Proving Partial Order Properties *

Doron Peled†
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974, USA

Amir Pnueli
Department of Applied Mathematics
The Weizmann Institute of Science
Rehovot 76100, Israel

April 5, 1994

Abstract

A temporal verification method which is based upon partial order semantics of *traces* [33] is presented. The semantic model used here can express the distributed nature of a program. E.g., properties such as serializability of database transactions, layering of a program, snapshots or the parallel execution of program segments. The proof rules are shown to be sound and relatively complete.

1 Introduction

Partial order semantics is recently becoming accepted as an intuitive formal model for representing distributed programs executions (see [35, 24, 41]). Among its many presentations, trace semantics [33] seems to be of a particular elegance.

While it is generally recognized that partial order semantics provides a significantly more faithful representation of concurrency than, say, interleaving semantics, the development of corresponding logics for specifying and verifying properties of programs, as they appear in the partial order semantics, has been slower. It is only recently [20, 42, 39] that several versions of temporal logics over partial orders have been defined and illustrated.

In this paper we continue the development of the temporal logic **ISTL*** [20] (Interleaving Set Temporal Logic), by presenting for it complete proof rules for proving properties of the form $AG\varphi$ (in all traces φ) and $EF\varphi$ (every run contains an observation, on which eventually φ). These two rules cover many of the safety and liveness properties one may wish to prove for a concurrent program.

Let us compare the expressive power of logics based on interleaving semantics with logics similar to the one presented here, that are based on partial order trace semantics [33] or similarly on *pomsets* [41, 15]. Interleaving semantics assign to a concurrent program a large set of *observations* (interleaving sequences) that represent all the possible executions of the program when linearized along a single time axis. Trace theory introduces more structure into this large set by defining an equivalence relation between the observations, such that two observations are considered equivalent if they differ

*This research was supported in part by the European Community ESPRIT Basic Research Action project 3096 (SPEC).

†This research was carried out while the first author was with the Department of Computer Science, The Technion, Haifa 32000, Israel.

only by the order they execute independent (concurrent) actions. We refer to the equivalence classes of this relation as *runs* or *computations* (interleaving sets in the terminology of [20]).

A temporal logic over interleaving semantics specifies a property of the program by requiring that *all* observations (i.e., all the interleaving sequences, as in linear interleaving semantics) satisfy a particular sequence predicate, that usually constrains the order in which certain events may occur. Branching time temporal logic over interleaving semantics may also require that *some* observation satisfies a particular sequence predicate.

Temporal logic over trace semantics has a two–level control over the range of observations that should satisfy the appropriate ordering constraints. It may require that *all* observations in *all* runs satisfy the constraints. It may also require that *every* run contains *some* observation that satisfies the constraints.

There are two main reasons why such more discriminating control is very advantageous. The first case is the existence of properties whose natural expression *require* the finer distinction between observations that belong to disjoint runs and observations that are equivalent, and hence belong to the same run. The canonical example of such properties is that of *serializability.* When we wish to state that a certain distributed program correctly implements a database, we do not expect all events pertaining to each transaction to occur within disjoint and non–overlapping intervals. But we expect the overall effect to be *as though* they did.

The best and most natural way of expressing this property is to claim that every run contains at least one observation in which the intervals spanned by transactions are non–overlapping. The literature is full of strange and contorted attempts to specify this property in logics that are inappropriate for the task, while its expression in the temporal logic over partial orders is simple and natural.

A second and similar example is the specification of a distributed snapshot algorithm, such as [5]. The natural specification is that if the algorithm prints $(y = 2, z = 3)$ at some point, then there must have been a preceding state at which y equals 2 and z equals 3. Unfortunately, if y and z belong to separate processes it could very well be that not all observations printing $(y = 2, z = 3)$ actually contain such a global state. The correct specification of this property is that every observation printing $(y = 2, z = 3)$ is *equivalent* to an observation containing a global state in which $y = 2, z = 3$.

Another important application of the partial order semantics is of course the ability to require that certain actions be implemented *concurrently.* If we view some of the attempts to synthesize programs from their specifications ([11, 32, 40]), we observe that in spite of the intention of synthesizing concurrent programs, the direct result of the given procedures is a sequential (often non–deterministic) program. To derive a concurrent program from this result, additional decomposition steps are called for. The reason for this shortcoming that is inherent in these works is simply that the logics on which they are based cannot express the requirement of concurrency. To satisfy the specification, the resulting program must respect the concurrency constraints.

Thus, our first motivation for studying temporal logic over partial orders is that there are some central and important properties, such as serializability, that *cannot* be naturally specified in any other way.

The second motivation is that of economy and efficiency. In some cases, it is sufficient and easier to prove that *one* observation out of every run satisfies certain ordering constraints. This case is amply demonstrated in [21], and we will only briefly mention one of these applications to the case of communication closed layers [10].

Sometimes, a program can be partitioned into several segments (orthogonal to its partition into processes), such that communication may occur only within a segment. For such a program, the most convenient behaviors to analyze are those in which all the processes move together from one layer (segment) into the next. It is certainly not true that all observations have this character, but clearly every run contains at least one such observation which we may therefore proceed to analyze.

The rest of the paper is organized as follows: In Section 2, trace semantics is defined for *transition programs* [26, 27]. Trace semantics is used as an interpretation for the temporal logic **ISTL*** in Section 3. Then, in Section 4, the motivation and intuition for the proof rules are given. In Section 5, a complete proof system for eventuality properties is given. The soundness and completeness of the system is proved in Section 6. Section 7 presents examples of proving serializability of database transactions and the correctness of the snapshot algorithm.

2 Trace Semantics

A *concurrent alphabet* is a pair $(T,\ D)$ where T is a finite nonempty set and $D \subseteq T \times T$ is a reflexive and symmetric binary relation, called the *dependency* relation. Two elements which are related by D are said to be *dependent*, otherwise they are *independent*. Denote by T^* the set of finite strings of the alphabet T. Concatenation is denoted by juxtaposition. We use $w,\ w',\ w_i,\ v,\ u\ \ldots$ to denote strings, ε to represent the empty string and $\tau,\ \tau',\ \alpha,\ \alpha_i,\ \beta \ldots$ to represent members of T. Denote $Id_T = \{(\alpha,\ \alpha) \mid \alpha \in T\}$.

Traces were introduced by Mazurkiewicz [33] as the equivalence classes of strings with respect to a concurrent alphabet. Two strings $v,\ w \in T^*$ are considered equivalent if one can transform v into w by repeatedly exchanging adjacent independent operations. They are used for giving semantics to finite state languages such as Petri-nets [38]. This is slightly different from our definition of accessible traces in that we allow only equivalence classes which are associated with first order interpretation. A further extension of traces which allows conditional dependency among operations appears in [22].

A program will be defined over a first order language $\mathcal{L}$, where the relation and function symbols are interpreted over a first order structure $\mathcal{A}$. We assume some fixed given interpretation to the relation and function symbols of $\mathcal{A}$. An *assignment* J over an ordered set of variables $\vec{y}$ is a mapping, associating with each variable $z \in \vec{y}$ a value $J(z)$ from its associated domain. If t is a term, denote the value obtained by evaluating t after assigning its variables values according to J by $J(t)$. Similarly, if $\vec{t}$ is a tuple (finite ordered set) of terms, $J(\vec{t})$ is a tuple of values evaluated from the terms $\vec{t}$ under the assignment J. The i^{th} component of $J(\vec{t})$ is denoted $J(\vec{t})_i$.

For an assignment J and a first order formula φ, denote the fact that φ holds over $\mathcal{A}$ and J by the standard notation $J \stackrel{\mathcal{A}}{\models} \varphi$. If φ is a sentence, i.e., it has no free variables, we write the fact that φ holds in $\mathcal{A}$ by $\stackrel{\mathcal{A}}{\models} \varphi$. We omit the structure $\mathcal{A}$ when it is understood from the context.

A *program* P is a quadruple $\langle T,\ D,\ \vec{y},\ \Theta\rangle$ where $(T,\ D)$ is a concurrent alphabet, $\vec{y}$ is a finite ordered set of variables, and $\Theta \in \mathcal{L}$ is a satisfiable predicate with free variables from $\vec{y}$ called the *initial condition*. In addition, the program associates with each element $\tau \in T$, which we call an *operation*, a pair $\langle en_\tau,\ f_\tau\rangle$ where $en_\tau \in \mathcal{L}$ (the *enabling condition*) and f_τ (the *transformation function*) is a tuple of $|\vec{y}|$ terms of $\mathcal{L}$. All the variables of f_τ and the free variables of en_τ are from $\vec{y}$. The enabling condition controls when the operation can execute, while the transformation dictates the new values of the program variables, calculated using the expressions in f_τ from the old values.

The following conditions are required for each pair of operations $\alpha,\ \beta$ such that $(\alpha,\ \beta) \notin D$:

- $(en_\alpha(\vec{y}) \wedge en_\beta(\vec{y})) \rightarrow f_\alpha(f_\beta(\vec{y})) = f_\beta(f_\alpha(\vec{y}))$ [commutativity of independent operations[1]].
- $en_\alpha(\vec{y}) \rightarrow (en_\beta(\vec{y}) \leftrightarrow en_\beta(f_\alpha(\vec{y})))$ [independent operations can neither disable nor enable each other].

An operation τ can be written as the guarded command $\langle en_\tau(\vec{y}) \longrightarrow \vec{y} := f_\tau(\vec{y})\rangle$. The intended meaning is that the condition before the arrow controls when the operation may execute and the effect of the operation is to simultaneously assign the $|\vec{y}|$ expressions $f_\tau(\vec{y})$ into the set of variables $\vec{y}$. For example, $\langle y_1 < y_2 \longrightarrow (y_2, y_1, y_3) := (y_1, y_2, y_3)\rangle$ is an operation that switches the values of the variables y_1 and y_2, if y_1 is smaller than y_2. The value of y_3 is unaltered. Since a typical operation changes only a subset of the program variables, we will usually denote explicitly only the expressions assigned to the variables that may be changed (e.g., $\langle y_1 < y_2 \longrightarrow (y_2, y_1) := (y_1, y_2)\rangle$).

A transformation from a shared-variables programming language, and a CSP-like [18] language into a set of operations can be found in [28]. In the sequel, it will be demonstrated how to transform Petri-Nets [38] into a set of operations. There are several ways to define the dependency relation D for a given program. The simplest definition (obviously not the most efficient) is to consider two operations τ_1 and τ_2 to be dependent if there exists at least one variable which is explicitly referenced by both τ_1 and τ_2. We consider a variable to be explicitly referenced by an operation τ if it occurs either in f_τ or free in en_τ.

We introduce the following definitions, taken from [22]. A *history* of a program P is a pair $h = \langle J, v\rangle$ where J is an assignment called the *initial assignment of* h, and $v \in T^*$. Furthermore, the following condition is satisfied: Let $n = |v|$, $v = \alpha_1\alpha_2\ldots\alpha_n$. There exists a sequence of assignments $J_0, J_1, \ldots, J_n$ with $J_0 = J$ such that for each $0 < i \leq n$,

1. $J_{i-1} \models en_{\alpha_i}$ [α_i is enabled in J_{i-1}], and
2. for each $0 \leq k < |\vec{y}|$, $J_i(y_k) = f_{\alpha_i}(J_{i-1}(\vec{y}))_k$ [i.e., $J_i(y_k)$ is the k^{th} component in the tuple obtained when applying the transformation f_{α_i} to the values assigned by J_{i-1}].

For each history $h = \langle J, v\rangle$, let the n^{th} assignment in the above sequence, namely $J_{|v|}$ (which is a function of J and v) be denoted by fin_h. This is called the *final assignment* of h. We use $h, h', h_i, \ldots$ for histories.

Two histories $h = \langle J, v\rangle$ and $h' = \langle J, w\rangle$ with a common initial assignment are *equivalent* iff there exists a sequence of histories $\langle J, v_1\rangle, \langle J, v_2\rangle, \ldots, \langle J, v_n\rangle$ with $v_1 = v$ and $v_n = w$, and for each $1 \leq i < n$ there exist $u, \bar{u} \in T^*$, $(\alpha, \beta) \notin D$ such that $v_i = u\alpha\beta\bar{u}$, $v_{i+1} = u\beta\alpha\bar{u}$. This is denoted $h \equiv h'$. For example, let $T = \{\alpha, \beta, \gamma\}$, $D = Id_T \cup \{(\alpha, \gamma), (\gamma, \alpha)\}$. Then, $\langle J, \alpha\alpha\beta\beta\gamma\gamma\rangle \equiv \langle J, \alpha\beta\alpha\gamma\beta\gamma\rangle \equiv \langle J, \beta\alpha\alpha\gamma\gamma\beta\rangle$, but $\langle J, \alpha\alpha\beta\gamma\gamma\rangle \not\equiv \langle J, \gamma\gamma\beta\alpha\alpha\rangle$ (because the α's and the γ's cannot be interchanged).

A *trace* is an equivalence class of histories. Denote a trace as $[J, w]$ where J is the common initial assignment and $\langle J, w\rangle$ is some member of the equivalence class. Traces will be denoted using $\sigma, \sigma', \sigma_i, \rho, \ldots$.

Obviously, if $h = \langle J, u\alpha\beta\bar{u}\rangle$ is a history of P, and $(\alpha, \beta) \notin D$, then $h' = \langle J, u\beta\alpha\bar{u}\rangle$ is also a history of P. Moreover, $fin_h = fin_{h'}$. Therefore, the final assignment of all the histories that belong to the same trace are identical. This permits defining fin_σ as $fin_{\langle J, v\rangle}$ for any $\langle J, v\rangle \in \sigma$. Concatenation between two traces $\sigma_1 = [J_1, v]$ and $\sigma_2 = [J_2, w]$, denoted $\sigma_1\sigma_2$, is defined when $fin_{\sigma_1} = J_2$, and is given by $[J_1, vw]$.

[1] Notice that the equivalence relation '=' is extended here to tuples.

The prefix relation '$\sqsubseteq$' between traces is defined as $\sigma_1 \sqsubseteq \sigma_2$ iff there exists some σ_3 such that $\sigma_1\sigma_3 = \sigma_2$. It is said that σ_1 is *subsumed* by σ_2. If in addition, the length of σ_1 is shorter than the length of σ_2 by exactly one, it is said that σ_2 is a *successor* of σ_1, and σ_1 is a *predecessor* of σ_2. If $\sigma' \sqsubseteq \sigma$, denote by σ/σ' the trace ρ such that $\sigma'\rho = \sigma$. It is said that ρ is a *suffix* of σ. Another notion of trace difference, denoted as $\sigma - \sigma'$ corresponds to removing the operations in σ' "from the end" of σ. That is, if $\rho\sigma' = \sigma$, then $\rho = \sigma - \sigma'$ (set difference is denoted with the '$\setminus$' symbol). We say that σ is an *intermediate trace between ρ and ρ'* when $\rho \sqsubseteq \sigma \sqsubseteq \rho'$.

If $\sigma = \rho[\tau]$ for some trace ρ, it is said that τ is *executed last* in σ (however, τ need not be unique, as other independent operations of τ can also be executed last in σ).

An *accessible* trace of a program is any trace obtained as the equivalence class of some of the histories of the program with an initial assignment satisfying Θ. The traces *generated by a program* are its accessible traces and the suffixes of such traces. In the sequel we consider only traces generated by programs. For historical reasons and for convenience, we omit in the sequel the initial assignment and denote a trace as $[w]$. We will denote $\sigma \models \varphi$ if $fin_\sigma \models \varphi$. Denote by $\Delta(\sigma)$ the set of operations that occur in (any string that is a member of) the trace σ.

Two traces σ_1, σ_2 are said to be *consistent* [34] iff there exists σ_3 such that $\sigma_1 \sqsubseteq \sigma_3$ and $\sigma_2 \sqsubseteq \sigma_3$. A set of traces Π is *directed* if for each pair of traces σ, $\rho \in \Pi$, it contains also a trace that subsumes both of them. A *run* (partial order execution) Π of P is a maximal directed set of traces. Let R_P be the set of all runs of P. For each pair of consistent traces σ_1, $\sigma_2 \in \Pi$, denote by $\sigma_1 \sqcup \sigma_2$ the minimal (with respect to '$\sqsubseteq$') trace $\sigma_3 \in \Pi$ subsuming both σ_1 and σ_2.

For $\varphi \in \mathcal{L}$, let $wp_\tau(\varphi) = en_\tau \wedge \varphi[f_\tau(\vec{y})/\vec{y}]$ (for $\vec{t}$ a tuple of terms such that $|\vec{t}|=|\vec{y}|$, $\varphi[\vec{t}/\vec{y}]$ is the predicate obtained from φ, by substituting for each $i \in 1 \ldots |\vec{y}|$ the element t_i of $\vec{t}$ instead of each free occurrence of y_i). The predicate $wp_\tau(\varphi)$ is the weakest (most general) predicate [9] satisfying that if $fin_\sigma \models wp_\tau(\varphi)$ then $fin_{\sigma[\tau]} \models \varphi$.

For each run Π, define an *interleaving sequence* as a sequence of traces $\sigma_0\,\sigma_1\,\sigma_2\ldots$ of Π such that $\sigma_0 = [\varepsilon]$ and for each $i \geq 0$, $\sigma_{i+1} = \sigma_i[\alpha_i]$ for some $\alpha_i \in T$. Interleaving sequences are denoted by o, $\hat{o}$, ...

An *observation* of Π is an interleaving sequence $\sigma_0\,\sigma_1\,\sigma_2\ldots$, where for each $\sigma \in \Pi$, there exists some σ_i, $i \geq 0$ such that $\sigma \sqsubseteq \sigma_i$. For each $\sigma \in \Pi$ there exists an observation of Π on which σ occurs (this is a simple consequence of the following theorem). Denote by $|o|$ the length of an observation (ω if o is infinite). The i^{th} trace in an observation o ($0 \leq i < |o|$) will be denoted by $o(i)$. The set $intset(\Pi)$ of all observations of the run Π is termed an *interleaving set*.

An interleaving sequence o is said to *ignores an operation* τ iff τ is enabled in a trace $o(i)$, $0 \leq i < |o|$, which appears in o, and for each $o(j)$, $i \leq j < |o|$, no operation α such that $(\tau, \alpha) \in D$ appears in $o(j)/o(i)$. It is said that τ is *ignored from* $o(i)$. Notice that by the requirements imposed on the relation D, if o ignores τ from $o(i)$, then it is enabled in each trace that occurs in o after $o(i)$.

Theorem 2.1 *An interleaving sequence is an observation iff it does not ignore any operation. (A similar proof for a related model appears in [23].)*

Proof. $\Longrightarrow$. Assume for the contrary that o is an observation of some run $\Pi \in R_P$ which ignores τ. Let σ be the minimal trace in o where τ is ignored. Define $A = \{\rho \mid \rho \in \Pi \wedge \sigma \sqsubseteq \rho\}$ (all the traces of Π subsuming σ). Observe that for each $\rho \in A$, $\rho \models en_\tau$, since for some σ' which appears in o later than σ, $\sigma \sqsubseteq \rho \sqsubseteq \sigma'$ (each trace of Π is subsumed by some trace of the observation) and no operation α, $(\tau, \alpha) \in D$ may appear in σ'/σ. Thus, no independent operation can disable τ. Let $B(\tau) = \{\rho[\tau] \mid \rho \in A\}$.

It will now be shown that the set $A \cup B(\tau)$ is directed which contradicts the maximality of Π. One case is where $\rho_1 \in A$ and $\rho_2[\tau] \in B(\tau)$. Hence, there exists a trace $\rho' \in A$, that subsumes both ρ_1 and ρ_2. The trace $\rho'[\tau] \in B(\tau)$ subsumes both ρ_1 and $\rho_2[\tau]$. The case with both $\rho_1[\tau]$, $\rho_2[\tau] \in B(\tau)$ is similar.

$\Longleftarrow$. Suppose o is an interleaving sequence of traces for some run $\Pi \in R_P$, but is not an observation. Let $K = \{\rho \mid \exists\sigma\, (\sigma \in o \wedge \rho \sqsubseteq \sigma)\}$ (all the traces subsumed by traces of o). Let $\sigma \in \Pi$ be some minimal trace which is not in K. Then, there exists some operation τ such that $\sigma = \rho[\tau]$ and $\rho \in K$ (otherwise, σ is not minimal). It will be shown that for each $\hat{\sigma} \in K$ satisfying $\rho \sqsubseteq \hat{\sigma}$ it holds that $\hat{\sigma} \models en_\tau$ and no operation dependent on τ appears in $\hat{\sigma}/\rho$.

To see this, assume for the contrary a minimal $\hat{\sigma}$ such that $\hat{\sigma} \in K$, $\hat{\sigma} = \sigma'[\alpha]$, $(\tau, \alpha) \in D$ and $\rho \sqsubseteq \sigma'$. It holds that $\sigma' \models en_\tau$ because $\rho \models en_\tau$ and from the minimality of $\hat{\sigma}$, no operation dependent on τ appears in σ'/ρ. Clearly $\alpha \neq \tau$, otherwise it is possible to commute τ with all the operations in σ'/ρ, obtaining that $\rho[\tau] \in K$, contrary to the assumption. It follows that $\sigma'[\tau]$ is the minimal trace subsuming both σ' and $\rho[\tau]$, hence $\sigma'[\tau] \in \Pi$. But $\sigma'[\tau]$ and $\sigma'[\alpha]$ cannot belong to the same run Π, a contradiction. Since all the traces of o belong to K, o ignores τ from the minimal trace in o that subsumes ρ. ∎

An interesting class of finite state programs can be represented by *Petri Nets* [38]. We will demonstrate verification programs that are described by Petri Nets because of their convenient graphical representation. However, notice that the verification method described in this paper is *not* limited to finite state programs. A Petri Net has circles called *places* and bars called *transitions*. A place p is called an *input place* of a transition τ if there exists an edge connecting p to τ. A place p is called an *output place* of a transition τ if there exists an edge connecting τ to p. The set of input places for a transition τ is denoted by ${}^\bullet\tau$, while the set of output places is denoted by $\tau^\bullet$. A place can be marked with a *token* or be unmarked. A transition is *enabled* if all its input places are marked and all its output places are unmarked. If a transition is enabled, it can be *fired* (executed). In this case, the token is removed from each of its input places and a token is put in each of its output places. (We only deal with *elementary net systems [46]* which can have no more than a single token in each place.)

In order to represent the transitions as operations, consider places as two valued variables. A place with a token represents the fact that the variable associated with the place has the value 1. Otherwise, it has the value 0. Define for each transition τ:

$$en_\tau = \bigwedge_{p \in {}^\bullet\tau} (p = 1) \wedge \bigwedge_{q \in \tau^\bullet} (q = 0)$$

The transformation function f_τ assigns zeros to the variables ${}^\bullet\tau$ and ones to the variables $\tau^\bullet$ while other variables remain unchanged. The relation D can be defined by letting $(\tau_1, \tau_2) \in D$ iff $({}^\bullet\tau_1 \cup \tau_1^\bullet) \cap ({}^\bullet\tau_2 \cup \tau_2^\bullet) \neq \phi$. It is sometimes convenient to denote the marking of the net by writing the set of places holding a token. Thus, $\{a, b\}$ means that $a = 1$ and $b = 1$ and all the other places are set to zero.

Example. Consider the Net A, depicted in Figure 1. Then,

$$T = \{\alpha_1, \alpha_2, \alpha_3, \beta_1, \beta_2, \beta_3\}$$

$$\begin{aligned} D = Id_T &\cup \{(\alpha_i, \beta_j) \mid 2 \leq i, j \leq 3\} \cup \{(\beta_i, \alpha_j) \mid 2 \leq i, j \leq 3\} \\ &\cup \{(\alpha_i, \alpha_j) \mid i + j \in \{3, 5\}\} \cup \{(\beta_i, \beta_j) \mid i + j \in \{3, 5\}\} \end{aligned}$$

The variables associated with the net are $\{p_i \mid 0 \leq i \leq 8\}$, and the initial condition Θ is

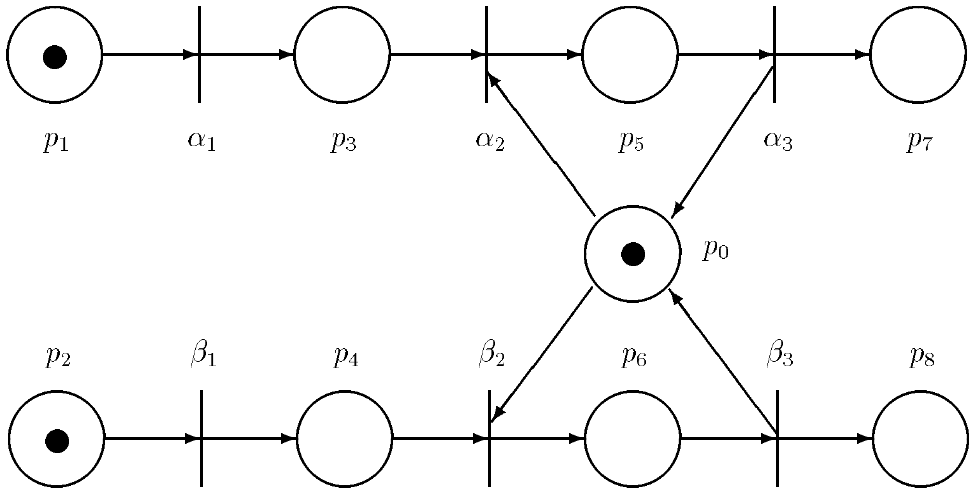

Figure 1: Net A.

$\{p_0, p_1, p_2\}$. The pairs of enabling conditions and transformation functions for each operation are listed below.

$$\begin{array}{ll}
\alpha_1 : \langle p_1 = 1 \wedge p_3 = 0 \longrightarrow (p_1, p_3) := (0, 1)\rangle & \beta_1 : \langle p_2 = 1 \wedge p_4 = 0 \longrightarrow (p_2, p_4) := (0, 1)\rangle \\
\alpha_2 : \langle p_0 = p_3 = 1 \wedge p_5 = 0 \longrightarrow (p_0, p_3, p_5) := (0, 0, 1)\rangle & \beta_2 : \langle p_0 = p_4 = 1 \wedge p_6 = 0 \longrightarrow (p_0, p_4, p_6) := (0, 0, 1)\rangle \\
\alpha_3 : \langle p_5 = 1 \wedge p_0 = p_7 = 0 \longrightarrow (p_0, p_5, p_7) := (1, 0, 1)\rangle & \beta_3 : \langle p_6 = 1 \wedge p_0 = p_8 = 0 \longrightarrow (p_0, p_6, p_8) := (1, 0, 1)\rangle
\end{array}$$

The place p_0 represents a semaphore [8]. The operations α_2 and β_2 represent the acquisition of the semaphore, while α_3 and β_3 represent its release. The traces for Net A, ordered according to prefixing relation are depicted in Figure 2. In this case there are two runs, Π_1 and Π_2. For example, the traces $[\alpha_1\alpha_2\beta_1] \in \Pi_1$ and $[\beta_1\beta_2\alpha_1] \in \Pi_2$ are not consistent. ▪

3 The Logic ISTL* and its Trace Semantics Interpretation

The logic **ISTL***(Interleaving Set Temporal Logic) was introduced in [20] as a logic to deal with global states (slices) generated from partial order executions. In this paper we interpret **ISTL*** over traces, but the resulting semantics is very similar to that of [20]. We also extend the logic by adding some *past* operators. The logic **ISTL*** strongly resembles the powerful branching time logic **CTL*** [12] in its interpretation over abstract structures. However, it differs from **CTL*** in the way it is interpreted over structures corresponding to programs and their executions. It is also related to **POTL** [39] in the introduction of the past operators.

An **ISTL*** structure is a branching structure, with a set of states and a set of sequences. In the application to programs, we will have an **ISTL*** structure corresponding to each run Π of the program, where traces of the run correspond to states of the structure.

Definition 3.1 *Syntax of* **ISTL*** *.*
We inductively define two sets of formulae: Γ (state formulae) and Φ (path formulae).

State Formulas (Γ)
- $\mathrm{T} \in \Gamma$.
- *if φ is a first order formula, then $\varphi \in \Gamma$.*
- *if φ, $\psi \in \Gamma$ then $\varphi \vee \psi$, $\neg\varphi \in \Gamma$.*
- *if $\varphi \in \Phi$ then $E\varphi \in \Gamma$.*

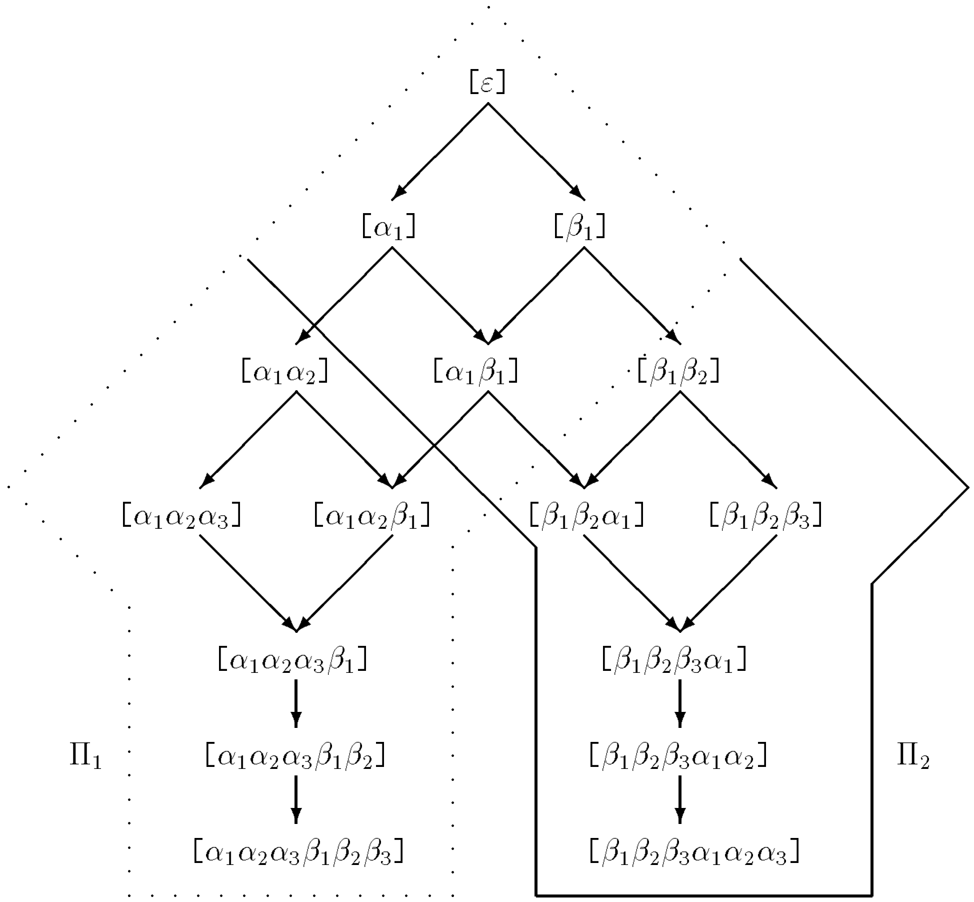

Figure 2: Runs of Net A.

Path Formulas (Φ) $\left\{ \begin{array}{ll} \bullet & \textit{if } \varphi \in \Gamma \textit{ then } \varphi \in \Phi. \\ \bullet & \textit{if } \varphi,\ \psi \in \Phi \textit{ then } \varphi \vee \psi,\ \neg\varphi \in \Phi. \\ \bullet & \textit{if } \varphi,\ \psi \in \Phi \textit{ then } (\varphi \mathcal{U} \psi),\ (\varphi \mathcal{S} \psi),\ X\varphi,\ Y\varphi \in \Phi. \end{array} \right.$

Let Q be a set of elements to which we refer as *states*. If ξ is a sequence of states $q_0\, q_1\, q_2 \ldots$, let $|\xi|$ be the length of ξ (ω if ξ is infinite), and $\xi(i)$ ($0 \leq i < |\xi|$) is the i^{th} element.

Definition 3.2 *An* **ISTL*** structure $\mathcal{M}$ *is a quadruple* $\langle \mathcal{A}, Q, \hat{q}, \Xi \rangle$ *where* $\mathcal{A}$ *is a first order structure,* Q *is a set of states,* $\hat{q}$ *is some distinguished state of* Q*,* Ξ *is a set of sequences of states starting at* $\hat{q}$*. Define a* labeling *function* L *that corresponds to a structure* $\mathcal{M}$ *over a set of variables* V *to be a function over the domain* Q *such that for each state* $q \in Q$*,* $L(q)$ *is an assignment over the set of variables* V*.*

Definition 3.3 Semantics of an ISTL* formula under a structure $\mathcal{M}$ and a corresponding labeling function L.

Satisfaction of a state formula under a state $q \in Q$*:*

- $q \models \mathbf{T}$
- $q \models \varphi \iff L(q) \models^{\mathcal{A}} \varphi$, *for φ a first order formula.*
- $q \models \varphi \vee \psi \iff q \models \varphi$ *or* $q \models \psi$.
- $q \models \neg\varphi \iff$ *not* $q \models \varphi$.
- $q \models E\varphi \iff$ *there exists a sequence $\xi \in \Xi$ and an integer $0 \le i < |\xi|$ with $\xi(i) = q$, such that $(\xi, i) \models \varphi$.*

Satisfaction of a path formula under a sequence $\xi \in \Xi$ and an integer $0 \le i < |\xi|$:

- $(\xi, i) \models \varphi \iff \xi(i) \models \varphi$, *if φ is a path formula which is also a state formula.*
- $(\xi, i) \models \varphi \vee \psi \iff (\xi, i) \models \varphi$ *or* $(\xi, i) \models \psi$.
- $(\xi, i) \models \neg\varphi \iff$ *not* $(\xi, i) \models \varphi$.
- $(\xi, i) \models (\varphi \mathcal{U} \psi) \iff$ *there exists $i \le j < |\xi|$ such that $(\xi, j) \models \psi$, and for every k, such that $i \le k < j$, $(\xi, k) \models \varphi$.*
- $(\xi, i) \models (\varphi \mathcal{S} \psi) \iff$ *there exists $0 \le j \le i$ such that $(\xi, j) \models \psi$, and for every k such that $j < k \le i$, $(\xi, k) \models \varphi$.*
- $(\xi, i) \models X\varphi \iff |\xi| > i+1$ *and* $(\xi, i+1) \models \varphi$.
- $(\xi, i) \models Y\varphi \iff i > 0$ *and* $(\xi, i-1) \models \varphi$.

Abbreviations: $\mathbf{F} = \neg\mathbf{T}$, $\varphi \wedge \psi = \neg((\neg\varphi) \vee (\neg\psi))$, $\varphi \rightarrow \psi = (\neg\varphi) \vee \psi$, $\varphi \leftrightarrow \psi = \varphi \rightarrow \psi \wedge \psi \rightarrow \varphi$, $A\varphi = \neg E(\neg\varphi)$, $F\varphi = (\mathbf{T}\mathcal{U}\varphi)$, $G\varphi = \neg(\mathbf{T}\mathcal{U}(\neg\varphi))$, $P\varphi = (\mathbf{T}\mathcal{S}\varphi)$, $H\varphi = \neg(\mathbf{T}\mathcal{S}(\neg\varphi))$. A useful notation in the *entailment* operator [30] "$\Rightarrow$", where $\varphi \Rightarrow \psi = AG(\varphi \rightarrow \psi)$. Similarly, $\varphi \Leftrightarrow \psi = AG(\varphi \leftrightarrow \psi)$.

Definition 3.4 *For every $\varphi \in \Gamma$, a structure $\mathcal{M} = \langle \mathcal{A}, Q, \hat{q}, \Xi \rangle$ and a corresponding labeling function L, $\langle \mathcal{M}, L \rangle \models \varphi$ iff $\hat{q} \models \varphi$. If $\langle \mathcal{M}, L \rangle \models \varphi$ for every pair of a structure $\mathcal{M}$ and a corresponding labeling function L, it is said that φ is* valid.

For example, $\langle \mathcal{M}, L \rangle \models EGF\psi$ asserts that there exists a sequence in which ψ holds infinitely often.

Let P be a program. Then, each run $\Pi \in R_P$ defines an **ISTL*** structure $\mathcal{M} = \langle \mathcal{A}, \Pi, [\varepsilon], intset(\Pi) \rangle$ such that $\mathcal{A}$ is the first order structure which includes the program's relation and function symbols, and a corresponding labeling function L such that for each $\sigma \in \Pi$, $L(\sigma) = fin_\sigma$.

Definition 3.5 *An* **ISTL*** *formula φ is said to be* valid over a program P *if it holds over all the pairs of structures and corresponding assignments that are defined by the runs of P. In this case, we also write $P \models \varphi$.*

4 Motivation and Intuition for the Proof Rules

The logic **ISTL*** extends the properties of concurrent programs that can be expressed using Linear Temporal Logic [30]. This section points out the obstacles which arise when formulating a relatively complete proof system for some of the new properties. Understanding the problems helps clarify the proof rules introduced in the next section. It also adds insight on what is involved in proving properties of traces and partial order semantics. Using some examples, we hope to demonstrate the intuition behind the proof rules.

In the logic **ISTL*** , $\varphi \Rightarrow EX\psi$ is valid in a program P if for every run $\Pi \in R_P$, for every trace $\sigma \in \Pi$ satisfying φ, there exists an operation $\tau \in T$ such that $\sigma[\tau] \in \Pi$ and $\sigma[\tau]$ satisfies ψ. It is said that ψ *holds immediately after* φ. The temporal assertion $\varphi \Rightarrow EF\psi$ means that in every run Π, for every trace $\sigma \in \Pi$ satisfying φ, there exists a trace $\rho \in \Pi$ subsuming σ and satisfying ψ. Thus,

ψ *holds sometimes after* φ. We call these properties partial order eventuality properties. Similarly, the formulas $\varphi \Rightarrow EY\psi$ and $\varphi \Rightarrow EP\psi$ mean ψ *holds immediately before* φ, and ψ *holds sometimes before* φ, respectively.

Properties of the form $\varphi \Rightarrow AF\psi$ have the meaning "for each observation on which there exists a trace satisfying φ, there exists a later trace satisfying ψ". Such properties can be expressed in Linear Temporal Logic by the formula $\varphi \Rightarrow \Diamond\psi$. The proof rules in [29, 30] can be exploited to prove such properties. Notice that Theorem 2.1 implies that the set of observations are exactly the interleaving sequences satisfying the *justice* [25] property that no operation is ignored.

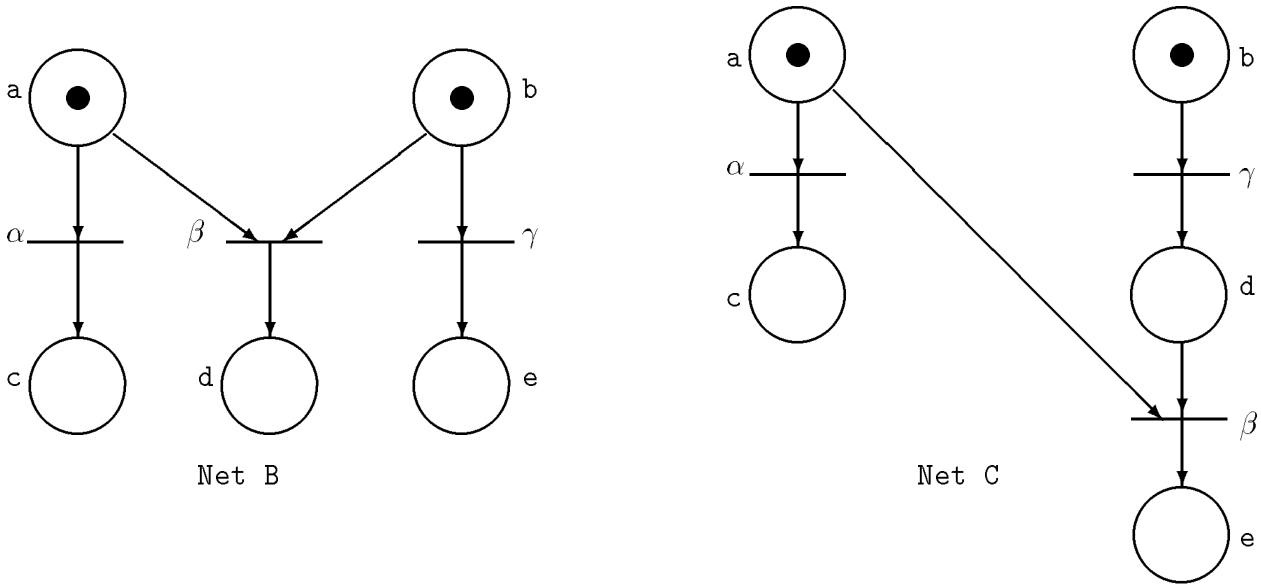

Figure 3: Nets B and C.

We deal first with proving properties of the form $\varphi \Rightarrow EX\psi$. In Net B (Figure 3) $D = Id_T \cup \{(\alpha, \beta), (\beta, \alpha), (\gamma, \beta), (\beta, \gamma)\}$. There are two runs: Π_β in which β is executed and $\Pi_{\neg\beta}$ in which α and γ are executed concurrently. In Net C (Figure 3) the dependency relation D is the same (that is, β is dependent on each one of the other two operations). In this net there are also two runs: Π_α in which α and γ are executed concurrently, and $\Pi_{\neg\alpha}$ in which β is executed after γ.

Let Θ (the initial condition) be $\{a, b\}$, for nets B and C, and also for Net D depicted in Figure 5. The following assertions are true for Net B.

$$\Theta \Rightarrow EX(\{b, c\} \vee \{d\}) \tag{1}$$

$$\Theta \Rightarrow EX(\{a, e\} \vee \{d\}) \tag{2}$$

$$\Theta \Rightarrow EX(\{b, c\} \vee \{d\} \vee \{a, e\}) \tag{3}$$

Notice that both assertions (1) (either α or β is executed) and (2) (either γ or β is executed) are stronger than (3) (either α or β or γ is executed). The assertion $\Theta \Rightarrow EX(\{c, e\} \vee \{d\})$ (either β is executed, or α **and** γ are executed) does not hold because the concurrent execution of α and γ cannot occur in one step.

For Net C in Figure 3, it holds that

$$\Theta \Rightarrow EX\{a, d\} \tag{4}$$

That is, γ is executed. On the other hand, the assertion $\Theta \Rightarrow EX\{b,\ c\}$ that asserts that α is executed in every run, does not hold (in the run $\Pi_{\neg\alpha}$, in which β is executed, followed by γ).

Thus, for each trace σ and each run Π containing σ, there is a set of operations executed immediately (and concurrently with each other) after σ. Similarly, there exists a set of operations executed immediately before σ. Notice that the operations that generate the successors of σ can vary from one run to another, while the operations executed last in σ do not depend on the run. This motivates the following definitions:

Definition 4.1 *Let Π be a run of P and $\sigma \in \Pi$. Define $succop_P(\Pi,\ \sigma) = \{\tau \mid \tau \in T \wedge \sigma[\tau] \in \Pi\}$ (i.e., the set of all operations executed in Π immediately after σ).*

Definition 4.2 *Let σ be a trace of P. Define $predop_P(\sigma) = \{\tau \mid \tau \in T \wedge \exists\rho\,(\sigma = \rho[\tau])\}$. (i.e., the set of all operations executed last in σ).*

For example, $succop_B(\Pi_\beta, [\varepsilon]) = \{\beta\}$, $succop_B(\Pi_{\neg\beta}, [\varepsilon]) = \{\alpha,\ \gamma\}$, $succop_C(\Pi_\alpha, [\varepsilon]) = \{\alpha,\ \gamma\}$, $succop_C(\Pi_{\neg\alpha}, [\varepsilon]) = \{\gamma\}$, $predop_B([\alpha\gamma]) = \{\alpha,\ \gamma\}$, $predop_C([\gamma\beta]) = \{\beta\}$. Notice that if $\tau_1,\ \tau_2 \in succop_P(\Pi,\ \sigma)$ (or $\tau_1,\ \tau_2 \in predop_P(\sigma)$), then $(\tau_1,\ \tau_2) \notin D$. The subscript P is omitted when clear from the context.

It is obvious that in order that the formula $\varphi \Rightarrow EX\psi$ will hold in a program, for each $\sigma \models \varphi$ of some $\Pi \in R_P$ there must exists some operation $\alpha \in succop_P(\Pi,\ \sigma)$ such that $\sigma[\alpha] \models \psi$. If $\varphi \Rightarrow EY\psi$ holds, then for each trace $\sigma \models \psi$, there exists a trace $\rho \models \psi$ and an operation $\alpha \in predop_P(\sigma)$ such that $\rho[\alpha] = \sigma$.

Definition 4.3 *A set $\mathcal{T} \subseteq T$ is called a* forward intercepting set for *φ in P if for each $\sigma \models \varphi$, for each run $\Pi \in R_P$ that contains σ, $succop_P(\Pi,\ \sigma) \cap \mathcal{T} \neq \phi$. Denote by $\varphi \searrow \mathcal{T}$ the fact that $\mathcal{T}$ is a forward intercepting set for φ. The negation of $\varphi \searrow \mathcal{T}$ is denoted by $\varphi \not\searrow \mathcal{T}$.*

Definition 4.4 *A set $\mathcal{T} \subseteq T$ is called a* backward intercepting set for *φ in P if for each $\sigma \models \varphi$, $predop_P(\sigma) \cap \mathcal{T} \neq \phi$. Denote by $\mathcal{T} \nwarrow \varphi$ the fact that $\mathcal{T}$ is a backward intercepting set for φ, and its negation by $\mathcal{T} \not\nwarrow \varphi$.*

In Net B of Figure 3, the assertions (1), (2) and (3) correspond to identifying $\{\alpha,\ \beta\}$, $\{\gamma,\ \beta\}$ and $\{\alpha,\ \beta,\ \gamma\}$ as the forward intercepting sets for Θ in B, respectively. The assertion (4) corresponds to the singleton $\{\gamma\}$ as a forward intercepting set for Θ in Net C. Hence, for proving $\varphi \Rightarrow EX\psi$ one needs (a) to find a forward intercepting set for the traces satisfying φ and (b) show that all the operations in this set transform φ to ψ. It might happen that not all the traces satisfying φ share such a forward intercepting set. In this case, it is possible to decompose the traces satisfying φ into subsets. That is, show $\varphi \Rightarrow (\varphi_1 \vee \ldots \vee \varphi_n)$. Then, find for each φ_i, $1 \leq i \leq n$ a forward intercepting set and use it to prove $\varphi_i \Rightarrow EX\psi$. Finding a forward intercepting set can rely on the following Lemma:

Lemma 4.5 *$\varphi \searrow \mathcal{T}$ iff for each $\sigma \models \varphi$, for each string $v \in T^*$ that can extend σ (i.e., $\sigma[v]$ is a trace of P), either*

1. *there exists some $\alpha \in \mathcal{T}$ such that $\sigma \models en_\alpha$ and for each $\beta \in T$ that appears in v, $(\alpha,\ \beta) \notin D$, or*
2. *$v = \bar{u}\alpha\hat{u}$, where $\bar{u},\ \hat{u} \in T^*$, $\alpha \in \mathcal{T}$, and all the operations of $\bar{u}$ are independent of α (hence, α can be permuted to occur immediately after σ).*

Proof. $\Longrightarrow$. Assume that $\varphi \searrow \mathcal{T}$. Assume for the contrary that there exists a trace $\sigma \models \varphi$, and a string v that can extend σ that does not satisfy the above two conditions. The trace $\sigma[v]$ belongs to some run $\Pi \in R_P$. We will show that for any $\alpha \in \mathcal{T}$ that $\sigma[\alpha] \notin \Pi$.

Assume that $\sigma[\alpha] \in \Pi$ for some $\alpha \in \mathcal{T}$. However, from the negation of the two conditions above, there exists some operation $\beta \in T$, $(\alpha, \beta) \in D$ that occurs in v, and if α also occurs in v, then β occurs before α (i.e., to the left of it). Then $v = \bar{w}\beta\hat{w}$, with $\bar{w}$ containing only operations independent of α. Then, $\sigma[\bar{w}\alpha] \in \Pi$, since $\sigma[\bar{w}\alpha]$ is the minimal trace that subsumes both $\sigma[\alpha]$ and $\sigma[\bar{w}]$, and $\sigma[\bar{w}\beta] \in \Pi$ since $\sigma[\bar{w}\beta] \sqsubseteq \sigma[v]$. However, since $(\alpha, \beta) \in D$, there cannot be a trace subsumming both $\sigma[\bar{w}\alpha]$ and $\sigma[\bar{w}\beta]$, contradictory to the fact that Π must be directed.

$\Longleftarrow$. Assume that at least one of the conditions of the Lemma holds. Choose a run Π that contains an observation o on which some trace σ that satisfies φ appears. Assume that the second condition holds for some string v in the equivalence class ρ/σ, where ρ appears in o after σ. This means that any string v in ρ/σ can be written as $\bar{u}\alpha\hat{u}$, with α independent of all the operations that occur in $\bar{u}$. Thus, $\sigma[\alpha] \sqsubseteq \sigma[v] = \rho$, and hence $\sigma[\alpha] \in \Pi$.

It remains to show that there always exist a trace ρ after σ in o and a string v in ρ/σ, such that the second condition holds. Assume for the contrary that for each such string v only the first condition holds. Notice that if this condition holds for v with some operation α, then it holds for any v' in ρ'/σ, where $\sigma \sqsubseteq \rho' \sqsubseteq \rho$ with the same operation α. Since $\mathcal{T}$ is finite, there exists some fixed α such that for every such string v the first condition holds with α. However, this means that o ignores α from σ, which by Theorem 2.1 contradicts the choice of o as an observation. ∎

When proving $\varphi \Rightarrow EY\psi$, it may not be enough to find a backward intercepting set $\mathcal{T}$ for φ such that the predecessor states generated by the operations in $\mathcal{T}$ satisfy ψ. This stems from the fact that the converse of an operation's transformation does not have to be a function. Hence, it is possible to have two traces σ and ρ, such that $\sigma[\tau]$ and $\rho[\tau]$ have identical final assignment, satisfying φ, yet σ may satisfy ψ while ρ does not. This means that it is not sufficient to find backward intercepting sets for φ.

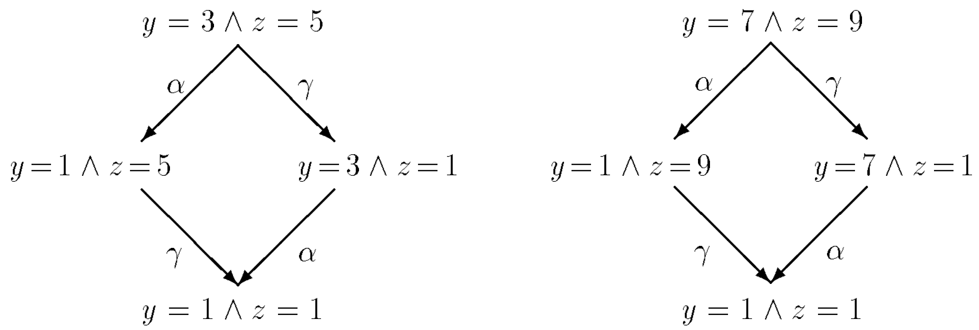

Figure 4: Two runs of $\{\alpha, \gamma\}$.

Example. To illustrate this point, let P be a program containing two independent operations $\alpha = \langle y \geq 2 \longrightarrow y := y \bmod 2\rangle$ and $\gamma = \langle z \geq 2 \longrightarrow z := z \bmod 2\rangle$. Let Θ be $(y = 3 \wedge z = 5) \vee (y = 7 \wedge z = 9)$. Thus, there exist two runs, as depicted in Figure 4. Consider the following assertion:

$$(y = 1 \wedge z = 1) \Rightarrow EY((y = 3 \wedge z = 1) \vee (y = 1 \wedge z = 9)) \tag{5}$$

Obviously, the program P always terminates with $y = 1$, $z = 1$ after both α and γ were executed. A trace $[\gamma]$ with final assignment $y = 3$, $z = 1$ exists in the first run (initiated with $y = 3$, $z = 5$). A trace $[\alpha]$ with final assignment $y = 1$, $z = 9$ exists in the second run. Thus, (5) holds for P. Now observe that the backward intercepting sets for $y = 1 \wedge z = 1$ are $\{\{\alpha\}, \{\gamma\}, \{\alpha, \gamma\}\}$. However, it is not the case that

$$wp_\tau(y = 1 \wedge z = 1) \Rightarrow ((y = 3 \wedge z = 1) \vee (y = 1 \wedge z = 9)) \tag{6}$$

where $\tau = \alpha$ or $\tau = \gamma$. For $\tau = \alpha$, the trace $[\gamma]$ with final assignment $y = 7$, $z = 1$ is a counter example for (6), while for $\tau = \gamma$, the trace $[\alpha]$ with $y = 1$, $z = 5$ falsifies (6). That is, in one run, the operation α is responsible for the validity of (5), while in the other run, it is γ. ▪

The solution is to augment the assignments of the program with additional variables that carry information about the history, separating between different traces that formerly had the same final assignment. In the above example, we can augment the assignments with two variables y' and z', holding the initial values of y and z. As a consequence, the final assignments of the two traces that are denoted by $[\alpha\gamma]$ (there are two such different traces with different initial assignment, each belongs to a different run) become different. Then,

$$(y = 1 \wedge z = 1) \Rightarrow ((y = 1 \wedge z = 1 \wedge y' = 3 \wedge z' = 5) \vee (y = 1 \wedge z = 1 \wedge y' = 7 \wedge z' = 9)) \tag{7}$$

$$wp_\alpha(y = 1 \wedge z = 1 \wedge y' = 3 \wedge z' = 5) \Rightarrow (y = 3 \wedge z = 1 \wedge y' = 3 \wedge z' = 5) \tag{8}$$

$$wp_\gamma(y = 1 \wedge z = 1 \wedge y' = 7 \wedge z' = 9) \Rightarrow (y = 1 \wedge z = 9 \wedge y' = 7 \wedge z' = 9) \tag{9}$$

Thus, (5) can be proven from (7), (8) and (9) by choosing $\{\alpha\}$ as a backward intercepting set for $y = 1 \wedge z = 1 \wedge y' = 3 \wedge z' = 5$ and $\{\gamma\}$ as a backward intercepting set for $y = 1 \wedge z = 1 \wedge y' = 7 \wedge z' = 9$.

Augmenting a trace with information about its history is done by adding to the program a set of auxiliary variables [6, 36]. That is, variables which do not change the behavior of the program, but merely collect information about the history. (We could have provided the machinery to assert about the initial assignment and the set of equivalent sequences of a trace by defining a distinguished set of variables that hold these values. As this would complicate the notation, we use instead the familiar construction of history variables to allow the extra expressiveness over traces only when needed).

Definition 4.6 *A program $P' = \langle \Theta', T', D', \vec{z} \rangle$ is an* augmentation *of $P = \langle \Theta, T, D, \vec{y} \rangle$, denoted $P \angle P'$ if the following conditions hold:*

1. *The sets of variables satisfy $\vec{z} \supseteq \vec{y}$. Denote $\vec{z} \setminus \vec{y} = \{z'_1, z'_2, \ldots, z'_m\}$. The variables $\vec{z} \setminus \vec{y}$ are called* auxiliary variable.
2. *$\Theta' = \Theta \wedge \theta$ where $\Theta \rightarrow \exists z'_1 \exists z'_2 \ldots \exists z'_m \, \theta$.*
3. *There is a bijection $\mathcal{I} : T \mapsto T'$ such that*
   - *$en_\tau = en_{\mathcal{I}(\tau)}$.*
   - *$f_{\mathcal{I}(\tau)}$ is a tuple of $\mid \vec{z} \mid$ terms, where the terms assigned to the original variables $\vec{y}$ are identical to the terms f_τ.*
4. *For each pair of operations $\alpha, \beta \in T$, $(\alpha, \beta) \in D$ iff $(\mathcal{I}(\alpha), \mathcal{I}(\beta)) \in D'$.*
5. *For each pair of operations $\alpha, \beta \in T'$ such that $(\alpha, \beta) \notin D'$,* $\forall \vec{y}(en_\alpha(\vec{y}) \wedge en_\beta(\vec{y})) \rightarrow f_\alpha(f_\beta(\vec{y})) = f_\beta(f_\alpha(\vec{y}))$. *(The other requirement from D is entailed from conditions 3 and 4.)*

It is allowed to prove temporal assertions where auxiliary variables do not occur free by using the augmentation P' instead of P. This is justified by the following Lemma.

Lemma 4.7 *Let $J\lceil\vec{y}$ be the restriction of J to the variables $\vec{y}$. Then, for a program P and its augmentation P', $\Pi \in R_P$ iff there exists a run $\Pi' \in R_{P'}$ such that the function $m : \Pi' \mapsto \Pi$ defined as $m([I, w]) = [I\lceil\vec{y}, w]$ is a bijection. A similar correspondence exists between observations of P and P'.*

Proof. Using simple inductions on the length of traces and interleaving sequences. ∎

Completeness proofs (unlike most actual proofs), often make use of the ability to record the entire history using auxiliary variables. This is traditionally done [36, 1] in arithmetical models by encoding finite sequences into integers [44].

It is possible to obtain all the information needed on the history of a trace by saving the initial values of the variables (this is done by adding for each $y_i \in \vec{y}$ an auxiliary variable z_i' and setting a conjunct of θ to be $z_1' = y_1 \wedge \ldots \wedge z_n' = y_n$) and the equivalence class of the string of operations occurred. In verification methods based on interleaving semantics, it is possible to collect the history of the computation by using an auxiliary variable z_h (besides the variables $z_1, \ldots, z_n$), initialized to the empty string ε. With any operation τ executed, the symbol τ is appended to the end of z_h (i.e., $z_h := z_h\,\tau$). However, commutativity among independent operations is violated this way, and therefore inappropriate for use with traces. Instead, one can use a function $min(v)$ that gives the minimal string v' under lexicographic order that is equivalent to v (up to commuting adjacent independent operations). Then, the assignment $z_h := min(z_h\,\tau)$ is added to each operation τ.

Next, let us consider proving properties of the form $\varphi \Rightarrow EF\psi$. It may seem at first, that it is possible to establish such a proof by an inductive argument on $\varphi \Rightarrow EX\psi$. Albeit this is sufficient in many cases, there are cases in which this is not possible. To exemplify such a situation, consider again Net A in Figure 1 (Section 2) and its two runs (Figure 2). In the following, we somewhat abuse our notation by allowing traces to stand for state assertions.

Assume one wants to prove that Net A of Figure 1 satisfies $[\varepsilon] \Rightarrow EF([\alpha_1\alpha_2\alpha_3] \vee [\beta_1\beta_2\beta_3])$. The trace $[\varepsilon]$ has two successors: $[\alpha_1]$ and $[\beta_1]$. At least one of $\{\alpha_1, \beta_1\}$ must be contained in every forward intercepting set to generate the successor of $[\varepsilon]$. Without loss of generality, assume $[\varepsilon] \Rightarrow EX[\alpha_1]$ was proved. Now, the forward intercepting sets of $[\alpha_1]$ are $\{\beta_1\}$ and $\{\alpha_2, \beta_1\}$. So the trace $[\alpha_1\beta_1]$ must be considered as a successor of the trace $[\alpha_1]$. However, the trace $[\alpha_1\beta_1]$ no longer satisfy $[\alpha_1\beta_1] \Rightarrow EF([\alpha_1\alpha_2\alpha_3] \vee [\beta_1\beta_2\beta_3])$.

The intuition behind "what went wrong" is as follows: The trace $[\alpha_1]$ is intermediate between $[\varepsilon]$ and $[\alpha_1\alpha_2\alpha_3]$ (both belong to Π_1). However, $[\alpha_1]$ also belongs to Π_2 and is not an intermediate trace between $[\varepsilon]$ and $[\beta_1\beta_2\beta_3]$ (which belong to Π_2). (A symmetric argument holds when exchanging the α's with the β's.) At the initial state, the two runs have the same enabled transitions, and it is only after executing either α_2 or β_2 that the two runs behave differently.

The solution for the above difficulty lies in using a forward—backward method. Define a *path of traces* to be a sequence $\sigma_1\,\sigma_2\,\ldots$ of length n ($n = \omega$ if the sequence is infinite) where for each $0 \leq i < n$ either σ_{i+1} is a successor of σ_i for a *forward path*, or σ_{i+1} is a predecessor of σ_i for a *backward path*. Thus, the forward–backward method is based upon showing that there exists a forward path from each trace σ satisfying φ ending with some trace ρ, and then a backward path starting with ρ and ending with a trace that subsumes σ and satisfies ψ.

This is done by choosing some $\hat{\psi}$ such that (1) $\hat{\psi}$ is satisfied by traces subsuming traces that satisfy φ, and (2) if σ is an intermediate trace between a trace that satisfies φ and a trace that satisfies $\hat{\psi}$ in some $\Pi \in R_P$, then it is an intermediate trace between such traces satisfying φ and $\hat{\psi}$ in every run $\Pi' \in R_P$ that contains σ. For Net A, $\hat{\psi} = ([\alpha_1\alpha_2\alpha_3\beta_1] \vee [\beta_1\beta_2\beta_3\alpha_1])$ can be used.

A naive attempt to verify $\varphi \Rightarrow EF\psi$ would be to prove $\varphi \Rightarrow EF\hat{\psi}$, and $\hat{\psi} \Rightarrow EP\psi$. However,

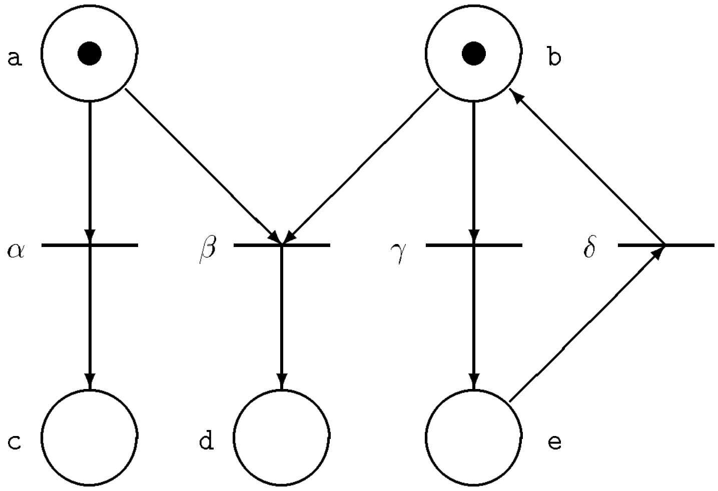

Figure 5: Net D.

unless $\varphi \Leftrightarrow \Theta$, proving $\varphi \Rightarrow EF\hat{\psi}$ and $\hat{\psi} \Rightarrow EP\psi$ guarantees only $\varphi \Rightarrow EF\,EP\psi$. This is true because the backward path from a $\hat{\psi}$–trace (i.e., a trace satisfying $\hat{\psi}$) to a ψ-trace is not restricted to reach a trace that subsumes the origin trace of the forwards path. To enforce this restriction, proving $\hat{\psi} \Rightarrow EP\psi$ is not done directly for the program P itself, but for a variant P_{φ} of P. The program P_{φ} is the same as P except that the initial assignments of P_{φ} are the final assignment of the traces of P that satisfy φ. When proving $P_{\varphi} \vdash \hat{\psi} \Rightarrow EP\psi$, the invariants of P_{φ} (which are stronger than those of P) are used, and in addition φ is used instead of Θ when the initial is required within a proof rule. P_{φ} is defined as follows:

Definition 4.8 *Let ν be a first order formula, and $P = \langle T, D, \Theta, \vec{y}\rangle$ be a program. Let*

$$\Lambda_{\nu} = \{\sigma \mid \sigma \models \nu \bigwedge \exists \Pi \in R_P\, \sigma \in \Pi\}.$$

*Define Θ_{ν} as the first order formula describing (i.e., satisfied exactly by) the traces Λ_{ν}. We define the ν–*variant *of the program P to be the program $P_{\nu} = \langle T, D, \Theta_{\nu}, \vec{y}\rangle$.*

Advanced comment. Notice that it is not always true that $\Theta_{\nu} \leftrightarrow \nu$, because the original program P begins with some initial condition Θ. This might limit the set of accessible traces satisfying ν, resulting in that Θ_{ν} is stronger than ν. However, it is not required that the underlying first order logic is expressive enough to formulate Θ_{ν} because Θ_{ν} is not used explicitly in the rules. (As remarked earlier, it is sufficient to use φ when the initial condition of P_{φ} is needed.) ▗

Another problem stems from the fact that paths which are not observations may generate infinite sequence of intermediate traces between φ and ψ. Consider Net D in Figure 5. This has infinitely many runs. One of the runs consists of α being executed concurrently with (independently of) an infinite sequence of $\gamma\delta$. All the other runs are finite with $\gamma\delta$ occurring a finite number of times followed by an execution of β.

Assume one wants to prove that $\Theta \Rightarrow EF(\{c, b\} \vee \{d\})$. Then, the traces with final assignments $\{a, b\}$ and the traces with final assignment $\{a, e\}$ are transformed one to the other (notice that γ is in every forward intercepting set of $\{a, b\}$ and δ is in every forward intercepting set of $\{a, e\}$). Hence, there is an infinite sequence (which is not an observation) of intermediate traces. Such a sequence defies using well founded induction, because progress is never made.

The solution for this problem is to use a proof rule which selects only the observations instead of the set of all the paths. If $\varphi \Rightarrow EF\psi$ holds in a program, then each observation that contains a trace $\sigma \models \varphi$ has some trace ρ' that subsumes some $\rho \models \psi$, where $\sigma \sqsubseteq \rho \sqsubseteq \rho'$. Thus, $\hat{\psi}$ can be defined as a predicate satisfied exactly by such traces ρ'. Notice that separating the traces satisfying φ to several cases might be necessary. Otherwise, it might happen that for another trace $\sigma' \models \varphi$, $\sigma' \sqsubseteq \rho'$, but no intermediate trace between σ' and ρ' satisfies ψ. (In this case, $P_\varphi \not\models \hat{\psi} \Rightarrow EP\psi$.) Thus, the traces σ and σ' must be handled separately. For Net D, we can choose $\hat{\psi} = \{d\} \vee \{c,\, b\} \vee \{c,\, e\}$.

Now, a trace satisfying $\hat{\psi}$ exists on each observation rather on some observations. To treat only observations, one can use proof rules for $\varphi \Rightarrow AF\hat{\psi}$ that deal with (weak) fairness, as those in [14, 16, 13]. This implies that the weaker property $\varphi \Rightarrow EF\hat{\psi}$ holds. To complete the proof, we establish, as before, $P_\varphi \vdash \hat{\psi} \Rightarrow EP\psi$.

5 A Complete Proof System for Safety and Eventuality Properties

In this section, a complete proof system is presented for some important properties expressible in the logic **ISTL***. The proof system is based upon classical (non–temporal) reasoning in first order logic. Soundness and completeness is proved in the next section. The symbols φ, ψ, η and ν denote first order formulas while κ stands for a temporal formula. We denote the fact that it is possible to prove (in the proof system given herein) that κ holds in a program P by $P \vdash \kappa$.

5.1 Basic Proof Rules

Rule 1 TAUT: This rule allows using each first order formula of $\mathcal{L}$ that holds in the structure $\mathcal{A}$ (over which the program's relation and function symbols are interpreted) to be used as an invariant of the program.

$$\frac{\models^{\mathcal{A}} \varphi}{AG\varphi}$$

Rule 2 INV: This rule is used to prove that an assertion η is an invariant of a program P with initial condition Θ. The case where η is an implication is useful as a premise in other rules when entailment is expected (although in many cases one can use the simpler rule TAUT). The premises of this rule can be proved using TAUT (recall that a formula of the form $\varphi \Rightarrow \psi$ is a shorthand for $AG(\varphi \rightarrow \psi)$).

$$\begin{array}{ll} \text{I1} & \Theta \Rightarrow \varphi \\ \text{I2} & (\varphi \wedge en_\tau) \Rightarrow wp_\tau(\varphi), \text{ for each } \tau \in T \\ \text{I3} & \varphi \Rightarrow \eta \\ \hline & AG\eta \end{array}$$

Rule 3 FCS and BCS: The rule FCS is used to partition the set of traces which satisfy the assertion φ into two sets, one set satisfying φ_1 and the other satisfying φ_2. Then, proving $\varphi_i \Rightarrow EX\psi$ is done separately for the traces satisfying φ_1 and the traces satisfying φ_2. To obtain the effect of generalizing the rules for arbitrary number of formulas φ_i, the rules can be applied repeatedly. Rule BCS is the past version of rule FCS.

$$\textbf{FCS}\quad \frac{\begin{array}{l}\varphi \Rightarrow (\varphi_1 \vee \varphi_2)\\ \varphi_1 \Rightarrow EX\psi\\ \varphi_2 \Rightarrow EX\psi\end{array}}{\varphi \Rightarrow EX\psi} \qquad\qquad \textbf{BCS}\quad \frac{\begin{array}{l}\varphi \Rightarrow (\varphi_1 \vee \varphi_2)\\ \varphi_1 \Rightarrow EY\psi\\ \varphi_2 \Rightarrow EY\psi\end{array}}{\varphi \Rightarrow EY\psi}$$

Rule 4 **NEXT:** Let $\mathcal{T} \subseteq T$ be a non–empty set. Find a directed graph G with set of nodes $g_i = \langle \varphi_i, \mathcal{T}_i \rangle$ where $i = 0, \ldots, n$ for some natural n, such that each φ_i is a first order formula, and $\mathcal{T}_i \subseteq \mathcal{T}$. For each pair of distinct nodes g_i, g_j, it holds that $\mathcal{T}_i \neq \mathcal{T}_j$. An empty set $\mathcal{T}_i = \phi$ is allowed only when $\varphi_i = \mathbf{F}$. The edges of G are labeled with single operations from T. For each node g_i and each operation $\tau \in T \setminus \mathcal{T}_i$ there exists a unique edge $g_i \xrightarrow{\tau} g_j$, for some node g_j. No other edges leave g_i.

$$\frac{\begin{array}{ll}\textbf{N1} & g_0{=}\langle \varphi_0,\, \mathcal{T}_0 \rangle, \text{ s.t. } \varphi \Rightarrow \varphi_0 \text{ and } \mathcal{T}_0 = \mathcal{T}\\ \textbf{N2} & \text{If } g_i \xrightarrow{\tau} g_j \text{ then } (\varphi_i \wedge en_\tau) \Rightarrow wp_\tau(\varphi_j) \text{ and } \mathcal{T}_j{=}\mathcal{T}_i \setminus \{\alpha \mid (\tau,\, \alpha) \in D\}\\ \textbf{N3} & \varphi \Rightarrow wp_\tau(\psi), \text{ for each } \tau \in \mathcal{T}\end{array}}{\varphi \Rightarrow EX\psi}$$

Notice that the only cycles allowed in G are self loops, since from **N2**, if $g_i \xrightarrow{\tau} g_j$ is an edge in G, then $\mathcal{T}_i \supseteq \mathcal{T}_j$ and if $i \neq j$, then $\mathcal{T}_i \neq \mathcal{T}_j$. Intuitively, the rule **NEXT** proves $\varphi \Rightarrow EX\psi$ by showing that in every run Π with a trace σ satisfying φ, one of the successors of σ is obtained by executing one of the operations from $\mathcal{T}$, i.e., $\varphi \searrow \mathcal{T}$. It is shown that for each $v \in T^*$ which extends a trace σ that satisfies φ (i.e., $\sigma[v]$ is a trace of P), at least one of the conditions of Lemma 4.5 holds. This is proved using a simple induction on the length of v, by considering a path in G that starts with g_0 and whose sequence of edge labels correspond to v or a prefix of it. The details of this induction appear in the next section. Finally, it is required (by **N3**) that for each $\tau \in \mathcal{T}$, if φ holds before its execution, then ψ holds after.

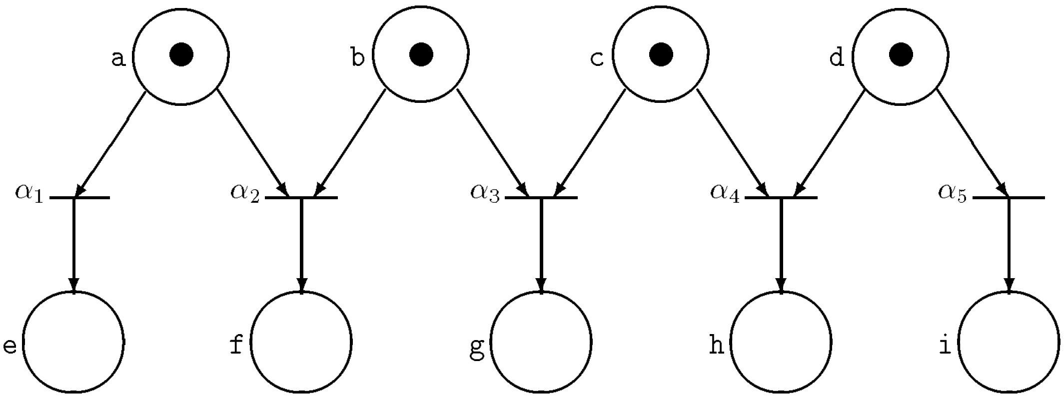

Figure 6: Net E.

Example. Consider the Net E presented in Figure 6. Each of the operations can be executed at most once. There are four possible runs where the following sets of operations are executed concurrently: $\{\alpha_1,\, \alpha_3,\, \alpha_5\}$, $\{\alpha_1,\, \alpha_4\}$, $\{\alpha_2,\, \alpha_5\}$ and $\{\alpha_2,\, \alpha_4\}$.

Obviously, at least one of $\{\alpha_1,\, \alpha_2\}$ appears in each run. Thus, $\{a,\, b,\, c,\, d\} \searrow \{\alpha_1,\, \alpha_2\}$. (This is not the only choice. For example, $\{\alpha_4,\, \alpha_5\}$ is also a forward intercepting set for $\{a,\, b,\, c,\, d\}$.) This can be formally proved using the graph depicted in Figure 7 (the self loop from the lower node s in the figure which have an empty set of operations to itself represents $|T|$ edges from s to s, each of which is labeled with an operation from T, i.e., the set of all the operations). Combining this with the last condition of **NEXT**, we obtain $\{a,\, b,\, c,\, d\} \Rightarrow EX(\{e,\, b,\, c,\, d\} \vee \{f,\, c,\, d\})$ ∎

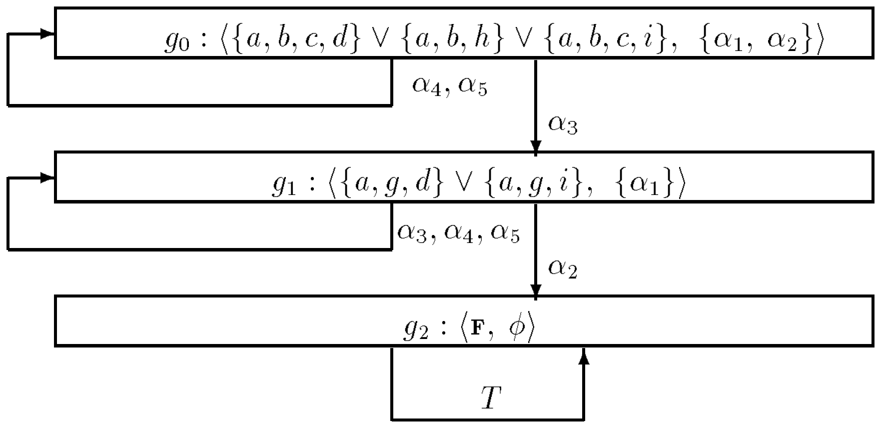

Figure 7: Graph for Net E.

Rule 5 PREV**:** Let $\mathcal{T} \subseteq T$ be a non–empty set. Find a directed graph G with set of nodes $g_i = \langle \varphi_i, \mathcal{T}_i\rangle$ where $i = 0, \ldots, n$ for some natural n, such that each φ_i is a first order formula, and $\mathcal{T}_i \subseteq \mathcal{T}$. For each pair of distinct nodes g_i, g_j, it holds that $\mathcal{T}_i \neq \mathcal{T}_j$. An empty set $\mathcal{T}_i = \phi$ is allowed only when $\varphi_i = \mathbf{F}$. The edges of G are labeled with single operations from T. For each node g_i and each operation $\tau \in T \setminus \mathcal{T}_i$ there exists a unique edge $g_i \xrightarrow{\tau} g_j$, for some node g_j. No other edges leave g_i.

$$\begin{array}{ll} \textbf{P1} & g_0 = \langle \varphi_0,\ \mathcal{T}_0\rangle \text{ s.t. } \varphi \Rightarrow \varphi_0 \text{ and } \mathcal{T}_0 = \mathcal{T} \\ \textbf{P2} & \text{If } g_i \xrightarrow{\tau} g_j, \text{ then } wp_\tau(\varphi_i) \Rightarrow \varphi_j \text{ and } \mathcal{T}_j = \mathcal{T}_i \setminus \{\alpha \mid (\tau,\ \alpha) \in D\} \\ \textbf{P3} & \Theta \Rightarrow \neg\varphi_i, \text{ for each node } g_i \\ \textbf{P4} & wp_\tau(\varphi) \Rightarrow \psi, \text{ for each } \tau \in \mathcal{T} \\ \hline & \varphi \Rightarrow EY\psi \end{array}$$

Rule 6 FIMM**:** This rule identifies $\varphi \Rightarrow \psi$ as a special case of $\varphi \Rightarrow EF\psi$.

$$\frac{\varphi \Rightarrow \psi}{\varphi \Rightarrow EF\psi}$$

Rule 7 CAUS**:** The rule CAUS is used to show that whenever φ holds in a trace, there exists a prefix of that trace satisfying ψ.

$$\begin{array}{ll} \textbf{C1} & \varphi \Rightarrow (\eta \vee \psi) \\ \textbf{C2} & \eta \Rightarrow EY(\eta \vee \psi) \\ \hline & \varphi \Rightarrow EP\psi \end{array}$$

Rule 8 FTRN**:** The rule FTRN allows proving $\varphi \Rightarrow EF\psi$ by finding first an assertion ν satisfied by intermediate traces between φ and ψ. Then, proving $\varphi \Rightarrow EF\nu$ and $\nu \Rightarrow EF\psi$.

$$\begin{array}{l} \varphi \Rightarrow EF\nu \\ \nu \Rightarrow EF\psi \\ \hline \varphi \Rightarrow EF\psi \end{array}$$

Rule 9 JWELL **[29, 31]:** Let $(\mathcal{W},\ \prec)$ be a well founded structure and ι a function from the state space of the program (the set of all possible assignments to its variables) to $\mathcal{W}$. Let $\Omega : \mathcal{W} \mapsto T$.

$$
\begin{array}{ll}
\textbf{J1} & \varphi \Rightarrow (\eta \vee \psi) \\
\textbf{J2} & \eta \Rightarrow (\iota \in \mathcal{W}) \\
\textbf{J3} & (\eta \wedge (\iota = k) \wedge en_\alpha) \Rightarrow wp_\alpha(\psi \vee (\eta \wedge \iota \preceq k)), \text{ for each } \alpha \in T \\
\textbf{J4} & (\eta \wedge (\Omega(\iota) = \alpha)) \Rightarrow en_\alpha, \text{ for each } \alpha \in T \\
\textbf{J5} & \text{for each } \alpha \in T, \text{ for each } \beta \text{ such that } (\alpha, \beta) \in D, \\
 & \quad (\eta \wedge (\iota = k) \wedge (\Omega(\iota) = \alpha) \wedge en_\beta) \Rightarrow wp_\beta(\psi \vee (\eta \wedge \iota \prec k)) \\
\hline
 & \varphi \Rightarrow AF\psi
\end{array}
$$

Premise **J1** states that a trace satisfying φ either also satisfies ψ or satisfies η. On any observation o, once η is satisfied by some trace σ, it continues to hold unless a trace satisfying ψ appears later in o (by **J3**). The function ι associates with every trace satisfying η a value from $\mathcal{W}$ (by **J2**) and its value does not increase when moving from an η–trace to a successor $\neg\psi$–trace (by **J3**). If σ is a trace of o satisfying η with $\iota = k$, then the operation $\Omega(k)$ is enabled in σ (by **J4**). If the value of ι does not decrease beyond σ and no trace satisfying ψ is reached, then no operation dependent on $\Omega(k)$ is taken beyond σ (by **J5**). Hence, the operation $\Omega(k)$ is ignored, a contradiction to the fact that o is an observation (see Theorem 2.1).

Rule 10 SMPF: This rule uses the fact that if a property holds on all the observations of a run, it holds on at least one of them.

$$
\frac{\varphi \Rightarrow AF\psi}{\varphi \Rightarrow EF\psi}
$$

Rule 11 FRBK: This rule allows proving that $\nu \Rightarrow EF\hat{\psi}$ holds by a forward–backward reasoning.

$$
\begin{array}{ll}
\textbf{F1} & \nu \Rightarrow EF\hat{\psi} \\
\textbf{F2} & P_\nu \vdash \hat{\psi} \Rightarrow EP\psi \\
\hline
 & \nu \Rightarrow EF\psi
\end{array}
$$

Premise **F1** of the rule asserts that for each trace σ of P such that $\sigma \models \nu$, there exists some trace $\sigma\rho$ of P such that $\sigma\rho \models \hat{\psi}$. It follows that ρ is a trace of P_ν. Premise **F2** states that there is a prefix ρ' of ρ which satisfies ψ. Hence, $\sigma\rho'$ is a trace of P and satisfies ψ.

Premise **F2** of **FRBK** requires that $\hat{\psi}$ holds in the ν–variant of P_ν (see Definition 4.8). In order to prove $P_\nu \models \kappa$, where κ is any formula is any formula dealt with in the above proof rules, the following changes are made:

Premises of type $\varphi \Rightarrow \psi$ where φ and ψ are classical formulas: In this case, $\varphi \rightarrow \psi$ needs to be an invariant of P_ν. Then rule **INV** is used as follows: in premise **I1**, Θ is replaced by ν (the initial condition of P_ν). Then all the premises of **INV** are verified with respect to the original program P (to prevent circularity), i.e., by using either other applications of **INV** (with Θ unchanged), or **TAUT**. (Notice that invariants of P also hold for P_ν, but for completeness, stronger invariants that hold for P_ν but not for P might be needed.)

Premise **P3** *of* **PREV**: Θ is replaced by ν.

Other premises are not changed.

Notice that although Θ_ν (which can be stronger than ν) is the initial condition of P_ν, it holds that $P_\nu \models \Theta_\nu \Leftrightarrow \nu$ (although $\Theta_\nu \leftrightarrow \nu$ need not be a tautology). Thus, for each η, $P_\nu \models \Theta_\nu \Rightarrow \eta$ iff $P_\nu \models \nu \Rightarrow \eta$, and hence the explicit use of Θ_ν is eliminated.

Rule 12 PGEN: This rule allows proving $\varphi \Rightarrow EF\psi$ by first splitting φ into a parametrized assertion $\nu(n)$.

$$\frac{\begin{array}{l}\varphi \Rightarrow \exists n\, \nu(n) \\ \nu(n) \Rightarrow EF\psi\end{array}}{\varphi \Rightarrow EF\psi}$$

Rule 13 REDC: If $P' = \langle \Theta', T', D', \vec{z} \rangle$ is an augmented version of the program $P = \langle \Theta, T, D, \vec{y} \rangle$ and κ is a temporal assertion with no free occurrences of variables from $\vec{z} \setminus \vec{y}$, then the augmentation P' can be used to prove that κ holds in P', and conclude that κ holds in P. Lemma 4.7 guarantees the soundness of this rule.

$$\frac{\begin{array}{l}P' \vdash \kappa \\ P \angle P'\end{array}}{\kappa}$$

5.2 Additional Rules

The following rules can be added to the proof system as syntactic sugar. That is, the proof system is complete even without them. However, they turn out to be convenient for verification.

Rule 14 BIMM: This rule is the past version of FIMM. It identify $\varphi \Rightarrow \psi$ as a special case of $\varphi \Rightarrow EP\psi$.

$$\frac{\varphi \Rightarrow \psi}{\varphi \Rightarrow EP\psi}$$

Rule 15 BTRN: The rule BTRN is the past version of FTRN.

$$\frac{\begin{array}{l}\varphi \Rightarrow EP\nu \\ \nu \Rightarrow EP\psi\end{array}}{\varphi \Rightarrow EP\psi}$$

Rule 16 FPRV **and** BPRV: These rules identify $\varphi \Rightarrow EX\psi$ and $\varphi \Rightarrow EY\psi$ as special cases of $\varphi \Rightarrow EF\psi$ and $\varphi \Rightarrow EP\psi$, respectively.

$$\text{FPRV} \quad \frac{\varphi \Rightarrow EX\psi}{\varphi \Rightarrow EF\psi} \qquad\qquad \text{BPRV} \quad \frac{\varphi \Rightarrow EY\psi}{\varphi \Rightarrow EP\psi}$$

Rule 17 FSPLIT **and** BSPLIT: The rule FSPLIT allows splitting the proof of $\varphi \Rightarrow EF\psi$ by decomposing the traces that satisfy φ into φ_1–traces and φ_2–traces. Then, proving separately $\varphi_i \Rightarrow EF\psi$ for $i = 1, 2$. The rule BSPLIT is the past version of FSPLIT.

$$\text{FSPLIT} \quad \frac{\begin{array}{l}\varphi \Rightarrow (\varphi_1 \vee \varphi_2) \\ \varphi_1 \Rightarrow EF\psi \\ \varphi_2 \Rightarrow EF\psi\end{array}}{\varphi \Rightarrow EF\psi} \qquad\qquad \text{BSPLIT} \quad \frac{\begin{array}{l}\varphi \Rightarrow (\varphi_1 \vee \varphi_2) \\ \varphi_1 \Rightarrow EP\psi \\ \varphi_2 \Rightarrow EP\psi\end{array}}{\varphi \Rightarrow EP\psi}$$

6 Soundness and Completeness

In this section, the proof rules presented above are shown to be sound and relatively complete for proving the various properties discussed at the beginning of Section 4.

We assume a first order logic $\mathcal{L}$ that can express for each program P defined over $\mathcal{L}$ the following:

- Encoding strings over the finite alphabet of P, and expressing

- the relation $v \equiv w$ (equivalence between strings under commuting adjacent independent operations), and
- $u\,w$ (concatenation).

- The following relation can be defined $\mathcal{N}(\vec{x},\, \alpha_1\,\alpha_2\,\ldots\,\alpha_n,\, \vec{y})$ iff $\exists \vec{z}_0\, \exists \vec{z}_1\, \ldots\, \exists \vec{z}_n\, (\vec{z}_0 = \vec{x} \wedge \vec{z}_n = \vec{y} \wedge \forall i\, (0 < i \leq n \rightarrow (en_\alpha(\vec{z}_{i-1}) \wedge \vec{z}_i = f_{\alpha_i}(\vec{z}_{i-1})\,)\,)\,)$ (where $\exists \vec{x}$ is an abbreviation for $\exists x_1\, \exists x_2\, \ldots\, \exists x_n$). This can be encoded for example in models which contain the standard arithmetic over the natural numbers, and the binary operations of addition and multiplication [17].

- The language $\mathcal{L}$ includes the *μ-calculus* [2], the recursive ordinals [43] and the order among them. Details of a related construction can be found in [45, 13].

The verification system presented here is complete for any formula of the forms $\varphi \Rightarrow EX\psi$, $\varphi \Rightarrow EP\psi$, $\varphi \Rightarrow AF\psi$, $\varphi \Rightarrow EY\psi$ or $\varphi \Rightarrow EP\psi$, relative to using a first order logic $\mathcal{L}$ that is expressive as required above and the ability of using any tautology of $\mathcal{L}$ as a premise of **TAUT**.

A first order formula φ *describes* a set of traces Σ if it is satisfied by exactly the traces in Σ. Formulating φ might require using an augmentation of the program with auxiliary variables to distinguish between different traces with the same final assignments (but different equivalence classes of sequences).

Proving relative completeness is done by showing that it is possible to formulate the assertions needed as the premises of the proof rules [31, 17]. In the following completeness proofs, we will demonstrate the construction of such assertions (using the relation $\mathcal{N}$) in some of the cases. It is simple to construct in the following proofs the other concrete formulas from the verbal description.

Theorem 6.1 *The rules* **INV** *and* **TAUT** *are relatively complete [27] for proving $AG\eta$.*

Proof. Let φ be a first order formula describing the accessible traces of a program P [31]. The formula φ can be expressed as $\varphi(\vec{y}) = \exists \vec{x} \exists v\, (\Theta(\vec{x}) \wedge \mathcal{N}(\vec{x},\, v,\, \vec{y}))$. The following premises of **INV** hold:

I1 $\Theta \Rightarrow \varphi$ because any initial trace $[\varepsilon]$ (satisfying Θ) is accessible.

I2 For each $\tau \in T$, $(\varphi \wedge en_\tau) \Rightarrow wp_\tau(\varphi)$, since accessible traces are closed under executing any operation of the program P.

I3 $\varphi \Rightarrow \eta$ because if η is invariant, it is satisfied by any accessible trace.

The premises of **INV** can be verified using the rule **TAUT**, since the above entailments can be replaced by (the stronger) first order implications (i.e., replacing the '$\Rightarrow$' symbol with '$\rightarrow$').

By a similar argument, the rules **INV** and **TAUT** are relatively complete for proving $P_\nu \vdash AG\eta$. Then, φ is the first order formula describing the accessible traces of P_ν, and Θ is replaced by ν. The premises **I1**, **I2** and **I3** are provable (in P) by using either **INV** (this time without replacing Θ in **I1**) or **TAUT**. ∎

The soundness of **INV** is proved using standard induction on the length of the traces and is not shown here.

Notation. Let G be a graph satisfying the conditions of rule **NEXT**. The nodes are indexed by non-negative integer numbers N. Let $w = \alpha_1\,\alpha_2\,\ldots\,\alpha_n$ be a string of T^* such that there exists a path $g_{i_0} \stackrel{\alpha_1}{\rightarrow} g_{i_1} \stackrel{\alpha_2}{\rightarrow} \ldots \stackrel{\alpha_n}{\rightarrow} g_{i_n}$ in G with $g_{i_0} = g_0$. Such a path is denoted l_w (if such a path exists, it is uniquely defined by the string w). Define a partial function $f : T^* \mapsto \mathsf{N}$ that returns for each string w that generates a path l_w the index of the last node in the path. For example, in Figure 7, $l_{\alpha_5\alpha_3}$ is the path $g_0 \stackrel{\alpha_5}{\rightarrow} g_0 \stackrel{\alpha_3}{\rightarrow} g_1$. Thus, $f(\varepsilon) = f(\alpha_5) = 0$, and $f(\alpha_5\alpha_2) = 1$.

Lemma 6.2 *Let G be a graph satisfying the conditions of rule* NEXT *for some $\mathcal{T} \subseteq T$. Let $\sigma[w] \in \Pi$ for some run $\Pi \in R_P$ such that $\sigma \models \varphi$, and for each $\tau \in \mathcal{T}$ enabled by fin_σ, $\sigma[\tau] \not\sqsubseteq \sigma[w]$ (i.e., it is impossible to permute w by repeatedly switching adjacent independent operations, resulting in some $\tau \in \mathcal{T}$ to appear first). Then, there exists a path l_w of G where the last node $g_{f(w)}$ satisfies*

1. $\sigma[w] \models \varphi_{f(w)} \wedge \bigwedge_{\tau \in \mathcal{T}_{f(w)}} en_\tau$, *and*
2. *for each $\gamma \in \mathcal{T}_{f(w)}$, no τ, $(\tau, \gamma) \in D$ appears in w.*

Proof. By induction on the length of the sequence w and using the conditions of NEXT. For $w = \varepsilon$, it holds that $f(\varepsilon) = 0$, and by N1, $\varphi \Rightarrow \varphi_{f(\varepsilon)}$. Hence, $\sigma[\varepsilon] \models \varphi_{f(\varepsilon)}$. By N3, for each $\tau \in \mathcal{T}$, $\varphi \Rightarrow wp_\tau(\psi)$ (thus, in particular $\varphi \Rightarrow en_\tau$), hence $\sigma[\varepsilon] \models \bigwedge_{\tau \in \mathcal{T}_{f(\varepsilon)}} en_\tau$. Now, assume that the lemma hold for any prefix of w of length smaller or equal to some m, $0 \leq m < |w|$. Let v be the prefix of w of length m, and β is the $(m+1)^{th}$ operation of w. From the inductive assumption, $\sigma[v] \models \varphi_{f(v)} \wedge \bigwedge_{\tau \in \mathcal{T}_{f(v)}} en_\tau$, and for each $\gamma \in \mathcal{T}_{f(v)}$, no τ, $(\tau, \gamma) \in D$ appears in v.

Now, $\beta \notin \mathcal{T}_{f(v)} \subseteq \mathcal{T}$, otherwise $\sigma[v\beta] = \sigma[\beta v]$ which implies $\sigma[w] = \sigma[\beta w']$, a contradiction to the conditions of the lemma. By N2, $\sigma[v\beta] \models \varphi_{f(v\beta)}$. Since $\mathcal{T}_{f(v\beta)} = \mathcal{T}_{f(v)} \setminus \{\alpha \mid (\beta, \alpha) \in D\}$, then for each $\gamma \in \mathcal{T}_{f(v\beta)}$, no τ, $(\tau, \alpha) \in D$ appears in $v\beta$. The fact that $\sigma[v\beta] \models \bigwedge_{\tau \in \mathcal{T}_{f(v\beta)}} en_\tau$ stems from the requirement on the dependency relation D that executing an operation independent of τ cannot disable τ. ∎

Theorem 6.3 *Rule* NEXT *is sound.*

Proof. First, we show that under the conditions of NEXT, it holds that $\varphi \searrow \mathcal{T}$. Let G be a graph satisfying the conditions of NEXT. We show that the conditions of Lemma 4.5 hold. That is, for each string $v \in T^*$ extending some trace $\sigma \models \varphi$, there exists some $\alpha \in \mathcal{T}$ such that either all the operations in v are independent of α, or $v = \bar{u}\alpha\hat{u}$, where the operations in $\bar{u}$ are independent of α. Let $\bar{u}$ be the maximal prefix of v such that for each $\alpha \in \mathcal{T}$ enabled by fin_σ, $\sigma[\alpha] \not\sqsubseteq \sigma[v]$. Then, by Lemma 6.2, there exists a path $l_{\bar{u}}$ such that for each $\gamma \in \mathcal{T}_{f(\bar{u})}$, $\sigma[v] \models en_\gamma$, and no τ, $(\tau, \gamma) \in D$ appears in $\bar{u}$. Now, if $v = \bar{u}$, then we are done, as α can be any operation of the set $\mathcal{T}_{f(\bar{u})}$. Notice that $\mathcal{T}_{f(\bar{u})}$ is non–empty, as v can extend σ and $\sigma[v] \not\models$ F ($\mathcal{T}_{f(\bar{u})}$ is only allowed to be empty if $\varphi_{f(\bar{u})}$ is F). Otherwise, by the maximality of $\bar{u}$, $v = \bar{u}\alpha\hat{u}$, and α is an operation from $\mathcal{T}$ that is independent of all the operations that occur in $\bar{u}$.

By premise N3 of NEXT, for each trace σ of P satisfying φ, and each $\tau \in \mathcal{T}$, it holds that $\sigma[\tau] \models \psi$. Since $\varphi \searrow \mathcal{T}$, it holds that $\varphi \Rightarrow EX\psi$. ∎

Theorem 6.4 *The rules* NEXT, FCS, INV *and* TAUT *constitute a relatively complete proof system for $\varphi \Rightarrow EX\psi$.*

Proof. Assume $\varphi \Rightarrow EX\psi$. For each accessible trace σ of P such that $\sigma \models \varphi$, let

$$\mathcal{R}(\sigma) = \{\mathcal{T} \mid \forall \Pi \in R_P\, (\sigma \in \Pi \rightarrow succop_P(\Pi, \sigma) \cap \mathcal{T} \neq \phi) \bigwedge \forall \tau \in \mathcal{T}\, (\sigma[\tau] \models \psi)\}$$

Notice that for each $\mathcal{T} \in \mathcal{R}(\sigma)$, for each run Π containing σ, there exists some $\alpha \in \mathcal{T}$ such that $\sigma[\alpha] \in \mathcal{T}$ and $\sigma[\alpha] \models \psi$. The set $\mathcal{R}(\sigma)$ is not empty, since $\varphi \Rightarrow EX\psi$. Let $\mu(\varphi) = \bigcup_{\sigma \models \varphi} \mathcal{R}(\sigma)$. (Notice that $\mu(\varphi)$ is finite as it contains subsets of T.) For each $\mathcal{T} \in \mu(\varphi)$, let $\varphi_{\mathcal{T}}$ be a formula describing the traces $\{\sigma \mid \sigma \models \varphi \wedge \mathcal{T} \in \mathcal{R}(\sigma)\}$. Based on Lemma 4.5, $\varphi_{\mathcal{T}}$ can be expressed as

$$\begin{aligned}\varphi_{\mathcal{T}}(\vec{y}) = \;& \exists \vec{x} \exists v\, (\Theta(\vec{x}) \wedge \mathcal{N}(\vec{x}, v, \vec{y}) \wedge \varphi(\vec{y}) \wedge \\ & \forall \vec{z} \forall u\, (\mathcal{N}(\vec{y}, u, \vec{z}) \rightarrow \exists \alpha((en_\alpha(\vec{y}) \wedge \alpha u \equiv u\alpha) \vee \exists w(u \equiv \alpha w))))\end{aligned}$$

Now, $\varphi \Rightarrow \bigvee_{\mathcal{T}\in\mu(\varphi)}\varphi_{\mathcal{T}}$, and for each $\mathcal{T} \in \mu(\varphi)$, $\varphi_{\mathcal{T}} \Rightarrow EX\psi$.

For each $\mathcal{T} \in \mu(\varphi)$, we will show that $\varphi_{\mathcal{T}} \Rightarrow EX\psi$ is verifiable. Enumerate the subsets of $\mathcal{T}$ by $\mathcal{T}_0$, $\mathcal{T}_1$, ... so that $\mathcal{T}_0 = \mathcal{T}$. Let $\Lambda_{\mathcal{T}_i}$ be the set of accessible traces $\sigma = [v][w]$ satisfying the following conditions:

- $[v] \models \varphi_{\mathcal{T}}$.
- The string w does not contain any occurrence of an operation τ such that for some $\alpha \in \mathcal{T}_i$, $(\tau, \alpha) \in D$.
- For each $\tau \in \mathcal{T} \setminus \mathcal{T}_i$, w contains an occurrence of some α such that $(\tau, \alpha) \in D$.
- An operation $\tau \in \mathcal{T} \setminus \mathcal{T}_i$ may occur in w only after some operation $\alpha \in T \setminus \mathcal{T}_i$ such that $(\tau, \alpha) \in D$.

This means that the operations of $\mathcal{T}_i$ are exactly the ones amongst the operations $\mathcal{T}$ that can be appended to the end of $[v][w]$ and be commuted with all the operations in w.

Clearly, for $\mathcal{T}_j = \phi$ it holds that $\Lambda_{\mathcal{T}_j} = \phi$ (i.e., no trace $[v][w]$ satisfying the above conditions exist), otherwise there exists a run Π such that $succop(\Pi, [v]) \cap \mathcal{T} = \phi$, and thus $\mathcal{T} \notin \mathcal{R}([v])$. Let $\varphi_{\mathcal{T}_i}$ be a predicate describing the traces of $\Lambda_{\mathcal{T}_i}$. Let $G_{\mathcal{T}}$ be a graph with set of nodes $\{\langle\varphi_{\mathcal{T}_i}, \mathcal{T}_i\rangle \mid \mathcal{T}_i \subseteq \mathcal{T}\}$, and set of edges $g_i \xrightarrow{\tau} g_j$, where $\tau \in T \setminus \mathcal{T}_i$ and $\mathcal{T}_j = \mathcal{T}_i \setminus \{\alpha \mid (\tau, \alpha) \in D\}$. We need to show that the premises of **NEXT** hold:

N1 $\varphi_{\mathcal{T}} \Rightarrow \varphi_{\mathcal{T}_0}$ holds because according to the above construction, $\Lambda_{\mathcal{T}_0}$ is a superset of the traces σ such that $\sigma \models \varphi_{\mathcal{T}}$.

N2 If $g_i \xrightarrow{\tau} g_j$ then $(\varphi_i \wedge en_\tau) \Rightarrow wp_\tau(\varphi_j)$. This stems from the fact that if $\sigma \in \Lambda_{\mathcal{T}_i}$, $\tau \in T \setminus \mathcal{T}_i$, and $\sigma \models en_\tau$, then $\sigma[\tau]$, belongs to $\Lambda_{\mathcal{T}_j}$ where $\mathcal{T}_j = \mathcal{T}_i \setminus \{\alpha \mid (\tau, \alpha) \in D\}$ (because $\sigma[\tau]$ satisfies the above conditions of $\Lambda_{\mathcal{T}_j}$).

N3 For each $\tau \in \mathcal{T}$, $\varphi_{\mathcal{T}} \Rightarrow wp_\tau(\psi)$ is satisfied because $\mathcal{T} \in \mu(\varphi)$.

Hence, one can apply the rule **NEXT** $|\mu(\varphi)|$ times for the various sets $\mathcal{T} \in \mu(\varphi)$, and then combine the separate proofs by using the rule **FCS** $|\mu(\varphi)|-1$ times. The entailments that appear in the premises of the mentioned proof rules are provable because of the relative completeness shown in Theorem 6.1. ∎

Denote by $\downarrow w$ the reverse of the string w (i.e., $\downarrow \alpha\beta\gamma$ is $\gamma\beta\alpha$).

Lemma 6.5 *Let G be a graph satisfying the conditions of rule* **PREV** *for some $\mathcal{T} \subseteq T$, and let $\sigma = [v][w]$ be a trace of some run $\Pi \in R_P$ such that $\sigma \models \varphi$. If no $\tau \in \mathcal{T}$ is executed last in $[w]$, then there exists a path $l_{\downarrow w}$ of G. The last node $g_{f(\downarrow w)}$ of $l_{\downarrow w}$ satisfies*

1. *$[v] \models \varphi_{f(\downarrow w)} \wedge \neg\Theta$, and*
2. *for each $\gamma \in \mathcal{T}_{f(\downarrow w)}$, no τ, $(\tau, \gamma) \in D$ appears in w.*

Proof. By induction on the length of w, similar to Lemma 6.2. ∎

Theorem 6.6 *Soundness of* **PREV**.

Proof. We first show that under the conditions P1, P2 and P3 of PREV, it holds that $\mathcal{T} \nwarrow \varphi$. Let G be a graph satisfying the conditions of PREV. Assume for the contrary that $\mathcal{T} \not\nwarrow \varphi$. Thus, there exists a trace $[w]$ of P such that $[w] \models \varphi$ and for no $\tau \in \mathcal{T}$, τ is executed last in $[w]$. But then it follows from Lemma 6.5, taking $\sigma = [\varepsilon][w]$ (i.e., $v = \varepsilon$), that the empty trace $[\varepsilon]$ of the run Π does not satisfy the initial condition Θ, a contradiction. Now, by premise P4 it holds that $\varphi \Rightarrow EY\psi$. ∎

Theorem 6.7 *The rules* PREV, BCS, INV *and* TAUT *constitute a relatively complete proof system for* $\varphi \Rightarrow EY\psi$.

Proof. Assume that $\varphi \Rightarrow EY\psi$. We show relative completeness even when the rule PREV is constrained to a singleton set $\mathcal{T} = \{\alpha\}$. For each trace σ of P that satisfy φ, let

$$\mathcal{B}(\sigma) = \{\alpha \mid \exists\rho\,(\sigma = \rho[\alpha] \bigwedge \rho \models \psi)\,\}$$

The set $\mathcal{B}(\sigma)$ is not empty, since $\varphi \Rightarrow EY\psi$. Let $\bar{\mu}(\varphi) = \bigcup_{\sigma\models\varphi} \mathcal{B}(\sigma)$. Let φ_α be the formula describing the traces $\Lambda_\alpha = \{\sigma \mid \exists\rho\,(\sigma = \rho[\alpha] \bigwedge \rho \models \psi)\}$. Then, $\varphi \Rightarrow \bigvee_{\alpha\in\bar{\mu}(\varphi)} \varphi_\alpha$, and for each $\alpha \in \bar{\mu}(\varphi)$, $\varphi_\alpha \Rightarrow EY\psi$. We will show that $\varphi_\alpha \Rightarrow EY\psi$ is verifiable, and thus it is possible to combine the separate proofs using $|\,\bar{\mu}(\varphi)\,| - 1$ applications of BCS.

Construct a graph G_α with two nodes: $g_0 : \langle \varphi_0^\alpha,\, \{\alpha\}\rangle$, where φ_0^α is a formula describing the traces $\{\rho \mid \exists\sigma\,(\sigma \in \Lambda_\alpha \wedge \rho \sqsubseteq \sigma \wedge \alpha \notin \Delta(\sigma/\rho)\,)\}$, and $g_1 : \langle \mathrm{F},\, \phi\rangle$. Let the set of edges E_α of G_α be

$$\{g_0 \xrightarrow{\beta} g_0 \mid (\alpha,\, \beta) \notin D\} \bigcup \{g_0 \xrightarrow{\beta} g_1 \mid (\alpha,\, \beta) \in D\} \bigcup \{g_1 \xrightarrow{\beta} g_1\} \mid \beta \in T\}$$

It can easily be checked that the conditions P1, P2 and P3 of PREV hold. Finally, P4 holds from the choice of Λ_α.

Theorem 6.8 *Soundness of* CAUS.

Proof. Let σ be a trace which satisfies φ. Then, by $\varphi \Rightarrow (\psi \vee \eta)$, it satisfies also η or ψ. Assume that $\sigma \not\models \psi$. By $\eta \Rightarrow EY(\eta \vee \psi)$ there exists a maximal backward path $\sigma_0,\, \sigma_1,\, \ldots \sigma_n$ such that $\sigma_0 = \sigma$, and for each i, $0 \le i \le n$, $\sigma_i \models \eta$. It is not possible that $\sigma_n = [\varepsilon]$ because $\eta \Rightarrow EY(\eta \vee \psi)$ forces the trace σ_n (which satisfies η) to have a predecessor. From maximality of the sequence and the fact that $\eta \Rightarrow EY(\eta \vee \psi)$, there exists some trace σ_{n+1} such that $|\,\sigma_n/\sigma_{n+1}\,| = 1$ and $\sigma_{n+1} \models \psi$. ∎

Theorem 6.9 *The rules* CAUS, REDC, PREV, INV *and* TAUT *constitute a relatively complete proof system for* $\varphi \Rightarrow EP\psi$.

Proof. Assume $\varphi \Rightarrow EP\psi$. Let η be a first order assertion describing the traces σ that subsume some trace $\rho \models \psi$. The formula η can be expressed as

$$\eta(\vec{y}) = \exists\vec{x}\exists v\exists\vec{z}\exists w(\Theta(\vec{x}) \wedge \mathcal{N}(\vec{x},\, v,\, \vec{z}) \wedge \psi(\vec{z}) \wedge w \neq \varepsilon \wedge \mathcal{N}(\vec{z},\, w,\, \vec{y}))$$

It is obvious that η satisfies the premises C1 and C2 of CAUS. Notice that augmenting the program with auxiliary variables might be needed in order that η will describe exactly the required traces. In this case, the rule REDC can be used as the last step of the proof to return to the original program. Premise C2 of CAUS can be proved using PREV, BCS, INV and TAUT, as shown in Theorem 6.7. ∎

Theorem 6.10 *The rule* JWELL *is sound.*

Proof. Observations where shown to be equivalently defined by a weak fairness (justice) property in Theorem 2.1. Thus, the souness of JWELL closely follows soundness proof such as in [13, 14, 16, 25, 31] with minor changes.

Theorem 6.11 *The rules* JWELL, INV *and* TAUT *are relatively complete for proving* $\varphi \Rightarrow AF\psi$.

Proof. Theorem 2.1 shows that that the set of observations are exactly the interleaving sequences in which no operation is ignored. This is obviously a justice property. Hence, completeness can then be shown along the lines of [13, 14, 16, 25, 31]. ∎

Theorem 6.12 *Relative completeness of the proof system for proving* $\varphi \Rightarrow EF\psi$.

Proof. Assume that $\varphi \Rightarrow EF\psi$. Let Λ be the set of accessible traces satisfying φ. Let $k : \Lambda \mapsto \mathcal{K}$ be a bijection from Λ to some set $\mathcal{K}$. Let $\varphi(n)$ be a parametrized formula, such that $\varphi(n)$ is satisfied exactly by the trace σ such that $k(\sigma) = n$. In order to formulate $\varphi(n)$, P can be augmented with auxiliary variables. Then, the verification is conducted with respect to the augmentation P', and then the rule REDC can be used to return to the original program P.

Let $\psi(n)$ be another parametrized formula such that $\psi(n)$ describes the set of traces $\{\rho \mid \exists\sigma\,(k(\sigma) = n \wedge \sigma \sqsubseteq \rho) \bigwedge \rho \models \psi\}$. Then, the following entailments hold:

$$\varphi \Rightarrow \exists n \in \mathcal{K}\, \varphi(n) \tag{10}$$

$$\varphi(n) \Rightarrow EF\psi(n) \tag{11}$$

$$\psi(n) \Rightarrow \psi \tag{12}$$

The assertions (10) and (12) can be verified using INV. Assume for the moment that (11) can be verified. Then, from (11) and (12) we can verify using FIMM and FTRN that

$$\varphi(n) \Rightarrow EF\psi \tag{13}$$

and then from (10) and (13) using PGEN that

$$\varphi \Rightarrow EF\psi \tag{14}$$

It remained now to show that (11) can be verified. Now, let $\hat{\psi}(n)$ be a parametrized formula such that $\hat{\psi}(n)$ describes the traces that subsume those satisfying $\psi(n)$. Since $\varphi(n) \Rightarrow EF\psi(n)$, every observation that has a trace satisfying $\varphi(n)$, has a trace subsuming some trace that satisfies $\psi(n)$. Hence,

$$\varphi(n) \Rightarrow AF\hat{\psi}(n) \tag{15}$$

which is verifiable as shown in Theorem 6.11. From (15) one can deduce using SMPF that

$$\varphi(n) \Rightarrow EF\hat{\psi}(n) \tag{16}$$

Furthermore,

$$P_{\varphi(n)} \models \hat{\psi}(n) \Rightarrow EP\psi(n) \tag{17}$$

holds and is verifiabled as shown in Theorem 6.9. Finally, (11) is obtained from (16) and (17) using FRBK. ∎

7 Examples

7.1 Proving a Serializability Property

Consider again Net A of Figure 1. Assume that the program's operations are partitioned into two database transactions $T_1 = \{\alpha_1, \alpha_2, \alpha_3\}$ and $T_2 = \{\beta_1, \beta_2, \beta_3\}$. Each operation can represent database read or write where everything except the relevant control part is abstracted out. Dependency between operations α and β means that either both operations belong to the same transaction, or they constitute conflicting reads or writes (i.e., at least one is a write, and both refer to the same database entity).

Serializability of the transactions T_1 and T_2 [3] means in terms of trace semantics the following: In every run, there exists an observation in which either all the operations of T_1 appear before those of T_2, or all the operations of T_2 appear before those of T_1. One can check indeed in Figure 2 that there exist two such representatives:

$$v = \alpha_1\alpha_2\alpha_3\beta_1\beta_2\beta_3,\ w = \beta_1\beta_2\beta_3\alpha_1\alpha_2\alpha_3$$

For simplicity, assume that (as indeed is the case in Figure 1) transactions are executed only once. Denote by "*before T*" and "*after T*" the assertions meaning that control is *before* or *after* the execution of transaction T, respectively. Then, serializability of T_1 and T_2 is expressed as:

$$(\mathit{before}\ T_1 \wedge \mathit{before}\ T_2) \Rightarrow EF((\mathit{before}\ T_2 \wedge \mathit{after}\ T_1) \vee (\mathit{before}\ T_1 \wedge \mathit{after}\ T_2)) \tag{18}$$

$$((\mathit{before}\ T_2 \wedge \mathit{after}\ T_1) \vee (\mathit{before}\ T_1 \wedge \mathit{after}\ T_2)) \Rightarrow EF(\mathit{after}\ T_1 \wedge \mathit{after}\ T_2) \tag{19}$$

The first part of the property (18) is the more difficult and interesting. The second part (19) is simple since once a transaction has executed, its operations are disabled, and the second transaction executes with no interference. In terms of the program's variables, the desirable property is:

$$\{p_0, p_1, p_2\} \Rightarrow EF(\{p_0, p_2, p_7\} \vee \{p_0, p_1, p_8\})$$

The following table summarizes the proof:

1.	$\{p_0,\ p_1,\ p_2\} \Rightarrow EX\{p_0,\ p_2,\ p_3\}$	**NEXT**	$\mathcal{T} = \{\alpha_1\}$
2.	$\{p_0,\ p_2,\ p_3\} \Rightarrow EX\{p_0,\ p_3,\ p_4\}$	**NEXT**	$\mathcal{T} = \{\beta_1\}$
3.	$\{p_0,\ p_3,\ p_4\} \Rightarrow EX(\{p_4,\ p_5\} \vee \{p_3,\ p_6\})$	**NEXT**	$\mathcal{T} = \{\alpha_2,\ \beta_2\}$
4.	$(\{p_4,\ p_5\} \vee \{p_3,\ p_6\}) \Rightarrow (\{p_4,\ p_5\} \vee \{p_3,\ p_6\})$	**TAUT**	
5.	$\{p_4,\ p_5\} \Rightarrow EX(\{p_0,\ p_4,\ p_7\} \vee \{p_0,\ p_3,\ p_8\})$	**NEXT**	$\mathcal{T} = \{\alpha_3\}$
6.	$\{p_3,\ p_6\} \Rightarrow EX(\{p_0,\ p_4,\ p_7\} \vee \{p_0,\ p_3,\ p_8\})$	**NEXT**	$\mathcal{T} = \{\beta_3\}$
7.	$(\{p_4,\ p_5\} \vee \{p_3,\ p_6\}) \Rightarrow EX(\{p_0,\ p_4,\ p_7\} \vee \{p_0,\ p_3,\ p_8\})$	**FCS**	4, 5, 6
8.	$\{p_0,\ p_1,\ p_2\} \Rightarrow EF\{p_0,\ p_2,\ p_3\}$	**FPRV**	1
9.	$\{p_0,\ p_2,\ p_3\} \Rightarrow EF\{p_0,\ p_3,\ p_4\}$	**FPRV**	2
10.	$\{p_0,\ p_1,\ p_2\} \Rightarrow EF\{p_0,\ p_3,\ p_4\}$	**FTRN**	8, 9
11.	$\{p_0,\ p_3,\ p_4\} \Rightarrow EF(\{p_4,\ p_5\} \vee \{p_3,\ p_6\})$	**FPRV**	3
12.	$\{p_0,\ p_1,\ p_2\} \Rightarrow EF(\{p_4,\ p_5\} \vee \{p_3,\ p_6\})$	**FTRN**	10, 11
13.	$(\{p_4,\ p_5\} \vee \{p_3,\ p_6\}) \Rightarrow EF(\{p_0,\ p_4,\ p_7\} \vee \{p_0,\ p_3,\ p_8\})$	**FPRV**	7
14.	$\{p_0,\ p_1,\ p_2\} \Rightarrow EF(\{p_0,\ p_4,\ p_7\} \vee \{p_0,\ p_3,\ p_8\})$	**FTRN**	12, 13
15.	$(\{p_0,\ p_4,\ p_7\} \vee \{p_0,\ p_3,\ p_8\}) \Rightarrow (\{p_0,\ p_4,\ p_7\} \vee \{p_0,\ p_3,\ p_8\})$	**TAUT**	
16.	$\{p_0,\ p_4,\ p_7\} \Rightarrow EY(\{p_0,\ p_2,\ p_7\} \vee \{p_0,\ p_1,\ p_8\})$	**PREV**	$\mathcal{T} = \{\beta_1\}$
17.	$\{p_0,\ p_3,\ p_8\} \Rightarrow EY(\{p_0,\ p_2,\ p_7\} \vee \{p_0,\ p_1,\ p_8\})$	**PREV**	$\mathcal{T} = \{\alpha_1\}$
18.	$(\{p_0,\ p_4,\ p_7\} \vee \{p_0,\ p_3,\ p_8\}) \Rightarrow EY(\{p_0,\ p_2,\ p_7\} \vee \{p_0,\ p_1,\ p_8\})$	**BCS**	15, 16, 17
19.	$(\{p_0,\ p_4,\ p_7\} \vee \{p_0,\ p_3,\ p_8\}) \Rightarrow EP(\{p_0,\ p_2,\ p_7\} \vee \{p_0,\ p_1,\ p_8\})$	**BPRV**	18
20.	$\{p_0,\ p_1,\ p_2\} \Rightarrow EF(\{p_0,\ p_2,\ p_7\} \vee \{p_0,\ p_1,\ p_8\})$	**FRBK**	14, 19

The graphs needed for the proofs of $\{p_0, p_1, p_2\} \Rightarrow EX\{p_0, p_2, p_3\}$ (line 1) and $\{p_0, p_4, p_7\} \Rightarrow EY(\{p_0, p_2, p_7\} \vee \{p_0,\ p_1,\ p_8\})$ (line 16), are depicted in Figure 8.

A more comprehensive treatment of serializability using **ISTL*** appears in [37].

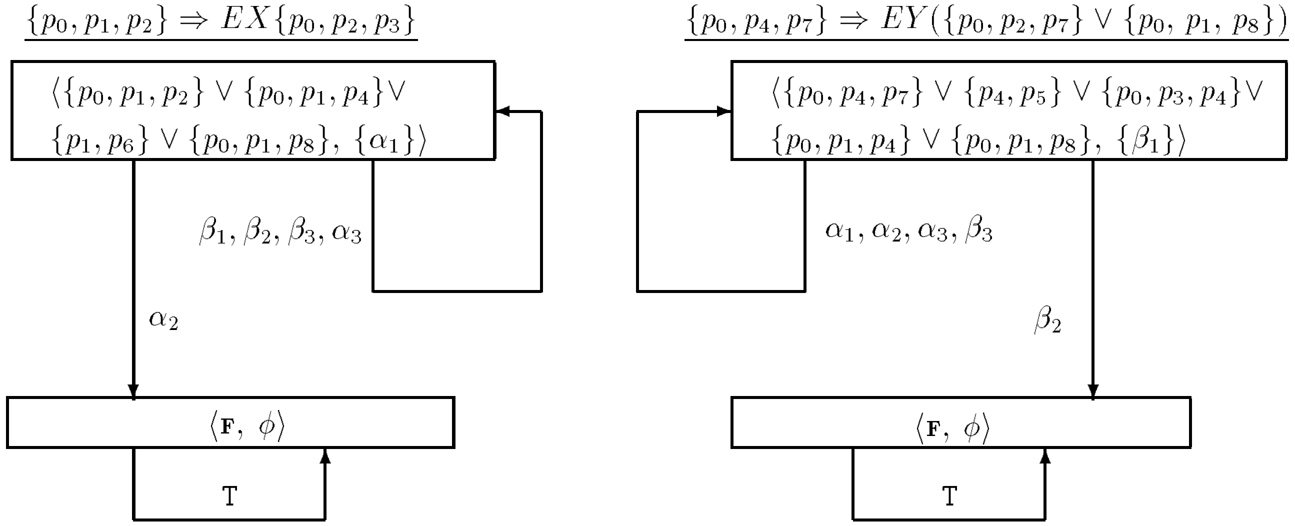

Figure 8: Graphs for **NEXT** and **PREV**.

7.2 The Snapshot Algorithm

We deal here with the well known *Snapshot Algorithm* of Chandy and Lamport [5]. The algorithm is superimposed on some basic algorithm. Its purpose is to record a global state of the underlying basic algorithm. The motivation to use trace semantics is that the global state which is recorded by the snapshot algorithm did not necessarily occur in the past of the same interleaving sequence in which it is eventually recorded. Rather, it occurs as the final interpretation of some trace subsumed by the trace in which the recording is completed. This can be written in **ISTL*** as

$$\text{“the global state recorded is } s\text{”} \rightarrow EP \text{ “the global state is } s\text{”}$$

Recall that from the above discussion, EP cannot be replaced with AP. We use a CSP-like notation which is a reduced set of the CSP language [18].

A program is constructed from a set of processes, each having its own set of local variables. Each process is written in the form

$$\mathit{Initiation};\ *[\mathit{guard}_1 \longrightarrow \mathit{action}_1 \Box \ldots \Box \mathit{guard}_n \longrightarrow \mathit{action}_n]$$

Where *Initiation* is a set of assignment operations which establish the initial values of the program variables. A *guard* may contain a boolean condition and an optional communication command $P!z$ (a *send*) or $P?z$ (a *receive*) for some process P. An *action* is a sequence of assignment operations (for simplicity we assume here that they can be executed in a single atomic step).

A *simple operation* τ is a pair $\mathit{guard}_i \rightarrow \mathit{action}_i$ where guard_i does not contain a communication command. In this case, $en_\tau = \mathit{guard}_i$. The transformation function f_τ is constructed from the assignment operations in action_i . A *communication operation* τ is constructed from two pairs: $\mathit{guard}_i \longrightarrow \mathit{action}_i$ in some process P and $\mathit{guard}_j \longrightarrow \mathit{action}_j$ in another process P', such that guard_i contains the command $P'!z$ and guard_j contains $P?y$ for some variables z and y. In this operation, process P sends its z value to the variable y in process P'. Then, $en_\tau = (\mathit{guard}_i \wedge \mathit{guard}_j)$, and the transformation function of τ contains the assignments included in action_i and action_j in addition to assigning z to y. Comments are inserted into the text following a pair of hyphens. The transformation of the program into a set of operations is straightforward. A transformation of full

CSP into operations appears in [28].

We say that an operation τ is *concerned with* a process P if τ is either a simple operation of P, or a communication operation where the send or the receive belongs to P. The dependency relation is defined such that $(\alpha,\ \beta) \in D$ iff α and β are concerned with a mutual process.

For simplicity, the example is limited to two processes P_1 and P_2. Since the snapshot algorithm should be superimposed on arbitrary programs, we used some abstraction concerning the basic algorithm. That is, we use an uninterpreted generic program, with only enough operations to consider all possible cases. Thus, each process P_i can do some local assignment operations $local_i$ when the condition $bool_i$ allows it. When $sendcond_i$ holds, P_i is ready to send some value to the other process. After sending, $sendlocal_i$ can be executed. Similarly, when $recvcond_i$ holds, P_i is ready to receive a value and thereafter to execute $recvlocal_i$. Notice that in order that communication takes place, one process must be willing to send while the other is willing to receive at the same time. The processes have the following structure:

$P_i :: Init_i;$
$\langle a\rangle$ $*[\ bool_i \longrightarrow local_i$
$\qquad\Box$
$\langle b\rangle$ $\qquad sendcond_i; P_{3-i}!val_i \longrightarrow sendlocal_i$
$\qquad\Box$
$\langle c\rangle$ $\qquad recvcond_i; P_{3-i}?var_i \longrightarrow recvlocal_i]$

The snapshot algorithm is implemented as a transformation imposed on the basic algorithm. A process is colored either white or red. Initially, it is colored white, and it turns red when entering the stage of collecting the snapshot of the global state. A process can turn red spontaneously ($\langle f\rangle$). A process can also turn red by receiving a special *marker* message from a red neighbor ($\langle g\rangle$). When a process turns red, it ceases to send messages according to the basic algorithm ($\langle b\rangle$), until it sends a *marker* message to all of its neighbors ($\langle d\rangle$). It also ceases to receive messages of the basic algorithm ($\langle c\rangle$) from each neighbor until it has received a *marker* message from it ($\langle e\rangle$). The *marker* message also transfer the local state recorded when the sending process turned red ($\langle e\rangle$, $\langle f\rangle$). The local guarded command $\langle a\rangle$ is not changed.

We present now the superimposition of the snapshot algorithm on the abstract basic algorithm according to the transformation described in [4]. The variable **turned_red** is an auxiliary variable, used only for the correctness proof. Assignment to auxiliary variables are added to the code of the program and are marked with an arrow ($\Longrightarrow$).

$P_i :: init_i;$
$\qquad red_i := false; received_i := false;$
$\qquad sent_i := false;$
$\Longrightarrow$ $\qquad \mathbf{turned\_red}_i := not\_yet;$
$\qquad Init_i;$
$\langle a\rangle$ $\qquad *[\ bool_i \longrightarrow local_i$ -- local action
$\qquad\quad\Box$
$\langle b\rangle$ $\qquad\quad (sent_i \vee \neg red_i) \wedge sendcond_i; P_{3-i}!val_i \longrightarrow sendlocal_i$ -- send action
$\qquad\quad\Box$
$\langle c\rangle$ $\qquad\quad (received_i \vee \neg red_i) \wedge recvcond_i; P_{3-i}?var_i \longrightarrow recvlocal_i$ -- receive action
$\qquad\quad\Box$
$\langle d\rangle$ $\qquad\quad red_i \wedge \neg sent_i; P_{3-i}!marker(state_i) \longrightarrow sent_i := true$ -- send marker
$\qquad\quad\Box$

$$
\begin{array}{lll}
\langle e\rangle & & red_i \wedge \neg received_i; P_{3-i}?marker(other\_state_i) \longrightarrow received_i := true \text{ -- receive marker} \\
& & \Box \\
\langle f\rangle & & \neg red_i \longrightarrow red_i := true; state_i := record\_local_i; \text{ -- spontaneously turn red} \\
\Longrightarrow & & \qquad \mathbf{turned\_red}_i = self \\
& & \Box \\
\langle g\rangle & & \neg red_i; P_{3-i}?marker(other\_state_i) \longrightarrow red_i := true; received_i := true; \\
& & \qquad state_i := record\_local_i; \text{ -- turn red when} \\
\Longrightarrow & & \qquad \mathbf{turned\_red}_i := hear \text{ -- receiving a marker} \\
& &]
\end{array}
$$

The boolean variable red_i is used to record when the process P_i has turned red. The variables $received_i$ and $sent_i$ keep track whether process P_i has yet received a marker or sent a marker to the other process, respectively. The function $record\_local_i$ (with no parameters) returns some designate of the process' local state at the moment of the call. This value is recorded in the local variable $state_i$ and transmitted (with the *marker* message) to the other process. The variable $other\_state_i$ holds this value transmitted from its companion process. As mentioned before, $\mathbf{turned\_red}_i$ is an auxiliary variable. Its possible values are *not_yet* (which means that the process is unaware of the beginning of a snapshot sample), *self* (the process turned red spontaneously), and *hear* (turned red as a result of receiving a message).

The labeling to the left of the process text will help later to identify the operations. For example, the operation $b_1 \longrightarrow c_2$ will identify the sending of value from P_1 to P_2 (the direction of the arrow indicates the sending direction), a_2 is P_2's local operation, and $g_1 \longleftarrow d_2$ is the operation where P_2 makes P_1 red by sending a marker together with reporting its recorded local state. The following invariants are used in the proof.

$$I_1 :: \bigwedge_{i\in\{1,2\}} \mathbf{turned\_red}_i \in \{not\_yet, self, hear\}$$

$$I_2 :: \bigwedge_{i\in\{1,2\}} (red_i \rightarrow (\mathbf{turned\_red}_i \neq not\_yet))$$

$$I_3 :: \bigwedge_{i\in\{1,2\}} ((received_i \vee sent_i) \rightarrow (red_1 \wedge red_2))$$

$$I_4 :: \bigwedge_{i\in\{1,2\}} ((\mathbf{turned\_red}_i = hear) \rightarrow received_i)$$

$$I_5 :: \neg((\mathbf{turned\_red}_1 = hear) \wedge (\mathbf{turned\_red}_2 = hear))$$

$$I_6 :: \bigwedge_{i\in\{1,2\}} (received_i \leftrightarrow sent_{3-i})$$

It is possible to prove that I_5 is an invariant using I_2, I_3, and I_4. Initially, I_5 holds. I_5 can turn false only from a state in which $(\mathbf{turned\_red}_1 = hear) \wedge ((\mathbf{turned\_red}_2 = not\_yet) \vee (\mathbf{turned\_red}_2 = self))$ or a state in which $(\mathbf{turned\_red}_2 = hear) \wedge ((\mathbf{turned\_red}_1 = not\_yet) \vee (\mathbf{turned\_red}_1 = self))$. Consider without loss of generality the first case. If $\mathbf{turned\_red}_2 = not\_yet$, then from I_2 it holds that $\neg red_2$ and thus from I_3, $\neg received_1$ and from I_4, $\mathbf{turned\_red}_1 \neq hear$. If $\mathbf{turned\_red}_2 = self$, then $\mathbf{turned\_red}_2$ cannot be changed any more, since from I_2, red_2 holds.

By I_1, I_2, I_3 and I_5, it is possible to conclude that the following invariant also holds:

$$
\begin{array}{rl}
received_1 \rightarrow & ((\mathbf{turned\_red}_1 = self) \wedge (\mathbf{turned\_red}_2 = self)) \\
& \vee((\mathbf{turned\_red}_1 = self) \wedge (\mathbf{turned\_red}_2 = hear)) \\
& \vee((\mathbf{turned\_red}_1 = hear) \wedge (\mathbf{turned\_red}_2 = self))
\end{array}
\tag{20}
$$

We want to prove the following property:

$$\begin{array}{c}(received_1 \wedge (state_1 = X) \wedge (other\_state_1 = Y)) \Rightarrow \\ EP((record\_local_1 = X) \wedge (record\_local_2 = Y))\end{array} \tag{21}$$

Notice that the asymmetry of this formula (asserting about the values recorded by process P_1) causes an asymmetry in the following proof.

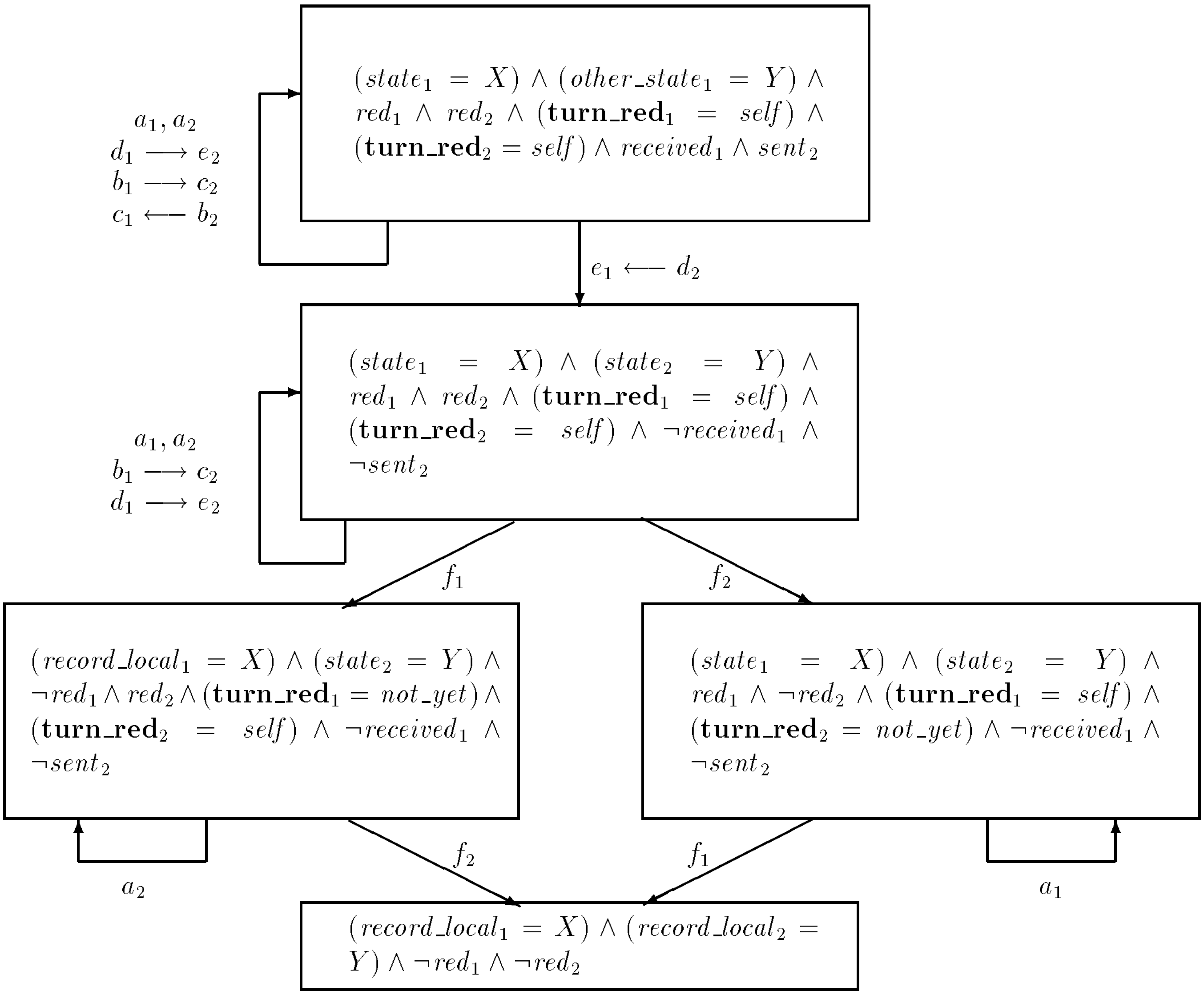

Figure 9: Both processes turn red spontaneously

There are three possible scenarios. In order to describe them, we use a directed graph $\langle V, E\rangle$ that contains no cycles other than self loops, whose nodes contain first order assertions, and edges are marked by operations. (Multiple edges are allowed, marked by different operations.) This graph has a single distinguished *root* node r and a single sink node k (a node with no outgoing edged). Denote the formula of a node s by $\varphi(s)$. The following conditions are satisfied:

1. If some edge from a node s to a node t is marked by τ, then $wp_\tau(\varphi(s)) \Rightarrow \varphi(t)$. That is, if reaching a state satisfying $\varphi(s)$ when executing τ, then the previous state prior to the execution of τ satisfied $\varphi(t)$.

2. The set of operations labeling the edges which emanate from s constitute a backward intercepting set for $\varphi(s)$.

3. For each node s which is not k, $\varphi(s) \Rightarrow \neg\Theta$.

Thus, if we denote the set of nodes which are connected to s by an edge which is directed from s by $out(s)$, we may conclude using rule **PREV**, that

$$\varphi(s) \Rightarrow EY \bigvee_{t \in out(s)} \varphi(t)$$

This leads to the conclusion that

$$\left(\bigvee_{t \in V \setminus \{k\}} \varphi(t)\right) \Rightarrow EY \bigvee_{s \in V} \varphi(s) \tag{22}$$

and together with the obvious fact that

$$\varphi(r) \Rightarrow \bigvee_{t \in V} \varphi(t) \tag{23}$$

we can conclude using rule **CAUS** (using (22) and (23) as premises **C2** and **C1**, respectively) that

$$\varphi(r) \Rightarrow EP\varphi(k).$$

The three scenarios of the algorithm are given below. We draw a small diagram for each scenario, depicting the order of the operations. In each such diagram, the time is running downwards. A small box containing a set of operations means that any of these operations can occur zero or more times.

- Both processes turn red spontaneously. The operations f_1 and f_2 can occur concurrently (the order between them does not matter). Then, the operations $\{a_1, a_2, d_1 \longrightarrow e_2, b_1 \longrightarrow c_2\}$ can occur, followed by $e_1 \longleftarrow d_2$. After that, the operations $\{a_1, a_2, d_1 \longrightarrow e_2, b_1 \longrightarrow c_2, c_1 \longleftarrow b_2\}$ can occur.

$$\boxed{\begin{array}{ccc} f_1 & & f_2 \\ & \boxed{\begin{array}{c} a_1, a_2 \\ d_1 \longrightarrow e_2 \\ b_1 \longrightarrow c_2 \end{array}} & \\ e_1 & \longleftarrow & d_2 \\ & \boxed{\begin{array}{c} a_1, a_2 \\ d_1 \longrightarrow e_2 \\ b_1 \longrightarrow c_2 \\ c_1 \longleftarrow b_2 \end{array}} & \end{array}}$$

The proof lattice of this scenario appears in Figure 9.

- Process P_1 turns red spontaneously and makes process P_2 turn red by sending a marker message.

$$\boxed{\begin{array}{ccc} f_1 & & \\ & \boxed{a_1} & \\ d_1 & \longrightarrow & g_2 \\ & \boxed{\begin{array}{c} a_1, a_2 \\ b_1 \longrightarrow c_2 \end{array}} & \\ e_1 & \longleftarrow & d_2 \\ & \boxed{\begin{array}{c} a_1, a_2 \\ b_1 \longrightarrow c_2 \\ c_1 \longleftarrow b_2 \end{array}} & \end{array}}$$

The corresponding proof lattice appears in Figure 10.

- Process P_2 turns red spontaneously and makes process P_1 turn red by sending a marker message.

$$\boxed{\begin{array}{ccc} & & f_2 \\ & \boxed{a_2} & \\ g_1 & \longleftarrow & d_2 \\ & \boxed{\begin{array}{c} a_1, a_2 \\ c_1 \longleftarrow b_2 \\ d_1 \longrightarrow e_2 \\ b_1 \longrightarrow c_2 \end{array}} & \end{array}}$$

The corresponding proof lattice appears in Figure 11.

Now, using (20), and the invariants $I_1 \ldots I_6$, it follows that when the left hand side of the implication (21) holds, one of the assertions at the roots of the above graphs holds. The right hand side of the implication (21) is implied by the assertion of the sink node of each of the graphs. Formally, this means that we can complete the proof using rule **BSPLIT**.

Acknowledgements

We appreciate the helpful conversations we have had on this subject with Shmuel Katz, Orna Grümberg and Nissim Francez. We also thank the referees of the 17^{th} ICALP and of Theoretical Computer Science for suggesting improvements in our presentation.

References

[1] K. R. Apt, Recursive Assertions and Parallel Programs, Acta Informatica 15(1981), 219–232.

[2] J. W. deBakker, Mathematical Theory of Program Correctness, Prentice–Hall, Englewood Cliffs, N. J, 1980.

[3] P. A. Bernstein, V. Hadzilacos, N. Goodman, Concurrency control and Recovery in Database Systems, Addison–Wesley, 1987.

[4] L. Bougé, Repeated snapshots in distributed systems with synchronous communications and their implementations in CSP, Theoretical Computer Science 49 (1987), 145–169.

[5] K. M. Chandy, L. Lamport, Distributed Snapshots: determining the global state of distributed systems, ACM Transactions on Computer Systems 3 (1985), 63–75.

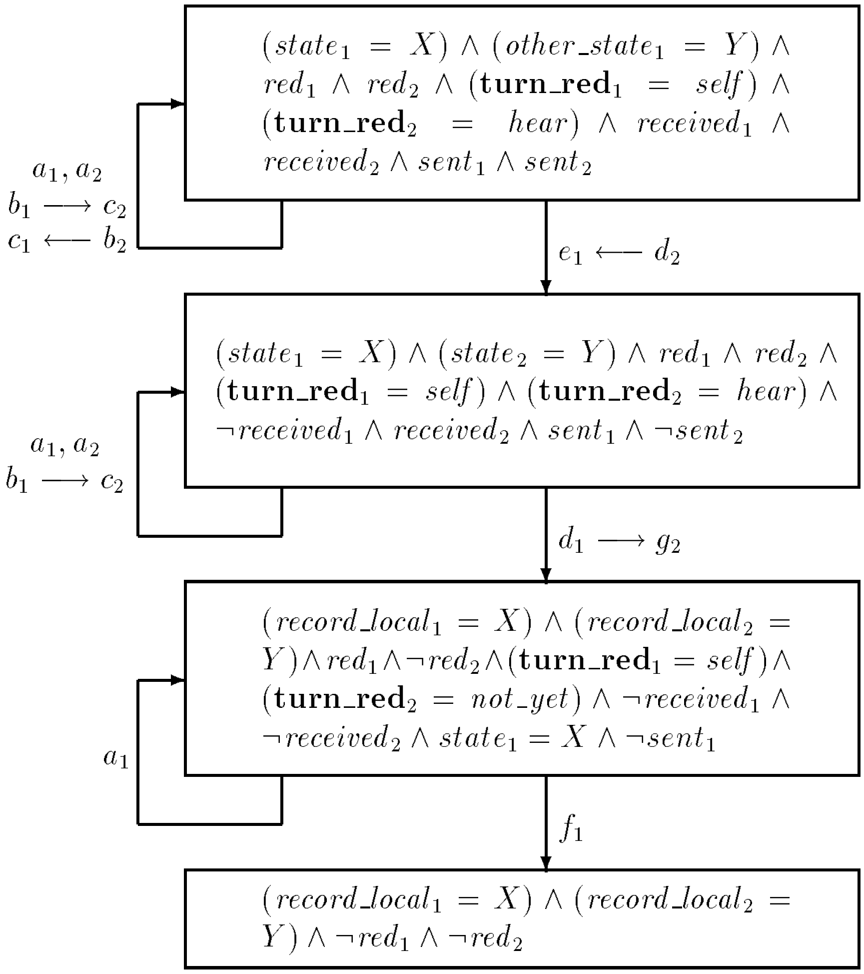

Figure 10: First process turns red spontaneously

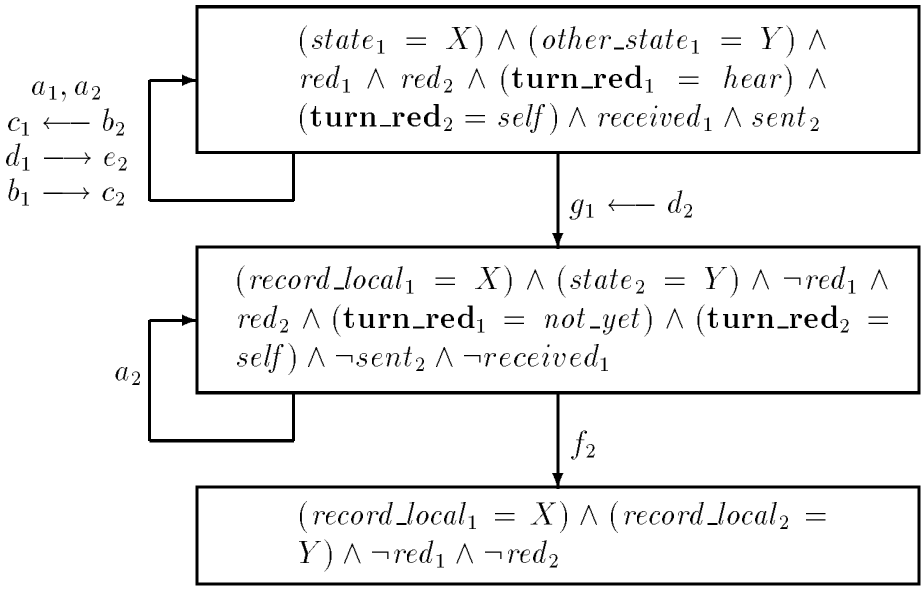

Figure 11: Second process turns red spontaneously

[6] M. Clint, Program proving: Coroutines, Acta Informatica 2 (1973) 50–63.

[7] S. A. Cook, Soundness and Completeness of an Axiom Systems for Program Verification, SIAM Journal of Computation 7 (1978), 70–90.

[8] E. W. Dijkstra, Cooperating Sequential Processes, Technical Report EWD123, Technological University, Eindhoven, The Netherlands, 1965, Reprinted in: F. Genuys (Editor), Programming languages, Academic Press, London, 1968, 43–112.

[9] E.W. Dijkstra, Guarded commands, Nondeterminancy and Formal Derivation of Programs, Communication of the ACM, 18(1975), 453–457.

[10] Tz. Elrad, N. Francez, Decomposition of distributed programs into communication–closed layers, Science of Computer Programming 2 (1982), 155–173

[11] E. A. Emerson, E. M. Clarke, Using branching time temporal logic to synthesize synchronization skeletons, Science of Computer Programming 2 (1982), 241–266.

[12] E.A. Emerson, J.Y. Halpern, "Sometimes" and "not never" revisited: on branching versus linear time temporal logic, Journal of the ACM 33(1986), 151–178.

[13] N. Francez, Fairness, Springer–Verlag, 1986.

[14] N. Francez, D. Kozen, Generalized Fair Termination, Proc. 11^{th} Symposium on Principles of Programming Languages, Salt Lake City, 1984, 46–53.

[15] H. Gaifman, V. Pratt, Partial Order Models of Concurrency and Computation of Functions, Symposium on Logic in Computer Science, 1987, 72–85.

[16] O. Grümberg, N. Francez, J. A. Makowski, W. P. de Roever, A proof rule for termination of guarded commands, Information and Control, 66 (1985), 83–102.

[17] D. Harel, First order Dynamic Logic, Lecture Notes in Computer Science 68, Springer–Verlag, 1979.

[18] C.A.R. Hoare, Communicating sequential processes, Communications of the ACM, 21(1978), 666–677.

[19] R. Janicki, P. E. Lauer, M. Koutny, R. Devillers, Concurrent and maximally Concurrent Evolution of Non–Sequential System, Theoretical Computer Science, 43(1986), 213–238.

[20] S. Katz, D. Peled, Interleaving Set Temporal Logic, Theoretical Computer Science, Vol. 75, Number 3, 21–43, also appeared in proceedings of the 6^{th} Annual ACM Symposium on Principles of Distributed Computing, Vancouver, Canada, August 1987, 178–190.

[21] S. Katz, D. Peled, Verification of Distributed Programs using Representative Interleaving Sequences, to appear in Distributed Computing. A preliminary version, titled An Efficient Verification Method for Parallel and Distributed Programs, appeared in: J. W. de Bakker, W. P. de Roever, G. Rozenberg, School/Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, Noordwijkerhout, The Netherlands, May/June 1988, Lecture Notes in Computer Science 354, Springer–Verlag 1989, 489–507.

[22] S. Katz, D. Peled, Defining Conditional Independence Using Collapses, Theoretical Computer Science, 101 (1992), 337–359, a preliminary version appeared in BCS–FACS Workshop on Semantics for Concurrency, Leicester, England, July 1990, Springer–Verlag, 262–280.

[23] M. Z. Kwiatkowska, Fairness for Non–interleaving Concurrency, Phd. Thesis, Faculty of Science, University of Leicester, 1989.

[24] L. Lamport, Paradigms for distributed programs: computing global states, in: M. Paul and H.J. Siegert (eds.), Distributed systems – Methods and tools for specification, An advanced course, Munich, Lecture notes in Computer Science 190, Springer–Verlag, 1985, 454–468.

[25] D. Lehman, A. Pnueli, J. Stavi, Impartiality, Justice and Fairness: The ethics of concurrent termination, in:S. Even, O. Kariv (eds.), Proceedings of the 8^{th} International colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science 115, 1981, 264–277.

[26] Z. Manna, A. Pnueli, Verification of concurrent programs: the temporal framework, in: R. S. Boyer and J. S. Moore (Eds.), The correctness problem in computer science, 1981, 215–273.

[27] Z. Manna, A. Pnueli, Verification of concurrent programs, a temporal proof system, Proceedings of the 4^{th} School on Advanced Programming, Amsterdam, Holland, June 1982, 162–255.

[28] Z. Manna, A. Pnueli, How to cook a temporal proof system for your pet language. Proceedings of the Symposium on Principles on Programming Languages, Austin, Texas, 1983, 141–151.

[29] Z. Manna, A. Pnueli, Adequate proof principles for invariance and liveness properties of concurrent programs, Science of Computer Programming 4 (1984), 257–289.

[30] Z. Manna, A. Pnueli, The Anchored Version of the Temporal Framework, in: J. W. de Bakker, W. P. de Roever, G. Rozenberg (eds.), School/Workshop on Linear Time, Branching Time and Partial order in Logics and Models May/June 1988, The Netherlands, Lecture Notes on Computer Science 354, Springer–Verlag, 1988, 201–281.

[31] Z. Manna, A. Pnueli, Completing the Temporal Picture, Proceedings 16^{th} International Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science 372 (1989), 534–558.

[32] Z. Manna, P. Wolper, Synthesis of communicating processes from temporal logic specifications, ACM Transactions on Programming Languages and Systems 6 (1984), 68–93.

[33] A. Mazurkiewicz, Trace semantics, in: W. Brauer, W. Reisig, G. Rozenberg (eds.) Proceedings of Advances in Petri Nets 1968, Bad Honnef, Lecture Notes in Computer Science 255, Springer–Verlag, 1987, 279–324.

[34] A. Mazurkiewicz, E. Ochmanski, W. Penczek, Concurrent Processes and Inevitability, Theoretical Computer Science 64 (1989), 281–304.

[35] M. Nielsen, G. Plotkin, G. Winskel, Petri Nets, Event Structures and Domains, Part I, Theoretical Computer Science 13(1981), 85–108.

[36] S. Owicki, A consistent and complete deductive system for the verification of parallel programs, Proceedings of the 8^{th} Annual Symposium on Theory of Computing, 1976, 73–86.

[37] D. Peled, S. Katz, A. Pnueli, Specifying and Proving Serializability in Temporal Logic, 6^{th} IEEE annual symposium on Logic in Computer Science, Amsterdam, The Netherlands, July 1991, 232–245.

[38] C. A. Petri, Kommunikation mit Automaten, Bonn: Institut für Instrumentelle Matematik, Schriften des IIM Nr. 2(1962).

[39] S. Pinter, P. Wolper, A temporal logic for reasoning about partially ordered computations, Proceedings of the 3^{rd} ACM Symposium on Principles of Distributed Computing, Vancouver, B. C., August 1984, 23–27.

[40] A. Pnueli, R. Rosner, On the Synthesis of an Asynchronous Reactive Module, 16^{th} International Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science, 372, Springer–Verlag, 1989, 652–671.

[41] V. Pratt, Modeling concurrency with partial orders, International Journal of Parallel Programming, 15 (1986), 33–71.

[42] W. Reisig, Temporal logic and causality in concurrent systems, Proceedings of CONCURRENCY 88, Hamburg, Lecture Notes in Computer Science 335, Springer–Verlag, 1988, 121–139.

[43] H. Rogers, Theory of recursive functions and effective computability, MgGraw–Hill, New York, 1967.

[44] J. R. Shoenfield, Mathematical Logic, Addison–Wesley 1976.

[45] F. A. Stomp, W. P. deRoever, R. T. Gerth, The μ-Calculus as an Assertion-Language for Fairness Arguments, Information and computation 82 (1989), 278–322.

[46] P. S. Thiagarajan, Elementary Net Systems, in: W. Brauer, W. Reisig, G. Rozenberg (eds.), Proceedings of Advances in Petri Nets 1986, Lecture Notes in Computer Science 254, Springer Verlag 1987, 26–59.