On using temporal logic for refinement and compositional verification of concurrent systems

Abdelillah Mokkedem*, Dominique Méry

CRIN-CNRS & INRIA-Lorraine, BP239, 54506 Vandœuvre-lès-Nancy, Cedex, France

Abstract

A simple and elegant formulation of compositional proof systems for concurrent programs results from a refinement of temporal logic semantics. The refined temporal language we propose is closed under \( w \)-stuttering and, thus, provides a fully abstract semantics with respect to some chosen observation level \( w \). This avoids incorporating irrelevant detail in the temporal semantics of parallel programs. Besides compositional verification, concurrent program design and implementation of a coarser-grained program by a finer-grained one, are easily practicable in the setting of the new temporal logic.

1. Introduction

A well-known problem for the verification and the construction of concurrent programs is that specifications that would be satisfied by a given process viewed in isolation, might be invalidated by actions performed by other processes executing in parallel. Composition principles provide a way to overcome this problem [2, 3, 27]. In compositional verification, properties of a composite system are established from properties of its components, without knowledge of their interior structure. Conversely, a compositional refinement method provides a mechanism for deriving refinements of a composed system from those of its components. Traditionally, composition principles for both specification and verification of concurrent systems are considered to be difficult to establish, and previous work [5, 7, 23] has shown that this difficulty

* Corresponding author.
lies in the formulation of a compositional rule for parallel composition. In our opinion, when formulating a compositional rule for parallel composition, one must be careful at the initial stage of defining the specification language semantics. This paper explores this point and proposes a new semantics for a temporal logic formalism, which we required to be fully abstract. This semantic criterion is used to define an appropriate basis for a compositional theory of specification and verification of concurrent programs [4].

A temporal theory for specifying programs and reasoning about them has three parts [14,16]: (1) a general part that provides axioms and rules for deriving general theorems, i.e. formulas which are valid over any model – no specific interpretation of symbols is given a priori, (2) a domain part that provides axioms and rules for reasoning about specific data domains to which both the program and the specification refer, and (3) a program part that restricts the set of considered models to those that correspond to the behaviour of the specific program being verified. The classical temporal logic [15] provides a powerful tool for global specification and noncompositional verification of existing concurrent programs. However, this logic offers very poor support for modular specification and verification and, consequently, systematic design of concurrent programs is hard (if not impossible) to do in such a setting. The lack of modularity comes from the fact that the semantics of the temporal formalism has been defined in terms of global state behaviours in such a way that the temporal properties of a given component, viewed within some context, do not abstract away from the invisible state changes performed by other components. Invariance under stuttering is a useful concept which may help us to find a solution to this problem. This notion means that whenever a behaviour $\tau$ satisfies a formula $F$, any behaviour that is equivalent to $\tau$ (modulo some state changes considered irrelevant) also satisfies $F$.

The purpose of this work is to provide a complete methodology for the compositional specification, verification and development of concurrent systems. Throughout the paper the term “concurrent systems” is used to refer to open systems which may involve several concurrent processes. An open system is one that interacts with its environment – in contrast to a closed system which is completely self-contained. A programming notation (IPL) for describing concurrent modules of an open system is introduced and a computational model for the representation of module semantics is defined. The obtained semantics is compositional in the sense that the semantics of a composite system is computed from a formal relation between semantics of its sub-modules. Next the temporal logic MTL is defined and a specification language derived by establishing a closed connection between computations of IPL programs and models of MTL formulas. Our logic is state-based. A system may be specified at many levels of abstraction; highest-level properties are described in terms of stuttering-invariant temporal formulas, while implementations are programs in the

---

1 An important concept of abstraction which we define later.
intermediate programming language IPL. A highest-level specification must deal only with the expected behavior of the system, avoiding references to efficiency or architectural details of its implementation. Such details can be introduced only in the last stage of the design process when a (parallel) algorithmic solution is already available.

This paper is organized as follows. In Section 2 a programming notation for concurrent systems is defined. In Section 3 we state problems we wish to overcome within the temporal logic. In Section 4 we introduce the concept of w-stuttering. The response to the abstraction problem is presented; this refines the temporal semantics of the basic operators that cause the trouble in abstraction. We show that the resulting temporal logic is fully abstract w.r.t. w-stuttering. In Section 5 we give an axiomatization for the refined temporal logic MTL and sketch its soundness. To justify the advantages of the new logic MTL, we give in Section 6 a formulation of IPL programs semantics within MTL, and show how a compositional proof system for the verification and the derivation of IPL programs can be built in the setting of the obtained temporal theory. Moreover, we show how implementation of a concurrent system by a finer grained equivalent system is formalized in an elegant way within this theory. In Section 7, we present an example illustrating compositional construction of proofs for properties of IPL programs within the developed theory. Finally, Section 8 concludes the paper, describing future and related work.

2. A programming notation for concurrent systems

Concurrent systems are described using the language IPL which is a slight modification of the language introduced in [17]. The purpose of these modifications is to give a compositional open semantics for IPL programs which aid in the design of a compositional proof system for IPL [20]. In particular, we introduce a dual mode, namely consum, to the environment mode external defined in [17]. external represents sends executed by the environment, whereas consum represents receives executed by the environment. We also use a uniform kind of statement, the primitive one, to describe programs, and this simplifies the technique needed to analyse these programs.

2.1. Syntax

The central notions of IPL are those of module statement and individual transition. An excerpt of the syntax is given below. A module statement has the form \(M : [\text{module}; \text{interface}; \text{body}]\) where interface declares moded channels through which the module communicates with its environment and body describes data and transitions of the module.
Interface of a module

\[
\text{interface} ::= \{\text{modes dcl}\_\text{ch}\}^*
\]

\[
\text{modes} ::= \{\text{in | out | consum | external}\}^+
\]

\[
dcl\_\text{ch} ::= \{\text{channel}\}^+:\text{type [where : init]}
\]

Concurrent modules communicate by asynchronous message passing via unbounded channels. Each module must communicate with the environment (other modules) through its interface, according to the modes assigned to channels. Let \(c\) be a channel declared in \(M\), a transition in \(M\) may have receive (resp. send) reference to \(c\) only if \(c\) is declared with the mode \(\text{in}\) (resp. \(\text{out}\)). A transition in a module parallel to \(M\) may have a receive (resp. send) reference to \(c\) only if \(c\) is declared as viewed in \(M\) with the mode \(\text{consum}\) (resp. \(\text{external}\)). So, modes \(\text{in}\) and \(\text{out}\) (which we call internal modes) declare the kind of references the module may have to the channel, while \(\text{external}\) and \(\text{consum}\) (which we call environment modes) declare the kind of references the environment may have to the channel. The module's \(\text{external}\) definition corresponds to environment's \(\text{out}\), and module's \(\text{consum}\) definition corresponds to environment's \(\text{in}\). This close correspondence permits us to define a fully compositional semantics for IPL.

Body of a module

\[
\text{body} ::= [\text{var dcl}\_\text{v;}] \text{init}_c; \text{statement}
\]

\[
\text{init}_c ::= \text{init}_c \text{variable} = \text{label}
\]

\[
dcl\_\text{v} ::= \{\text{variable}\}^+:\text{type [local [where : init]]}
\]

\[
\text{statement} ::= \{\text{transition}\}^*
\]

\[
\text{transition} ::= (\text{label}, \text{guard} \rightarrow \text{action}, \text{label})
\]

\[
\text{action} ::= \text{skip} | \text{assignment} | \text{random} | \text{send} | \text{receive}
\]

\[
\text{guard} ::= \text{expression}
\]

\[
\text{send} ::= \text{channel}! \text{expression}
\]

\[
\text{receive} ::= \text{channel}? \text{variable}
\]

A concurrent system \(Net\) has the following syntax:

\[
Net ::= M | Net || Net | v.c.Net | Net[d/c]
\]

where \(c\) and \(d\) are channels and \(v\) and \([.]/.\) denote, respectively, channel hiding and channel renaming.

Each module operates on a finite set of unshared variables. Modules communicate explicitly through channels. Certain variables may be \(\text{local}\), which, from an operational point of view, means that the observational behaviour of the module should be abstracted from them. Local variables are used in the execution of the module (like the
other variables), but the observable behaviour does not depend on the values they
take during the execution. Similarly, channels can be hidden using the binder v.

\( \text{Net}[d/c] \), where \( c \) and \( d \) are either channels or variables, is obtained from \( \text{Net} \) by renaming \( c \) into \( d \).

**Interface compatibility**

Modules can operate in parallel provided they have compatible interfaces. We first
define *interface compatibility* and we complete this definition with a *compositional*
definition of interfaces of networks. For instance, the interface of \( M_1 \parallel M_2 \) is obtained from the interfaces of \( M_1 \) and \( M_2 \).

**Notation 2.1.** Let \( M \) be a module.
- \( \text{interface}(M) \) denotes the interface of \( M \),
- \( \text{View}(M) \) denotes the of viewed channels and variables of a module \( M \); it contains nonlocal variables and nonhidden channels declared in \( M \),
- \( \text{chan}(\text{inter}) \) denotes the set of channels declared in the interface \( \text{inter} \).

**Definition 1.** Let \( M_1 \) and \( M_2 \) be two modules. \( M_1 \) and \( M_2 \) are interface compatible \( (M_1 \text{ compat-with } M_2) \) if the declaration for any channel \( c \in \text{View}(M_1) \cap \text{View}(M_2) \) satisfies the following requirements: the types of \( c \) in both declarations match, the conjunction of the where clauses (supposed true when not specified) is consistent, and if one of the declarations specifies an out (resp. in) mode, the other specifies an external (resp. consum) mode.

**Notation 2.2.** Let \( M \) be a module, \( i\_\text{mode}_M(c) \) and \( e\_\text{mode}_M(c) \) denote, respectively, the set of internal modes and the set of environment modes assigned to the channel \( c \) in module \( M \). \( \text{mode}_M(c) = df i\_\text{mode}_M(c) \cup e\_\text{mode}_M(c) \).

**Interface of networks**

1. **Parallel composition:** \( M = M_1 \parallel M_2 \), where \( M_1 \) and \( M_2 \) are two channel-hiding free modules (i.e. modules in which the binder \( v \) does not occur) and \( \text{inter}_1 \) and \( \text{inter}_2 \) their respective interfaces.
   - \( \text{interface}(M_1 \parallel M_2) = df \text{inter}_1 \oplus \text{inter}_2 \) with \( \text{chan}(\text{inter}_1 \oplus \text{inter}_2) = \text{chan}(\text{inter}_1) \cup \text{chan}(\text{inter}_2) \) such that
     (a) for every \( c \in \text{chan}(\text{inter}_1) \cap \text{chan}(\text{inter}_2) \)
     \[ i\_\text{mode}_M(c) = i\_\text{mode}_{M_1}(c) \cup i\_\text{mode}_{M_2}(c) \]
     \[ e\_\text{mode}_M(c) = e\_\text{mode}_{M_1}(c) \cap e\_\text{mode}_{M_2}(c) \]
     (b) for every \( c \) s.t. \( c \in \text{chan}(\text{inter}_1) \) and \( c \notin \text{chan}(\text{inter}_2) \), \( \text{mode}_M(c) = \text{mode}_{M_1}(c) \)
     (c) for every \( c \) s.t. \( c \notin \text{chan}(\text{inter}_1) \) and \( c \notin \text{chan}(\text{inter}_2) \), \( \text{mode}_M(c) = \text{mode}_{M_2}(c) \)
This definition assume that $M_1$ and $M_2$ are binder-free, equations which move $v$ over $||$ are needed to complete this definition. Indeed, the following equations hold; they are proved later, when semantics is defined:

$$M_1 || vc.M_2 \approx vc.(M_1 || M_2) \quad \text{if} \quad c \notin View(M_1)$$

$$M_1 || vc.M_2 \approx vd.(M_1 || M_2[d/c]) \quad \text{if} \quad c \in View(M_1), \text{where} \quad d \text{is a new channel variable.}$$

2. **Channel hiding:** $M = vc.M_1$, where $M_1$ is a module and $\text{inter}_1$ its interface.
   - interface$(vc.M) = df vc.\text{inter}_1$ with $\text{chan}(vc.\text{inter}) = \text{chan}(\text{inter}_1)$ such that
     (a) $\text{mode}_M(c) = \text{mode}_{\text{inter}_1}(c) \setminus e_\text{mode}_{M_1}(c)$
     (b) for every $d \in \text{chan}(\text{inter}_1) \setminus \{c\}$, $\text{mode}_M(d) = \text{mode}_{M_1}(d)$

3. **Renaming:** $M = M_1[d/c]$, where $M_1$ is a module and $\text{inter}_1$ its interface.
   - interface$(M[d/c]) = df \text{inter}_1[d/c]$

**Additional notation**

$$[\text{module}; \text{inter}_1; B_1] || [\text{module}; \text{inter}_2; B_2] \triangleq [\text{module}; \text{inter}_1 \oplus \text{inter}_2; B_1 || B_2]$$

$$vc.[\text{module}; \text{inter}; B] \triangleq [\text{module}; vc.\text{inter}; vc.B]$$

$$[\text{module}; \text{inter}; B][d/c] \triangleq [\text{module}; \text{inter}[d/c]; B[d/c]]$$

This notation will be useful when we relate IPL modules to specification modules in Section 6.2.

**Proposition 2.** The relation $\text{compat}_{-\text{with}}$ satisfies the following properties:

(i) Let $M_1$ and $M_2$ be two modules, if $M_1 \text{compat}_{-\text{with}} M_2$ then $M_2 \text{compat}_{-\text{with}} M_1$,

(ii) Let $M_1$ and $M_2$ be two compatible modules. $M \text{compat}_{-\text{with}} [M_1 || M_2]$ iff $M \text{compat}_{-\text{with}} M_1$ and $M \text{compat}_{-\text{with}} M_2$

**Proof.** Straightforward from Definition 1, and the definition of interfaces of networks. □

**Example 1.** Let us consider the following modules:

$$M_1 ::=
\begin{cases}
\text{module;}
\text{external in } c_2: \text{channel } [1..] \text{ of integer}
\text{consume out } c_1: \text{channel } [1..] \text{ of integer}
\text{var } x: \text{integer where } x=0
\text{init}_c \pi_1 = l_0
(l_0, \text{true} \rightarrow x := x + 1, l_1),
(l_1, \text{true} \rightarrow c_1! x, l_2),
(l_2, \text{true} \rightarrow c_2? x, l_0),
\end{cases}$$
M3 :: module;
  out c1 : channel [1..] of integer
  var z : integer
  init c \pi_3 = l_4
  \[ (l_4, \text{true}\rightarrow z := 0, l_4) \]

M2 :: module;
  external in c1 : channel [1..] of integer
  consum out c2 : channel [1..] of integer
  consum out d : channel [1..] of char
  var y : integer
  init c \pi_2 = m_0
  \[ (m_0, \text{true}\rightarrow c_1 ? y, m_1), \]
  \[ (m_1, \text{true}\rightarrow y := y - 1, m_2), \]
  \[ (m_2, \text{true}\rightarrow c_2 ! y, m_0), \]

M1 and M2 are interface compatible, but M3 is not compatible with either M1 nor with M2. So, the concurrent programs M1 || M2 and \( v_1 \cdot [M_1 || M_2] \) are syntactically well defined, however M1 || M3 and M2 || M3 are not.

interface(M1 || M2) = in out c1 : channel[1..] of integer
out in c2 : channel [1..] of integer
consum out d : channel [1..] of char

interface(vd \cdot [M_1 || M_2]) = in out c1 : channel[1..] of integer
out in c2 : channel [1..] of integer
out d : channel [1..] of char

2.2. Semantics

The basic computational model used to assign meanings to concurrent programs is that of fair transition system (FTS). We associate with each IPL module M a fair transition system \( S_M = (\Pi_M, \Sigma_M, \Theta_M, \Omega_M, \mathcal{F}_M) \) which consists of the following components:

\( \Pi_M \) (State variables): \( = \{ \pi_M \} \cup C_M \cup Y_M \), \( \pi_M \) is a control variable which ranges over \( L_M \), where \( L_M \) denotes the set of locations in M. \( C_M \) denotes the set of channels declared in the interface of M. \( Y_M \) denotes variables in M.
\( \Sigma_M \) (States): All the possible interpretations of variables in \( \Pi_M \) consistent with their types.

\( \mathcal{I}_M \) (Transitions): \( (= \mathcal{I}_M^I \cup \mathcal{I}_M^E) \)

1. Internal transitions: \( (= \mathcal{I}_M^I) \) are transitions \( \tau_a \) associated with individual transitions \( a \) in the body of \( M \). We characterize individual transitions by relations that express their operational semantics as shown in Table 1.

Note that in each relation \( \rho_a \) associated to a transition \( a \), we do not express what is kept unchanged by the transition. For instance, when \( a \) is \( x := e \), we omit to say that \( y' = y \) for every other variable \( y \). We assume that implicitly expressed in the relation \( \rho_a \).

2. Environment transitions: \( (= \mathcal{I}_M^E) \) We characterize environment transitions as follows:

- The idling transition \( \tau_i \) represented by the transition relation \( \rho_i : \text{true} \). It corresponds to a stuttering step in Abadi–Lamport’s terminology [1] which characterizes internal transitions executed by the environment.

- The environment receiving transition \( \tau_{bR} \) for any consum channel \( b \in C_M \), represented by the relation \( \rho_{bR} ^E : (|b| > 0) \land (b' = tl(b)) \). This corresponds to a receive executed by the environment on the channel \( b \).

- The environment sending transition \( \tau_{cS} \) for any external channel \( c \in C_M \), represented by the relation \( \rho_{cS} ^E : \exists u. (c' = c \oplus u) \). This corresponds to a send executed by the environment on the channel \( c \).

\( \Theta_M \) (Initial condition): It consists of \( \Theta_M \equiv \varphi_c \land \varphi \), where \( \varphi \) represents the where parts of the declarations of out channels and variables, and \( \varphi_c \) represents the initial locations of the control declared in the clause init \( c \). The environment controls the initial value of external channels.

\( \mathcal{J}_M \) (Just transitions): It contains transitions which cannot be enabled continually but taken only finitely many times. This consists of all the (internal) transitions associated with noncommunication actions of \( M \).

\( \mathcal{F}_M \) (Fair transitions): It contains transitions which cannot be enabled infinitely often but taken only finitely many times. This consists of all the internal transitions associated with communication actions of \( M \). Environment transitions \( \mathcal{I}_M^E \) are contained neither in \( \mathcal{J}_M \) nor in \( \mathcal{F}_M \).

---

**Table 1**

<table>
<thead>
<tr>
<th>Transitions ( \alpha )</th>
<th>Associated relations ( \rho_{\alpha} )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Noncommunication</strong></td>
<td></td>
</tr>
<tr>
<td>((l, b \rightarrow \text{skip}, m))</td>
<td>( \pi_M = l \land b \land \pi_M = m )</td>
</tr>
<tr>
<td>((l, b \rightarrow x := e, m))</td>
<td>( \pi_M = l \land b \land \pi_M = m \land x' = e )</td>
</tr>
<tr>
<td>((l, b \rightarrow x := ?, m))</td>
<td>( \pi_M = l \land b \land \pi_M = m \land \exists v. x' = v )</td>
</tr>
<tr>
<td><strong>Communication</strong></td>
<td></td>
</tr>
<tr>
<td>((l, b \rightarrow c!e, m))</td>
<td>( \pi_M = l \land b \land \pi_M = m \land c' = c \land e )</td>
</tr>
<tr>
<td>((l, b \rightarrow c?x, m))</td>
<td>( \pi_M = l \land b \land \pi_M = m \land</td>
</tr>
</tbody>
</table>
Comment 2.1. Note that an important feature of the semantics we are adopting is that it is an open semantics, i.e. the meaning of a program takes into account what happens in its environment; which is formalized by means of environment transitions $\mathcal{E}_M$. The closed semantics corresponds to the reduced case where $\mathcal{E}_M$ is removed from the set of transitions $\mathcal{F}$. The idea of introducing environment steps in the computations of a module is not new (see e.g. [25]). However, in contrast with other styles of open computations, we do not need explicit transition labelling. Indeed, by looking at which variable or channel has been modified in a transition, we can decide which component (module or environment) is responsible for that.

Example 2. Let us consider the module $M_1$ presented in Example 1. The associated FTS is $S_{M_1} = (I_{M_1}, \Sigma_1, \mathcal{I}_1, \Theta_1, \mathcal{F}_1, \mathcal{F}_1)$ such that

$$
I_{M_1} = \{ \pi_1, x, c_1, c_2 \} \\
\Theta_1: \pi_1 = l_0 \land x = 0
$$

$$
\mathcal{I}_1 = \{ \tau_{i_0}, \tau_{i_1}, \tau_{i_2}, \tau_{i_3}, \tau_{c_1}, \tau_{c_2} \} \\
\mathcal{F}_1 = \{ \tau_{i_1}, \tau_{i_2} \}
$$

$$
\rho_{i_0}: \pi_1 = l_0 \land x' = x + 1 \land \pi'_1 = l_1
$$

$$
\rho_{i_1}: \pi_1 = l_1 \land c'_1 = c_1 \cdot x \land \pi'_1 = l_2
$$

$$
\rho_{i_2}: \pi_1 = l_2 \land |c_2| > 0 \land x' = \text{hd}(c_2) \land c'_2 = \text{tl}(c_2) \land \pi'_1 = l_3
$$

$$
\rho_{ES}: \exists u. (c'_2 = c_2 \cdot u)
$$

$$
\rho_{ER}: (|c_1| > 0) \land (c'_1 = \text{tl}(c_1))
$$

Definition 3 (Computations of $S_M$). A (possible) computation of $S_M$ is an infinite sequence of states $\sigma$: $s_0, s_1, \ldots$, such that

1. $s_0$ satisfies the initial condition $\Theta_M$;
2. For each $i \geq 0$, $s_{i+1} \in \tau(s_i)$, for some $\tau \in \mathcal{F}_M$;
3. $\sigma$ satisfies justice and fairness requirements imposed by the sets $\mathcal{I}_M$ and $\mathcal{F}_M$.

Two computations $\sigma, \tau$ are said to be stuttering equivalent (denoted $\sigma \simeq \tau$) if they are equal modulo idling steps. Recall that in such a semantic model, finite computations are represented by infinite sequences by adding an infinite number of idling steps ($\tau_1$) which take the halting state into itself.

Definition 4 (Behaviour of $S_M$). The behaviour of $S_M$ is defined to be the set of all possible computations of $S_M$ closed under stuttering and variance to local variables and hidden channels. $\sigma$ belongs to the behaviour of $S_M$ if and only if there exists $\tau$ such that $\sigma|_M (\text{View}(M)) \simeq \tau|_M (\text{View}(M))$ and $\tau$ is a (possible) computation of $S_M$. We also say, sometimes, that $\sigma$ is a behaviour of $S_M$. 
$s\restr V$ denotes the restriction (or the projection) of the state $s$ on the set of variables $V$ ($\restr$ is extended to sequences of states in the usual way).

The semantics of a concurrent program $N_1 \parallel \cdots \parallel N_n$ is a fair transition system resulting from a composition of fair transition systems associated with modules $^2 N_i$; in notation, $S_{N_1 \parallel \cdots \parallel N_n} = S_{N_1} \otimes \cdots \otimes S_{N_n}$. Executions in $S_{N_1 \parallel \cdots \parallel N_n}$ are represented as interleaving concurrent actions in the different modules respecting fairness constraints in each component $S_{N_i}$ (together with the limited-critical-reference (LCR) restriction [17]) in order to capture a closed connection between interleaving and overlapped executions. (The LCR restriction is satisfied by all programs in the class of asynchronously communicating modules we consider.)

**Definition 5.** Let $M_1$ and $M_2$ be two modules and $c$ be a shared channel (i.e. $c \in \Pi_{M_1} \cap \Pi_{M_2}$), we define $\text{loc\_share}(c)$ to be environment modes that are not declared for $c$ in both modules, i.e. $\text{loc\_share}(c) = \{e_{\text{mode}_{M_1}}(c) \cup e_{\text{mode}_{M_2}}(c)\} \setminus (e_{\text{mode}_{M_1}}(c) \cap e_{\text{mode}_{M_2}}(c))$.

**Definition 6.** Let $S_{M_i} = (\Pi_{M_i}, \Sigma_{M_i}, \mathcal{T}_{M_i}, \Theta_{M_i}, \mathcal{I}_{M_i}, \mathcal{F}_{M_i})$, $i \in \{1, 2\}$, be the FTS associated with modules $M_1$ and $M_2$. The FTS associated with the composed module $M_1 \parallel M_2$ is defined as follows:

$S_{M_1 \parallel M_2} = \Pi_{M_1} \otimes \Pi_{M_2}$, where modes of channels of $\Pi_{M_1}$ are defined according to $\text{interface}(M_1) \otimes \text{interface}(M_2)$;

1. $\Pi_{M_1} = \Pi_{M_1} \cap \Pi_{M_2}$ where modes of channels of $\Pi_{M_1}$ are defined according to $\text{interface}(M_1)$;

2. $\Sigma_{M_1} = \{s: \Pi_{M_1} \rightarrow D_M / s[I_{M_1} \in \Sigma_{M_1} \text{ and } s[I_{M_1} \in \Sigma_{M_1}]\}$;

3. $\mathcal{T}_{M_1} = (\mathcal{T}_{M_1}^{\Pi_{M_1}} \cup \mathcal{T}_{M_2}^{\Pi_{M_1}} \setminus \tau_{\text{c}}^{\text{ER}} \setminus \tau_{\text{d}}^{\text{ES}} / c, d \in \Pi_{M_1} \cap \Pi_{M_2} \wedge \text{external}\in \text{loc\_share}(c) \wedge \text{consum}\in \text{loc\_share}(d))$,

where $\mathcal{T}_{M_1}^{\Pi_{M_1}}$ are transitions of $\mathcal{T}_{M_1}$ strengthened by $y' = y$ for every in $\Pi_{M_1} \setminus \Pi_{M_1}$ (and similarly for $\mathcal{T}_{M_2}^{\Pi_{M_1}}$);

4. $\Theta_{M_1} = \Theta_{M_1} \wedge \Theta_{M_2}$ (consistency is guaranteed by the interface compatibility requirement);

5. $\mathcal{I}_{M_1} = \mathcal{I}_{M_1} \cup \mathcal{I}_{M_2}$ and $\mathcal{F}_{M_1} = \mathcal{F}_{M_1} \cup \mathcal{F}_{M_2}$.

**Definition 7.** The semantics of hiding and renaming of channels is defined as follows:

1. $S_{\text{vc}.M} = (\Pi'_{M}, \Sigma_{M}, \mathcal{T}'_{M}, \Theta_{M}, \mathcal{I}_{M}, \mathcal{F}_{M})$ with $\mathcal{T}'_{M} = \mathcal{T}_{M} \setminus \{\tau_{\text{c}}^{\text{ER}}, \tau_{\text{c}}^{\text{ES}}\}$ and $\Pi'_{M} = \Pi_{M}$ where modes of channels of $\Pi'_{M}$ are defined according to $\text{vc}.\text{interface}(M)$.

2. $S_{M[d/c]} = S_{M}[d/c]$ (renaming is extended to tuples in the usual way).

Showing that both separated modules and composite concurrent programs semantics are formalized in terms of the same structure of fair transition systems, we shall

---

$^2$ Although components $N_i$ are arbitrary concurrent systems, they are called modules; semantically a concurrent system is considered as a new composed module.
use the term "module" for both composite concurrent programs and separated modules.

**Definition 8 (Compatible computations).** Let $M_1$ and $M_2$ be two compatible modules such that $V_1 = \text{View}(M_1)$ and $V_2 = \text{View}(M_2)$, and let $\sigma_1$ and $\sigma_2$ be two computations of $S_{M_1}$ and $S_{M_2}$, respectively. $\sigma_1$ and $\sigma_2$ are said to be compatible iff $\sigma_1 \cap V_1 \cap V_2 = \sigma_2 \cap V_1 \cap V_2$ (Fig. 1).

**Proposition 9.** Behaviour closure w.r.t. idling steps. Let $S_M$ be a FTS associated with a module $M$. Further, let $\sigma$ and $\tau$ be two sequences over $\Pi_M$ such that $\tau \simeq \sigma$. $\sigma$ is a behaviour of $S_M$ iff $\tau$ is a behaviour of $S_M$.

**Proof.** Follows from Definition 4.

**Proposition 10** (Compositionality). Let $M_1$ and $M_2$ be two compatible modules, $M = [M_1 \parallel M_2]$ and $S_M$ the FTS associated with $M$ according to the relation $S_M = S_{M_1} \otimes S_{M_2}$, and $s_0, s_1, s_2, \ldots$ be a sequence of states over $\Pi_1 \cup \Pi_2$. The two following propositions are equivalent:

1. $\sigma$ is a computation of $S_M$.
2. $\sigma \cap \Pi_1$ and $\sigma \cap \Pi_2$ are two compatible computations of $S_{M_1}$ and $S_{M_2}$, respectively.

**Proof.** It is a straightforward consequence of the use of open computations. A complete proof is given in [21].

2.3. Program equivalence and module congruence

Properties with regard to the composition operators $\parallel$ and $\vee$ are elucidated by the notions of program equivalence and module congruence. Manna and Pnueli [17] use

![Diagram showing computations and operations](image-url)
the notion of reduced behaviours w.r.t. a set of observable variables to define *program equivalence*. Briefly, two programs \( P_1 \) and \( P_2 \) are said to be *equivalent* (relative to a set of variables \( \emptyset \)), denoted by \( P_1 \sim P_2 \), iff \( \mathcal{R}_\emptyset(P_1) = \mathcal{R}_\emptyset(P_2) \), where \( \mathcal{R}_\emptyset(P_1) \) denotes the set of all reduced behaviours w.r.t. \( \emptyset \) generated by the program \( P_1 \). This definition is adequate for comparing entire programs, considered as transition systems. However, when we consider components, like modules in IPL, which are expected to be parts of large systems, we need a more stringent notion of equivalence. In short, we wish to be able to interchange two (or more) modules in any context without changing the semantics of the whole system w.r.t. some set of observable variables.

Two modules \( M_1 \) and \( M_2 \) are said to be *congruent*, denoted by \( M_1 \cong M_2 \), iff they are interchangeable in any program context \([17]\). Such a definition seems rather intuitive and we prefer to give a strong (but sufficient) formal condition: we say that two modules \( M_1 \) and \( M_2 \) are *congruent* if they have the same associated transition systems \( S_{M_1} = (\Pi_{M_1}, \Sigma_{M_1}, \mathcal{I}_{M_1}, \Theta_{M_1}, \mathcal{T}_{M_1}, \mathcal{S}_{M_1}) \) modulo a renaming of local variables and hidden channels, i.e. \( \text{View}(M_1) = \text{View}(M_1) \) and \( S_{M_1}^{\overline{c}}(\text{View}(M_1)) = S_{M_2}^{\overline{c}}(\text{View}(M_2)) \).

**Proposition 11.** (i) \( M_1 \parallel c . M_2 \approx \parallel c . (M_1 \parallel M_2) \) if \( c \notin \text{View}(M_1) \).

(ii) \( M_1 \parallel c . M_2 \approx \parallel d . (M_1 \parallel M_2[d/c]) \) if \( c \in \text{View}(M_1) \), where \( d \) is a new channel variable.

**Proof.** Follows obviously from Definitions 6 and 7. \( \square \)

Another important consequence of Definition 6 is the associative law of parallel construction.

**Proposition 12.** Let \( M_1, M_2, M_3 \) be three interface compatible modules then,

(i) \( M_1 \parallel M_2 \approx M_2 \parallel M_1 \),

(ii) \( (M_1 \parallel M_2) \parallel M_3 \approx M_1 \parallel (M_2 \parallel M_3) \).

**Proof.** Because \( S_{M_1} \otimes S_{M_2} = S_{M_2} \otimes S_{M_1} \) and \( (S_{M_1} \otimes S_{M_2}) \otimes S_{M_3} = S_{M_1} \otimes (S_{M_2} \otimes S_{M_3}) \) (straightforward from Definition 6). \( \square \)

Throughout this section we have defined a modular programming notation, namely IPL, for concurrent systems and elaborated a computational model that compositionally models semantics of concurrent systems described in IPL. In the next section, we describe a logical framework which adequately permits the expression of some desired properties of concurrent systems and their verification in a compositional way. In the same framework it can be decided whether an IPL program implements another (refinement) and, more generally, IPL implementations for concurrent systems can be derived from their abstract specifications in a systematic way.
3. What’s the problem with TL?

The linear discrete temporal logic TL has been perceived to be an appropriate tool for both the semantic description of concurrent (and sequential) programs and the reasoning about them [15]. It relies on the fact that concurrent program behaviour can be easily modelled by all possible totally ordered execution sequences arising from interleavings of actions in the separate “sequential” processes of the concurrent program (interleaving semantics). However, serious problems arise when one wants to apply TL to parallel programs of realistic size. Proofs are not compositional and consequently are very hard to master. Moreover, one cannot develop a program together with its correctness proof. In a constructive fashion, we aim to be able to (1) decompose a proof of a large program into lemmas associated to its components (lemmas that remain valid for any context where these components are used), and (2) ignore details of the reasoning and, if required, to take them into account later without losing proved properties. The logic TL does not provide an appropriate tool to support these notions and has been strongly criticized from this point of view [8, 9].

In [8, 9] Lamport objects to the use of the next operator as the cause of trouble in abstraction, forcing too much irrelevant detail to be present in the semantic description. It turns out that the lowest level of atomicity must be visible, which should not occur in a properly abstract semantics. This remark also holds for quantification over flexible variables [13]. The semantics of these operators does not make abstraction to stuttering (i.e. invisible steps) [13, 17].

On the other hand, Manna and Pnueli [13] state some points of dissatisfaction with the temporal logic presented in [12], due to the floating interpretation which does not assign any special significance to the initial state so that satisfiability and validity are evaluated at all positions in models. In fact, they presented an anchored temporal logic [13] in which they consider that a formula \( \varphi \) is defined to be valid (resp. satisfiable) over a set of sequence \( \mathcal{C} \), if it holds at position 0 of every (resp. some) sequence of \( \mathcal{C} \).

Example 3. Let us consider the following programs:

\[ P_1: \text{var } x: \text{integer}; \quad P'_1: \text{var } x: \text{integer}; \quad P_2: \text{var } y: \text{integer}; \]
\[
\begin{align*}
  x &:= x + 1 \\
  t &:= 1; x := x + t \\
  t &:= \text{integer local;} \\
  y &:= y - 1 \\
  \end{align*}
\]

Let \( p \rightarrow q = \text{df } (p \triangleright \triangleright q) \), where \( \triangleright \) and \( \triangleright \triangleright \) denote, respectively, the operators next and always of the temporal logic TL [17].

Remark 1 (Abstraction problem).

\[ P_1 \models (x = 0) \rightarrow (x = 1) \quad P'_1 \not\models (x = 0) \rightarrow (x = 1) \]
Although $P_1$ and $P'_1$ are (observationally) equivalent, $P_1$ satisfies a safety property which $P'_1$ does not satisfy. The lesson is that TL, especially its operator $\circ$, is too (operationally) precise - even w.r.t. invisible changes; TL lacks abstractness.

**Remark 2 (Compositionality problem).**

$$P_2 \models (y = 1) \rightarrow (y = 0) \quad P_1 \parallel P_2 \not\models (x = 0) \rightarrow (x = 1)$$

$$P_1 \parallel P_2 \not\models (y = 1) \rightarrow (y = 0)$$

Although the two programs ($P_1$ and $P_3$) do not share any variables (i.e. it is concurrency without communication), in the composition $P_1 \parallel P_2$ the behaviour of each program disturbs the safety property of each other. TL does not provide an efficient tool for compositionally reasoning about concurrent programs.

**Remark 3 (Refinement problem).** It would be desirable for $P'_1 \models \exists t. \ x = 0 \rightarrow (t = 1 \land x = t)$ to hold since the only difference between $P_1$ and $P'_1$ is that $P'_1$ is finer-grained than $P_1$. The behaviour that concerns the invisible variable $t$ should be completely hidden by the binder $\exists$, if that is so then we have success since we can easily check that $\exists t. \ x = 0 \rightarrow (t = 1 \land x = t)$ implies $x = 0 \rightarrow x = 1$.

But, unfortunately, this is not the case: according to the classical definition of $\exists$ in the logic TL, $P'_1 \not\models \exists t. \ x = 0 \rightarrow (t = 1 \land x = t)$. The lesson of this is that the logic TL does not provide an adequate mathematical tool for formalizing refinement with implication.

### 4. The logic MTL

This paper is concerned with the problems mentioned above. We propose a refined temporal logic MTL\(^4\) in which notions of abstraction, compositionality and refinement turn out to be rigorously treated. In this logic we assume an anchored version of a future-fragment with flexible quantifiers, and the semantics of the next operator and the quantifiers are refined in such a way as to be abstract w.r.t. some invisible steps [22]. The temporal semantics of programs is formulated in terms of the refined temporal logic MTL. Notice that the design decisions have been especially motivated by the need to reach a sufficient abstraction for the temporal language semantics which should enable the design of composition principles for (compositionally) reasoning about concurrent programs. Moreover, we are interested in an open semantic model in which the temporal semantics of a program $S$ describes the execution sequences of $S$ in all (possible) environments. The resulting logic does not require suffix closure of program behaviour, and guarantees invariance under

---

\(^3\)This suggests that refinement can be formalized by logical implication!

\(^4\)MTL stands for modular temporal logic.
stuttering of properties. Besides allowing semantic description of open systems, it provides a good abstraction for compositional specification and verification of concurrent systems and also offers a good support for systematic design of concurrent programs.

4.1. Syntax and semantics

We first describe the basic syntax of state formulas and models to define the syntax and semantics of MTL. State formulas (also called assertions) are formulas expressed in some fragment of the predicate logic language, they describe properties at individual states. We assume an infinite (countable) set of flexible variables \( \forall_s \) (\( x, y, z, \ldots \in \forall_s \)) and an infinite (countable) set of rigid variables \( \forall_r \) (\( u, v, n, \ldots \in \forall_r \)). A flexible variable may assume different values in different states of the model, while values of rigid variables does not depend on states. From a computer-science rather than mathematical point of view, rigid variables are intended to represent constants, while flexible variables represent program variables. We assume a set \( \text{Val} \) of values including the booleans true and false, natural numbers, strings, ... We assume that contains all the values needed for example considered. In addition to variables we also assume concrete predicates and concrete functions over their respective domains included in \( \text{Val} \). We agree to view constants as 0-ary functions and \( \text{propositions} \) as boolean variables. We also assume boolean connectives \( \neg, \land, \lor, \Rightarrow, \text{and} \) equality \( = \).

Values of both flexible variables and rigid variables range over \( \text{Val} \); however, since they do not have the same status (i.e. variables may assume different values in different states, while rigid variables may assume only a fixed value), we prefer to interpret them in two different ways.

**Definition 13.** We define a state \( s \) over \( V \) (\( V \subseteq \forall_s \)) (resp. a valuation \( \xi \) over \( V, (V_r \subseteq \forall_r) \)) to be an assignment of values to variables in \( V \) (resp. in \( V_r \)), i.e. a mapping from \( V \) to \( \text{Val} \) (resp. from \( V_r \) to \( \text{Val} \)).

We denote by \( \alpha[x] \) the value that the mapping \( \alpha \) assigns to the variable \( x \) (where \( x \) is either a flexible or a rigid variable). Let \( \alpha \) and \( \alpha' \) be two mappings over \( V \) and \( x \) be a variable in \( V \); we say that \( \alpha' \) is an \( x \)-variant of \( \alpha \) (\( \alpha' =_x \alpha \) in notation) if \( \alpha[y] = \alpha'[y] \) for every \( y \in V \\setminus \{x\} \). Let \( \alpha \) be a mapping over \( V \) and \( w \subseteq V \). \( \alpha|_w \) denotes the projection of \( \alpha \) onto the set of variables \( w \) (i.e. the partial mapping over \( V \) which takes the same values as \( \alpha \) on \( w \) and is undefined in \( V \setminus w \)). State formulas are interpreted by couples \( (\xi, s) \) in the usual way, e.g. let \( s = \langle x:0, y:3 \rangle \) and \( \xi = \langle u:1 \rangle \), \( (\xi, s)[x = u-1 \land y > u+1] = \alpha'[x] = \xi[u] = 0 \land s[y] > \xi[u] + 1 = 0 \land 3 > 2 = \text{false} \).

We denote by \( (s, s', s_1, s_2, \ldots) \Sigma \) and \( (\sigma, \sigma', \sigma_1, \sigma_2, \ldots) \Gamma \) the set of all states and the set of all infinite sequences of states, respectively. The set of valuations is denoted by

\[ 5 \text{ We call "flexible variables" simply "variables" from now on.} \]
Let \( \sigma: s_0, s_1, \ldots \) and \( \sigma': s'_0, s'_1, \ldots \) be two sequences of states over \( V \), and \( x \in V \) be a variable; we say that \( \sigma' \) is an \( x \)-variant of \( \sigma \) (we write this \( \sigma' =_x \sigma \)) if for each \( j \geq 0 \), \( s'_j =_x s_j \). Let \( \sigma \) be a sequence of states over \( V \) and \( w \) a subset of \( V \); we denote by \( \sigma \parallel w \) the sequence \( s_1 \mid w, s_1 \mid w, \ldots \). Let \( \sigma: s_0, s_1, s_2, \ldots \) be a sequence over \( V \); \( \sigma^i \) denotes the sequence \( s_i, s_{i+1}, \ldots \) and \( \sigma^< j \) denotes the sub-sequence \( s_0, s_1, s_2, \ldots, s_j \). We denote by \( \sigma_i \) the \( i \)th state \( s_i \) in \( \sigma \).

**Definition 14 (Stuttering).** Let \( \sigma: s_0, s_1, \ldots \) be a sequence over \( V \). A step \((s_{i-1}, s_i)\) in \( \sigma \) is called a **stuttering** step iff \( s_{i-1} = s_i \). We call **finite stuttering** (resp. **infinite stuttering**) a finite number of stuttering steps \( s_0, s_1, \ldots, s_i \) (resp. an infinite number of stuttering steps \( s_0, s_1, \ldots \)). We define \( \xi \sigma \) to be the sequence obtained from \( \sigma \) by removing all finite stutterings.

**Definition 15 (Stuttering removal).** Let \( \sigma: s_0, s_1, s_2, \ldots \) be a sequence;

\[
\xi \sigma = \begin{cases} 
\sigma^j & \text{if } \forall i > j, s_i = s_j \\
\sigma^j s_{j+1} & \text{if } s_j = s_{j+1} \\
\text{(s_j) } \xi \sigma^j & \text{else}
\end{cases}
\]

**Definition 16 (w-stuttering).** Let \( \sigma, \tau \) be two sequences over \( V \) and \( w \subseteq V \). \( \sigma, \tau \) are said to be **w-stuttering equivalent** (in notation \( \sigma \approx^w \tau \)) if \( \xi (\sigma \parallel w) = \xi (\tau \parallel w) \). We simply say that \( \sigma \) and \( \tau \) are stuttering equivalent for the case \( w = V \) and we write this \( \approx^V \).

**Proposition 17.** \( \forall k \geq 0. \exists j \geq k. (\xi (\sigma \parallel w))^k = \xi (\sigma^j \parallel w) \).

**Proof.** A consequence of Definition 15. \( \square \)

**Definition 18.** A **temporal model** (or a Kripke model) over \( U \cup V \) for MTL is a couple \((\xi, \sigma')\) that consists of a valuation \( \xi \) over \( U \) and an infinite sequence of states \( \sigma \) over \( V \) and a positive index \( i \) which is used as now.

The new and central concept in the definition of MTL lies in the introduction of a new kind of **next** operator, denoted \( \otimes_w \) (and its dual, denoted \( \oplus_w \)) indexed by a set of **flexible** variables \( w \). An important feature of \( \otimes_w \) is that it is **insensitive** to finite \( w \)-stuttering and **sensitive** to infinite \( w \)-stuttering (with respect to a given set of variables \( w \)), while its dual, \( \oplus_w \), is **insensitive** to both finite and infinite \( w \)-stuttering. We define the **index** of a formula to be the set of **flexible** variables that freely occur in the formula. When applying MTL to programs, the index represents observable variables. Another new concept, similar to the one introduced by Lamport in [11], is defined to consist of flexible quantification modulo stuttering steps. We then define the other temporal operators (always \( \Box \), sometimes \( \exists \), etc.) according to these new concepts in order to obtain a temporal logic that will enable semantic descriptions which are **invariant** under \( w \)-stuttering, where \( w \) represents the set of variables viewed by the component. This is one of the major results to ensure a **desired** level of
abstraction necessary for modular specification and compositional verification of concurrent systems. The syntax and semantics of MTL, along with the additional notation we use to write MTL formulas, are summarized below. Assuming the meaning of state formulas, which can easily be defined within the predicate logic (see [17]), we provide all one needs to understand MTL formulas.

We inductively define MTL formulas and their indexes. We will name MTL formulas by symbols from \{ p, q, f, g, F(w), G(w), G(w), \ldots \}; names like \( F(w) \) precise that \( w \) is the index of the formula denoted by \( F \). In the following, we denote by \( i(F) \) (resp. \( r(F) \)) the set of flexible variables (resp. rigid variables) that freely occur in \( F \), \( i(F) \) is also called the index of \( F \).

**Syntax:**

\[
\langle formula \rangle := \langle state\_formula \rangle \mid \bigoplus_w \langle formula \rangle \mid \bigotimes_w \langle formula \rangle
\]

\[
\mid \Box \langle formula \rangle \mid \neg \langle formula \rangle \mid \langle formula \rangle \lor \langle formula \rangle \mid \exists \langle flexible\_variable \rangle . \langle formula \rangle \mid \exists \langle rigid\_variable \rangle . \langle formula \rangle
\]

\[
\langle w \rangle := \{ \langle flexible\_variable \rangle \}^* \]

provided \( w \) includes all free flexible variables that occur in \( formula \).

- Let \( p \) be a state formula;
  - \( i(p) = \{ x \mid x \) is a flexible variable and \( x \) is free in \( p \} \) and
  - \( r(p) = \{ u \mid u \) is a rigid variable and \( u \) is free in \( p \} \)
  - \( i(\bigoplus_w f) = w \) and \( r(\bigoplus_w f) = r(f) \) (syntax requires that \( i(f) \) is included in \( w \))
  - \( i(\Box f) = i(f) \) and \( r(\Box f) = r(f) \)
  - \( i(\exists u . f) = i(f) \) and \( r(\exists u . f) = r(f) \setminus \{ u \} \)
  - \( i(\exists x . f) = i(f) \setminus \{ x \} \) and \( r(\exists x . f) = r(f) \)
  - \( i(\Box f) = i(f) \) and \( r(\Box f) = r(f) \)
  - \( i(f \lor g) = i(f) \cup i(g) \) and \( r(f \lor g) = r(f) \cup r(g) \)

**Semantics:**

\[(\xi, \sigma) \models p \quad \text{iff} \quad (\xi, \sigma_0)[p] \quad \text{for a state formula} \quad p\]

\[(\xi, \sigma) \models \bigoplus_w p \quad \text{iff} \quad \forall k > 0 . \sigma_k[w] = \sigma_0[w] \quad \text{or} \]

\[
\exists k > 0 . \sigma_k[w] \neq \sigma_0[w] \quad \text{and} \quad (\xi, \sigma^k) \models p \quad \text{and} \quad \forall i : 0 \leq i < k . \sigma_0[w] = \sigma_i[w]
\]

\[(\xi, \sigma) \models \bigotimes_w p \quad \text{iff} \quad \exists k > 0 . \sigma_k[w] \neq \sigma_0[w] \quad \text{and} \quad (\xi, \sigma^k) \models p \quad \text{and} \quad \forall i : 0 \leq i < k . \sigma_0[w] = \sigma_i[w]
\]

\[(\xi, \sigma) \models \Box p \quad \text{iff} \quad (\xi, \sigma^k) \models p \quad \text{for every} \quad k \geq 0\]

\[(\xi, \sigma) \models \exists_1 u . p \quad \text{iff} \quad \exists \xi' \in \Delta . \xi' = u \xi \quad \text{and} \quad (\xi', \sigma) \models p\]
\[(\xi, \sigma) \models \exists x \cdot p \quad \text{iff} \quad \exists \tau \in \Gamma \cdot \rho \ni \sigma \text{ and } \tau = \cdot x \rho \text{ and } (\xi, \tau) \models p\]

\[(\xi, \sigma) \models \neg p \quad \text{iff} \quad (\xi, \sigma) \not\models p\]

\[(\xi, \sigma) \models p \lor q \quad \text{iff} \quad (\xi, \sigma) \not\models p \text{ or } (\xi, \sigma) \models q\]

provided \((\xi, \sigma)\) is over \(E\) such that \(E\) contains all free variables of the interpreted formula.

**Notation 4.1.**

1. \(p \land q =_{df} \neg (\neg p \lor \neg q)\)
2. \(p \lor q =_{df} \neg p \lor q\)
3. \(p \leftrightarrow q =_{df} (p \rightarrow q) \land (q \rightarrow p)\)
4. \(\Diamond p =_{df} \neg \Box \neg p\)
5. \(p \rightarrow q =_{df} \Diamond (p \rightarrow q)\)
6. \(p \leftrightarrow q =_{df} (p \rightarrow q) \land (q \rightarrow p)\)
7. \(\forall \cdot u \cdot p =_{df} \neg \exists \otimes \cdot p\)
8. \(\exists \cdot x \cdot p =_{df} \neg \exists \cdot \neg p\)
9. \(\text{shuffle}(w_1, w_2) =_{df} \forall \cdot u \cdot \vec{v} \cdot (w_1 = \vec{u} \land w_2 = \vec{v} \land \otimes w_{1 \cup w_2} \text{true}) \Rightarrow \otimes w_{1 \cup w_2} (w_1 = \vec{u} \lor w_2 = \vec{v})\)

**Comment 4.1.** We insist on the main difference between \(\otimes\) and its dual \(\oplus\): \(\otimes\) includes a liveness part whereas \(\oplus\) does not. It is not difficult to show that \(\otimes p \Rightarrow \neg \Box \neg p\). In comparison with Lamport’s TLA, \(\oplus\) looks like \([\bullet]\_w\) and \(\otimes\) looks like \(<\bullet>_w\). The predicate \(\text{shuffle}\) is used in the axiom formalizing conjunction of independent transitions. Intuitively, it asserts that sequence changes leading to the next state may not involve variables from both \(w_1\) and \(w_2\).

**4.2. Abstractness**

We consider the abstraction problem, stated in Section 3, that arises when applying temporal logic to describe concurrent program behaviour. Our suggestion aims at solving this problem with the new semantics by abstracting state changes of invisible variables (i.e. variables outside the index). First, we show that the truth-value of a formula depends only on free variables (explicitness). Then, we show that the meaning of every formula \(F\) is insensitive to w-stuttering, i.e. steps keeping values of all variables in \(w\) unchanged, where \(w\) is the index of \(F\) (w-stuttering invariance).

**Proposition 19** (Explicitness). Let \(F\) be a formula, \(w = \imath(F)\) and \(\vec{u} = \rho(F)\). For every \(\xi\) over \(U\) and \(\sigma\) over \(V\), s.t. \(\vec{u} \subseteq U\) and \(w \subseteq V\), \((\xi, \sigma) \models F\) iff \((\xi \mid \vec{u}, \sigma \mid w) \models F\).

**Proof.** The proof is equivalent to showing that for every \(x \not\in \imath(F)\) and every \(u \not\in \rho(F)\), whenever \(\xi =_u \xi'\) and \(\sigma =_x \sigma'\), \((\xi, \sigma) \models F\) iff \((\xi', \sigma') \models F\).

This is proved by induction on the formula structure. We focus only on the pertinent cases. We consider arbitrary \(\xi'\) and \(\sigma''\) such that \(\xi' =_u \xi\) and \(\sigma' =_x \sigma\).

Case 0: \(F\) is a state formula \(p\),

\[(\xi, \sigma) \models p\]
Case 1: $F$ is of the form $\otimes_w P$,

$$(\xi, \sigma) \models \otimes_w P$$

iff {definition}

$$(\xi, \sigma_0)[p]$$

iff {definition, with $(\sigma'_0) = (\sigma_0)'$}

$$(\xi', \sigma') \models p$$

Case 2: $F$ is of the form $\exists y. P$, where $y$ is a flexible variable,

$$(\xi, \sigma) \models \exists y. P$$

iff {definition}

$$\exists \rho, \tau. (\rho \models \sigma \land \tau =_y \rho) \text{ and } (\xi, \tau) \models P$$

iff {x$\notin w$ (which implies $x \notin i(p)$), $u \notin r(P)$, $(\sigma^x)' =_x \sigma^x$ and $(\sigma^y)' =_x \sigma^y$, ind. hyp.}

there is some $j > 0$ s.t. $(\sigma_j)' \models w \notin (\sigma_0)' \models w$ and for every $i$, $0 \leq i < j$, $(\sigma_i)' \models w = (\sigma_i)' \models w$

iff {definition}

$$(\xi', \sigma) \models \otimes_w P$$

Case 3: $F$ is of the form $\exists 1 u. P$, where $u$ is a rigid variable,

$$(\xi, \sigma) \models \exists 1 u. P$$

iff {definition}

there exists $\xi_1 \in \Delta$ s.t. $\xi_1 =_v \xi$ and $(\xi_1, \sigma) \models P$

iff {definition, with $u \models v$ or $u \notin r(P)$}

there exists $\xi_1 \in \Delta$ s.t. $\xi_1 =_v \xi'$ and $(\xi_1, \sigma) \models P$
iff \( \{ x \notin F \) implies \( x \notin P \), ind. hyp.\}

there exists \( \xi_1 \in \Delta \) s.t. \( \xi_1 = \varepsilon \xi' \) and \( (\xi', \sigma') \models P \)

iff \{definition\}

\( (\xi', \sigma') \models \exists_1 v . P \)

\[ \square \]

**Proposition 20** (Stuttering invariance). Given a formula \( F \), for every valuation \( \xi \) and sequence \( \sigma \), \( (\xi, \sigma) \models F \) iff \( (\xi, \exists \sigma') \models F \).

**Proof.** We show by induction on the structure of the formula \( F \) that, for every \( \xi \) and \( \sigma \),

\( (\xi, \sigma) \models F \) iff \( (\xi, \exists \sigma) \models F \).

**Case 0:** \( F \) is a state formula \( p \),

\( (\xi, \sigma) \models p \)

iff \{definition\}

\( (\xi, \sigma_0)[p] \)

iff \{since \( (\exists \sigma)_0 = \sigma_0 \} \)

\( (\xi, (\exists \sigma_0)[p] \)

iff \{definition\}

\( (\xi, \exists \sigma) \models p \)

**Case 1:** \( F \) is of the form \( \otimes_{\omega} P \),

\( (\xi, \sigma) \models \otimes_{\omega} P \)

iff \{definition\}

there is some \( j > 0 \) s.t. \( \sigma_j[\omega] \neq \sigma_0[\omega] \) and for every \( i, 0 \leq i < j \), \( \sigma_0[\omega] = \sigma_i[\omega] \) and \( (\xi, \sigma_i) \models P \)

iff \{by definition of \( \exists \) and ind. hyp. \( (\xi, (\exists \sigma^j)) \models P \}\}

there is some \( 0 < k < j \) s.t. \( (\exists \sigma)^k = (\exists \sigma^j) \) and \( (\exists \sigma)[\omega] \neq (\exists \sigma_0)[\omega] \) and for every \( i, 0 \leq i < k \), \( (\exists \sigma_0)[\omega] = (\exists \sigma)[\omega] \) and \( (\xi, (\exists \sigma)[\omega]) \models P \)

iff \{definition\}

\( (\xi, \exists \sigma) \models \otimes_{\omega} P \)

**Case 2:** \( F \) is of the form \( \square P \),

\( (\xi, \sigma) \models \square P \)

iff \{definition\}

\( \forall k \geq 0. (\xi, \sigma^k) \models P \)

iff \{by ind. hyp.\}

\( \forall k \geq 0. (\xi, \exists (\sigma^k)) \models P \)

\[ \forall k \geq 0. (\zeta, \| \sigma \|)^k \models P \]

iff \{definition\}

\( (\zeta, \| \sigma \|) \models \Box P \)

Case 3: \( F \) is of the form \( \neg P \),

\( (\zeta, \sigma) \models \neg P \)

iff \{definition\}

\( (\zeta, \sigma) \not\models P \)

iff \{by ind. hyp.\}

\( (\zeta, \| \sigma \|) \not\models P \)

iff \{definition\}

\( (\zeta, \rho) \models \neg P. \)

Case 4: \( F \) is of the form \( P \lor Q \),

\( (\zeta, \sigma) \models P \lor Q \)

iff \{definition\}

\( (\zeta, \sigma) \models P \) or \( (\zeta, \sigma) \models Q \)

iff \{ind. hyp.\}

\( (\zeta, \| \sigma \|) \models P \) or \( (\zeta, \| \sigma \|) \models Q \)

iff \{definition\}

iff \( (\zeta, \| \sigma \|) \models P \lor Q \)

Case 5: \( F \) is of the form \( \exists x. P \), where \( x \) is a variable,

\( (\zeta, \sigma) \models \exists x. P \)

iff \{definition\}

\( \exists \sigma', \tau. (\sigma' \simeq \sigma \land \tau =_x \sigma') \) and \( (\zeta, \tau) \models P \)

iff \{by definition \( \| \sigma \| \simeq \sigma \)\}

\( \exists \sigma', \tau. (\sigma' \simeq \| \sigma \| \land \tau =_x \sigma') \) and \( (\zeta, \tau) \models P \)

iff \{definition\}

\( (\zeta, \| \sigma \|) \models \exists x. P \)

Case 6: \( F \) is of the form \( \exists_{1} u. P \), where \( u \) is a rigid variable,

\( (\zeta, \sigma) \models \exists_{1} u. P \)

iff \{definition\}

\( there \ exists \ \xi' \in \Delta \ st. \ \xi' =_{u} \xi \ and \ (\xi', \sigma) \models P \)

iff \{by ind. hyp.\}

\( there \ exists \ \xi' \in \Delta \ st. \ \xi' =_{u} \xi \ and \ (\xi', \| \sigma \|) \models P \)

iff \{definition\}

\( (\zeta, \| \sigma \|) \models \exists_{1} u. P \)
Proposition 21 (w-stuttering invariance). Given a formula $F$ and a valuation $\xi$, $w = i(F)$, for every pair of sequences $\sigma, \tau$ such that $\sigma \equiv^w \tau$, $(\xi, \sigma) \models F$ iff $(\xi, \tau) \models F$.

Proof. Let $\sigma, \tau$ be two sequences s.t. $\sigma \equiv^w \tau$.

$$(\xi, \sigma) \models F$$

iff {Proposition 19}

$$(\xi, \tau) \models F$$

iff {Proposition 20}

$$(\xi, h(\sigma[w])) \models F$$

iff {Propositions 19 and 20}

$$(\xi, \tau) \models F \quad \Box$$

Proposition 19 asserts that the truth-value of $F$ does not depend on variables that do not occur free in $F$. Proposition 21 asserts that the meaning of any formula $F$ is insensitive to steps preserving the value of all variables in the index of $F$. This is what allows the description of temporal semantics of concurrent programs in a modular way.

4.3. Validity and provability

As in [13, 17] two types of validity are considered. A state formula is defined to be assertionally valid, denoted by $\models_s F$, if $s[F]$ for every state $s \in \Sigma$.

A temporal formula $F$ is defined to be temporally valid, denoted by $\models_T$, if

$$(\xi, \sigma) \models F$$

for every valuation $\xi \in A$ and every sequence $\sigma \in \Gamma$.

Corresponding to these two types of validity, two possible deductive proof systems may be considered. The first proof system supports proving assertional validity of state formulas, while the second system support proving temporal validity of temporal formulas. This leads to two notions of provability. We say that a state formula $F$ is assertionally provable, denoted by $\vdash_s F$, if its assertional validity can be proved using the assertional proof system. Similarly, we say that a formula $F$ is temporally provable, denoted by $\vdash_T F$, if its temporal validity can be proved by the temporal proof system. Since we are mainly interested in temporal validity and provability, we assume an underlying assertional proof system and we give only axioms and rules dealing with temporal validity. But for the famous results of Gödel [6] the set of valid assertions (allowing quantification and interpretation into concrete structures including natural numbers) is in general nonrecursive and, consequently, any temporal proof system based on it must also be nonrecursive. To circumvent this situation, we assume that
we have a so-called oracle to decide whether some assertion of our assertion language is valid or not. The temporal proof system, that we present in Section 5, is recursive relatively to this oracle, i.e. the set of temporally valid formulas may be described by a recursive proof system where we may call upon the oracle to decide the validity of assertions. Focusing on the temporal part, we will omit the subscript $T$ and interpret the simpler $\models$ and $\vdash_T$ as $\models_T$ and $\vdash_T$, respectively.

4.4. More about quantifiers

Section 3 discussed a problem that concerns implementing a program by one which is finer-grained. This problem is referred to as the action refinement problem in [26] which precisely rises when an action in a program is decomposed into two or more actions in another one. Lamport's logic TLA [11] solves this problem by defining the semantics of quantification taking into account possible stuttering steps. We have adopted the same definition for flexible quantification in the logic MTL conserving the classical laws of quantifiers. This results from the fact that all the temporal operators are insensitive to stuttering (even the next operator).

Now let us define $\text{Free}(F)$ to be the set of (flexible and rigid) variables that occur free in the formula $F$, i.e. $\text{Free}(F) =_{df} \text{Free}(F)$.

Example.

- $\text{Free}(\oplus_{(x,y)}(y=0)) =_{df} \{x, y\}$
- $\text{Free}(\exists x . \oplus_{(x,y)}(x=0)) =_{df} \{y, u\}$

We have the following theorem.

**Theorem 22.** Let $x,u$ be, respectively, a variable and a rigid variable:

1. $\models \exists x . F \iff F$ if $x \notin \text{Free}(F)$
   $\models \exists u . F \iff F$ if $u \notin \text{Free}(F)$
2. $\models \exists x . (F \lor G) \iff \exists y . F \lor G$ if $x \notin \text{Free}(G)$
   $\models \exists u . (F \lor G) \iff \exists y . F \lor G$ if $u \notin \text{Free}(G)$

**Proof.** Let $x,u$ be a variable and a rigid variable, respectively, and $F$ be a formula and $w$ its index,

1. (i) $(\xi, \sigma) \models \exists x . F$ iff $\exists \rho, \tau . (\rho \approx \sigma \land \tau = x \rho)$ and $(\xi, \tau) \models F$
   \{because for any $\rho, \tau . s.t \rho = x \tau$ and $x \notin \text{Free}(F)$, $(\xi, \rho) \models F$ iff $(\xi, \tau) \models F$\}
   iff $\exists \rho, \tau . (\rho \approx \sigma \land (\rho, \tau) \models F$
   \{by proposition 21 and $\rho \approx \sigma$ implies $\rho \approx^w \sigma$\}
   iff $(\xi, \sigma) \models F$

(ii) $(\xi, \sigma) \models \exists u . F$ iff there exists $\xi' \in \Delta$ s.t. $\xi' =_u \xi$ and $(\xi', \sigma') \models F$
   \{since $u \notin \text{Free}(F)$\}
   iff $(\xi, \sigma) \models F$

2. Similar to 1. □
We have presented, above, the syntax and semantics of the logic MTL and have given interesting properties of this semantics. For instance, we have shown that the truth-value of any formula $F$ does not relate to any interleaving with steps changing values of variables outside its index.

5. A proof system for MTL

We give now a system of axioms and rules, namely $\emptyset$, dedicated to mechanizing theorem proving within MTL. An important notion connected to the construction of proofs is instantiation.

**Definition 23.** Let $\psi$ be a formula (scheme) and $p_1, \ldots, p_k$ some of the (propositional) sentence symbols appearing in $\psi$. A temporal replacement $\alpha: [p_1 \mapsto \varphi_1, \ldots, p_k \mapsto \varphi_k]$, specifies for $p_i$ a replacing formula $\varphi_i$.

We denote by $\psi[\alpha]$ the formula obtained from $\psi$ by replacing all occurrences of $p_i$ with $\varphi_i$, $i=1, \ldots, k$, respectively. We refer to $\psi[\alpha]$ as an instantiation of $\psi$. For example, the formula $\Box p \lor \neg(\Box p)$ is an instantiation of $p \lor \neg p$ obtained by the replacement $p \mapsto \Box p$.

Working with variables we wish to extend temporal replacement to parametrize sentence symbols, but additional restrictions are required in order to make the instantiation rule sound. The problem that can arise from an uncontrolled temporal replacement of parametrized sentence symbols is clearly stated in [17]. We give here only the restriction undertaken to overcome this problem (for more details see [17]). One restricts the temporal replacement to rigid parametrized sentence symbols and we require that all variables appearing in the replacing formulas are not captured by quantifiers in the instantiated formula $\psi$.

**Definition 24.** We define a parametrized occurrence of a sentence symbol $p(u_1, \ldots, u_k)$ to be rigid if the variables $u_1, \ldots, u_k$ are rigid. Let $\psi$ be a formula and $p(u_1, \ldots, u_m)$ be a rigid parametrized sentence symbol occurring in $\psi$, we define a general temporal replacement

$$p(u_1, \ldots, u_m) \mapsto \varphi(u_1, \ldots, u_m), \quad m \geq 0$$

to be admissible if $\varphi(u_1, \ldots, u_m)$ does not contain any variable that is quantified in $\psi$.

Note that by taking $m=0$, this definition also covers the case of unparametrized replacement $p \mapsto \varphi$. The preceding discussion considers replacing a sentence symbol $p$ with a formula $\varphi$. When dealing with quantifiers and equality we also need to replace variables by expressions. In the following, we write $p(y)$ to imply that $p(y)$ has one or more free occurrence of the variable $y$, and we use the term “$w$-next operator” to designate either $\bigoplus_w$ or $\bigotimes_w$. 

Definition 25. Let \( x, u \) be a flexible variable and a rigid variable, respectively; \( e \) be an expression; and \( V(e) = V_\text{f}(e) \cup V_\text{r}(e) \), where \( V_\text{f}(e) \) and \( V_\text{r}(e) \) are, respectively, the set of flexible variables and rigid variable of \( e \).

1. The replacement \( x \mapsto e \) is said to be compatible for \( p(x) \) if for any free occurrence of \( x \) within the scope of a \( w \)-next operator, \( V_\text{f}(e) \subseteq w \).

2. The replacement \( x \mapsto e \) is said to be admissible for \( p(x) \) if it is compatible for \( p(x) \), and none of the variables appearing in \( e \) is quantified in \( p(u) \).

3. The replacement \( u \mapsto e \) is said to be admissible for \( p(u) \) if \( V_\text{f}(e) = \emptyset \) and none of the variables appearing in \( e \) is quantified in \( p(u) \).

Definition 26 (Instantiation). Let us consider an admissible replacement \( x \mapsto e \) for \( p(x) \).

1. \( p(x) \) is a state formula: \( p(x)[x \mapsto e] \) is defined similarly to substitution in first-order logic.

2. \( (\otimes w p(x))[x \mapsto e] =_{df} \otimes w (p(x)[x \mapsto e]) \)

3. \( (\ominus p(x))[x \mapsto e] =_{df} \ominus (p(x)[x \mapsto e]) \)

4. for \( \exists, \exists!, \neg, \) and \( \vee \), instantiation is defined according the same law as in first-order logic.

Example 4. \( \otimes (x, y, z)[x = y][y \mapsto z + x] = \otimes (x, y, z)(x = z + x) \), however, \( y \mapsto z + x \) is not admissible for \( \otimes (x, y)(x = y) \).

5.1. The proof system

We write \( p(e/v) \) for the instantiated formula \( p(v)[v \mapsto e] \) where \( v \) is either a variable or a rigid variable.

**Axioms** for temporal operators:

(A1) \( \Box p \supset p \)

(A2) \( \Box w_1 \neg p \iff \neg \otimes w_1 p \)

(A3) \( \otimes w (p \supset q) \iff (\otimes w p \supset \otimes w q) \)

(A4) \( \Box (p \supset q) \Rightarrow (\Box p \supset \Box q) \)

(A5) \( \Box p \supset \Box \otimes w_1 p \)

(A6) \( (p \Rightarrow \otimes w_1 p) \supset (p \Rightarrow \Box p) \)

(A7) \( \otimes w_1 p \wedge \otimes w_1 \text{true} \iff \otimes w_1 p \)

(A8) \( \otimes w_1 p \vee \otimes w_2 q \iff \otimes (w_1 \cup w_2) (p \vee q) \)

(A9) \( \otimes w_1 p \wedge \otimes w_2 q \Rightarrow \otimes (w_1 \cup w_2) (p \vee q) \)

(A10) \( \otimes \text{true} \Rightarrow \text{false} \)

(A11) \( (\otimes w_1 \wedge w_2 \text{true} \Rightarrow \otimes w_1 \wedge \text{true}) \Rightarrow (\otimes w_1 p \wedge \otimes w_2 q \iff \otimes (w_1 \cup w_2) (p \vee q)) \)

(A12) \( \text{shuffle}(w_1, w_2) \supset (p \wedge q \Rightarrow \otimes (w_1 \cup w_1) (p \vee q)) \)
Axioms for quantifiers:

(A14) \( \neg \forall x. p(x) \iff \exists x. \neg p(x) \)

(A15) \( p(e/x) \Rightarrow \exists x. p(x) \) provided \( x \mapsto e \) is admissible for \( p(x) \)

(A16) \( \neg \forall u. p(u) \iff \exists u. \neg p(u) \)

(A17) \( p(e/u) \Rightarrow \exists u. p(u) \) provide \( u \mapsto e \) is admissible for \( p(u) \)

(A18) \( \bigoplus_w \exists x. p \Rightarrow \exists x. \bigoplus_w p \)

(A19) \( \bigoplus_w \exists u. p \Rightarrow \exists u. \bigoplus_w p \)

where \( x \) and \( u \) are, respectively, a variable and a rigid variable, and \( e \) is an expression.

Interference rules:

**GEN**: For \( \varphi \), a state formula,

\[
\models_A \varphi \quad \text{decided by an oracle} \quad \Box \varphi
\]

**INS**: \( \psi \quad \psi[x] \) where \( x \) is an admissible general temporal replacement

**MP**: \( p \supset q, \quad p \rightarrow q \)

**EXI 1**: \( p \Rightarrow q \quad \exists x. p \Rightarrow q \)

**EXI 2**: \( p \Rightarrow q \quad \exists u. p \Rightarrow q \)

Here, \( x \) and \( y \) are, respectively, a variable and a rigid variable which do not occur freely in \( q \).

**Comment.** Observe that, from a programming point of view, axioms (A9) and (A12) may be assumed as modelling concurrency without communication and the axiom (A11) as modelling concurrency with communication. In the axioms (A9) and (A12) changes concern variables not shared by the indexes \( w_1 \) and \( w_2 \); however in (A11) changes involve variables common to \( w_1 \) and \( w_2 \).

5.2. Theorems and derived rules

The axioms and rules given above are used to derive some additional theorems and rules. A theorem is a statement of the form \( \vdash F \), claiming that the formula \( F \) is provable in the presented deductive system, and hence is valid (assuming soundness,
as defined below). The proof of a theorem $\phi$ under assumptions $\Gamma$ is a finite sequence $\phi_0, \ldots, \phi_n$ so that for all $i \in \{0, \ldots, n\}$:
1. $\phi_i$ is an assertionally valid assertion, or
2. $\phi_i$ is an axiom, or $\phi_i$ is in $\Gamma$, or
3. $\phi_i$ is derived from $\{\phi_0, \ldots, \phi_{i-1}\}$ using a rule of the deductive system $\mathcal{G}$,
4. $\phi_n$ is $\phi$.

We write $\Gamma \vdash \phi$ a proof of $\phi$ from $\Gamma$, we write simply $\vdash \phi$ when $\Gamma$ is empty. We say that $\frac{p_1, \ldots, p_k}{q}$ is a derived rule of $\mathcal{G}$ if there is a proof $p_1, \ldots, p_k \vdash q$ in $\mathcal{G}$. Once a theorem, or a derived rule, is proved, we may use it in subsequent proofs to justify additional steps. We give some examples of theorems and derived rules in Appendix A.

5.3. The soundness of the $\mathcal{G}$ system

It can be shown that the axioms and rules dealing only with the propositional fragment of the temporal language are complete. That is, any valid propositional temporal formula can be proved using our proof system. This is because when we drop variables and quantifiers we obtain exactly the same propositional fragment of the regular temporal logic TL [13, 17] where the $w$-next operator $\otimes_w$ is equivalent to $\circ$. For this fragment the proof system provides the same axioms and rules presented in TL which are proved complete for the propositional fragment [13]. However, whilst the axioms and rules we provide to deal with variables and quantifiers allow derivation of a large number of valid formulas, they do not lead to a complete proof system. This is not surprising since the underlying assertional language assumes variables that range over concrete structures. This includes integers, and no complete deductive system to reason about them exists.

Theorem 27. The proof system $\mathcal{G}$ is sound, i.e. if $\vdash F$ then $\models F$ for any formula $F$.

The proof of this theorem is given in Appendix B.

6. Properties of IPL programs

In order to relate a formula to an IPL module it is supposed to specify, it is necessary that behaviours of the module can serve as models (in the logical sense) for the formula. This means that we can evaluate the formula on each of these behaviours and ascertain whether or not it holds on the behaviour. We thus should introduce a more specific validity (we call program-validity) concerning behaviours. To do this we augment the MTL logic by

1. some program specific predicates and functions, referring to the additional IPL domains and constructs needed to fully describe a state in the behaviour of a concurrent program, for instance, functions upon integers, booleans, lists like $\geq, +, \cdot, \text{hd, tl, ...}$, control-predicate at $\pi$,
2. rules that allow the derivation of theorems about IPL program behaviours.
Definition 28 (Semantics of atₜ). For every state \( s \), \( s[\text{at}_τ(l)] \) if \( s[π_M] = l \). Intuitively, \( \text{at}_τ(l) \) holds in a state \( s \) if and only if the control \( π_M \) is at the location \( l \) in \( s \).

Definition 29 (Program-validity). Let \( S = (Π, Σ, F, Θ, J, ☐) \) be the FTS associated with the IPL module \( N \) and \( V = V_ε \cup V_r \) be a vocabulary that consists of a set of variables \( V_ε \) that contains \( Π \), and a set of rigid variables \( V_r \) (\( V_ε \) and \( V_r \) are disjoint). Let \( σ \), \( s_0, s_1, s_2, \ldots \) be a sequence of states over \( V_ε \) and \( ξ \) be a valuation over \( V_r \). We say that the model \( ⟨(ξ, σ)⟩ \) corresponds to the behaviour \( ρ \): \( s'_0, s'_1, s'_2, \ldots \) of \( S \) iff \( σ[Π] = ρ[Π] \), and we refer to \( ⟨(ξ, σ)⟩ \) as a \( V \)-model of \( N \).

We denote by \( \text{open}_V[N] \) the set of all \( V \)-models of \( N \).

Let \( F \) be a formula and let us denote its vocabulary by \( V_F \), we define \( F \) to be \( N \)-valid or, equivalently, \( F \) is valid over the program \( N \) (denoted \( N \models F \)), iff \( \forall σ ∈ \text{open}_V[N] \), \( ∀ \xi ∈ Α. (ξ, σ) \models F \), where \( V = V_F \cup Π_M \).

Observe that program variables are always considered as flexible variables in specifications. A specification may contain variables that do not appear in the program, together with rigid variables. Rigid variables that appear in the specification are interpreted by a valuation that does not relate to the dynamic behaviour of the program. As already stated, they are mainly used for specification purposes, i.e. they are used to relate values at different states in sequences. Our program-validity \( \models \) is defined according to the open semantics \( \text{open}_V[N] \). We establish a relationship between this strong notion of program-validity and the classical one defined according to the close semantics (usually denoted \( \models \)). We also establish a close relationship between our program-validity \( \models \) and the modular-validity relation in the sense of Manna and Pnueli [17]. They define a property \( F \) to be modularly valid for a module \( M_1 \) if \( M_1 ∥ M_2 \models F \) for any module \( M_2 \), interface compatible with \( M_1 \). The following theorem justifies these informal notes.

Theorem 30. (1) \( M \models F \) iff \( \forall \bar{c}. M \models F \) where \( \bar{c} = \text{View}(M) \).

(2) \( M \models F \) iff \( M \parallel M' \models F \) for all \( M' \) interface compatible with \( M \).

Proof. Let \( \bar{c} = \text{View}(M) \). To prove (1) we show that

(i) \( σ ∈ \text{close}_V[\bar{c}. M] \) iff there exists \( τ ∈ \text{open}_V[\bar{c}. M] \) s.t. \( σ \models V. τ \).

Then, since \( V = V_F \cup Π_M \) and the validity of \( F \) is invariant to \( V \)-stuttering (so to \( V \)-stuttering too), we can deduce that \( \text{close}_V[\bar{c}. M] \models F \) iff \( \text{open}_V[\bar{c}. M] \models F \). Now we state (i) as follows. First, recall that \( \text{close}_V[M] \) is the set of \( V \)-models that correspond to behaviours of \( S_M \) whose set of transitions is reduced by removing all the environment transitions \( F^e_M \). \( \text{open}_V[\bar{c}. M] \) is the set of \( V \)-models that correspond to behaviours of \( S_{\bar{c}. M} \), but according to the semantics of \( \bar{c}. M \), \( S_{\bar{c}. M} \) corresponds to \( S_M \) reduced by removing all the environment transitions associated with channels in \( \bar{c} \). However, since \( \bar{c} = \text{View}(M) \) no environment transitions are kept in \( S_{\bar{c}. M} \), except \( τ_1 \), and thus \( \text{close}_V[M] \) and \( \text{open}_V[\bar{c}. M] \) differ only by \( V \)-models which differ by \( V \)-stuttering.
The proof of (2) is sketched as follows. Let $\text{comp}(M)$ be the set of all modules interface compatible with $M$ (it may be infinite), we have to show that $\text{open}_V(M) = \bigcup_{M' \in \text{comp}(M)} \text{close}_V(M \parallel M')$.

(a) $\bigcup_{M' \in \text{comp}(M)} \text{close}_V(M \parallel M') \subseteq \text{open}_V(M)$?

Let $M'$ be an arbitrary module which is interface compatible with $M$. It is not difficult to show that whenever $\sigma \in \text{close}_V(M \parallel M')$ then $\sigma \in \text{open}_V(M)$. It is satisfy by taking, for each viewed channel $c$ of $M$, the environment transitions $\mathcal{F}_c^E$ associated with $c$ to be communication transitions executed by $M'$ on this channel. This is made possible by the interface compatibility hypothesis.

(b) $\text{open}_V(M) \subseteq \bigcup_{M' \in \text{comp}(M)} \text{close}_V(M \parallel M')$?

This seems more difficult to prove. However, it is sufficient to show that for every behaviour $\sigma \in \text{open}_V(M)$ we are able to construct a module $M'$ (interface compatible to $M$) that simulates, for each viewed channel $c$ of $M$, all the environment transitions $\mathcal{F}_c^E$ that occur in $\sigma$. (i) When it is a matter of a transition $\tau_{c}^{\text{BR}}$ this is achieved by including any receive transition $c?x$ in $M'$. (ii) When it is a matter of a transition $\tau_{c}^{\text{BS}}$ the simulation is achieved by sending a nondeterministic value by the module $M'$; this corresponds to including in $M'$ a nondeterministic assignment $x := ?$ followed by the send transition $c!x$.

6.1. Invariance and eventuality properties

Two types of properties of IPL programs are of interest: Invariance (also called safety) properties and eventuality properties (a subclass of liveness). Invariance expresses a property which is true at every state of the behaviour and is expressed by a formula of the form $\Box P$, where $P$ is a state formula; informally, it states that $P$ holds throughout all the initialized behaviours. To prove invariance properties we use the usual computational induction based on an invariance principle. According to this principle one must find an invariant $I$ such that $\text{Init} \Rightarrow I$, $I \Rightarrow \bigoplus_\nu I$, $I \Rightarrow P$, where $\text{Init}$ is the predicate specifying initial states of the program. Examples of invariance are partial correctness, mutual exclusion, deadlock freedom, ... Eventuality asserts that something eventually happens. The two forms of eventuality considered are $F \iff G = dtf F \Rightarrow \Diamond G$ and $F \iff G = dtf F \Rightarrow \Diamond G$. The first operator is called “weak eventuality” (examples are termination, total correctness) and the second one “strong eventuality” (examples are absence of starvation, response to service). It is not difficult to see that, while the operator $\iff$ is transitive, the operator $\iff$ is not. Explicit induction, also called structural induction, is used to prove eventuality properties. This induction is often represented as an application of a well-founded argument on some element of the state [13].

6.2. Modular specification

Large systems contain several components (modules), and a separate specification is given for each of them, specifying their desired behaviour in the whole system. For
specifying concurrent modules in a convenient way we explore Lamport's modular specification method [10] and similar notions introduced in [17]. We emphasize, in particular, the relevance of complementing a specification module with the specification of the interface — the mechanism by which the module communicates with its environment. The interface specification of a module stipulates the constraints the environment must satisfy for a correct interaction with the module. The information that the interface contains is essential for the completeness of the specification module and is intended to eliminate the need for any communication between the user of the module and its implementor. Thereby, while the behavioural part can be a highest-level specification, the interface part will be a low-level specification.

**Specification module.** A specification module is an object of the form \( \langle \text{inter}, F_w \rangle \), where \( \text{inter} \) specifies the interaction constraints on the environment and \( F_w \) is a formula that specifies initial states, safety properties, and liveness properties of the expected behaviour of the module \( M \) when it runs within an environment interacting according to \( \text{inter} \). Since we wish to be sensitive to all changes viewed through the interface, the index set \( w \) should contain all the channels appearing in \( \text{inter} \).

**Definition 31 (Correctness formulas).** Let \( M = [\text{module}; \text{inter}; \text{body}] \) be an IPL module and \( \langle \text{inter}, F \rangle \) be a specification module. We define the correctness formula written \( \{\text{body}\}_{\pi_M} \langle \text{inter}, F \rangle \) as follows:

\[
\models \{\text{body}\}_{\pi_M} \langle \text{inter}, F \rangle \iff M \models F
\]

Intuitively, \( \{\text{body}\}_{\pi_M} \langle \text{inter}, F \rangle \) means that whenever a body of a module (i.e. its behaviour description) \( \text{body} \) operates within an environment compatible with the interface \( \text{inter} \), the module's behaviour satisfies the property \( F \). Note the indexing of the body by the variable \( \pi_M \). To any module (identified by \( M \)) we associate the specific variable \( \pi_M \), which serves to specify its control (thus taking values from locations of the module \( M \)). Names of control variables associated to different modules differ from each other.

We shall see in the next section that rules in the proof system associated to the program part focus on the algorithmic aspect. That is, the rules are essentially intended to reason about the behaviour content of both specifications and programs. This does not mean that reasoning about domains is excluded. Indeed, reasoning about data types can be formulated in our logic as proofs of invariance properties. For instance, to prove that \( x \) ranges over integers during any execution of the program \( M \), it is sufficient to derive the correctness formula \( M \langle \text{inter}, \text{Init} \wedge \Box(x \in \mathbb{N}) \rangle \).

6.3. The proof system — program part

The proof system for MTL, presented in Section 5, provides axioms and rules to derive temporal tautologies — formulas that are true regardless of the meanings of their elementary formulas. In this section, the logic MTL is augmented by a collection
of axioms and rules to deal with MTL formulas whose elementary formulas are
instantiated by assertions about domains and control of IPL programs. This exten-
sion permits the derivation, for a given IPL program \( \mathcal{M} \), of theorems that are valid
over the set of models corresponding to the behaviour of \( \mathcal{M} \). Clearly, every temporal
tautology of the basic logic MTL is a theorem for any program \( \mathcal{M} \), but there are
formulas which are valid for a given program \( \mathcal{M} \) but not valid in general. For the
search to establish a proof system that should support both compositional verification
and incremental (and modular) construction of IPL programs, composition rules are
needed where both the program part and specification part of the correctness formulas in
premises reduce in complexity w.r.t. the conclusion. According to this criterion, given a
large specification to be implemented, rules allow the implementor to decompose it
into more elementary ones that can be implemented separately. Conversely, given the
correctness proofs of some small modules, they allow the verifier to establish the
correctness of bigger modules. The first collection of axioms given below consists of
program axioms that describe the temporal semantics of basic transitions of IPL
modules. We then present the main rules mechanizing compositional reasoning about
IPL modules.

**Axioms for transitions:**

(E1) \( \{ \text{var} \ w; \text{id} \} \langle \text{consum}(c), \ a_{-\pi}(l) \land |c| > 0 \land p[\text{tl}(c)/c] \Rightarrow \bigoplus_{(\pi, w, c)} (a_{-\pi}(l) \land p) \rangle \)

(E2) \( \{ \text{var} \ w; \text{id} \} \langle \text{external}(c), \ a_{-\pi}(l) \land \forall u. p[c \bullet u/c] \Rightarrow \bigotimes_{(\pi, w, c)} (a_{-\pi}(l) \land p) \rangle \)

(E3) \( \{ \text{var} \ w; (l, b \rightarrow \text{skip}, m) \} \langle \text{false}, \ a_{-\pi}(l) \land b \land p \Rightarrow \bigotimes_{(\pi, w)} (a_{-\pi}(m) \land p) \rangle \)

(E4) \( \{ \text{var} \ w; (l, b \rightarrow x := e, m) \} \langle \text{false}, \ a_{-\pi}(l) \land b \land p[e/x] \Rightarrow \bigotimes_{(\pi, w)} (a_{-\pi}(m) \land p) \rangle \)

(E5) \( \{ \text{var} \ w; (l, b \rightarrow x := ?, m) \} \langle \text{false}, \ a_{-\pi}(l) \land b \land \forall x. p \Rightarrow \bigotimes_{(\pi, w)} (a_{-\pi}(m) \land p) \rangle \)

(E6) \( \{ \text{var} \ w; (l, b \rightarrow c! e, m) \} \langle \text{out}(c), \ a_{-\pi}(l) \land b \land p[c \bullet e/c] \Rightarrow \bigotimes_{(\pi, w, c)} (a_{-\pi}(m) \land p) \rangle \)

(E7) \( \{ \text{var} \ w; (l, b \rightarrow c? x, m) \} \langle \text{in}(c), \ a_{-\pi}(l) \land |c| > 0 \land b \land p[\text{hd}(c)/x, \text{tl}(c)/c] \Rightarrow \bigotimes_{(\pi, w, c)} (a_{-\pi}(m) \land p) \rangle \)

Globally, these transition axioms associate to each transition \( \alpha \) an MTL formula
\( F \) providing a temporal description of the behaviour of \( \alpha \) when it communicates with
the environment through an interface \( \text{inter} \). The first two axioms require more
explanation. They serve to give the individual temporal semantics of the environment
transitions corresponding to an external and a consum channel. To do so we use the
idling transition (denoted by \( \text{id} \)), whose transition relation is \( \rho_I: \text{true} \), and conse-
quently the temporal formulas that appear in the axioms (E1) and (E2) denote exactly
the semantics of the environment transitions in question. We then give a rule (rule Env
below) to combine this pure environment semantics with semantics of individual transitions that appear in the module's body. The other axioms describe temporal semantics of each individual transition \( \alpha \) when operating in a close system, which
means that no environment mode (external, consum) appears in the interface. As claimed, this closed semantics is combined with the environment semantics (using the rule \textbf{Env}) to achieve an open semantics of transitions. Note the use of $\oplus$ for environment transitions and the use of $\otimes$ for internal transitions. In operational semantics, fairness conditions concern internal transitions but not environment transitions.

Note the use of the predicate false to represent the empty interface and the disjunction $\lor$ to represent the union of interfaces. The soundness of these axioms is proved according to the IPL’s semantics defined in the Section 2.2. It is not difficult to check that for every axiom $E_i$ of the form $\{\text{dcl}; \alpha\} \times \langle \text{inter}, F \rangle, [\text{inter}; \text{dcl}; \alpha] \supseteq F$ holds. Equivalently, we have to verify that every model in open, $[M]$ satisfies $F$, where $M = \{\text{inter}; \text{dcl}; \alpha\}$ and $V = \Pi_M \cup V_r$. That reduces to verify that the formula $F$ specifies the transition relation and fairness requirements associated to $\alpha$. Consider axiom (E4) to illustrate this. Intuitively [E4] states that a proper execution of the transition $(l, b \rightarrow x := e, m)$ must always be describable as follows: whenever the control of the module is at the location $l$, and the current state satisfies the guard $b$, then there is eventually a next state (possibly preceded by a finite number of stuttering steps) obtained by executing the assignment $x := e$ (which is correctly expressed in the instantiated precondition $p[e/x]$ according to Hoare’s axiom). The eventual existence of the next state (asserted by the use of the strong next operator $\diamond$) expresses the fairness constraint requiring that the transition $(l, b \rightarrow x := e, m)$ must be eventually executed once is enabled.

Rules for modules:

\textbf{Env}:

\[
\{\text{dcl}; \text{id}\} \times \langle \text{inter}_1, F_1 \rangle
\]

\[
\{\text{dcl}; \alpha\} \times \langle \text{inter}_2, F_2 \rangle
\]

\[
\{\text{dcl}; \alpha\} \times \langle \text{inter}_1 \lor \text{inter}_2, F_1 \land F_2 \rangle
\]

To argue the soundness of this rule, let $\sigma$ be a model corresponding to an execution sequence of $[\text{inter}_1 \lor \text{inter}_2, \text{dcl}; \alpha]$, where $\text{inter}_1$ consists of environment modes. Then $\sigma$ must correspond to an execution sequence of both $[\text{inter}_1; \text{dcl}; \text{id}]$ and $[\text{inter}_2; \text{dcl}; \alpha]$. The first module collects environment transitions associated with modes of $\text{inter}_1$ and the second one collects internal transitions associated with $\alpha$.

\textbf{Body}:

\[
\{\text{dcl}_1; \alpha_1\} \times \langle \text{inter}_1, F_1 \rangle
\]

\[
\vdots
\]

\[
\{\text{dcl}_n; \alpha_n\} \times \langle \text{inter}_n, F_n \rangle
\]

\[
\{\text{dcl}_1, \ldots, \text{dcl}_n; (\alpha_1, \ldots, \alpha_n)\} \times \langle \text{inter}_1 \lor \cdots \lor \text{inter}_n, F_1 \land \cdots \land F_n \rangle
\]
By this rule, the execution of a list of transitions consists of execution sequences which satisfy transition relations and fairness requirements of both individual transitions of the list. Note that the combination of the different transitions in a module using the conjunction $\wedge$ (in rules $\text{Env}$ and $\text{Body}$) relates to the expression of individual transition semantics in an implicative form (i.e. using the entailment $\Rightarrow$), and the soundness of these rules strongly relate to this fact.

**Init\_c:**

$$
\{\text{dcl}; S\}_{\pi} \langle \text{inter}, F \rangle \\
\{\text{dcl; init\_c } \pi = l; S\}_{\pi} \langle \text{inter}, \text{at}_{-\pi}(l) \wedge F \rangle
$$

The soundness of this rule is trivial. When we give the starting location of the module control, the rule only strengthens the specification $F$ of the module by the proposition $\text{at}_{-\pi}(l)$ restricting the set of possible execution sequences to those of which the initial control-state is at $l$.

**Where:**

$$
\{\text{dcl}; S\}_{\pi} \langle \text{inter}, F \rangle \\
\{\text{dcl where } \varphi; S\}_{\pi} \langle \text{inter}, \varphi \wedge F \rangle
$$

This rule is similar to the previous one, it concerns, however, initialization of (program) variables. In the same way, it strengthens the specification $F$ of the module with the assertion $\varphi$ which must hold at the first state of any execution sequence.

**local:**

$$
\{\text{dcl; S}\}_{\pi} \langle \text{inter}, F \rangle \\
\{\text{dcl; local } x; S\}_{\pi} \langle \text{inter}, \exists x. F \rangle
$$

Let the formula $F$ specify the right behaviour for the module $[\text{inter}; \text{dcl}, S]$. It also specifies the visible value of the variable $x$ declared in $\text{dcl}$. However, in the module $[\text{inter}; \text{dcl, local } x; S]$ we wish to specify that variables, except $x$, change in exactly the same way as in the first module and we do not mind how the variable $x$ changes. We therefore require a formula (for specifying the second module) asserting that variables, except $x$, behaves as described in $F$, but that it does not matter which values that $x$ assume. Such an intuitive meaning is exactly the one given by the quantified formula $\exists x. F$.

**Localization:**

$$
\{l_i, l_j\} \subseteq L_B, \ l_i \neq l_j \vdash \{B\}_{\pi} \langle \text{inter}, \text{at}_{-\pi}(l_i) \wedge \text{at}_{-\pi}(l_j) \Rightarrow \text{false} \rangle \\
\vdash \{B\}_{\pi} \langle \text{inter}, \Box(\bigvee_{l \in L_B} \text{at}_{-\pi}(l)) \rangle
$$

provided $L_B$ denotes the set of locations in $B$. 
These two axioms concern localization of the control in a module. The first one asserts that for any module, the control cannot be at two different locations in the same time. The second one asserts that at any time the module control must be in some location.

**Rules for networks:**

**Bind:**

\[
{B}_\pi \langle \text{inter}, F_w \rangle \\
\vdash {vc.B}_\pi \langle vc \cdot \text{inter}, \exists c. F \rangle
\]

The soundness of this rule can be argued in a similar way as the previous one. Hiding a channel \(c\) is equivalent to considering it to be local to the module and thus values that \(c\) assumes become insignificant outside the module. This meaning can be captured in terms of the flexible quantifier \(\exists\).

**Rename:**

\[
{B}_\pi \langle \text{inter}, F \rangle \\
\vdash {B[d/c]}_\pi \langle \text{inter}[d/c], F^d \rangle
\]

provided \(d\) is a new channel.

where \(F^y_x\) denotes renaming of the variable \(x\) to a new variable \(y\) in the formula \(F\). Renaming is extended to MTL formulas in the usual way, e.g., \(\oplus (x,y)x = y\) = \(\oplus (x,z) x = z\).

Renaming a channel \(c\) in a module \(M\) to be a new channel \(d\) is a simple case of implementing a channel by renaming another. The channel \(d\) must assume the same behaviour as the channel \(c\). Let \(F\) specifies the behaviour of the channel \(c\) within \(M\), and renaming \(c\) to \(d\) corresponds exactly to renaming \(c\) to \(d\) in the specification \(F\), which is denoted \(F^d_c\).

**PAR:**

\[
{B}_1\pi_1 \langle \text{inter}_1, F(w_1) \rangle \\
{B}_2\pi_2 \langle \text{inter}_2, G(w_2) \rangle \\
F(w_1) \land G(w_2) \land \text{shuffle}(w'_1, w'_2) \supset \Psi \\
\vdash {B_1 \parallel B_2}_\pi \langle \text{inter}_1 \oplus \text{inter}_2, \Psi \rangle
\]

provided,

\(\text{inter}_1 \text{ compat with } \text{inter}_2\)

\(\bar{c} = \text{View}(\text{inter}_1) \cap \text{View}(\text{inter}_2)\)

\(w'_1 = w_1 \setminus \bar{c}\), \(w'_2 = w_2 \setminus \bar{c}\)

\(\pi = (\pi_1, \pi_2)\)

The expression \(\text{inter}_1 \oplus \text{inter}_2 (\neq \text{inter})\) in the **PAR** rule, is defined according to the definition given in Section 2.1 which combines two compatible interfaces \(\text{inter}_1\) and \(\text{inter}_2\) to compute the global interface of the parallel module \(M_1 \parallel M_2\).

Let \(M_1 = [\text{inter}_1; B_1]\) and \(M_2 = [\text{inter}_2; B_2]\) provided \(\text{inter}_1\) and \(\text{inter}_2\) are compatible. The general idea underlying the **PAR** rule is that any execution of the module \(M_1 \parallel M_2\) can always be viewed both as an \(M_1\) execution in which both the \(M_2\) steps and the true environment steps are regarded as environmental, and symmetrically, as
an $M_2$ execution in which both the $M_1$ steps and the true environment steps are regarded as environmental, with an additional constraint that the only variables that can be shared are shared channels. This additional constraint is expressed by the formula $\text{shuffle}(w_1, w_2)$ which asserts that steps involving $w_1'$ and steps involving $w_2'$ can only interleave, i.e. $w_1'$ and $w_2'$ cannot change together. The third requirement concerns control; it postulates that the control variable $\pi$ of the parallel program $\{B_1 \parallel B_2\}$ is implemented by the pair formed by the two control variables $\pi_1$ and $\pi_2$ of the components $B_1$ and $B_2$.

**Adaptation rules:**

**Conjunction:**

\[
M \langle \text{inter}, F_1 \rangle \quad M \langle \text{inter}, F_2 \rangle \\
\frac{}{M \langle \text{inter}, F_1 \land F_2 \rangle}
\]

**Consequence:**

\[
M \langle \text{inter}, F \rangle, \quad F \Rightarrow G \\
\frac{}{M \langle \text{inter}, G \rangle}
\]

The two last rules are very useful for proof adaptation and for incremental proofs. The first rule allows us to decompose the proof of a large specification into proofs of elementary specifications. The consequence rule allows us to adopt as a valid specification any formula that follows logically from a valid specification. Moreover, these rules are of interest for a bottom-up verification approach.

7. A small example

This section presents an example illustrating compositional construction of proofs for properties of IPL programs within the developed theory. To simplify formulas we assume, in the following, that rigid variables ($e, v_0, v_1, u, v, h, t, a$) are implicitly universally quantified in any formula where they appear, otherwise we should give their quantification explicitly, and let $w_1 = (\pi_1, x, c_1, c_2, c)$ and $w_2 = (\pi_2, y, c_1, c_2)$. Let us first name formulas needed for the proof construction. $x_1$ and $x_2$ range, respectively, over $\{l_0, l_1, l_2\}$ and $\{m_0, m_1\}$.

**init** $t_1$: $at_{-\pi_1}(l_0) \land c_1 = e \land c = e$

$F_{01}$: $at_{-\pi_1}(a) \land c_1 = e \land h \land c_2 = v \land c = t \land x = v_0$

$\Rightarrow \bigoplus_{w_1}(at_{-\pi_1}(a) \land c_1 = h \land c_2 = v \land c = t \land x = v_0)$

$F_{02}$: $at_{-\pi_1}(a) \land c_1 = u \land c_2 = v \land c = t \land x = v_0$

$\Rightarrow \bigoplus_{w_1}(at_{-\pi_1}(a) \land c_1 = u \land c_2 = v \land c = t \land x = v_0)$
The following theorems are directly derived from axioms (E1, ..., E7) of the (program-part) proof system.

1. \{\text{var } x; \text{id} \}_n, \langle \text{consum}(c_1), F_{01} \rangle
2. \{\text{var } x; \text{id} \}_n, \langle \text{external}(c_2), F_{02} \rangle
3. \{\text{var } x; (l_0, c_1 ! l_1) \}_n, \langle \text{out}(c_1), F_1 \rangle
4. \{\text{var } x; (l_1, c_2 ? x, l_2) \}_n, \langle \text{in}(c_2), F_2 \rangle
5. \{\text{var } x; (l_2, c_1 ! l_0) \}_n, \langle \text{out}(c), F_3 \rangle
6. \{\text{var } y; \text{id} \}_n, \langle \text{consum}(c_2), G_{01} \rangle
7. \{\text{var } y; \text{id} \}_n, \langle \text{external}(c_1), G_{02} \rangle
8. \{\text{var } y; (m_0, c_1 ! y, m_1) \}_n, \langle \text{in}(c_1), G_1 \rangle
9. \{\text{var } y; (m_1, c_2 ! 1, m_0) \}_n, \langle \text{out}(c_2), G_2 \rangle

We use rules \text{Env} and \text{Body} and Theorems 1–5 in order to derive a specification for a large module. Similarly, we use rules \text{Env} and \text{Body} and Theorems 6–9 in order to derive a specification for another module. These are represented in the two following theorems.

\begin{align*}
\text{Th}_1: & \{\text{var } x; (l_0, c_1 ! 1, l_1), (l_1, c_2 ? x, l_2), (l_2, c_1 ! l_0) \}_n, \\
& \langle \text{consum}(c_1) \lor \text{out}(c_1) \lor \text{external}(c_2) \lor \text{in}(c_2) \lor \text{out}(c), \\
& F_{01} \land F_{02} \land F_1 \land F_2 \land F_3 \rangle
\end{align*}

\begin{align*}
\text{Th}_2: & \{\text{var } y; (m_0, c_1 ! y, m_1), (m_1, c_2 ! 1, m_0) \}_n, \\
& \langle \text{consum}(c_2) \lor \text{out}(c_2) \lor \text{external}(c_1) \lor \text{in}(c_1), \\
& G_{01} \land G_{02} \land G_1 \land G_2 \rangle
\end{align*}
Now we use rules \texttt{Init\_c} and \texttt{Where} to specify initial values of the control and some variables. The following theorems are derived from \texttt{Th\_1} and \texttt{Th\_2}.

\begin{align*}
\text{\texttt{Th\_3}: } & \{ \text{var } x, \text{ where } c_1 = \emptyset, c = \emptyset, \text{ init\_c } \pi_1 = l_0; (l_0, c_1!1, l_1), (l_1, c_2?x, l_2), (l_2, c!1, l_0) \} \\
& \langle \text{consum}(c_1) \lor \text{out}(c_1) \lor \text{external}(c_2) \lor \text{in}(c_2) \lor \text{out}(c), \text{ init}_1 \land (F_0 \land F_3) \rangle \\
\text{\texttt{Th\_4}: } & \{ \text{var } y, \text{ where } c_2 = \emptyset, \text{ init\_c } \pi_2 = m_0; (m_0, c_1?y, m_1), (m_1, c_2!1, m_0) \} \\
& \langle \text{consum}(c_2) \lor \text{out}(c_2) \lor \text{external}(c_1) \lor \text{in}(c_1), \text{ init}_2 \land (G_0 \land G_3) \rangle
\end{align*}

At this level, we have derived a temporal specification for each of the following modules:

\begin{align*}
M_1 & : \begin{cases}
\text{module;}
\text{external in } c_2: \text{channel}[1..] \text{ of integer}
\text{consum out } c_1, c: \text{channel}[1..]
\text{of integer where } c_1 = \emptyset, c = \emptyset
\text{var } u: \text{integer}
\end{cases}
\end{align*}

\begin{align*}
[(l_0, c_1!1, l_1), (l_1, c_2?u, l_2), (l_2, c!1, l_0)]
\end{align*}

\begin{align*}
M_2 & : \begin{cases}
\text{module;}
\text{external in } c_1: \text{channel}[1..] \text{ of integer}
\text{consum out } c_2: \text{channel}[1..]
\text{of integer where } c_2 = \emptyset
\text{var } v: \text{integer where } v = 0
\end{cases}
\end{align*}

\begin{align*}
[(m_0, c_1?v, m_1), (m_1, c_2!1, m_0)]
\end{align*}

The temporal specification of each module describes its semantics. Now the pure temporal logic MTL can be used to derive many other theorems from the basic ones \texttt{Th\_3} and \texttt{Th\_4}. The derived theorems represent valid specifications of each module which follows logically (i.e. using the logic MTL) from its basic specification contained in \texttt{Th\_3} (or \texttt{Th\_4}). Often, we are interested in deriving specifications describing desired safety and liveness properties of a module. To illustrate this, let us first use the rule \texttt{PAR} to derive the basic specification of the parallel module $M_1 \parallel M_2$, and then derive some desired properties for this program within the logic MTL. It is not difficult to
verify that conditions of the rule PAR are satisfied, e.g. the interfaces of \( M_1 \) and \( M_2 \) are compatible, and thus the following theorem can be derived from Th3 and Th4:

\[
\text{Th5:}
\]

\[
\{ \text{var } x, \text{where } c_1 = e, \ c = e, \ \text{init } c_1 \pi_1 = l_0; \ \{ \text{var } y, \text{where } c_2 = e, \ \text{init } c_2 \pi_2 = m_0; \ (l_0, c_1 ! 1, l_1), (l_1, c_2 ? x, l_2), (l_2, c_1 ! 1, l_0) \}_{\pi_1} \parallel (m_0, c_1 ! y, m_1), (m_1, c_2 ? 1, m_0) \}
\]

\[
\langle \text{out}(c_1) \lor \text{in}(c_1) \lor \text{out}(c_2) \lor \text{in}(c_2) \lor \text{out}(c), \ \text{init}_1 \land \text{init}_2 \land F \land G \land \text{shuffle}(w'_1, w'_2) \rangle
\]

where \( F =_{df} F_{01} \land F_{02} \land F_1 \land F_2 \land F_3 \) and \( G =_{df} G_{01} \land G_{02} \land G_1 \land G_2 \)

and \( w'_1 = (\pi_1, x, c) \) and \( w'_2 = (\pi_2, y) \)

Let us now consider the logic MTL to derive consequences of the formula \( \text{init}_1 \land \text{init}_2 \land F \land G \land \text{shuffle}(w'_1, w'_2) \) in theorem Th5. The derived consequences are also valid specifications of the module \( M_1 \parallel M_2 \).

1. \( \text{init}_1 \land \text{init}_2 \land F \land G \land \text{shuffle}(w'_1, w'_2) \)

\[
\{ \text{axioms A11, MP} \}
\]

2. \( \text{at } _{-\pi_1}(l_0) \land \text{at } _{-\pi_2}(m_0) \land c_1 = e \land c = e \land c_2 = e \land \bigoplus_{w_1 \cup w_2} \text{at } _{-\pi_1}(l_1) \land \text{at } _{-\pi_2}(m_0) \land c_1 = 1 \land e \land c = e \land c_2 = e \land c_2 = e \land y = 1 \)

\[
\{ \text{axioms A3, A4, A7, MP} \}
\]

3. \( \bigotimes_{w_1 \cup w_2} \bigotimes_{w_1 \cup w_2} \text{at } _{-\pi_1}(l_1) \land \text{at } _{-\pi_2}(m_1) \land c_1 = e \land c = e \land c_2 = e \land y = 1 \)

\[
\{ \text{axioms A3, A4, A7, MP} \}
\]

4. \( \bigotimes_{w_1 \cup w_2} \bigotimes_{w_1 \cup w_2} \bigotimes_{w_1 \cup w_2} \bigotimes_{w_1 \cup w_2} \text{at } _{-\pi_1}(l_1) \land \text{at } _{-\pi_2}(m_0) \land c_1 = e \land c = e \land c_2 = e \land y = 1 \)

\[
\{ \text{axioms A3, A4, A7, MP} \}
\]

5. \( \bigotimes_{w_1 \cup w_2} \bigotimes_{w_1 \cup w_2} \bigotimes_{w_1 \cup w_2} \bigotimes_{w_1 \cup w_2} \text{at } _{-\pi_1}(l_2) \land \text{at } _{-\pi_2}(m_0) \land c_1 = e \land c = e \land c_2 = e \land x = 1 \land y = 1 \)

Finally, note that liveness properties are directly derived from the previous ones, since \( \diamond p \) is a consequence of \( \bigotimes_w p \) in MTL. For example, the following theorem is derivable from ones given previously.

\[
M_1 \parallel M_2 \langle \text{inter}, \text{init}_1 \land \text{init}_2 \land (\text{at } _{-\pi_1}(l_2) \land \text{at } _{-\pi_2}(m_0))c_1 = e \land c = e \land c_2 = e \land x = 1 \land y = 1 \rangle
\]

where \( \text{inter} =_{df} \text{out}(c_1) \lor \text{in}(c_1) \lor \text{out}(c_2) \lor \text{in}(c_2) \lor \text{out}(c) \)

In this section we have just sketched the compositional proof of some desired properties of a small concurrent program. To construct proofs, rules have been used forwards in order to verify properties of a composite program on the basis of properties of its elementary components. When using the rules backwards, we find ourselves doing another task which consists of incrementally deriving pieces of a large system by decomposing specifications into smaller ones. In [20] an example is developed from this point of view.
8. Conclusion and related work

In this paper we have presented the preliminary concepts of a refined temporal logic that guarantees a fully abstract semantics w.r.t. to the chosen level of observation. We have shown how a compositional temporal proof system for concurrent programs can be derived. The resulting full logic provides a practicable method for both compositional verification and modular construction of concurrent programs. The novelty of the refined temporal logic lies mainly in its ability to express properties with any chosen level of abstraction. Many versions of Pnueli's temporal logic [24] have been proposed to describe a program by a temporal formula [17]. Some differ from others by their expressiveness, but all of them represent programs by formulas that are not invariant under stuttering. Consequently, a compositional rule for parallel composition was hard to obtain, and where it was possible the result was very complex. Moreover, a finer-grained program could not implement a coarser-grained one in these logics.

Lamport's TLA [11] is the first logic in which programs are described by formulas that are invariant under stuttering. With the refined semantics for the basic temporal operators proposed here, it is shown that results equivalent to those in TLA may be reformulated in the regular temporal logic with the advantage, in our logic, that temporal quantifiers behave like the first-order quantifiers. Another attempt to tackle the problem of stuttering within the classical temporal logic is done by Pnueli in [26]. The main difference between Pnueli's work and this lies in the fact that, contrary to our discrete temporal logic, Pnueli deals with the temporal logic TLR [26] which is based on a dense time domain (isomorphic to reals). Our proposal mainly intends to achieve results equivalent to Lamport's (for TLA) and Pnueli's ones (for TLR) for discrete temporal logic (TL [17]). We thus define a discrete temporal logic that supports refinement and systematic development of concurrent systems.

Finally, this work is undertaken with the idea of refining a previous logic [18, 19] which we felt was cumbersome for reasoning about real-size concurrent programs. Closure under stuttering is aimed at reaching a more modular (and hence more practical) method which can support systematic design of concurrent programs starting from their desired properties.

Appendix A

Ex-1. Rule GMP: \( p \Rightarrow q, \Box p \) \\
\[ \Box q \]

Proof.

1. \( \Box(p \Rightarrow q) \Rightarrow (\Box p \Rightarrow \Box q) \) \text{ A4}
2. \( \Box(p \Rightarrow q) \Rightarrow (\Box p \Rightarrow \Box q) \) \text{ MP A1, 1}
3. $\Box(p \supset q)$  premise, definition
4. $(\Box p \supset \Box q)$  MP 2, 3
5. $\Box p$  premise
6. $\Box q$  MP 4, 5

Ex-2. Rule $\bigoplus^w G$: \[
\frac{\Box p}{\Box \bigoplus^w p}
\]

Proof.
1. $\Box p$  premise
2. $\Box p \supset \Box \bigoplus^w p$  A5
3. $\Box \bigoplus^w p$  MP 2, 1

Ex-3. Rule $\bigoplus^w M$: \[
\frac{p \supset q}{\bigoplus^w q \bigoplus^w p}
\]

Proof.
1. $p \supset p$  premise
2. $\Box \bigoplus^w (p \supset q)$  $\bigoplus^w G$
3. $\bigoplus^w (p \supset q) \supset (\bigoplus^w p \supset \bigoplus^w q)$  A3
4. $\bigoplus^w p \supset \bigoplus^w q$  MP 3, 2

Ex-4. Theorem T1: $\bigoplus^w p \land \bigoplus^w q \iff \bigoplus^w (p \land q)$

Proof.
1. $\Box(p \supset p)$  GEN
2. $\Box(\bigotimes^w \text{true} \supset \bigotimes^w \text{true})$  INS
3. $\bigoplus^w p \land \bigoplus^w q \iff \bigoplus^w (p \land q)$  GMP 2, A12 with $w_1 = w_2 = w$

Ex-5. Theorem T2: $\bigoplus^w p \lor \bigoplus^w q \iff \bigoplus^w (p \lor q)$

Proof.
1. $\bigoplus^w \neg p \land \bigoplus^w \neg q \iff \bigoplus^w (\neg p \land \neg q)$  T1
2. $\bigotimes^w (p \otimes q) \iff \bigotimes^w q \lor \bigotimes^w q$  Prop, definition
3. $\bigoplus^w \text{false} \iff \bigoplus^w \text{false}$  INS
4. $\bigotimes^w (p \lor q) \lor \bigoplus^w \text{false} \iff \bigotimes^w q \lor \bigotimes^w q \lor \bigoplus^w \text{false}$  Prop
5. $\bigoplus^w (p \lor q) \iff \bigoplus^w q \lor \bigoplus^w q$  Prop, definition
Ex-6. Theorem T3: $\bigoplus_w p \lor \bigoplus_w \text{false} \Rightarrow \bigoplus_w p$

Proof.
1. $\text{false} \Rightarrow p$ \hspace{1cm} GEN
2. $p \Rightarrow p$ \hspace{1cm} GEN
3. $\bigoplus_w \text{false} \Rightarrow \bigoplus_w p$ \hspace{1cm} $\bigoplus_w \text{M1}$
4. $\bigoplus_w p \Rightarrow \bigoplus_w p$ \hspace{1cm} $\bigoplus_w \text{M2}$
5. $\bigoplus_w p \lor \bigoplus_w \text{false} \Rightarrow \bigoplus_w p$ \hspace{1cm} Prop

Ex-7. Theorem T4: $\bigotimes_w (p \Rightarrow q) \Rightarrow (\bigotimes_w p \Rightarrow \bigotimes_w q)$

Proof.
1. $\bigoplus_w (p \Rightarrow q) \Rightarrow \bigoplus_w p \Rightarrow \bigoplus_w q$ \hspace{1cm} A3
2. $\bigoplus_w (p \Rightarrow q) \land \bigotimes_w \text{true} \Rightarrow ((\neg (\bigotimes_w p) \lor \bigotimes_w q) \land \bigotimes_w \text{true})$ \hspace{1cm} Prop
3. $\bigotimes_w (p \Rightarrow q) \Rightarrow (\neg (\bigoplus_w p) \land \bigotimes_w \text{true}) \lor (\bigotimes_w q \land \bigotimes_w \text{true})$ \hspace{1cm} Prop, A7
4. $\bigotimes_w (p \Rightarrow q) \Rightarrow (\bigotimes_w \neg p \bigotimes_w q)$ \hspace{1cm} Prop, A2
5. $\bigotimes_w (p \Rightarrow q) \Rightarrow (\bigoplus_w \neg p \lor \bigotimes_w q)$ \hspace{1cm} Prop, A2
6. $\bigotimes_w (p \Rightarrow q) \Rightarrow ((\neg \bigotimes_w p) \lor \bigotimes_w q)$ \hspace{1cm} Prop, A2
7. $\bigotimes_w (p \Rightarrow q) \Rightarrow (\bigotimes_w p \Rightarrow \bigotimes_w q)$ \hspace{1cm} Prop

Often, we use the step Prop in the proof constructions without explicit explanation. This is done whenever the step corresponds to a propositional reasoning generalized by the rule GEN and then particularized using MP, A1, INS.

Appendix B

Proof (sketch). We prove the soundness of all axioms and rules of $\mathcal{G}$, directly from the definition of the semantics. Classically, we have to show that each axiom is temporally valid and that each rule preserves temporal validity, i.e.

$$\forall \xi \in \Delta, \forall \sigma \in \Gamma. \left( \bigwedge_{i<n} (\xi, \sigma) \models p_i \right) \Rightarrow (\xi, \sigma) \models p_n$$

Since formal proofs of the soundness of most axioms and rules are lengthy without being difficult to feel intuitively, we give proofs of only some of them (the complete proof is given in [21]).

Axioms A1 and A4 are trivially valid (definition of $\Box$)
Axiom A5

\((\xi, \sigma) \models \square p \text{ iff } \forall k \geq i. (\xi, \sigma^k) \models p \text{ then } \forall k \geq i. (\xi, \sigma^k) \models \bigoplus_w p \text{ then } (\xi, \sigma) \models \square \bigoplus_w p \)

Axiom A7 for every position \(i\)

\((\xi, \sigma^i) \models \bigoplus_w p \land \bigotimes_w \text{true} \text{ iff } \forall j > i. \sigma_j \models w = \sigma_1 \models w \land \forall k.i \leq k < j. \sigma_k \models w = \sigma_i \models w \land (\xi, \sigma^j) \models p) \land \)

\((\exists j > i. \sigma_j \models w \neq \sigma_i \models w) \text{ iff } (\exists j > i. \sigma_j \models w \neq \sigma_i \models w \land \forall k.i \leq k < j. \sigma_k \models w = \sigma_i \models w \land (\xi, \sigma^j) \models p) \text{ iff } (\xi, \sigma^i) \models \bigotimes_w p \)

Axiom A8 for every position \(i\)

\((\xi, \sigma^i) \models \bigoplus_w p \lor \bigoplus_w q \text{ iff } ((\forall j > i. \sigma_j \models w_1 = \sigma_1 \models w_1) \lor (\exists j > i. \sigma_j \models w_1 \neq \sigma_i \models w_1 \land \forall k.i \leq k < j. \sigma_k \models w_1 = \sigma_i \models w_1 \land (\xi, \sigma^j) \models p)) \lor \)

\(((\forall j > i. \sigma_j \models w_2 = \sigma_1 \models w_2) \lor (\exists j > i. \sigma_j \models w_2 \neq \sigma_i \models w_2 \land \forall k.i \leq k < j. \sigma_k \models w_2 = \sigma_i \models w_2 \land (\xi, \sigma^j) \models q)) \text{ iff } ((\forall j > i. \sigma_j \models w_1 \lor w_2 = \sigma_1 \models w_1 \lor w_2) \lor \)

\((\exists j > i. \sigma_j \models w_1 \lor w_2 \neq \sigma_i \models w_1 \lor w_2 \land \forall k.i \leq k < j. \sigma_k \models w_1 \lor w_2 = \sigma_i \models w_1 \lor w_2 \land (\xi, \sigma^j) \models q) \text{ by Proposition 19} \text{ iff } (\xi, \sigma^i) \models \bigoplus_{w_1 \lor w_2} p \lor q \)

Axiom A9 for every position \(i\)

\((\xi, \sigma^i) \models \bigotimes_{w_1 \lor w_2} p \land \bigoplus_{w_1 \lor w_2} q \text{ iff } ((\forall j > i. \sigma_j \models w_1 \land w_2 = \sigma_1 \models w_1 \land w_2) \lor \)

\(((\forall j > i. \sigma_j \models w_2 = \sigma_1 \models w_2) \lor (\exists j > i. \sigma_j \models w_2 \neq \sigma_i \models w_2 \land \forall k.i \leq k < j. \sigma_k \models w_2 = \sigma_i \models w_2 \land (\xi, \sigma^j) \models q) \text{ then } (\forall j > i. \sigma_j \models w_1 \land w_2 = \sigma_1 \models w_1 \land w_2) \lor \)

\((\exists j > i. \sigma_j \models w_1 \land w_2 \neq \sigma_i \models w_1 \land w_2 \land \forall k.i \leq k < j. \sigma_k \models w_1 \land w_2 = \sigma_i \models w_1 \land w_2 \land (\xi, \sigma^j) \models q) \text{ then } (\xi, \sigma^i) \models \bigotimes_{w_1 \land w_2} p \land q) \)

Axiom A11 for every position \(i\)

let \((\xi, \sigma^i) \models \bigotimes_{w_1 \land w_2} \text{true} \Rightarrow \bigotimes_{w_1 \land w_2} \text{true} \text{ then } (\xi, \sigma^i) \models \bigoplus_{w_1 \land w_2} \text{false} \lor \bigotimes_{w_1 \land w_2} \text{true} \text{ iff } ((\forall j > i. \sigma_j \models w_1 \land w_2 = \sigma_1 \models w_1 \land w_2) \lor (\exists j > i. \sigma_j \models w_1 \land w_2 \neq \sigma_i \models w_1 \land w_2) \text{ hyp}) \)
(ξ, σ') |= Σ_{w_1} p ∧ Σ_{w_2} q \iff
\((\forall j > i. \sigma_j[w_1 = σ_j[w_1]) \lor (\exists j > i. \sigma_j[w_1 \neq σ_i[w_1 \land \forall k. i \leq k < j. σ_k[w_1
= σ_i[w_1 \land (ξ, σ') = p]) \land
\((\forall j > i. \sigma_j[w_2 = σ_i[w_2]) \lor (\exists j > i. \sigma_j[w_2 \neq σ_i[w_2 \land \forall k. i \leq k < j. σ_k[w_2
= σ_i[w_2 \land (ξ, σ') = q) \) \iff
(∀j > i. σ_j[w_1 \cup w_2 = σ_i[w_1 \cup w_2] \lor
(∃j > i. σ_j[w_1 \cup w_2 \neq σ_i[w_1 \cup w_2 \land \forall k. i \leq k < j. σ_k[w_1 \cup w_2
= σ_i[w_1 \cup w_2 \land (ξ, σ') = p) \land
(∃j > i. σ_j[w_1 \cup w_2 \neq σ_i[w_1 \cup w_2 \land \forall k. i \leq k < j. σ_k[w_1 \cup w_2
= σ_i[w_1 \cup w_2 \land (ξ, σ') = q) \) \) \iff
(ξ, σ') |= Σ_{w_1 \cup w_2} p \land q

Axioms for quantifiers follow obviously from definitions.

Rules GEN, INS, MP concern reasoning in the general level. They are similar to those found in any classical temporal proof system like Manna and Pnueli's [13, 17] and their soundness is preserved.

The soundness of the rule EXT follows from the Theorem 22. □

References