of 37. It says that if $w$ is true now and is false sometime in the future, then there exists some instant such that $w$ is true at that instant and false at the next.

Statement 39 says that if $w_1$ and $w_2$ are both guaranteed to happen, then either $w_1$ will happen first, followed by $w_2$ or $w_2$ will happen first, followed by $w_1$.

40. $\vdash \Box w \equiv (w \land \Box \Box w)$  

41. $\vdash \Diamond w \equiv (w \lor \Box \Diamond w)$  

42. $\vdash w_1 \cup w_2 \equiv w_2 \lor (w_1 \land \Box (w_1 \cup w_2))$  

Statements 40 to 42 explain the $\Box$, $\Diamond$, and $\cup$ operators respectively by distributing their effect into what is implied for the present and what is implied for the next instant.

Statement 40 says that $w$ is true for all future instants iff $w$ is true for the present and for all instants lying in the future of the next instant.

Statement 41 says that $w$ is true in some future instance iff it is either true now or true at an instant not earlier than the next.

Statement 42 says that 'w₁ until w₂' is presently true iff either w₂ is true now or w₁ holds now and 'w₁ until w₂' is true for the next instant.

43. $\vdash (\sim w \cup w) \equiv \Diamond w$

44. $\vdash (\Box w_1 \land \Diamond w_2) \supset (w_1 \cup w_2)$

45. $\vdash ((w_1 \cup w_2) \cup w_3) \supset ((w_1 \cup w_3) \cup (w_2 \cup w_3))$

46. $\vdash ((w_1 \cup w_2) \land (\sim w_2 \cup w_3)) \supset (w_1 \cup w_3)$

47. $\vdash (w_1 \cup (w_2 \land w_3)) \supset ((w_1 \cup w_2) \cup w_3)$
48. \[ \vdash ((w_1 \cup w_2) \cup w_3) \supset ((w_1 \lor w_2) \cup w_3) \]

49. \[ \vdash (\square w_1 \land \square w_2) \supset ((\neg w_1 \cup w_2) \lor (\neg w_2 \cup w_1)) \]

This list of statements illustrates some properties of the until operator.

Statement 43 says that \( w \) is guaranteed to happen if\( f \) there is an instant in which \( w \) is true and until this instant \( w \) is false. This states that \( w \) happens if\( f \) there is an earliest occurrence of \( w \).

Statement 44 says that if \( w_2 \) is guaranteed to happen and \( w_1 \) is constantly true, then it will be true until a guaranteed occurrence of \( w_2 \).

Statement 45 says that if \( w_1 \) implies \( w_2 \) until \( w_3 \) happens and \( w_1 \) is true until an instance of \( w_3 \) (not necessarily the same instance) then \( w_2 \) will hold until an instance of \( w_3 \) (which can be taken as the earlier of the two).

Statement 46 says that if \( w_1 \) holds until \( w_2 \) and \( w_2 \) is false until \( w_3 \) then \( w_1 \) is true until \( w_3 \). To justify this let (a) \( w_1 \cup w_2 \) and (b) \( \neg w_2 \cup w_3 \) be the two clauses given as premises. By (b) we know that \( w_3 \) will happen say at \( t_3 \) and \( w_2 \) will be false until then. By (a) \( w_2 \) must happen, say at \( t_2 \) and \( w_1 \) must be true until then. By (b) \( t_2 \geq t_3 \) so that \( w_1 \) must certainly be true until \( t_3 \), an instance of \( w_3 \).

Statement 47 can be justified as follows. The premise guarantees an instant \( t_2 \) such that \( w_2 \) and \( w_3 \) are both true at \( t_2 \) and \( w_1 \) is true until then. Clearly, taking any \( 0 \leq t_1 < t_2 \) we know that \( w_2 \) will be true at \( t_2 \) and \( w_1 \) is true for every \( t \), \( t_1 \leq t < t_2 \), thus \( w_1 \cup w_2 \) at \( t_1 \). Since \( w_1 \cup w_2 \) is true for every \( t_1 \), \( 0 \leq t_1 < t_2 \), and \( w_3 \) is true at \( t_2 \), \( w_1 \cup w_2 \) is true until \( w_3 \).

Statement 48 says that if \( w_1 \cup w_2 \) is continuously true until an instance of \( w_3 \) then so is \( w_1 \lor w_2 \).

Statement 49 says that if both \( w_1 \) and \( w_2 \) are guaranteed to happen then one of them will happen "first"; that is, either \( w_2 \) happens first and \( w_1 \) is false until then, or \( w_1 \) happens first and \( w_2 \) is false until then. (In both cases we allow the possibility that both \( w_1 \) and \( w_2 \) occur for the first time at the same instant.)

50. \[ \vdash \diamond \exists x w \equiv \exists x \diamond w \]

51. \[ \vdash \square \forall x w \equiv \forall x \square w \]

52. \[ \vdash \circ \exists x w \equiv \exists x \circ w \]

53. \[ \vdash \circ \forall x w \equiv \forall x \circ w \]

54. \[ \vdash (\forall x w_1) \cup w_2 \equiv \forall x(w_1 \cup w_2) \quad \text{provided } x \text{ is not free in } w_2 \]

55. \[ \vdash (w_1 \cup (\exists x w_2)) \equiv \exists x(w_1 \cup w_2) \quad \text{provided } x \text{ is not free in } w_1 \]

Statements 50 to 55 indicate the commutativity relations between the temporal operators and the quantifiers. They follow from our restriction that the quantifiers \( \forall \) and \( \exists \) are to be applied only to global individual variables. Statements 50 and 51 are known as Barcan's formulas.

Statement 50 demonstrates once more the existential character of the operator \( \diamond \). It says that in some instant there exists an \( x \) satisfying \( w(x) \) if\( f \) there exists an \( x \) such that at some instant \( w(x) \) is satisfied.

18
Statement 51 demonstrates the universal character of the $\Box$ operator. It says that $w$ is true in all instants for all values of $x$ if $w$ is true for all values of $x$ for every instant.

Statements 52 and 53 demonstrate the dual character of the $\Diamond$ operator, which is both universal and existential.

Statements 54 and 55 demonstrate that the $\text{until}$ operator has a universal character with respect to its first argument and an existential character with respect to its second argument.

The preceding statements were all of the form

$$\models w$$

and they stated formulas which are true in every model. The next list of statements contains inferences of the form

$$\models w_1 \Rightarrow \models w_2.$$  

They state that if $w_1$ has been shown to be a valid statement then so is $w_2$. The inference statements enable us to deduce the validity of one formula from the other. For every valid formula $\models w_1 \supset w_2$ there is a corresponding inference $\models w_1 \Rightarrow \models w_2$, and this is a standard way of justifying an inference. However, there are inferences $\not\models w_1 \Rightarrow \models w_2$ such that $\not\models w_1 \supset w_2$ is not a valid statement (see, for example, the following inference 56).

56. $\models w \Rightarrow \models \Box w$ \hspace{2cm} $\Box$-insertion

57. $\models w \Rightarrow \models \Diamond w$ \hspace{2cm} $\Diamond$-insertion

58. $\models w \Rightarrow \models O w$ \hspace{2cm} $O$-insertion

Inference 56 states that if $w$ is valid then so is $\Box w$. The fact that $w$ is valid means that it is true for every sequence and therefore for all suffixes $\sigma^{(i)}$ of a given sequence. Thus $\Box w$ is true for every sequence $\sigma$ and is therefore a valid statement.

Inference 57 may be deduced by inferring first $\models \Box w$ and then using the valid statement $\models \Box w \supset \Diamond w$ (number 8 in our list) to infer $\models \Diamond w$.

Inference 58 may be deduced similarly by using Statement 7, $\not\models \Box w \supset \Box w$.

59. $\not\models w_1 \supset w_2 \Rightarrow \not\models \Box w_1 \supset \Box w_2$ \hspace{2cm} $\Box\Box$-insertion

60. $\not\models w_1 \supset w_2 \Rightarrow \not\models \Diamond w_1 \supset \Diamond w_2$ \hspace{2cm} $\Diamond\Diamond$-insertion

61. $\not\models w_1 \supset w_2 \Rightarrow \not\models O w_1 \supset O w_2$ \hspace{2cm} $O\hspace{1mm}O$-insertion

These inferences are all obtained by inferring first $\not\models \Box(w_1 \supset w_2)$ by Inference 56 and then using statements 29 to 31, respectively.

62. $\not\models w_1 \supset \Box w_2 \supset w_2 \supset \Box w_3 \Rightarrow \not\models w_1 \supset \Box w_3$ \hspace{2cm} $\Box$-concatenation

63. $\not\models w_1 \supset \Diamond w_2 \supset w_2 \supset \Diamond w_3 \Rightarrow \not\models w_1 \supset \Diamond w_3$ \hspace{2cm} $\Diamond$-concatenation
Inference 62 is obtained by first deriving $\vdash \Box w_2 \supset \Box \Box w_3$ by Inference 59, observing that $\Box \Box w_3 \equiv \Box w_3$, and then using propositional reasoning. Inference 63 is obtained similarly by applying Inference 60. Note that the corresponding $\bigcirc$-concatenation inference does not hold.

\[
\begin{align*}
64. \quad \vdash w_1 \supset w_2 \quad &\Rightarrow \quad \vdash w_1 \supset \Box w_4 & \text{□-consequence} \\
&\vdash w_3 \supset w_4 \\
65. \quad \vdash w_2 \supset \Diamond w_3 \quad &\Rightarrow \quad \vdash w_1 \supset \Diamond w_4 & \text{○-consequence} \\
&\vdash w_3 \supset w_4 \\
66. \quad \vdash w_2 \supset \bigcirc w_3 \quad &\Rightarrow \quad \vdash w_1 \supset \bigcirc w_4 & \text{○-consequence} \\
&\vdash w_3 \supset w_4
\end{align*}
\]

Inference 64 is obtained by deriving first $\vdash \Box w_2 \supset \Box w_4$ by $\Box \Box$-introduction (59) and then applying propositional reasoning. Similarly, inferences 65 and 66 are obtained by deriving $\vdash \Diamond w_3 \supset \Diamond w_4$ and $\vdash \bigcirc w_3 \supset \bigcirc w_4$ by 60 and 61, respectively.
2. CONCURRENT PROGRAMS AND THEIR EXECUTION

In the following we introduce the model of concurrent programs that we will study here. (For simpler models see [KEL] and [LAMI].)

\[ y := f_0(x) \]

In our model, a concurrent program

\[ y := f_0(x); \{P_1||\ldots||P_m\} \]

consists of an initial value assignment \( y := f_0(x) \) followed by the parallel execution of \( m, m \geq 1 \), processes \( P_1, \ldots, P_m \). The processes operate on a set of program variables \( y = (y_1, \ldots, y_n) \) which are shared between the processes. The variables \( y \) are accessible to all the processes for both referencing and modifying. Each process \( P_i, i = 1, \ldots, m \), is an independent transition graph with nodes (locations) labeled by \( \ell_0, \ell_1, \ldots, \ell_\ell \). The sets of labels \( L_i = \{\ell_0^i, \ldots, \ell_\ell^i\} \) of the different processes are disjoint. The edges (or transitions) in each process are labeled by instructions of the form:

\[ \ell \xrightarrow{c_{\alpha}(y) \rightarrow [y := f_\alpha(y)]} \ell' \]

where \( c_{\alpha}(y) \) is a condition called the enabling condition of the transition \( \alpha \), and \( f_\alpha \) is the transformation associated with the transition \( \alpha \). If \( c_{\alpha}(y) \) is true we say that the transition \( \alpha \) is enabled for \( y = \bar{y} \).

For a given node \( \ell \) with \( k \) outgoing transitions

\[ c_1(y) \rightarrow [y := f_1(y)] \]

\[ c_k(y) \rightarrow [y := f_k(y)] \]

\[ \ell \]

\[ \alpha_1 \]

\[ \alpha_k \]

\[ \ell_1 \]

\[ \ell_k \]
we define \( E_\ell(y) = c_1(y) \lor \ldots \lor c_k(y) \) to be the full-exit condition at node \( \ell \). We do not require that the individual conditions are exhaustive, i.e., that \( E_\ell(y) = \text{true} \) for every \( y \); thus, deadlocks (or blockings) are allowed in our semantics. Nor do we require the conditions to be exclusive; thus, each process can be nondeterministic. A location whose individual conditions are mutually exclusive is called a deterministic location. If \( E_\ell(y) \) is true, i.e. at least one of the \( c_i, i = 1, \ldots, k, \) transitions originating from \( \ell \) is enabled, we say that the location \( \ell \) is enabled for \( y = \bar{y} \). If a process \( P_j \) is currently at \( \ell \in L \), which is enabled, we say that the process is enabled.

The set of program variables \( \bar{y} = (y_1, \ldots, y_n) \) is accessible and shared by all the processes. This model of concurrent programs is therefore called the shared-variables model. In this model, communication and synchronization between processes are managed via the shared variables.

The initial assignment \( \bar{y} := f_0(\bar{x}) \) assigns initial values to the shared program variables prior to the beginning of the concurrent execution. The parameters \( \bar{x} = (x_1, \ldots, x_t) \) that appear in this initial assignment, as well as other parameters appearing in the bodies of the processes, are the input parameters of the program. The behavior of the program naturally depends on the input parameters.

We will often represent a process in a linear-text form instead of a graph. In such a case the nodes are the places (labels) just before each statement, and the transitions are the statements themselves.

We list below the types of statements that we allow in the linear-text form and their representation in the graph model:

- \( \ell : \quad \bar{y} := f(\bar{y}) \)
  \( \ell' \):

  is represented as

  \[
  \begin{array}{c}
  \ell \\
  \text{true} \rightarrow [\bar{y} := f(\bar{y})] \\
  \ell'
  \end{array}
  \]

- \( \ell : \quad \text{if } p(\bar{y}) \text{ then go to } m \)
  \( \ell' \):

  is represented as

  \[
  \begin{array}{c}
  \ell \\
  p(\bar{y}) \rightarrow [\ ] \\
  m \\
  \sim p(\bar{y}) \rightarrow [\ ] \\
  \ell'
  \end{array}
  \]

- \( \ell : \quad \text{if } p(\bar{y}) \text{ then } \bar{y} := f(\bar{y}) \)
  \( \ell' \):

  is represented as

  \[
  \begin{array}{c}
  \ell \\
  p(\bar{y}) \rightarrow [\bar{y} := f(\bar{y})] \\
  \sim p(\bar{y}) \rightarrow [\ ] \\
  \ell'
  \end{array}
  \]
\( \ell : \text{loop until } p(\bar{y}) \)
\( \ell' : \)

This statement loops until the condition \( p(\bar{y}) \) becomes true. It is represented as

\[
\begin{array}{c}
\ell \\
\downarrow \quad p(\bar{y}) \rightarrow [] \\
\sim p(\bar{y}) \rightarrow [] \\
\ell'
\end{array}
\]

\( \ell : \text{loop while } p(\bar{y}) \)
\( \ell' : \)

This statement is the complement of the above statement: it loops until condition \( p(\bar{y}) \) is false. It is represented as

\[
\begin{array}{c}
\ell \\
\downarrow \quad \sim p(\bar{y}) \rightarrow [] \\
p(\bar{y}) \rightarrow [] \\
\ell'
\end{array}
\]

\( \ell : \text{compute } u_1, \ldots, u_r \text{ using } v_1, \ldots, v_s \)
\( \ell' : \)

This statement represents a segment of terminating computation in whose details we are not interested. The only facts we assume about this segment are:

1. The segment may modify only the program variables \( u_1, \ldots, u_r, r \geq 0 \),
   and may reference only the program variables \( v_1, \ldots, v_s, s \geq 0 \).
2. The segment must eventually terminate.

The statement is represented as

\[
\begin{array}{c}
\ell \\
\downarrow \quad \text{true} \rightarrow [(u_1, \ldots, u_r) := f(v_1, \ldots, v_s)] \\
\ell'
\end{array}
\]

where \( f \) represents an unspecified function.

We will often use compute segments of the form

\( \ell : \text{compute} \)
\( \ell' : \)

for the case \( r = s = 0 \) to refer to a segment of terminating computation that does not modify or access any program variables.
This statement represents an arbitrary program segment that may modify only the program variables \( u_1, \ldots, u_r, r \geq 0 \), and may reference only \( v_1, \ldots, v_s, s \geq 0 \). Here we do not require that the segment must eventually terminate. Consequently its representation is given by:

\[
\begin{align*}
\ell & : \quad \text{execute } u_1, \ldots, u_r \text{ using } v_1, \ldots, v_s \\
\ell' & :
\end{align*}
\]

\[
\begin{align*}
\text{true} & \rightarrow \left[ (u_1, \ldots, u_r) := f(v_1, \ldots, v_s) \right] \\
\text{false} & \rightarrow \ell'
\end{align*}
\]

- \( \ell_e : \) halt

is represented as:

\[
\begin{align*}
\ell_e & :
\end{align*}
\]

i.e., a node with no exits.

Note that for all the statements considered so far, except for the halt statement, the full-exit condition is always identically true. Also all the instructions (and their corresponding locations), except for the execute \( u_1, \ldots, u_r \) instruction, are deterministic, i.e., they have mutually exclusive transitions.

Example:

Consider the following concurrent program for computing the binomial coefficient \( \binom{n}{k} \) for integers \( n \) and \( k \), such that \( 0 \leq k \leq n \):

Program BC (Binomial Coefficient):

\[
\begin{align*}
y_1 & := n, \quad y_2 := 0, \quad y_3 := 1 \\
& \\
\ell_0 & : \text{if } y_1 = (n - k) \text{ then go to } \ell_e \\
\ell_1 & : y_3 := y_3 \cdot y_1 \\
\ell_2 & : y_1 := y_1 - 1 \\
\ell_3 & : \text{go to } \ell_0 \\
\ell_e & : \text{halt}
\end{align*}
\]

- Process \( P_1 \) -

\[
\begin{align*}
m_0 & : \text{if } y_2 = k \text{ then go to } m_e \\
m_1 & : y_2 := y_2 + 1 \\
m_2 & : \text{loop until } y_1 + y_2 \leq n \\
m_3 & : y_3 := y_3/y_2 \\
m_4 & : \text{go to } m_0 \\
m_e & : \text{halt}
\end{align*}
\]

- Process \( P_2 \) -

The input parameters to this program are \( n \) and \( k \). Note that \( n \) appears in the initial assignment while both \( n \) and \( k \) appear in statements of the processes.

We have not yet discussed the execution of concurrent programs in our model. Assume for a moment that each instruction in this program is atomic and that at any instant only one such
atomic instruction is executed. Once it is completed, another instruction (from either process) is executed to its completion, and so on. Under this assumption, the program $BC_0$ computes the binomial coefficient

$$\binom{n}{k} = \frac{n \cdot (n-1) \cdot \ldots \cdot (n-k+1)}{1 \cdot 2 \cdot \ldots \cdot k}$$

The values of $y_1$, i.e., $n, n-1, \ldots, n-k+1$, are used to compute the numerator in $P_1$ (the last value of $y$, $n-k$, is not used), and the values of $y_2$, i.e., $1, 2, \ldots, k$, are used to compute the denominator (the first value of $y_2$, 0, is not used). The process $P_1$ multiplies $n \cdot (n-1) \cdot \ldots \cdot (n-k+1)$ into $y_3$ while $P_2$ divides $y_3$ by $1 \cdot 2 \cdot \ldots \cdot k$.

The instruction

$$m_2: \text{ loop until } y_1 + y_2 \leq n$$

guarantees even divisibility. It synchronizes $P_2$'s operation with that of $P_1$ to ensure that $y_3$ is divided by $i$ only after it has been multiplied by $n - i + 1$. We rely here on the mathematical theorem that the product of $i$ consecutive positive integers: $k \cdot (k+1) \cdot \ldots \cdot (k+i-1)$ is always divisible by $i!$.

Now, consider the intermediate expression at $m_2$:

$$y_3 = \frac{n \cdot (n-1) \cdot \ldots \cdot (n-j+1)}{1 \cdot 2 \cdot \ldots \cdot (i-1)},$$

where $1 \leq i \leq j \leq n$, $y_1 = n - j$ and $y_2 = i$. The numerator consists of the product of $j$ consecutive positive integers and is therefore divisible by $i$ since $i \leq j$. If $j = i$, we have to wait until $y_1$ is decremented by the instruction in $\ell_2$ from $n - i + 1$ to $n - i$ before we can be absolutely sure that $(n - i + 1)$ has been multiplied into $y_3$. Thus, process $P_2$ waits at $m_2$ until $y_1 + y_2$ drops to a value less than or equal to $n$.

In order to keep track of the progress of the execution in each process we use a vector of location variables $\vec{v} = \{\pi_1, \ldots, \pi_m\}$ where each $\pi_i$ ranges over the label set $L_\alpha$ of process $P_\alpha$.

$$y := f_0(x)$$
The location variable \( \pi \) points to the location in \( P_i \) which is to be executed next.

CONCURRENCY AND ITS MODELLING BY INTERLEAVING

Before defining the execution of concurrent programs in our model, we should first study in more detail the actual behavior of a physically concurrent system.

As our motivating real-life situation we consider a system consisting of \( m \) physically separate processors \( \Pi_1, \ldots, \Pi_m \). Each of the processors \( \Pi_i \) is responsible for executing the process program \( P_i \). The shared program variables \( y_1, \ldots, y_n \) reside in a common memory \( M \) to which each of the processors must gain access in order to retrieve or store a value of a shared variable. In addition, each of the processors has its own set of private variables (registers). These are used to hold intermediate results of the computation or values which are not needed by the other processes. We will refer to these private registers as \( t_0, t_1, \ldots \). We assume that the shared memory, \( M \), is hardware protected to allow only one processor to access a shared variable at a certain instant. While the access is taking place, the particular variable accessed is unavailable to all other processors. Each access is restricted to a single operation, a value retrieval or a value update, but not both.

Consider for example the joint operation of two processors \( \Pi_1 \) and \( \Pi_2 \) which are executing the following concurrent program:

**Elementary Program EP**

\[
\begin{align*}
\ell_0 &: \ t_1 := y \\
\ell_1 &: \ t_1 := t_1 - 1 \\
\ell_2 &: \ y := t_1 \\
\ell_e &: \ \text{halt}
\end{align*}
\]

Each processor \( \Pi_i \) has its private register \( t_i, \ i = 1, 2 \). This program has been carefully constructed so that it uses only three standardized types of elementary instructions:

a. A shared retrieval (reference), transferring the current value of a shared variable into a private register:

\[
t_1 := y \quad \text{and} \quad t_2 := y.
\]

b. A shared update (modification), storing the value of a private register into a shared variable:

\[
y := t_1 \quad \text{and} \quad y := t_2.
\]

c. An internal computation of the form \( t_i := f(i) \) assigning to one register of a processor a value which is a function of the registers \( t \) of the same processor:

\[
t_1 := t_1 - 1 \quad \text{and} \quad t_2 := t_2 + 1.
\]
We also frequently use a fourth type of elementary instruction:

d. An internal test of the form

\[
\text{if } p(t) \text{ then go to } \ell,
\]

where \( \ell \) are registers of the same processor.

With the execution of the instructions of types a and b we can associate a unique event which is the actual access to the shared memory \( M \). We refer to these events as shared access events. For the simple program presented above we can associate the events \( r_i, i = 1, 2 \), with the retrieval of the value of the shared \( y \) at the instruction in locations \( l_0 \) and \( m_0 \) respectively. Similarly, we associate the events \( u_i, i = 1, 2 \), with the updating of the shared variable \( y \) at the instructions \( l_2 \) and \( m_2 \) respectively. No access event is associated with internal computations such as those at \( l_1 \) and \( m_1 \).

Since in our example all four accesses refer to the same variable \( y \), no two of them can occur exactly at the same time because of the exclusivity mechanism provided by the memory unit \( M \). Thus in any possible concurrent execution of this program we will observe a linear sequence of the occurrences of these four events. The only possible sequences are:

\[
\begin{align*}
& r_1, u_1, r_2, u_2 \quad \text{leading to a final value of } y = 0 \\
& r_2, u_2, r_1, u_1 \quad \text{leading to a final value of } y = 0 \\
& r_1, r_2, u_1, u_2 \quad \text{leading to a final value of } y = 1 \\
& r_1, r_2, u_2, u_1 \quad \text{leading to a final value of } y = -1 \\
& r_2, r_1, u_1, u_2 \quad \text{leading to a final value of } y = 1 \\
& r_2, r_1, u_2, u_1 \quad \text{leading to a final value of } y = -1.
\end{align*}
\]

For this program, the sequence of access events uniquely determines the final state of the computation.

While the access events themselves are constrained by the memory protection mechanism to form a linear sequence in which no two events coincide, the execution of the non-accessing part of the instructions will generally overlap in time. In fact, many different executions which greatly differ in the timing and overlaps of their non-accessing parts and instructions correspond to the same linear timing sequence of the accessing events, and hence yield the same final state. This proliferation of executions which all yield the same result and display essentially the same behavior makes the analysis of concurrent executions unnecessarily complicated.

Consequently, in order to reduce the complexity of analysis we use a simplified model in which the executions are restricted to be interleaved. An interleaved execution is one in which at any instant only one processor is executing an elementary instruction to its completion. Once the elementary instruction is completed, another processor may initiate an elementary instruction and proceed to complete it. Under this model, the execution proceeds as a sequence of discrete steps. In each step one enabled transition (instruction) is selected in one of the processes and is executed to completion.

The selection of the next process to be executed is personified by a scheduler who performs the selection. At each step of the computation the scheduler selects one process which has an enabled
transition and lets that process execute one instruction (transition). For the sake of completeness we also allow the scheduler to arbitrarily insert an idling step in which no process is scheduled, no instruction is performed, and the values of all program and location variables remain the same. In the case that no enabled transition is available, an idling step is the only choice that the scheduler has thereafter. In such a case we say that the program is deadlocked. A special case of this situation is when the program has terminated, i.e., all the processes have terminated.

When first encountered the model of interleaved execution may appear to be artificial and counterintuitive. In fact it seems to defeat the whole idea of concurrency — concurrent (overlapping) execution of instructions in different processes. Therefore we emphasize that the interleaving model is only a mathematical device for simplifying the analysis which proves to be adequate for the kind of non-quantitative analysis we consider here. That is, as long as we are not interested in questions about the timing of instructions and the running time of a program and make no assumptions about the relative speeds of the processors, the model of interleaved executions faithfully represents all the possible behaviors of the program.

We use the following definitions:

- An access to a variable in an instruction of a process $P_i$ is defined to be critical if it is either a modification of a variable which is accessed by other processes or an access to a variable which is modifiable by other processes.

- An instruction is said to obey the single (critical) access rule if it contains at most one critical access.

We can then state the following result:

**Proposition (single (critical) access):** Interleaved executions of a program $P$, all of whose instructions obey the single (critical) access rule, faithfully represent all concurrent executions of $P$.

Thus, it is possible to represent by interleaving all possible situations arising under concurrency. Since this approach greatly simplifies the analysis, we will adopt this it in our treatment of concurrent programs.

One necessary exception to the single access rule is semaphores.

**SEMAPHORES**

Semaphores are devices for achieving synchronization in concurrent systems ([DIJ1]). They are special atomic instructions denoted by $\text{request}(y)$ (also known as $P(y)$), and $\text{release}(y)$ (also known as $V(y)$), operating on the semaphore variable $y$.

The request instruction

- $t : \text{request}(y)$
  
  $t' :$

is equivalent to the single transition
The release instruction

\[ \ell : \text{release}(y) \]

\[ \ell': \]

is equivalent to the transition

\[ \ell \xrightarrow{\text{true}} \{ y := y + 1 \} \]

Semaphores are considered atomic (primitive) even under concurrent execution. Therefore when programs are transformed to single access form, the semaphore instructions should be preserved as atomic and not broken up into single access instructions. No other operations can be performed on semaphore variables.

Usually the semaphore variable \( y \) is initialized to 1. A process reaching a \( \text{request}(y) \) instruction will proceed beyond it only if \( y > 0 \), and then it will decrement \( y \) by 1, setting it to 0. Thus a location containing a \( \text{request}(y) \) instruction can be used as a checkpoint, synchronizing the process with other processes containing \( \text{request}(y) \) and \( \text{release}(y) \) instructions operating on the same \( y \).

Consider a concurrent program of form

\[ y = 1 \]

\[ \ell^1 : \text{request}(y) \]

\[ \ell^2 : \text{request}(y) \]

\[ \ell^k : \text{request}(y) \]

\[ m^1 : \text{release}(y) \]

\[ m^2 : \text{release}(y) \]

\[ m^k : \text{release}(y) \]

Assume, for example, that \( P_1 \) arrived first at \( \ell^1 \) when \( y \) was 1. It then went beyond \( \ell^1 \) and set \( y \) to 0. As long as \( P_1 \) is between \( \ell^1 \) and \( m^1 \), \( y \) will remain 0, and any other process, say \( P_2 \), which will attempt to go beyond its \( \text{request} \) statement \( \ell^2 \) will be held there since the enabling condition \( y > 0 \) is false. It must wait there for \( y \) to turn positive, which can only be caused by \( P_1 \) performing the \( \text{release}(y) \) operation at \( m^1 \). Even if \( P_1 \) and \( P_2 \) reach \( \ell^1 \) and \( \ell^2 \) simultaneously, the atomicity of the \( \text{request} \) instruction (which is required for exactly this reason) ensures that only one process can gain access to its region lying between \( \ell \) and \( m \). This region is called a critical section, and our use of semaphores in this example ensures mutual exclusion of access to the critical sections; that is, at most one of the processes may execute its critical section at any instant. Semaphores may also be used for a variety of other signalling and synchronization tasks.
Mutual exclusion of critical sections is necessary whenever two or more processes need to access a shared variable or device (such as disk) and wish to be protected from interference or attempts by the other processes to access the same resource while doing so.

Example:

Consider once more Program BC₀ (Binomial Coefficient). In order to recast it in the single access form we notice that the variable \( V₃ \) is the critically shared variable. Hence, we have to break the instruction

\[
\ell₁ : \quad y₃ := y₃ \cdot y₁
\]

into the sequence

\[
\ell₁ : \quad t₁ := y₃ \cdot y₁ \\
\ell'₁ : \quad y₃ := t₁
\]

Note that \( y₁ \) is modified only by \( P₁ \); hence its access at \( \ell₁ \) is non-critical.

Similarly we have to break the instruction

\[
m₃ : \quad y₃ := y₃/y₂
\]

into

\[
m₃ : \quad t₂ := y₃/y₂ \\
m₃' : \quad y₃ := t₂
\]

Note that both the assignments \( y₁ := y₁ - 1, y₂ := y₂ + 1 \) and the test \( y₁ + y₂ ≤ n \) already satisfy the single access rule.

The problem now is that of interference between the two new processes. Consider for example an execution which includes the sequence:

\[
\ell₁, \quad m₃, \quad \ell'₁, \quad m₃'.
\]

Following this execution we find that while the instruction at \( \ell'₁ \) stores a certain value into \( y₃ \), it is immediately overwritten by the value stored into it by the instruction at \( m₃' \). Thus the value of the computation performed in \( \ell₁ \) is completely lost and the result is of course invalid. To prevent such a mishap we must protect each of the sequences \((\ell₁, \ell'₁)\) and \((m₃, m₃')\) from interference by the other. The protection is done by using a semaphore variable \( y₄ \); the modified programs appears below:

**Program BC (modified Binomial Coefficient):**

\[
y₁ := n, \quad y₂ := 0, \quad y₃ := 1, \quad y₄ := 1
\]

\[
\ell₀ : \quad \text{if } y₁ = (n - k) \text{ then go to } \ellₑ \]
\[
ℓ₁ : \quad \text{request}(y₄) \]
\[
ℓ₂ : \quad \ell₁ := y₃ \cdot y₁ \\
ℓ₃ : \quad y₃ := \ell₁ \\
ℓ₄ : \quad \text{release}(y₄) \]
\[
ℓ₅ : \quad y₁ := y₁ - 1 \\
ℓ₆ : \quad \text{go to } \ell₀ \\
ℓₑ : \quad \text{halt}
\]

\[
m₀ : \quad \text{if } y₂ = k \text{ then go to } mₑ \]
\[
m₁ : \quad y₂ := y₂ + 1 \\
m₂ : \quad \text{loop until } y₁ + y₂ ≤ n \\
m₃ : \quad \text{request}(y₄) \]
\[
m₄ : \quad t₂ := y₃/y₂ \\
m₅ : \quad y₃ := t₂ \\
m₆ : \quad \text{release}(y₄) \]
\[
m₇ : \quad \text{go to } m₀ \\
mₑ : \quad \text{halt}
\]
The mutually protected critical sections are \((\ell_2, \ell_3, \ell_4)\) and \((m_4, m_5, m_6)\) respectively. Their exclusion ensures that each computed value of \(y_3\) is assigned to \(y_3\) without any interference. Under interleaved executions, \(BC\) computes the binomial coefficient and is in single reference form.

Example:

Consider the following program \(CP\) modelling a consumer-producer situation:

Program \(CP\) (Consumer Producer):

\[
\begin{align*}
b &:= A, \quad s := 1, \quad cf := 0, \quad ce := N \\
\ell_0 &:= \text{compute } y_1 \quad m_0 := \text{request(cf)} \\
\ell_1 &:= \text{request(cf)} \quad m_1 := \text{request(s)} \\
\ell_2 &:= \text{request(s)} \\
\ell_3 &:= t_1 := b \circ y_1 \\
\ell_4 &:= b := t_1 \\
\ell_5 &:= \text{release(s)} \\
\ell_6 &:= \text{release(cf)} \\
\ell_7 &:= \text{go to } \ell_0 \\
m_2 &:= y_2 := \text{head(b)} \\
m_3 &:= t_2 := \text{tail(b)} \\
m_4 &:= b := t_2 \\
m_5 &:= \text{release(s)} \\
m_6 &:= \text{release(cf)} \\
m_7 &:= \text{compute using } y_2 \\
m_8 &:= \text{go to } m_0 \\
p_1 &:= \text{Producer} \\
p_2 &:= \text{Consumer}
\end{align*}
\]

The program is in single access form. The producer \(P_1\) computes a value into \(y_1\) without using any other program variables; the computation details are irrelevant. It then adds \(y_1\) to the end of the buffer \(b\). The consumer \(P_2\) removes the first element of the buffer into \(y_2\) and then uses this value for its own purposes (at \(m_7\)). It is assumed that the maximal capacity of the buffer \(b\) is \(N > 0\). The ‘compute using \(y_2\)’ instruction references \(y_2\) but does not modify any of the shared program variables.

In order to ensure the correct synchronization between the processes we use three semaphore variables:

- The variable \(s\) ensures that the accesses to the buffer are protected and provides exclusion between the sections \((\ell_3, \ell_4, \ell_5)\) and \((m_2, m_3, m_4, m_5)\).

- The variable \(ce\) ("count of empties") counts the number of free available slots in the buffer \(b\). It protects the buffer \(b\) from overflowing. The producer cannot deposit a value in the buffer if \(ce = 0\), and when it does deposit a value, it decrements \(ce\) by 1. Since we start with \(ce = N\), the producer cannot deposit more than \(N\) items before the consumer has removed any of them. The consumer, on the other hand, increments \(ce\) by 1 whenever it removes an item and creates a new vacancy.

- The variable \(cf\) ("count of fulls") counts how many items the buffer currently holds. It is initialized to 0, incremented by the producer whenever a new item is deposited, and decremented by the consumer whenever an item is removed. It ensures that the consumer does not attempt to remove an item from an empty buffer.
FAIRNESS

Another problem with modelling concurrency by interleaving is fairness. Consider first a program with no semaphore instructions, and where the full-exit condition $E_d(y)$ at each nonterminal location $l$ (i.e., $l \neq l_e$) is identically true, i.e., $E_d(y) = \text{true}$ for every $y$. Note that the latter is true for every linear-text program without semaphores. Under these restrictions every process that has not yet terminated is enabled, i.e., it always has an enabled transition, and if selected by the scheduler can always execute this transition. Running under true concurrency, every process will go on executing until it reaches its termination label $l_e$.

In order to model the same property under interleaving execution we require the scheduler to be fair. By that we mean that no process which is ready to run (i.e., enabled) will be neglected forever. Stated more precisely, we exclude infinite executions in which a certain process which has not terminated is never scheduled from a certain point on. Note that all finite terminating sequences are necessarily fair. This will also prevent the scheduler from going on an infinite spree of idling steps when at least one process is enabled.

Coming back to the more general situation which allows semaphore instructions, we have to consider the possibility that a nonterminated process is not continuously enabled. Furthermore, its being enabled may depend on the action of the other processes, since in general the full-exit condition $E_d(y)$ depends on the shared variables $y$.

Our requirement of fairness for this more general case will be formulated as:

We disallow infinite sequences in which a certain process is enabled infinitely often and is scheduled only a finite number of times.

Example:

Consider the simplest case of two processes synchronized by a semaphore:

\[
y := 1
\]

\[
l_0 : \text{request}(y) \quad m_0 : \text{request}(y)
\]
\[
l_1 : \text{release}(y) \quad m_1 : \text{release}(y)
\]
\[
l_2 : \text{go to } l_0 \quad m_2 : \text{go to } m_0
\]
\[
-P_1- \quad -P_2-
\]

Obviously the infinite execution sequence (where we only mention the label arrived at as a result of the current transition)

\[
l_1, l_2, l_0, m_1, m_2, m_0, l_1, l_2, l_0, m_1, m_2, m_0, \ldots
\]

is fair. On the other hand the sequence:

\[
l_1, l_2, l_0, l_1, l_2, l_0, \ldots
\]

while constantly $\pi_2 = m_0$ is unfair. This is because whenever $\pi_1 = l_0$ or $\pi_1 = l_2$, $P_2$ is enabled. Thus in this sequence, even though $P_2$ is not continuously enabled (it is not enabled when $\pi_1 = l_1$),

32
it is enabled infinitely often. Since \( P_2 \) is never scheduled while being enabled infinitely often this sequence is unfair.

In practice every scheduler which is fair satisfies a stronger requirement: it is fair within a finite bound, i.e., no enabled process may be neglected for more than \( k \) instants of being enabled. Here \( k \) is a constant, characteristic of the scheduler.

Generalizing the semaphore instruction \( \text{request}(y) \) which waits for \( y \) to turn positive and then decrements it, we have the 'wait until \( p(y) \)' and 'wait while \( p(y) \)' instructions. They are modelled as follows:

\[ \text{\checkmark} \quad \ell : \ \text{wait until } p(\bar{y}) \]
\[ \ell' : \]

is represented by

\[ \begin{array}{c}
\ell \\
\quad p(\bar{y}) \rightarrow [1] \\
\quad \ell'
\end{array} \]

and

\[ \text{\checkmark} \quad \ell : \ \text{wait while } p(\bar{y}) \]
\[ \ell' : \]

is represented by

\[ \begin{array}{c}
\ell \\
\quad \neg p(\bar{y}) \rightarrow [1] \\
\quad \ell'
\end{array} \]

The wait instructions are similar to the request instruction in that the full-exit condition is not identically true. Thus for the 'wait until \( p(y) \)' instruction, the full-exit condition \( E_{\ell}(y) \) is equal to \( p(y) \). Consequently fairness considerations ensure that if \( p(y) \) turns true infinitely often while a process is waiting at \( \ell \) it will eventually be scheduled (exactly when \( p(y) \) is true) and proceed to \( \ell' \).

Let us compare the 'wait until \( p(y) \)' instruction with the 'loop until \( p(y) \)' instruction whose graph representation is

\[ \begin{array}{c}
\ell \\
\quad \neg p(\bar{y}) \rightarrow [1] \\
\quad p(\bar{y}) \rightarrow [1] \\
\quad \ell'
\end{array} \]

Note that the full-exit condition for this instruction is \( E_{\ell} = \text{true} \). Thus even if \( p(y) \) turns true infinitely often we are not assured of ultimately reaching \( \ell' \). This is so because the only requirement implied by fair scheduling is that if \( E_{\ell} \) is infinitely often true the process waiting at \( \ell \) must eventually be scheduled at an instant in which \( E_{\ell} \) is true. However this instant may always happen to be one in which \( p(y) = \text{false} \) and the instruction executed is a transition back to \( \ell \).

The only condition that will guarantee for a loop instruction the eventual exit to \( \ell' \) is that \( p(y) \) becomes permanently true beyond a certain stage in the computation.

There are practical implications to the distinction between the wait and loop instructions. If we wish to implement an actual fair interleaving scheduler, it is easier to be fair to the loop instruction than to the wait instruction. Since for the loop instruction, \( E_{\ell} \) is identically true, in order to be fair
to a process which is at \( z \), the scheduler just has to make sure it does not neglect it and eventually comes around to scheduling it. In order to be fair to a \textit{wait} instruction, whose full-exit condition is \( p(y) \), we have to monitor the instants in which \( p(y) \) is true. Then when it is observed that \( p(y) \) is true many times the relevant process has to be eventually scheduled.

On the other hand, the use of a \textit{wait} instruction implies greater efficiency since the scheduler may place the process executing a \textit{wait} instruction on a suspension list, from which it will be removed only when \( p(y) \) is true and the scheduler decides to schedule that process.
3. THE TEMPORAL DESCRIPTION OF PROGRAM PROPERTIES

As we have seen, the behavior of a concurrent program is characterized by the set of its fair execution sequences. We have also developed the formalism of temporal logic whose formulas are interpreted over sequences. We now combine the two and utilize temporal logic to state properties of the execution sequences of a given program, thus describing properties of the dynamic behavior of the program ([PNU1], [MP]).

In order to apply the general temporal formalism to execution sequences, it is necessary to introduce additional structure and special notation into the temporal language. For states we will consider "execution states" which each consist of the vector of current locations in the program and of the current values of all program variables at a certain stage in the execution. The accessibility relation between execution states will represent "derivability" by the program's execution. We will use predicates and propositions to describe properties of a single state, and modalities to describe properties of the execution leading from one state to another.

Consider a typical concurrent program

\[ P = y := f_0(x); [P_1] \ldots [P_m] \]

with input parameters \( x = (x_1, \ldots, x_k) \) and shared program variables \( y = (y_1, \ldots, y_n) \) over a domain \( D \). (For simplicity, we do not consider many-sorted domains.)

An execution state for this program has the general structure

\[ s = (\bar{x}; \bar{y}), \]

where

- \( \bar{x} = (\lambda_1, \ldots, \lambda_m) \) is the vector of current values held by the location variables \( \bar{x} \). Thus \( \lambda_i \in L_i \) is the label of the node in the transition graph of process \( P_i \) where execution is to resume next. (It is the label of the next instruction to be executed in the linear-text representation.)
- \( \bar{y} = (y_1, \ldots, y_n) \in D^n \) is the vector of data values assumed by the program variables \( \bar{y} \) in the state \( s \). Thus \( y_i \in D \) is the current value of \( y_i \) in \( s \).

An execution sequence of a concurrent program is an infinite sequence of states:

\[ \sigma = s_0, s_1, s_2, \ldots \]

Corresponding to the structure of execution states and sequences we will consider temporal formulas with the following individual variables:

(a) Local program variables: \( y_1, \ldots, y_n \).

These represent the current values of the program variables which of course may vary from one execution state to the other.

(b) Local location variables: \( \pi_1, \ldots, \pi_m \).
These represent the location of each process in a given state. Each $\pi$, will range over the set $L_i$.

(c) Global variables: $x_1, \ldots, x_k, u_1, u_2, \ldots$

These are the input parameters $x_1, \ldots, x_k$, and auxiliary variables $u_1, u_2, \ldots$ which stay constant over the complete execution, i.e., they do not vary from state to state. The auxiliary variables are used to express relations between local values in different states. For example:

$$\forall u[(y = u) \circ (y = u + 1)]$$

expresses the statement that there will be a future instant in which the value of the variable $y$ will be greater by 1 than its current value.

For a label $\ell \in L_j$, we abbreviate the atomic formula $\pi_j = \ell$ to $\text{at}_\ell$, i.e.,

$$\text{at}_\ell \text{ is true iff } \pi_j = \ell,$$

which may therefore be considered a local proposition. Thus, for a given state $s = (\bar{x}; \bar{\eta})$ and location $\ell \in L_j$, $\text{at}_\ell$ is true at $s$ if the process $P_j$ is currently at $\ell$, i.e., $\lambda_j = \ell$.

More generally, for a set of labels $L \subseteq L_j$ the local proposition $\text{at}_L$ is defined to be true if $P_j$ is anywhere within $L$, i.e.,

$$\text{at}_L \text{ is true iff } \pi_j \in L.$$

If $L$ consists of all the labels $\ell_i$ within a segment, i.e., $L = \{\ell_a, \ell_{a+1}, \ldots, \ell_b\}$ for some $0 \leq a \leq b$, we will also write $\text{at}_L$ as $\text{at}_{[a..b]}$. Thus,

$$\text{at}_{[a..b]} = \text{at}_{\{\ell_a, \ell_{a+1}, \ldots, \ell_b\}} = \bigvee_{i=a}^{b} \text{at}_{\ell_i}.$$

We proceed to give a precise definition for the set of legal execution sequences $\sigma$, corresponding to a given program $P$ with input values $\bar{x} = \bar{\xi}$. There are three requirements which a legal execution sequence ought to fulfill:

A. Initialization

An execution sequence

$$\sigma = s_0, s_1, s_2 \ldots$$

is properly initialized if $s_0 = (\bar{x}_0; \bar{\eta}_0)$ has the structure:

- $\bar{x}_0 = (\ell_0^1, \ldots, \ell_0^m)$, the set of initial locations in each of the processes;
- $\bar{\eta}_0 = f(\bar{\xi})$, the initial values assumed by the program variables on initialization.

B. State to state transitions

An execution sequence $\sigma$ is admissible if each $s_{k+1} = (\bar{x}_i'; \bar{\eta}_i')$ is related to $s_k = (\bar{x}_i; \bar{\eta}_i)$ by one of the following rules:
(a) **Idling step:** \( s_{k+1} = s_k \) (i.e., \( \lambda' = \lambda, \eta' = \eta \))

(b) **An i-step:** For some \( i, 1 \leq i \leq m \), we have the following: The process \( P_i \) contains a transition

\[
\begin{array}{c}
\lambda_i \\
\xrightarrow{c(\eta)} [y := f(\eta)] \\
\lambda'_i
\end{array}
\]

such that \( \pi_i = \lambda_i, c(\eta) = \text{true} \) (i.e., the transition is enabled) and \( \eta' = f(\eta) \). For all \( j, j \neq i \), we have \( \lambda'_j = \lambda_j \).

Note that in the presence of self loops, i.e.,

we cannot always uniquely decide whether an idling step or a trivial i-step led from state \( s_k \) to state \( s_{k+1} \).

### C. Fairness or Justice

- An admissible sequence \( \sigma \) is **just** if there is no process \( P_t \) which is continuously enabled beyond a certain state \( s \) in the sequence \( \sigma \), and only a finite number of steps of \( \sigma \) are i-steps.

Thus the notion of justice ensures that no process is indefinitely neglected. This notion is adequate for programs with no semaphore instructions.

- An admissible sequence \( \sigma \) is **fair** if there is no process \( P_t \) which is enabled an infinite number of times in \( \sigma \), and only a finite number of steps of \( \sigma \) are i-steps.

Note that a fair sequence is also a just sequence. In addition to the assurances given by justice, fairness guarantees that no process will remain blocked at a semaphore instruction whose exit condition turns true infinitely often. For programs without semaphore instructions the notions of fairness and justice coincide. Consequently, our treatment will concentrate on fair executions.

Note that in checking for fairness we are allowed to take a given step both as an i-step and as a j-step if both interpretations are possible. Thus the following degenerate program

\[
\ell_0 : \text{go to } \ell_0 \\
m_0 : \text{go to } m_0
\]

possesses the legal execution sequence

\[
((\ell_0, m_0); ()}, ((\ell_0, m_0); ()}, \ldots
\]

Each step here may be interpreted as an idling step, a 1-step or a 2-step. Because of this possible multiple interpretation the sequence is indeed fair.

Consider the sequence corresponding to a terminating computation, i.e., all processes have terminated. Since in a terminating state \( \pi_t = \ell_e \) the process \( P_1 \) is never enabled, the fairness criterion does not require further scheduling of \( P_1 \), and the only possible steps from that point on are idling steps. Thus our representation of a terminating computation as an infinite sequence in which from a certain point on all states are identical is consistent with fairness. This state, to which the sequence has "converged," is the terminal state.
Every suffix of a properly $\xi$-initialized, admissible, fair execution sequence is defined to be a $(P, \xi)$-computation. The set of all $(P, \xi)$-computations is denoted by $\mathcal{F}(P, \xi)$. By definition, this set is suffix closed, i.e., if $\sigma \in \mathcal{F}(P, \xi)$, then $\sigma^i \in \mathcal{F}(P, \xi)$ for every $i \geq 0$.

For a given program $P$ let $\varphi(x)$ be a restriction (precondition) on the input parameters $x$. Usually $\varphi$ characterizes the inputs we expect the program to operate on.

A computation is said to be a $(P, \varphi)$-computation (proper computation) if it is a $(P, \xi)$-computation for some $\xi$ such that $\varphi(\xi)$ is true.

We define the set $\mathcal{F}(P, \varphi)$ to be the set of all $(P, \varphi)$-computations. Obviously $\mathcal{F}(P, \varphi)$ also has the suffix closure property.

A formula $w$ is $\mathcal{F}(P, \varphi)$-valid if it is true for every computation in $\mathcal{F}(P, \varphi)$. Such a formula is obviously an established valid property of all $(P, \varphi)$-computations. In the following sections we study the expression of program properties as $\mathcal{F}(P, \varphi)$-valid formulas.

Since most of our reasoning will be done in the context of a fixed program $P$ and a fixed precondition $\varphi$, we introduce a special notation for $\mathcal{F}(P, \varphi)$ validity. We denote

$$\mathcal{F}(P, \varphi) \models w$$

by $\models w$.

The statement $\models w$ thus means that $w$ is true for every suffix of a fair, admissible execution of $P$ which is initiated at $\xi_0 = (t_0, \ldots, t_n)$ with $\varphi(x)$ holding and $y = j_0(x)$.

Facts of the form $\models w$ will serve as the basic statements in our specification and description of program properties. Consequently, we will discuss in later reports proof rules for deriving such statements.

The following is an important derivation:

$$\models w \Rightarrow \models w.$$

It states that if $w$ is true for every possible sequence it is true in particular for every $(P, \varphi)$-computation. This enables us to transport all the generally known valid temporal statements ($\models$-valid) into reasoning about a particular program ($\models$-valid). Thus the following are $\models$-valid formulas:

$$\models \Box \sim w \equiv \sim \Diamond w$$
$$\models \Box (w_1 \sqcup w_2) \supset (\Box w_1 \supset \Box w_2)$$
$$\models \Box (w \sqcup \Diamond w) \supset (w \supset \Box w)$$

etc.

Another valid inference is

$$\models w \Rightarrow \models \Box w$$

This rule states that if $w$ is true for all the $(P, \varphi)$-computations then $\Box w$ is also true for them. This rule is a direct consequence of the suffix closure property of $\mathcal{F}(P, \varphi)$. One can prove similarly that
all the inference rules (numbers 56 to 66) proven in the earlier repertoire still hold after replacing \( \mathsf{P} \) by \( \mathsf{P} \).

We will now review the expression of program properties by temporal formulas. The properties will be classified according to the form of the temporal formulas expressing them.

**INVARIANCE (SAFETY) PROPERTIES**

Consider first the class of program properties that hold continuously throughout all computations. They are expressible by formulas of the form:

\[ \mathsf{P} \Box w. \]

Such a formula states that \( \Box w \) holds for every computation, i.e., \( w \) is an invariant of every computation. By the generalization rule this could have been written as \( \mathsf{P} w \), but we prefer the above form since it emphasizes the invariant character of the properties in this class.

Note that the initial condition associated with the proper computation is:

\[ s \in \mathcal{L}_0 \land \bar{y} = f_0(x) \land \varphi(x) \]

which characterizes the initial state for inputs \( x \) satisfying the precondition \( \varphi(x) \). Here, \( \mathcal{L}_0 = (L_0^1, \ldots, L_0^n) \) is the set of initial locations in each of the processes. To emphasize the precondition \( \varphi(x) \) we sometimes express \( \mathsf{P} \Box w \) as

\[ \mathsf{P} \varphi(x) \supset \Box w. \]

A formula of this form therefore expresses an invariance property. The properties in this class are also known as safety properties, based on the premise that they ensure that “nothing bad will ever happen” ([LAM1]).

More generally, invariance properties can be expressed by formulas of the form

\[ \mathsf{P} w_0 \supset \Box w. \]

This form may be used to state that a certain event implies the invariance of some other condition from that moment on. Under this interpretation \( w_0 \) is the triggering event whose occurrence causes the subsequent invariance of the property \( w \).

We give below a sample of important properties falling under this category.

a. **Partial Correctness**

This property is meaningful only for programs in which each process contains a terminal location \( L \). We call such programs terminating programs, in contrast with continuous (or cyclic programs) whose proper behavior does not call for termination and therefore do not contain terminal locations.
Let $\varphi(\bar{x})$ be the precondition that restricts the set of inputs for which the program is supposed to be correct, and $\psi(\bar{x}, \bar{y})$ the statement of its correctness, i.e., the relation that should hold between the input values $\bar{x}$ and the output values $\bar{y}$. Then in order to state partial correctness with respect to a specification $(\varphi, \psi)$ we can write:

$$\varphi(\bar{x}) \supset \Box (at\bar{t}_e \supset \psi(\bar{x}, \bar{y})),$$

where $\bar{t}_e = (t^1_e, \ldots, t^m_e)$ is the vector of terminal locations in each of the processes. This formula claims that if the initial state satisfies the precondition, then in any state accessible from it: If that state happens to be an exit state, i.e. $\bar{x} = \bar{t}_e$, then the relation $\psi(\bar{x}, \bar{y})$ holds between the input parameters $\bar{x}$ and the current values of $\bar{y}$. Thus this formula states that all convergent $\varphi$-computations terminate in a state satisfying $\psi$, but it does not guarantee termination itself. Note that we rely on $\bar{x}$ being global and retaining its original value throughout the computation.

**Example:**

Let us consider as a concrete example, a single process program for computing $x!$ over the nonnegative integers.

**Program F (Factorial Program):**

$$
\begin{align*}
y_1 & := x, \quad y_2 := 1 \\
l_0 & : \text{if } y_1 = 0 \text{ then goto } l_e \\
l_1 & : (y_1, y_2) := (y_1 - 1, y_1 \cdot y_2) \\
l_2 & : \text{goto } l_0 \\
l_e & : \text{halt.}
\end{align*}
$$

The statement of its partial correctness is

$$
\Box (x \geq 0) \supset \Box (at\bar{t}_e \supset y_2 = x!),
$$

where the initial condition associated with the proper computation is actually

$$atl_0 \land y_1 = x \land y_2 = 1 \land x \geq 0.$$

We are justified in regarding partial correctness as an invariance property since it is actually a part of a "network of invariants" normally used in the Invariant-Assertion Method; namely, for the Program $F$ above:

$$
\Box (x \geq 0) \supset \Box \{ \begin{align*}
& atl_0 \supset (y_1 \geq 0) \land (y_2 \cdot y_1! = x!) \\
& atl_1 \supset (y_1 > 0) \land (y_2 \cdot y_1! = x!) \\
& atl_2 \supset (y_1 \geq 0) \land (y_2 \cdot y_1! = x!) \\
& atl_e \supset (y_1 = 0) \land (y_2 = x!).
\end{align*} \}.
$$

And in fact, in order to prove the partial correctness property, we usually prove the invariance of this larger formula, from which partial correctness follows. 

**Example:**

40
As another example consider a program $TN$ counting the number of nodes in a binary tree $X$.

**Program $TN$ (Counting the nodes of a tree):**

\[
S := (X), \quad C := 0
\]

\[
\ell_0 : \quad \text{if } S = () \text{ then goto } \ell_4
\]

\[
\ell_1 : \quad (T, S) := (\text{hd}(S), \text{tl}(S))
\]

\[
\ell_2 : \quad \text{if } T = \Lambda \text{ then goto } \ell_0
\]

\[
\ell_3 : \quad C := C + 1
\]

\[
\ell_4 : \quad S := \ell(T) \cdot r(T) \cdot S
\]

\[
\ell_5 : \quad \text{goto } \ell_0
\]

\[
\ell_6 : \quad \text{halt.}
\]

The program operates on a tree variable $T$ and a variable $S$ which is a stack of trees. The input variable $X$ is a tree. The output is the value of the counter $C$. Each node in a tree may have zero, one or two descendants.

The available operations on trees are the functions $\ell(T)$ and $r(T)$ which yield the left and right subtrees of a tree $T$ respectively. If the tree does not possess one of these subtrees the functions return the value $\Lambda$.

The stack $S$ is initialized to contain the tree $X$. Taking the head and tail of a stack (functions hd and tl respectively) yields the top element and rest of the stack respectively. The operation in $\ell_1$ pops the top of the stack into the variable $T$. The operation at $\ell_4$ pushes both the right subtree and the left subtree of $T$ onto the top of the stack.

At any iteration of the program, the stack $S$ contains the list of subtrees of $X$ whose nodes have not yet been counted. The iteration removes one such subtree from the stack. If it is the empty subtree, $T = \Lambda$, we proceed to examine the next subtree on the stack. If it is not the empty subtree we add one to the counter $C$ and pushes the left and right subtrees of $T$ to the stack. When the stack is empty, $S = ()$, the program halts.

Denoting by $|T|$ the number of nodes in a tree $T$ we can express the statement of partial correctness of the program $TN$ by:

\[
\models [\text{at } \ell_4 \supset C = |X|].
\]

The actual initial condition associated with the proper computation is

\[
\text{at } \ell_0 \land S = (X) \land C = 0.
\]

**Example:**

As a more complex example consider again the program $BC$ for the concurrent computation of a binomial coefficient.

The statement of partial correctness to be proved there is:

\[
\models (0 \leq k \leq n) \supset [\text{at } \ell_6 \land \text{atm}_s \supset y_3 = \binom{n}{k}].
\]
That is, every properly initialized execution of the program BC that terminates satisfies $y_3 = \frac{n}{2}$ at its termination point. The actual initial condition associated with the proper computation is

$$ at\ell_0 \land atm_0 \land y_1 = n \land y_2 = 0 \land y_3 = 1 \land y_4 = 1 \land 0 \leq k \leq n. $$

b. Clean Behavior

For every location in a program we can formulate a cleanness condition that states that the instruction at this location will execute successfully and will generate no execution faults (exceptions). Thus if the statement contains a division, the cleanness condition will include the clause specifying that the divisor is nonzero or not too small (to avoid arithmetic overflow). If the statement contains an array reference, the cleanness condition will state that the subscript expressions are within the declared range. Denoting the cleanness condition at location $\ell$ by $\alpha_\ell$, the statement of clean behavior is:

$$ \varphi(x) \supset \Box \bigwedge_\ell (at\ell \supset \alpha_\ell). $$

The conjunction is taken over all “potentially dangerous” locations in the program.

Example:

The factorial program $F$ above should produce only natural number values during its computation. A cleanness condition at $\ell_1$, which is clearly a critical point, is (under the precondition $x \geq 0$)

$$ \varphi(x) \supset \Box [at\ell_1 \supset (y_1 > 0)], $$

guaranteeing that the subtraction performed at $\ell_1$ always yields a natural number. Note that we have not indicated that $y_1$ is an integer; such type considerations will be ignored in our discussions.

Example:

If a program contains the instruction

$$ \ell: \text{if } y_1 > y_2 \text{ then } y_1 := (S[i] \div y_2), $$

where $\div$ is the integer-division operator and the range of the array subscript $i$ is between 1 and $m$, then the cleaness condition at $\ell$ can be expressed as follows:

$$ \Box \{ [at\ell \land (y_1 > y_2)] \supset [(1 \leq i \leq m) \land (y_2 \neq 0)] \}. $$

Example:

A clean behavior statement for the tree node counting program $TN$ is given by:

$$ \Box (at\ell_1 \supset S \neq (\_)) \land (at\ell_4 \supset T \neq \_). $$
This ensures that no attempt is made to pop an empty stack or to decompose an empty tree.

Example:

In the binomial coefficient program $BC$ an appropriate and crucial cleanliness statement is given by:

$$\varphi \left( 0 \leq k \leq n \right) \supset \Box \left\{ \text{atm}_4 \supset \left[ \left( y_2 \neq 0 \right) \land \left( y_3 \mod y_2 = 0 \right) \right] \right\}.$$ 

That is, whenever we reach the location $m_4$ in a proper computation of $BC$, $y_3$ is evenly divisible by $y_2$.

A general concern in the considerations of clean behavior is the compatibility of values with types. In the presence of dynamic types we should also worry about the compatibility of types.

c. Global and Local Invariants

Very frequently, invariant properties are not related to any particular location. In general, some properties may be invariant independent of the location. In these cases we speak of global invariants, i.e., invariants unattached to any particular location. The expression of global invariance is even more straightforward. Thus, we write

$$\varphi(\pi) \supset \Box \beta,$$

to state that property $\beta$ holds at all times during a proper computation.

Example:

In the factorial program $F$ above, to claim that $y_1$ is always a nonnegative integer, we may write:

$$\varphi \left( x \geq 0 \right) \supset \Box \left( y_1 \geq 0 \right).$$

Another valid global invariant for this program is:

$$\varphi \left( x \geq 0 \right) \supset \Box \left( y_2 \cdot y_1! = x! \right),$$

which states that $y_2 \cdot y_1! = x!$ at all steps of the execution.

Example:

For the binomial coefficient program $BC$, an appropriate global assertion would be:

$$\varphi \left( 0 \leq k \leq n \right) \supset \Box \left\{ \left[ n - k \leq y_1 \leq n \right] \land \left[ 0 \leq y_2 \leq k \right] \right\}.$$ 

Another interesting set of properties are invariance properties which are attached to particular locations, but not necessarily to the exit locations of the program. These properties are particularly important for programs which have no exits and are expected to run indefinitely.
We refer to such properties as *local invariants* and write

\[ \Box (at \ell \triangleright \beta) \]

to indicate that a statement \( \beta \) is true whenever we are at a certain location \( \ell \). Partial correctness is actually a local invariant referring to the exit locations.

**Example:**

In the \( TN \) program for counting the nodes in a tree, we can express as a local invariant the fact which is true whenever we visit the location \( \ell_0 \); namely,

\[ \Box (at \ell_0 \triangleright (\sum_{t \in S} |t| + C = |X|)), \]

i.e., the sum of the number of nodes in all the subtrees currently in the stack plus the current value of the counter \( C \) is invariant at \( \ell_0 \) and equals the number of nodes in the tree \( X \).

Invariants can also be used in the context of a program whose output is not necessarily apparent at the end of the execution; for example, a sequential program whose output is printed on an external file during the computation.

**Example:**

Consider the following program \( PR \) for printing the infinite sequence of successive prime numbers

\[ 2, 3, 5, 7, 11, 13, 17, \ldots \]

*Program PR (Printing the prime numbers):*

\[
\begin{align*}
y_1 & := 2 \\
\ell_0 & : print(y_1) \\
\ell_1 & : y_1 := y_1 + 1 \\
\ell_2 & : y_2 := 2 \\
\ell_3 & : if \ (y_2)^2 > y_1 then \ goto \ \ell_0 \\
\ell_4 & : if \ (y_1 \ mod \ y_2) = 0 then \ goto \ \ell_1 \\
\ell_5 & : y_2 := y_2 + 1 \\
\ell_6 & : \ goto \ \ell_3
\end{align*}
\]

A part of the correctness statement for this program is:

\[ \Box (at \ell_0 \triangleright \text{prime}(y_1)); \]

it indicates that only primes are printed.

Next we will examine some properties which are meaningful only for concurrent programs.
d. Mutual Exclusion

The notions of critical sections and mutual exclusion were introduced earlier, but let us briefly review them.

Consider two processes $P_1$ and $P_2$ being executed in parallel. Assume that each process contains a section $C_i \subseteq L_i$, for $i = 1, 2$, which includes some task critical to the cooperation of the two processes. For example, it might access a shared device (such as a disk) or a shared variable. If the nature of the task is such that it must never be done by both of them simultaneously, we call these sections \textit{critical sections}. The property stating that the processes will never simultaneously execute their respective critical sections is called \textit{mutual exclusion} with respect to this pair of critical sections.

The property of mutual exclusion for $C_1$ and $C_2$ can be described by:

$$\varphi(x) \supset \square \neg (atC_1 \land atC_2).$$

This states that it is never the case that the joint execution of the processes reaches $C_1$ and $C_2$ simultaneously.

\textit{Example:}

Consider again the consumer-producer program $CP$. The sections

$$C_1 = \{t_3, t_4, t_5\} \text{ in } P_1$$

and

$$C_2 = \{m_2, m_3, m_4, m_5\} \text{ in } P_2$$

are obviously critical sections since they make several accesses to the shared variable $b$. In order to obtain the correct result it must be ensured that no other accesses to $b$ are made during the computation involving $b$.

The mutual exclusion property in this case can be expressed by:

$$\square \neg (atC_1 \land atC_2),$$

where the initial condition associated with the proper computations is:

$$at\ell_0 \land atm_0 \land b = \Lambda \land s = 1 \land cf = 0 \land ce = N.$$  

The formula states that we can never simultaneously be in both critical sections $C_1$ and $C_2$. Note that actually it suffices to prove

$$\square \neg (at\ell_3 \land atm_2).$$

This is so because there exists an execution in which $at\ell_3 \land atm_2$ in some state if and only if there exists an execution in which $atC_1 \land atC_2$ in some state.
Example:

Similarly a statement of mutual exclusion for the program BC computing the binomial coefficient is given by:

\[ \neg (0 \leq k \leq n) \supset \square \neg (atl_{2,4} \land atm_{4,6}). \]

Here, we follow our convention,

\[ atl_{2,4} \] denotes \( \pi_1 \in \{ \ell_2, \ell_3, \ell_4 \} \]

and

\[ atm_{4,6} \] denotes \( \pi_2 \in \{ m_4, m_5, m_6 \} \).

Deadlock Freedom

A concurrent program consisting of \( m \) processes is said to be deadlocked if no process is enabled. This leaves the idling step as the only possible choice of the scheduler. The rest of the computation will therefore consist of an endless repetition of the current deadlocked state. Clearly in a deadlock situation each process \( P_j \) must be blocked at a location \( \ell \in L_j \) whose full-exit condition \( E_\ell \) is false for the current value \( y \) of \( y \). Therefore the only potential deadlock locations are those \( \ell \) for which \( E_\ell \) is not identically true. We refer to such locations as \textit{waiting locations}. The terminal location \( \ell_t \) is also considered to be a waiting location. However, the special case in which \textit{all} processes are at their respective \( \ell_t \) locations is not considered to be a deadlock but rather a \textit{termination}.

Let us therefore consider a tuple \( \vec{\ell} = (\ell_1, \ldots, \ell_m) \) of waiting locations, \( \ell \in L_j \), not all of which are terminal locations. Let \( E_1, \ldots, E_m \) be their associated full-exit conditions. To prevent a deadlock at \( \vec{\ell} \) we require:

\[ \varphi(\vec{\ell}) \supset \square (\bigwedge_{i=1}^m atl^i \supset \bigvee_{j=1}^m E_j(y)). \]

This indicates that whenever all the processes are each at \( \ell^i, j = 1, \ldots, m \), at least one of them is enabled. The corresponding process can then proceed and deadlock is averted.

In order to eliminate the possibility of a deadlock in the full program, we must impose a similar requirement for every possible \( n \)-tuple of waiting locations, excluding \( \vec{\ell}_c = (\ell^1_c, \ldots, \ell^m_c) \).

Example:

In the consumer producer program \( CP \), the complete deadlock freedom condition will be expressed as

\[ \square \{ (atl_1 \land atm_0) \supset (ce > 0 \lor cf > 0) \} \]
\[ \land \{ (atl_1 \land atm_1) \supset (ce > 0 \lor s > 0) \} \]
\[ \land \{ (atl_2 \land atm_0) \supset (s > 0 \lor cf > 0) \} \]

46
\[ \wedge \left[ (a \ell_2 \land a t m_1 \supset (s > 0)) \right]. \]

f. generalised deadlock

We may generalize the definition of waiting locations to also include looping instructions of the form:

\[ \ell : \text{loop until } p(y) \quad \text{or} \quad \ell : \text{loop while } \sim p(y). \]

Obviously, being trapped at a tuple \((\ell^1, \ldots, \ell^m)\) some of whose locations are looping locations, with \(y = \eta\) such that \(p(\eta) = \text{false}\) for their escape conditions, is just as bad as a deadlock. Formally such a situation is not a deadlock since the execution of the self-transitions in the looping locations is not officially an idling step. But it is also self-evident that these steps cannot alter the state and the computation will remain trapped forever.

Let us therefore call a generalised deadlock situation to be a state \(s = (\ell^1, \ldots, \ell^m; \eta)\) such that each \(\ell^i\) is either a waiting location or a looping location, and such that \(E_\ell(\eta) = \text{false}\) for each \(i = 1, \ldots, m\). The escape condition \(E_\ell(\eta)\) corresponding to location \(\ell\) is taken as the exit condition \(E_\ell(\eta)\) if \(\ell\) is a semaphore location, \text{false} if \(\ell\) is a terminal location \(\ell^*_e\), and the condition for getting out of the self-loop if \(\ell\) is a looping instruction of the form

\[ \ell : \text{loop until } E_\ell(\eta) \quad \text{or} \quad \ell^* : \text{loop while } \sim E_\ell(\eta). \]

Then again the statement ensuring prevention of generalised deadlock at a tuple \(\ell = (\ell^1, \ldots, \ell^m)\) is the requirement

\[ \square \left( \bigwedge_{j=1}^m \text{at } \ell^j \supset \bigvee_{j=1}^m E_\ell(\eta) \right). \]

Example:

Consider the binomial coefficient program \(BC\). A statement of the impossibility of general deadlock at the potentially dangerous locations is given by:

\[ \square (0 \leq k \leq n) \supset \square \left\{ \begin{array}{l} [a t \ell_1 \land a t m_3] \supset (y_4 > 0) \\
\wedge [a t \ell_1 \land a t m_3] \supset (y_4 > 0) \\
\wedge [a t \ell_2 \land a t m_2] \supset (y_1 + y_2 \leq n) \\
\wedge [a t \ell_2 \land a t m_3] \supset (y_4 > 0) \\
\wedge [a t \ell_3 \land a t m_2] \supset (y_4 > 0 \lor y_1 + y_2 \leq n) \end{array} \right\}. \]

This statement ensures that if execution is at \((\ell_1, m_3)\) then \(y_4 > 0\) and one of the processes is able to proceed; if one of the processes is ever at its terminal location the other process is not deadlocked at its request instruction or trapped at its loop instruction; and if the execution is ever at \((\ell_1, m_2)\) then either \(y_4 > 0\) or \(y_1 + y_2 \leq n\), thus either enabling \(P_1\) or permitting \(P_2\) to exit from its self-loop. 

47
EVENTUALITY (LIVENESS) PROPERTIES

A second category of properties are those expressible by formulas of the form:

\[ w_1 \supset \Diamond w_2. \]

This formula states that for every proper computation, if \( w_1 \) is initially true then \( w_2 \) must eventually be realized. In comparison with invariance properties that only describe the preservation of a desired property from one step to the next, an eventuality property guarantees that some event will finally be accomplished. It is therefore more appropriate for the statement of goals which may need many steps to be realized.

Note that because of the suffix closure of the set of proper computations this formula is equivalent to:

\[ \Box (w_1 \supset \Diamond w_2) \]

which states that whenever \( w_1 \) arises during the computation it will eventually be followed by the realization of \( w_2 \).

A property expressible by such a formula is called an eventuality (liveness) property ([OL]). Following are some samples of eventuality properties.

A. Total Correctness

This property, like partial correctness, is meaningful only for programs with terminal locations, i.e., programs that are expected to terminate in contrast to continuous (cyclic) programs.

A program is said to be totally correct with respect to a specification \((\varphi, \psi)\), if for all input values \( \bar{x} \) satisfying \( \varphi(\bar{x}) \), termination is guaranteed, and the output values \( \bar{y} \) upon termination satisfy \( \psi(\bar{x}, \bar{y}) \). Once more, let \( \ell_e \) denote the exit points of the program. Total correctness w.r.t. \((\varphi, \psi)\) is expressible by:

\[ \varphi(\bar{x}) \supset \Diamond (at\ell_e \land \psi(\bar{x}, \bar{y})). \]

This says that if we have an admissible execution sequence beginning in a state which is at locations \( \ell_0 \) and has values \( \bar{y} = f_0(\bar{x}) \) where \( \varphi(\bar{x}) \) is true, then later in that execution sequence we are guaranteed to have a state which is at \( \ell_e \) and satisfies \( \psi(\bar{x}, \bar{y}) \).

Example:

The statement of total correctness for the factorial program \( F \) is:

\[ (x \geq 0) \supset \Diamond (at\ell_e \land y_2 = x!). \]

Example:

The expression of total correctness for the tree node counting program \( TN \) is given by:

\[ \Diamond (at\ell_e \land C = |X|). \]
Example:

The statement of total correctness for the binomial coefficient program $BC$ is given by:

$$
\begin{align*}
(0 \leq k \leq n) \Rightarrow \Diamond [at\ell_e \land atm_e \land y_3 = \binom{n}{k}] .
\end{align*}
$$

b. Intermittent Assertions

Eventuality formulas enable us to express a causality relation between any two events, not only between program initialization and termination but also between events arising during the execution. This becomes especially important when discussing continuous (cyclic) programs, i.e., programs that are not supposed to terminate but are to operate continuously. The general form of such an eventuality is:

$$
(\text{all} \land \phi) \Rightarrow \Diamond (\text{all}\ell' \land \phi')
$$

and it claims that whenever (in a proper computation) $\phi$ arises at $\ell$ we are guaranteed of eventually reaching $\ell'$ with $\phi'$ true. This is the exact formalization of the basic \textit{Intermittent-Assertion statement} ([BUR], [MW]):

"If sometime $\phi$ at $\ell$ then sometime $\phi'$ at $\ell'$.”

Example:

Consider the program $TN$ for counting the number of nodes in a tree. An important intermittent assertion that serves as a basis for the proof of its correctness is:

$$
[\text{all}_0 \land S = u \cdot s \land C = c] \Rightarrow \Diamond [\text{all}_0 \land S = s \land C = c + |u|].
$$

Here, $u$, $s$, and $c$ are used in the role of \textit{global} variables, while $S$ and $C$ are local program variables. This statement says that being at $\ell_0$ with a nonempty stack ensures a later arrival to $\ell_0$. In a subsequent arrival (not necessarily the next one), the top element of the stack will be removed and the value of $C$ will have been incremented by the number of nodes in the top element.

Example:

Consider again the program $PR$ for printing successive prime numbers. Under the invariance properties we expressed the claim that \textit{nothing but primes is printed}

$$
(1) \quad \Box (\text{all}_0 \Rightarrow \text{prime}(y_1)).
$$

Now we can state that the proper sequence of primes is produced. The property that \textit{every prime number is printed} can be expressed by

$$
(2) \quad \Box (\text{all}_0 \land y_1 = 2 \land \text{prime}(u)) \Rightarrow \Diamond (\text{all}_0 \land y_1 = u).
$$
In conjunction with the invariance property (1), this statement guarantees that all printed results are primes

\[2, 3, 5, 7, 11, 13, 17, \ldots,\]

but they do not guarantee that some primes are not printed more than once or out of sequence. For example, the sequence of integers

\[3, 2, 5, 3, 7, 11, 7, 13, 11, \ldots\]

\[\uparrow \quad \uparrow \quad \uparrow \quad \uparrow \quad \uparrow \]

satisfies the statements above.

We thus have to add an additional statement that will guarantee that the printed sequence is exactly the desired one. We have to be careful in devising a solution: Note that the statement

\[[at\ell_0 \land y_1 = u] \supset [at\ell_0 \supset y_1 > u]\]

does not resolve the problem! Why?

The property that the primes are printed in order can be expressed by

\[(3) \quad [at\ell_1 \land y_1 = u] \supset [at\ell_0 \supset y_1 > u].\]

This ensures monotonicity for any future visit to \(\ell_0\).

The following properties are of interest mainly for concurrent programs having more than one process.

c. Accessibility

Consider again a process that has a critical section \(C\). In the previous discussion we have shown how to state exclusion (or protection) for that section. A related and complementary property is accessibility. That is, if a process wishes to enter its critical section it will eventually get there and will not be indefinitely held up by the protection mechanism. Obviously a foolproof protection mechanism is worthless if it does not eventually admit the process into its critical section.

Let \(\ell_1\) be a location just before the critical section. The fact that the process is at \(\ell_1\) indicates an intention to enter the critical section. Let \(C\) be the set of locations in the critical section. The property of accessibility can then be expressed by

\[\supset at\ell_1 \supset [at\ell_0 \supset atC];\]

namely, whenever the program is at \(\ell_1\), it will eventually get into \(C\).

A correct construction of critical sections should ensure these two complementary properties: protection (exclusiveness) and accessibility.
Example:

For the consumer-producer program \( CP \), we wish to express the property that whenever the producer is at \( \ell_1 \) it will eventually get to \( \ell_3 \) and be able to deposit \( y_1 \) in the buffer. A symmetric statement expresses accessibility for the consumer: whenever the consumer is at \( m_0 \) it will eventually get to \( m_2 \). The conjunction of these two properties, expressing the accessibility property of the program, is given by:

\[
[\text{at}\ell_1 \rightarrow \Diamond \text{at}\ell_3] \land [\text{at}m_0 \rightarrow \Diamond \text{at}m_2].
\]

d. Liveness

A more general class of eventuality properties arises when we consider the notion that the computation of any particular process must eventually progress. Here we do not necessarily restrict ourselves to locations containing semaphore instructions.

Consider an arbitrary non-terminal location \( \ell \) in some process \( P_1 \), i.e., \( \ell \neq \ell_0 \) for that process. If the computation of this process is to proceed we cannot remain blocked at \( \ell \) due to a failure of the scheduler to schedule process \( P_1 \). Assuming that our program contains self-loops only for waiting purposes, such as in the loop instruction, progress in \( P_1 \) is observable by seeing \( P_1 \) moving from a state of \( \text{at}\ell \) to a state of \( \sim \text{at}\ell \). Consequently, the property of liveness for a general location \( \ell, \ell \neq \ell_0 \), can be expressed by:

\[
[\text{at}\ell \rightarrow \Diamond \sim \text{at}\ell],
\]

i.e., if we arrive at this location we will eventually move out. In fact we can simplify this formula to

\[
[\Diamond \sim \text{at}\ell],
\]

which is equivalent to

\[
[\sim \square \text{at}\ell],
\]

meaning that we cannot get blocked at the location \( \ell \).

The property of liveness is also known as absence of livelock or freedom from individual starvation. A livelock (or individual starvation) is defined as a situation in which some processes which are not in a terminal location cannot proceed even though the full program may still progress by having some other processes execute. Note that this is a stronger requirement than the absence of a (generalized) deadlock. As long as at least one of the processes can proceed the program is not deadlocked.

e. Responsiveness

A very important class of programs that are usually modeled as concurrent programs are operating systems and real-time programs such as airline reservation systems and other online
data-base systems. These programs can conveniently be considered as continuous (cyclic) programs which are to run forever. A halt in these programs usually indicates an error condition. Consequently these programs are not run for their end results but for the effects produced during their endless operation. Thus the notions of total and partial correctness are meaningless and have to be replaced by statements about the programs' continuous behavior.

A property usually expected of such programs is responsiveness.

Example:

Consider a continuous program (grantor) $G$ modelling an operating system. Assume that it serves a number of customer programs (requesters) $R_1, \ldots, R_t$ by scheduling a shared resource between them. The resource here can be a shared disk, main memory, etc. Let the customer programs communicate with the operating system concerning the resource via a set of boolean variables $\{r_i, g_i\}$, for $i = 1, \ldots, t$. Here, $r_i$ is set to true by the customer program $R_i$ to signal a request for the resource; $g_i$ is set to true by $G$ signalling to $R_i$ that it has been granted (allocated) the resource. After using the resource, the customer $R_i$ releases the resource back to the system $G$ by setting $r_i$ to false. This release is then acknowledged by the system $G$ by setting $g_i$ to false.

To summarize:

- $R_i$ signals a request $\Rightarrow r_i := \text{true}$
- $G$ allocates a resource $\Rightarrow g_i := \text{true}$
- $R_i$ releases the resource $\Rightarrow r_i := \text{false}$
- $G$ acknowledges the release $\Rightarrow g_i := \text{false}$.

The statement that the operating system fairly responds to the customer requests – responsiveness – is given by:

$$a_i : r_i \supset \Diamond g_i,$$

i.e., whenever $r_i$ becomes true, eventually $g_i$ will turn true. Note that this statement does not stipulate that $r_i$ becomes true when $G$ is at a particular location. Consequently it can express events such as interrupts or unsolicited signals which may occur at any arbitrary moment.

Similarly we have to ensure that the system acknowledges the release of the resource by turning $g_i$ to false:

$$b_i : \neg r_i \supset \Diamond \neg g_i.$$

Furthermore, the system cannot hope to operate successfully if it does not enjoy the cooperation of the customer programs. For example, the system cannot promise $R_2$ an eventual grant of the resource if $R_1$, who currently holds the resource, does not ever intend to release it. Consequently we will expect the $R_i$'s to satisfy some proper behavior requirements, namely for each $i$:

$$c_i : g_i \supset \Diamond \neg r_i.$$
This statement ensures that when the resource is granted to \( R_i \), it will eventually be released.

To these statements we will usually add some invariance statements ensuring the correct continuous behavior of \( G \). One such statement is

\[
d : \quad \square \left( \sum_{i=1}^{t} g_i \leq 1 \right)
\]

meaning that at any particular time the system grants the resource to at most one requester. This is a type of a mutual exclusion.

Denote the correct behavior statement of \( G \) by

\[
\psi = \bigwedge_{i=1}^{t} a_i \land \bigwedge_{i=1}^{t} b_i \land d
\]

and the correct behavior expected from the \( R_i \)'s by

\[
\varphi = \bigwedge_{i=1}^{t} c_i
\]

The problem of proving the correct behavior of \( G \) can be approached in two different ways:

- Consider a concurrent program \( P \) that consists of \( G \) alone. The \( r_i \)'s and \( g_i \)'s are then considered as input/output variables, where the \( r_i \)'s are supposed to be set by the external agents \( R_1, \ldots, R_t \).

  For this program we would prove:

  \[
  \text{**} \quad \square \varphi \supset \square \psi.
  \]

  That is, provided the external communication \( \varphi \) continuously behaves properly we can promise the correct behavior \( \psi \) of \( G \).

- As another alternative consider the concurrent program \( P \) that consists of \( G \) running together with \( R_1, \ldots, R_t \), i.e.

  \[
  P = (r, g) := (false, \ldots, false); \quad [G||R_1|| \ldots ||R_t].
  \]

  For each \( R_i \) here we substitute a simplified model that guarantees to maintain \( \square c_i \). Such a model can be represented as:

  \[
  \begin{align*}
  \ell_0 : & \text{ execute} \\
  \ell_1 : & \quad r_i := true \\
  \ell_2 : & \text{ wait until } g_i \\
  \ell_3 : & \text{ compute } \{ \text{use resource} \} \\
  \ell_4 : & \quad r_i := false \\
  \ell_5 : & \text{ wait until } \neg g_i
  \end{align*}
  \]
\( \ell_0 \): \text{ go to } \ell_0 \\

— Customer Program \( R_i \) —

If we believe that our model for \( R_i \) faithfully represents the real \( R_i \) as far as communication with \( G \) is concerned, we can proceed to prove

\[ \Box (\varphi \land \psi) \]

to ensure the correct behavior of \( P \).

Thus the two modelling alternatives available to us are the following: either considering \( G \) alone communicating with the external world via the \( r, g \) variables, or considering a combined system of \( G \) together with \( R_1, \ldots, R_t \). In the first case the proper behavior of the external world has to be promised through a continuous maintainance of \( \varphi \). In the second case the proper behavior of the \( R_i \)'s is proven at the same time as the proper behavior of \( G \).

The same analysis can of course be conducted for other situations where a program communicates with external devices and is expected to respond properly to incoming signals.

The application of the temporal formalism to the problems of responsiveness points out its power. Invariances and total correctness are long-known properties and many special formal systems and methodologies have been proposed and successfully implemented for their analysis and proofs. The temporal logic contribution to this problem is a uniform treatment and an explicit direct expressibility. In contrast, the discussion of responsiveness is relatively recent; no prior formalism addressed itself to the description and proof of these properties.

**PRECEDENCE (UNTIL) PROPERTIES**

The third class of properties to be considered are those properties which are expressible using the \( \text{until} \) operator.

In their simplest form they will be expressed by statements of the type:

\[ w_1 \text{ \text{until} } w_2. \]

This statement says that in all proper computations of \( P \) there will be a future instance in which \( w_2 \) holds and such that \( w_1 \) will hold until that instance. Recall that the formal meaning of the until operator was given by

\[ w_1 \text{ until} w_2 |^\sigma = \text{true } \iff \text{for some } k \geq 0, \begin{cases} w_2 |^\sigma(k) = \text{true and} \\
\text{for all } i, 0 \leq i < k, \ w_1 |^\sigma(i) = \text{true.} \end{cases} \]

Note that we require \( i < k \) and not \( i \leq k \). Thus, the formula \( w_1 \text{ until} w_2 \) expresses the exclusive form of the until operator since \( w_1 \) is required to hold until the instant that \( w_2 \) becomes true but not including that instant. The corresponding inclusive until property that requires \( w_1 \) to be true up to and including the instant in which \( w_2 \) becomes true can be expressed by the formula

\[ w_1 \text{ until} (w_1 \land w_2). \]
The until operator is also very useful in expressing precedence relations between events. We define the derived precede operator $P$ by:

$$w_1 P w_2 \text{ is } \sim((\neg w_1) \cup w_2).$$

This makes $P$ the dual of $\cup$ in a similar way to $\Box$ being the dual of $\Diamond$. The statement $w_1 P w_2$, read $w_1$ precedes $w_2$, states that if $w_2$ ever happens it will not happen until $w_1$ happens first. This is equivalent to stating that the first instance of $w_1$ (observed from the present) strictly precedes the first instance of $w_2$. The formal meaning of the precede operator can be given by

$$w_1 P w_2|_\alpha = \text{true} \text{ iff } \text{for every } k \geq 0, \begin{cases} \text{if } w_2|\alpha(k) = \text{true} \\ \text{then for some } i, 0 \leq i < k, w_1|\alpha(i) = \text{true} \end{cases}$$

Note that we have again $i < k$ and not $i \leq k$. Thus, the precedes operator $P$ is again an exclusive operator, expressing strict precedence between $w_1$ and $w_2$.

If we wish to express inclusive precedence, allowing the first instances of $w_1$ and $w_2$ to coincide, we may use

$$w_1 P (\sim w_1 \land w_2).$$

To show that this indeed expresses inclusive precedence, we may substitute $\sim w_1 \land w_2$ for $w_2$ in the definition above to obtain after some manipulation:

$$w_1 P (\sim w_1 \land w_2)|_\alpha = \text{true} \text{ if and only if } \begin{cases} \text{for every } k \geq 0, \\ \text{if } w_2|\alpha(k) = \text{true} \\ \text{then } w_1|\alpha(k) = \text{true} \\ \text{or for some } i, 0 \leq i < k, w_1|\alpha(i) = \text{true} \end{cases}$$

showing that the first instance of $w_2$ either coincides with an instance of $w_1$ or is preceded by such an instance.

While $w_1 \cup w_2$ implies that $w_2$ is bound to happen, this is not guaranteed by $w_1 P w_2$. In fact, if $w_2$ never happens then $w_1 P w_2$ holds for every $w_1$.

Several obvious properties of the precedes operator may be derived from corresponding properties of the $\cup$ operator and the definition of $P$. Among them are:

1. $\vDash w P w \equiv \Box \sim w$
2. $\vDash w_1 P w_2 \land w_2 P w_3 \supset w_1 P w_3$
3. $\vDash w_1 P w_2 \equiv \sim w_2 \land [w_1 \lor \Diamond(w_1 P w_2)]$
4. $\vDash \Box \sim w_2 \supset w_1 P w_2$
5. $\vDash w_1 P w_2 \lor w_2 P w_1 \lor \Diamond(w_1 \land w_2)$
6. $\vDash w_1 P w_2 \lor w_2 P (\sim w_2 \land w_1)$
7. $\vDash w_1 \cup w_2 \equiv \sim(\sim w_1 P w_2)$. 

55
Statement 1 says that \( w \) may precede itself if and only if it never happens, since no event can come before the first occurrence of that event.

Statement 2 indicates the transitivity of the precedence relation. It says that if \( w_1 \) precedes \( w_2 \) which precedes \( w_3 \) then \( w_1 \) precedes \( w_3 \).

Statement 3 gives an inductive characterization of the \( P \) operator. It says that \( w_1 \) precedes \( w_2 \) if and only if \( w_2 \) is presently false and either \( w_1 \) is true now or \( w_1 \) precedes \( w_2 \) when observed from the next instant.

Statement 4 says that if \( w_2 \) never happens then obviously \( w_1 \) precedes \( w_2 \), for every \( w_1 \).

Statements 5 and 6, each characterize the linearity of time. Statement 5 says that for every two events \( w_1 \) and \( w_2 \), either \( w_1 \) precedes \( w_2 \) or \( w_2 \) precedes \( w_1 \) or both occur at the same time. Statement 6 says that for every two events \( w_1 \) and \( w_2 \), either \( w_1 \) strictly precedes \( w_2 \) or \( w_2 \) weakly precedes \( w_1 \).

Statement 7 shows that the \( \mathcal{U} \) operator itself is expressible by the \( P \) operator.

We will consider formulas involving the \( P \) operator as belonging to the class of \textit{until} properties. We discuss below several subclasses of properties involving the \( \mathcal{U} \) and \( P \) operators.

\textbf{a. Safe Liveness}

We may interpret invariance properties as an assurance that nothing bad will happen, and liveness properties as a promise that something good will eventually happen. Consistent with this, we may want to ascertain that nothing bad happens until something good happens. This is exactly expressible by

\[ \equiv w_1 \mathcal{U} w_2, \]

where \( w_1 \) is a safety property that we wish to maintain (e.g., clean behaviour and global assertions), while \( w_2 \) is a liveness property that we want ultimately to achieve (e.g., termination and correctness). It is recommended that a full specification of a program should always be expressed as an until expression \( \equiv w_1 \mathcal{U} w_2 \), i.e., achieve \( w_2 \) while maintaining \( w_1 \).

In some cases the "until" notation is just a conveniently expressed combination of safety and liveness properties since:

\[ \equiv (\Box w_1 \land \Diamond w_2) \supset w_1 \mathcal{U} w_2. \]

However the more interesting case is when \( w_1 \) holds up to but not including the instant in which \( w_2 \) happens. Then it is no longer true that \( \Box w_1 \) is a program-valid statement.

The \textit{until} operator can also be used to express "first-time" properties. Recall that a formula of form

\[ \equiv \left( \text{at} \ell \land \phi \right) \supset \Diamond \left( \text{at} \ell' \land \phi' \right) \]

expresses the \textit{some-time} property: If the program is at \( \ell \) and \( \phi \) is true, then sometime (eventually) the program must reach \( \ell' \) with \( \phi' \) being true. Similarly, a formula of form

\[ \equiv \left( \text{at} \ell \land \phi \right) \supset \left( \neg \text{at} \ell' \right) \supset \left( \text{at} \ell' \land \phi' \right) \]

expresses the \textit{first-time} property: If the program is at \( \ell \) and \( \phi \) is true, then the program must first reach \( \ell' \) with \( \phi' \) being true.
expresses the first-time property: If the program is at $\ell$ and $\phi$ is true, then sometime the program must reach $\ell'$, and on the first visit, $\phi'$ will be true.

Example:

The safety and liveness properties for the binomial coefficient program $BC$ can be stated as:

$$\exists \ (0 \leq k \leq n) \supset (\{ ([atm_4 \supset (y_2 \neq 0) \land (y_3 \mod y_2 = 0)] \\
\land (n - k \leq y_1 \leq n) \land (0 \leq y_2 \leq k)) \\
\cup [at_{t_e} \land atm_e \land y_3 = (\binom{n}{k})]. \}

That is, achieve termination and correct result while maintaining clean behavior and global invariances.

b. Absence of Unsolicited Response

Let $w_1 \supset \Diamond w_2$ be a statement of responsiveness which guarantees that to every situation in which $w_1$ is true the program responds by making $w_2$ true. We often wish to complement this statement by requiring that on the other hand, $w_2$ will never happen unless preceded by $w_1$, i.e. the program does not respond unless explicitly requested. This of course is expressible as:

$$\forall w_1 P w_2,$$

meaning that there is always a $w_1$ preceding every $w_2$.

There is however a problem associated with the interpretation of the formal statement above as expressing our intuitive requirement. Assume a situation in which $w_1$ occurs at $t_1$ and $w_2$ indeed follows at $t_2$, $t_2 > t_1$, and neither $w_1$ nor $w_2$ is true between $t_1$ and $t_2$. If we try to test the statement: “$w_1$ precedes $w_2$” at any $t_3$, $t_1 < t_3 < t_2$, it will turn out to be false, since the first event following $t_3$ is $w_2$ rather than $w_1$. Thus we have to be careful to restrict our statement to only such reference points from which the precedence relation can be safely observed.

Thus a more careful description of the no-request-no-response statement is:

$$\exists (at_{t_0} \supset w_1 P w_2) \land [(w_2 \land \Diamond \neg w_2) \supset \Diamond (w_1 P w_2)].$$

This selects as good reference points from which the precedence of $w_1$ to $w_2$ may be observed either the starting point of the computation, or an instant in which $w_2$ is true and is changing to false in the next instant. In the later case $w_1 P w_2$ begins to hold only in the next instant.

In most practical cases we have additional information about the behavior of $w_1$ and $w_2$ that helps us formulate the requirements in simpler terms. Thus if we knew that once $w_1$ was raised and not yet answered by a $w_2$ it stays true until answered, the above problem would not have risen. Instead we could use the simpler

$$\exists (at_{t_0} \lor \neg w_1) \supset w_1 P w_2.$$
Example:

Let us reconsider the example of the operating system model: an allocator (grantor) $G$ that allocates a resource between customers (requesters) $R_1, \ldots, R_t$. Customer $R_i$ signals its request by setting $r_i$ to true. The allocator $G$ eventually responds by setting $g_i$ to true. The customer eventually releases the resource by setting $r_i$ to false which the allocator acknowledges by setting $g_i$ to false.

This simple communication protocol between a particular customer $R_i$ and the allocator can be specified by the following four invariants:

1. $\models (r_i \land \neg g_i) \Rightarrow r_i$.

This says that if $r_i$ is true and $g_i$ is false, meaning that $R_i$ is requesting the resource but has not yet been granted its request, $R_i$ should persist in its request by leaving $r_i$ on for the next instant. Note that we exclude instantaneous response by using the current values of $r_i$ and $g_i$ to determine the next value of $r_i$.

2. $\models (r_i \land g_i) \Rightarrow \neg \neg g_i$.

This states that if the resource has been granted to $R_i$, then the allocator is not allowed to withdraw its grant until the resource is released by $R_i$, by setting $r_i$ to false.

3. $\models (\neg r_i \land g_i) \Rightarrow \neg \neg r_i$.

This states that if the allocator has not yet acknowledged the release of the resource by $R_i$, then $R_i$ may not issue a new request.

4. $\models (\neg r_i \land \neg g_i) \Rightarrow \neg \neg g_i$.

This states that if the resource is not currently allocated to $R_i$ nor is $R_i$ requesting it, the allocator should not grant the resource to a process which is not requesting it. This is exactly our requirement of no unsolicited responses for this case.

These four demands with the additional responsiveness requirement

5. $\models r_i \Rightarrow \Diamond g_i$  
6. $\models g_i \Rightarrow \Diamond \neg r_i$  
7. $\models \neg r_i \Rightarrow \Diamond \neg g_i$

ensure the correct and proper behavior of the system.

The four statements 1-4 above characterize the behavior of the program by immediate transition rules. Since it is not always obvious what are the global consequences of such local constraints, we would prefer to specify them in a more global style. Such specifications can be given by:

(a) $\models r_i \Rightarrow [r_i \cup (g_i \land r_i)]$

(b) $\models g_i \Rightarrow [g_i \cup (\neg r_i \land g_i)]$
which replace 1–7.

Statement (a) says that if \( r_1 \) is true it will remain true until \( g_1 \) is granted. Statement (b) says that if the resource is granted it will remain granted until released. Statement (c) says that if the resource has been released it will not be requested again until the release has been acknowledged. Statement (d) says that if \( g_1 \) is not currently allocated, its next allocation must be preceded by a request.

c. Fair Responsiveness

In many situations we have the precedence of two events \( \psi_1 \) and \( \psi_2 \), i.e., \( \psi_1 \) precedes \( \psi_2 \) only when two earlier events \( \phi_1 \) and \( \phi_2 \) occurred in the same order, i.e. \( \phi_1 \) precedes \( \phi_2 \). We will refer to such situations as conditional precedence. It is expressible by the statement:

\[ (\phi_1 P \phi_2) \supset (\psi_1 P \psi_2). \]

This says that if \( \phi_1 \) (strictly) precedes \( \phi_2 \) then \( \psi_1 \) will (strictly) precede \( \psi_2 \).

Coupled with the implications

\[ \phi_1 \supset \diamond \psi_1 \quad \text{and} \quad \phi_2 \supset \diamond \psi_2 \]

which ensure responsiveness, the conditional precedence sharpens our commitment to fair responsiveness. That is, if we interpret \( \phi_1 \supset \diamond \psi_1 \) and \( \phi_2 \supset \diamond \psi_2 \) as describing a response \( \psi_1 \) to a request \( \phi_1 \), then responsiveness says that every request will eventually be honored by a response. The fair responsiveness establishes a first-come-first-serve discipline by ensuring that if \( \phi_1 \) preceded \( \phi_2 \) then the response to \( \phi_1 \), namely \( \psi_1 \), will precede the response to \( \phi_2 \), i.e. \( \psi_2 \).

Example:

Let us consider again the problem of the granter (allocator) and his serviced customers (requesters). We may impose a fairness requirement on his responsiveness obligations by insisting on a first-come-first-serve policy. This would be expressed by:

\[ (r_1 P r_j) \supset (g_1 P g_j). \]

This means that if customer \( R_i \) placed his request before customer \( R_j \) he will be serviced prior to customer \( R_j \). However, we again must be careful to state this only in "quiescent" reference points. For example, if \( g_j \) is currently true, while both \( r_i = r_j = false \), a situation which may occur just at the end of a granting period to \( R_j \), we certainly cannot promise that \( g_j \) will precede \( g_j \).

A reasonable set of reference points is such instants in which \( g_j \) is currently false. Thus the conditional precedence statement restricted to these observation points is:

\[ (\neg g_j) \supset [(r_1 P r_j) \supset (g_1 P g_j)]] \]

59
Example:

Consider a pair of processes where the critical sections $C_1 = \{\ell_2, \ell_3\}$ and $C_2 = \{m_2, m_3\}$ are mutually protected by semaphores:

\[
y := 1
\]

\[
\begin{array}{ll}
\ell_0 & : \text{execute} \\
\ell_1 & : \text{request}(y) \\
\ell_2 & : \text{compute} \\
\ell_3 & : \text{release}(y) \\
\ell_4 & : \text{go to } \ell_0
\end{array}
\quad
\begin{array}{ll}
m_0 & : \text{execute} \\
m_1 & : \text{request}(y) \\
m_2 & : \text{compute} \\
m_3 & : \text{release}(y) \\
m_4 & : \text{go to } m_0
\end{array}
\]

\[P_1\]

\[P_2\]

We discussed previously the statement of accessibility for such a program; namely, that if $P_1$ is waiting at $\ell_1$ it will be eventually admitted into $C_1$. This ensures only the absence of infinite overtaking, i.e., the possibility of $P_1$ waiting at $\ell_1$ forever while $P_2$ enters its own critical section infinitely often. Yet, can we prevent overtaking altogether; i.e., can we prevent $P_2$ from overtaking $P_1$ and entering $C_2$ even though $P_1$ reached $\ell_1$ before $P_2$ reached $m_1$?

We may impose fair responsiveness on this situation by requiring that the first process to reach its request instruction will be the first to be admitted into its critical section. We may attempt to state this property by:

\[
\equiv [(\ell_1 \triangleright atm_1) \triangleright (atC_1 \triangleright atC_2)] \land [(atm_1 \triangleright at\ell_1) \triangleright (atC_2 \triangleright atC_1)].
\]

This states that if $P_1$ gets to $\ell_1$ before $P_2$ gets to $m_1$ then $P_1$ will gain access to $C_1$ before $P_2$ gets to $C_2$, and similarly for the dual case in which $P_2$ gets to $m_1$ before $P_1$ gets to $\ell_1$.

However we again face the question of appropriate reference points. The statement would certainly not be true if $P_2$ is currently at $C_2$. In the above example we may be aided by the location variables in order to select appropriate reference points. One correct specification of fairness of the semaphores in this case is:

\[
\equiv [(\ell_1 \land at\{m_4, m_0\}) \triangleright (at\ell_2 \triangleright atm_3)] \land [(atm_1 \land at\{\ell_4, \ell_0\}) \triangleright (atm_2 \triangleright at\ell_2)].
\]

This says that if we are at an instant in which $P_1$ is already at $\ell_1$ while $P_2$ is both out of $C_2$ and has not yet arrived at $m_1$ then $P_1$ will be admitted to its critical section first, and similarly for the dual case.

One should not be confused by the double appearance of the notion of fairness, once when discussing fair scheduling and fair execution sequences, and here when discussing fair responsiveness as a program property. The concepts are very similar, but previously we assumed fairness as a restriction on execution sequences, since we were interested only in fair execution sequences. Here we consider (and later prove) fairness as a property of the program that gives rise to those sequences. A badly designed program could fail to achieve fairness in responding even when each of the executions we examine is fair as a computation, i.e., the scheduler may be doing its best but the program failed to ensure correct (and timely) response to each request.
Consequently, when we prove that a program has the fair responsiveness property for every proper computation, we assume that the computation is scheduled fairly and prove that it responds fairly.

Acknowledgement

We thankfully acknowledge the help extended to us by Yoni Malachi, Ben Moszkowski, Richard Schwartz, Pierre Wolper, Frank Yellin, Rivi Zarhi, and the CS256 students (Spring 1981) at Stanford University in reading the earlier drafts of the manuscript. Special thanks are due to Connie Stanley and Evelyn Eldridge-Dias for TEXing the infinitely often ($\Box \Diamond$) changing versions of the manuscript.
REFERENCES


