INTERLEAVING SET TEMPORAL LOGIC*

Shmuel KATZ** and Doron PELED

Department of Computer Science, Technion, Haifa, Israel

Communicated by Z. Manna
Received April 1987
Revised July 1988, April 1989

Abstract. A new temporal logic and interpretation are suggested which have features from linear temporal logic, branching time temporal logic, and partial order temporal logic. The new logic can describe properties essential to the specification and correctness proofs of distributed algorithms, such as those for global snapshots. It is also appropriate for the justification of proof rules and for ascribing temporal semantics to properties such as layering of a program. These properties cannot be described with existing temporal logics. The semantic model of the logic is based on a collection of sets of interleaving sequences which reflect partial orders from the underlying semantics of the computational model. For the common partial order derived from sequentiality in execution of each process, the logic will distinguish between nondeterminism due to the parallel execution and nondeterminism due to local nondeterministic choices. The difference in expressive power is thus qualitative, and not merely due to the presence or absence of a particular temporal operator. In the logic, theorems are proven which clarify when it is possible to establish a property $P$ for some of the interleaving computations, and yet conclude the truth of $P$ for every interleaving.

1. Introduction

Many attempts have been made to design a logic which allows expressing specifications and proving correctness for distributed systems, and the relative expressibility of each has been studied. We deal here with a new kind of temporal logic, called Interleaving Set Temporal Logic (ISTL), which can express various new properties that existing logics can not.

The simplest version of ISTL is syntactically identical to a branching-time temporal logic [9], but the formulas have a different semantic interpretation, one which is more appropriate for distributed computation and specification.

In any (temporal or other) logic, the semantic objects over which the logic operates determine what can be expressed about the "reality" outside the logic. The logic here is in essence over a collection of triples, where each triple is a set of events, a partial order on the events, and an initial assignment. Each such triple is used to generate a branching structure, equivalent to a collection of all interleavings of the


** The research of this author was supported by the Technion V.P.R. Fund—F. and J. Bishop Research Fund.
events in the set which are consistent with the partial order and satisfy an additional fairness criterion called acceptability. Within each such set (called an interleaving set) global states and explicit reasoning on the paths are used. Because all the paths in the same interleaving set are generated from a single partial order and event set, they have certain uniformity properties that all of the interleavings of events generated from a program do not have.

We demonstrate that this semantic interpretation is appropriate for expressing and reasoning about many known phenomena of distributed computation which cannot be expressed in other temporal logics.

This logic is particularly appropriate for specifying and reasoning about distributed programs and languages which have both independently executing processes, and local nondeterministic choices within the processes. Such nondeterministic constructs are common in most languages for distributed programming, with two well-known examples being the select construct of Ada, and the guards of CSP [20]. Among other benefits, the explicit nondeterminism allows choosing one of a number of possible communications without imposing an arbitrary ordering in advance. In the partial order that appears frequently in the literature [7, 5, 22, 26, 38], it is natural to distinguish between the nondeterminism which arises because of the independent execution of events in different processes, and the nondeterminism due to the explicit choices of the nondeterministic control constructs. However, other partial orders can be considered in the framework of ISTL. In Section 2, several alternative partial orders are described.

No matter which partial order is adopted, in our view, a program will be represented by a collection of partial orders, each defining a set of (interleaved) execution sequences. For each partial order there can exist several interleaving sequences (paths) which are completions of that partial order to a total order. The abstraction of a distributed system does not allow us to prove that one path is more correct than another (we do not depend on a global clock). This is a nondeterminism that is due to the execution of unrelated events in different processes. On the other hand, paths that are different because of choosing a different continuation out of several nondeterministic choices inside a process have different event sets and are completions of different partial orders. We can sometimes say that a distributed program is correct under a given specification if for each partial order, there exists a path that satisfies the specification. Using the grouping into substructures and the branching modalities, we will show when it is justified to apply this correctness criterion to distributed programs.

This approach, similar to the semantic view suggested in [22, 33, 38, 39], has not previously served as the basis of a global temporal logic. Rather, an assertion in linear temporal logic (denoted LTL [30, 31]) is easiest understood as having a set of (nonbranching) sequences as models, each one being a structure. This logic expresses correctness properties such as liveness and partial correctness. An LTL formula is a path formula. That is, it makes an assertion about an interleaved path. It has the feature of always talking about "all the executions" of a program at the same time. In other words, a formula is valid for a certain program iff it is true in
each of the structures (which are interleaved paths) which its possible executions generate.

An assertion in branching temporal logic (denoted by UB, CTL or CTL\* [4, 9, 15, 14]) is modeled by a single DAG structure, where we can express the possibility that a program will choose a given alternative. Therefore properties like "there can be an execution that terminates with $x = 2$" may be expressed. A UB or CTL formula is a state formula, that is, it makes an assertion about all the possible continuations from a global state. A formula is said to be valid for a program if it is true in every global state of the DAG structure generated by its possible executions.

Partial order temporal logic (denoted POTL [36]) allows treating the relative precedence of local events in a single partial order execution. It is meaningless to talk in POTL about global states, since the logic reasons directly on the partial order and does not generate the global states. Although some global assertions can be "simulated" by assertions about the local states of the partial order, this is not true for many global assertions.

ISTL inherits some of the features of each of these logics. From LTL we take the transition between global states and the property that a formula is satisfied by a structure iff it is satisfied in any (partial order) execution. The branching structure is clearly from UB or CTL, as is the ability to choose between alternative continuations. From POTL, the view of a single partial order execution as a stand-alone (sub)structure is adopted.

By dividing the structure into such substructures and using the branching modalities we are able to express properties that the other temporal logics can not. Among the new properties that we can now consider are: the specification of the distributed snapshot algorithm for finding the global states of a distributed system as seen in [8], the behavior of communication closed layers [13], or the temporal semantics for distributed languages like CSP.

The rest of this paper is organized as follows. In Section 2 a short description is given of the two models of distributed computation, as a partial order among events and as an interleaving, and the semantic model is defined. In Section 3 the model of partial order is used to define a structure in the sense of logic and set theory, and a formal definition is given for the syntax and semantics of the new logic. In Section 4 the expressive power of the new logic is compared with that of some existing ones through a few examples. Section 5 deals with deduction in the new framework. In Section 6 correctness criteria for distributed programs are examined, and the subject of giving a specification and correctness proof for a distributed program in the new logic is discussed. In Section 7 we present some applications which demonstrate the utility of ISTL. Section 8 gives some conclusions.

2. Preliminaries

The underlying semantics of distributed programs is clearly dependent on the set of operations defined. In addition to standard operations such as assignment, testing,
and local control, the set of operations will depend on whether, e.g., there are a variable or a fixed number of processes (that is, fork and join operations to spawn or terminate processes are either allowed or disallowed), synchronous or asynchronous communication is present, or communication is defined only between two processors or is generalized to $N$-way communication. In order to abstract away from these issues, we will assume that for each model a set of primitive operations is given.

There are at least two ways to connect the possible executions of a distributed program to partial orders. In the first one, the entire program is considered as defining one large partial order (or branching semantics) of all possible choices. This is the classic branching view. In the other view, whenever there is an explicit nondeterministic choice in the code of the program, a single (partial order) execution will include only the specific choice made in that execution. Thus in general there will be many partial orders associated with a program. We will adopt and further explain this latter view. Both of the above are contrasted with the modeling of a program as a collection of sequences, i.e., as a total order (interleaving) of the events. An excellent description of these views appears in [22].

The following definitions are aimed at defining a general semantics for various program models. The semantic model is related to several works, among them, the partial order semantics for CSP given in [38], the traces model of Mazurkiewicz [33], and the event structures of Winskel [39].

**Definition 2.1.** A snapshot is an assignment of values to variables according to their domains. The set of variables assigned by the snapshot $s$ is denoted by $\text{vars}(s)$.

The domains of variables used are dependent upon the specific model of computation. For example, in most programming languages, the variables are real, integer or boolean. The variables may also include those with values from the domains of program counters, communication queues, tokens in places (firing conditions) of Petri nets [35], etc.

**Definition 2.2.** An operation is a transformation from snapshots of some fixed set of variables to snapshots of another (possibly identical) fixed set of variables. For each operation $\tau$, let $\text{prevars}(\tau)$ and $\text{postvars}(\tau)$ denote these two sets of variables. A program is a set of operations. The set of variables $\text{vars}(P)$ associated with a program $P$, is the union of the variables associated with all its operations (i.e., $\text{vars}(P) = \bigcup_{\tau \in P} \text{prevars}(\tau) \cup \bigcup_{\tau \in P} \text{postvars}(\tau)$).

When the text of the program is composed of processes, certain variables can be considered as local, i.e., accessed by the operations associated with the code of a single process. In a synchronous communication model, a communication event is mutual to two (or more) processes. Thus, such an operation might access variables
of both processes. The mapping of each specific computational model into the definition of operations is simple and its technicalities should be carried out before specifying or verifying programs.

Definition 2.3. Events are single executions of operations. That is, there exists a function $M_P$ associated with a program $P$ which maps each event to a single operation. Each event $e$ is associated with a presnapshot and postsnapshot such that the operation $M_P(e)$ transforms $\text{presnapshot}(e)$ to $\text{postsnapshot}(e)$. (Therefore, $M_P$ maintains the sets of variables, i.e., $\text{vars}(\text{presnapshot}(e)) = \text{prevars}(M_P(e))$, and $\text{vars}(\text{postsnapshot}(e)) = \text{postvars}(M_P(e))$.)

The ordering among events is also a part of the basic semantics of the model. For example, among events that occur in the same process during a single execution, there exists a natural total order of occurrence. Apart from that, the event of sending a message in one process precedes the event of receiving it in another (or in a synchronous communication model, this is done within a single joint event). Other natural orderings are defined for events such as join and fork. The transitive closure of this irreflexive relation among events is a partial order that is taken as the model of computation in the partial order approach. This partial order, which is due to Lamport [22] is only one possibility, although it is natural and popular among researchers.

Another reasonable partial order may be defined which takes into account only real causality among events. In such a partial order, two sequentially executed events can be considered as unordered if they do not affect each other (and their relative order is not an interesting factor in the description of the program). This partial order, which can be called the essential partial order, is weaker than the one discussed previously, but still expresses the temporal relations which are required by the underlying semantics.

Although the partial order model is mainly discussed in connection with distributed systems [7, 22, 26, 38], it is also a useful tool for shared memory parallel programs [5]. One way to introduce shared variables into the execution model is by adding the following constraint: in each partial order, each shared memory location imposes a total order among the events which reference it (both read and write). Because of the total order among the events of accessing a shared variable, one can view each of the shared variables as a process by itself. Each of the events of this process is mutual to another process (the one that reads or writes) similarly to a synchronous communication between processes. Therefore, nondeterminism in the access to a shared variable creates alternative events that belong to different executions. Again, this is only one proposed semantics for using a shared variable. A different underlying semantics may allow, for example, concurrent reading.

The choice of a particular construction of partial order to be used later in the definition of the logic is orthogonal to the following definitions and the logic in the next section. This choice will follow from the semantics. In the continuation, we
assume that the partial order at least includes the essential partial order described above.

**Definition 2.4.** Let \( AFF(v, R) = \{ f : f \in R \land v \in \text{vars}(\text{postsnapshot}(f)) \} \) where \( R \) is a set of events. That is, \( AFF \) is the subset of events of \( R \) which may affect the value of the variable \( v \). If the set of events of \( AFF \) are totally ordered by the relation \( < \), define \( VAL(v, R) \):

\[
VAL(v, R) = \begin{cases} 
\text{postsnapshot}(\max(AFF(v, R)))(v) & \text{if } AFF(v, R) \neq \emptyset, \\
\theta(v) & \text{otherwise.}
\end{cases}
\]

That is, if \( AFF(v, R) \) is not empty, \( VAL(v, R) \) is the value assigned to \( v \) by the maximal event in \( R \) which has \( v \) in the set of variables it affects; otherwise, it is the initial value \( \theta(v) \).

**Definition 2.5.** A partial order execution is a triple \( \text{POEX} = (E, <, \theta) \), where \( < \) is a partial order over the set of events \( E \), and \( \theta \) is a snapshot where \( \text{vars}(\theta) = \text{vars}(P) \). The snapshot \( \theta \) represents the initial value from which the execution begins. It may assign the value \( \bot \) (undefined) to some of the variables. Partial order executions must satisfy various semantical restrictions (dictated by the programming model under consideration), which include at least the following:

1. The set \( E \) is countable.
2. No event has an infinite number of predecessors (that is, for each \( e \in E \), the set \( \{ f : f \in E \land f < e \} \) is finite.
3. If for \( e_1, e_2 \in E \), \( \text{vars}(\text{postsnapshot}(e_1)) \cap \text{vars}(\text{postsnapshot}(e_2)) \neq \emptyset \), then either \( e_1 < e_2 \) or \( e_1 > e_2 \). That is, if two events affect the same variable, they are ordered. In addition, since we assume the essential partial order, if one event uses a variable and another event changes the same variable, the two events are also ordered by \( < \).
4. Let \( v \in \text{vars}(\text{presnapshot}(e)) \). Then, the value assigned to it by \( \text{presnapshot}(e) \) is consistent with the latest update of \( v \). That is, \( \text{presnapshot}(e)(v) = VAL(v, \{ f : f < e \}) \). Notice that \( VAL(v, R) \) is well defined because of (3).

The precedence relation among events represents exactly what our abstraction of distributed systems allows us to infer about the order of events. If two unrelated (according to the relation \( < \)) events \( e_1, e_2 \in E \) were executed, no conclusions may be reached about their relative order of execution as long as no global clock is given. Two such events are said to be concurrent.

A global state does not exist directly in the partial order model. In order to incorporate such states, for each possible partial order \( (E, <) \) the following terms [22] may be defined.

**Definition 2.6.** A slice \( S \) is a downward-closed finite subset of \( E \). That is, for each pair of events \( x \) and \( y \) in \( E \), such that \( x < y \), if \( y \in S \) then \( x \in S \). Let \( \text{SLICES(POEX)} \)
be the set of all slices defined on \( \text{POEX} \). Slices are considered to represent the global states of the program.

The notion of a slice will be used as a state of a program in the sense that without a global clock, no one can disprove the claim that the program really passed through a synchronous interval of time in which exactly all the events in the slice have already executed and the values of all the variables (and set of messages in transit) agree with the related snapshot. This is similar to choosing between different observers moving in constant speeds relative to each other in modern physics.

While slices are convenient in the definition of the semantic model, and in defining properties of the logic, in most cases, specification and verification of a program will need only a subset of the information about the values of variables, program counters and queues. For this reason, the following definition extracts from each slice the information referring to program variables.

**Definition 2.7.** The snapshot characterizing a slice \( S \) called the **global snapshot** of \( S \) (denoted \( \xi_S(S) \)) assigns values to each of the program’s variables (i.e., \( \text{vars}(P) \)). The value assigned by \( \xi_S(S) \) to a variable \( v \) agrees with the latest change of \( v \) appearing in \( S \). That is, \( \xi_S(S)(v) = \text{VAL}(v, S) \).

In many definitions of temporal logic, the term “global state” refers to the above definition of a snapshot characterizing a slice. Recall from Definition 2.6 that this is not the case here, where a global state is identified with a slice. Thus, it is possible that two or more different global states of a program have exactly the same snapshots in a single program execution. For example, a loop inside a program may cause the set of variables (propositions, sets of messages) to repeatedly have the same values during a single execution. However, different global states occur because the set of events accumulated into the appropriate slices is extended each time events are executed. Thus different slices may be characterized by the same snapshot.

**Definition 2.8.** Let \( \rho \subseteq (\text{SLICES}(\text{POEX}) \times \text{SLICES}(\text{POEX})) \) be a relation such that \( s \rho t \) iff \( s \preceq t \). That is, \( s \) has fewer events than \( t \) and can be said to **precede** \( t \). Note that \( s \) and \( t \) must be finite by the definition of \( \text{SLICES}(\text{POEX}) \).

**Definition 2.9.** Let \( \text{TRAN}(\text{POEX}) \subseteq \rho \) be the **transition relation** between slices such that

\[
\text{TRAN}(\text{POEX}) = \{(s, t) : s \rho t \land \neg \exists r (s \rho r \land r \rho t) \lor (s = t \land \neg \exists r s \rho r)\}.
\]

That is, two global states \( s \) and \( t \) are related by \( \text{TRAN}(\text{POEX}) \) if \( t \) is different from \( s \) by the execution of a single additional event or if they are both the same maximal slice according to the relation \( \rho \) [38]. If \( (s, t) \in \text{TRAN}(\text{POEX}) \), \( s \) is said to **immediately precede** \( t \) (\( \rho \) is the transitive closure of \( \text{TRAN}(\text{POEX}) \)).
The relation \( \text{TRAN}(\text{POEX}) \) between global states generates a branching structure (DAG) because for a single state, we generally have more than one successor. The alternative possible successors are caused by the occurrence of unrelated (according to the partial order) events in different processes and not by nondeterminism inside a process.

According to the *interleaving view*, we look at sequences of events. Each sequence is a total order among the (beginning of the) events. The interleaving model can describe an idealization of a reality in which there is a global clock and the events are "timeless", or an implementation of processes using multiprogramming, with no actual parallel execution. Note that if events are not instantaneous and can overlap in time, then none of the interleavings need describe what "actually" occurred. The model is justified by the assertion that the program will behave "as if" one (or more) of the interleavings represents reality.

**Definition 2.10.** A single maximal sequence \((s_0, s_1, s_2, \ldots)\) such that \(\forall i \geq 0 \ s_i \in \text{SLICES}(\text{POEX}) \land (s_i, s_{i+1}) \in \text{TRAN}(\text{POEX})\) is called an interleaving sequence or a path. (Maximality of a sequence means that it is not a proper prefix of another sequence.) A finite and contiguous portion of such a path will be called in the sequel a finite path.

An interesting property of the relation \(\rho\) that will be used later is as follows.

**Proposition 2.11.** If \(s \rho t\) (thus \(s \leq t\) and \(s\) and \(t\) are finite) then there is a finite path from \(s\) to \(t\).

**Proof.** Consider \(s\) and \(t\) as sets of events, then by acyclicity of the partial order (since it is transitive closed and irreflexive) there is a minimal event \(e\) in \(t - s\). Since all its predecessors are included in \(s\), \(s' = s \cup \{e\}\) is a slice. The rest of the proof follows by a simple induction on the finite set \(t - s\). \(\square\)

**Definition 2.12**

\[
\text{PATMS}(\text{POEX}) = \{ (s_0, s_1, s_2, \ldots) : \forall i \geq 0 \ s_i \in \text{SLICES}(\text{POEX}) \land (s_i, s_{i+1}) \in \text{TRAN}(\text{POEX})\}.
\]

It can be easily seen [22, 38] that the set of paths generated by the relation \(\text{PATMS}(\text{POEX})\) is exactly the set of interleaved sequences which are related to a single (partial order) execution. Note that such a relation is constructed separately for each partial order execution.

**Definition 2.13.** An acceptable path \(x\) on a partial order execution \(\text{POEX} = (E, <, \theta)\) satisfies the condition that for each event \(e \in E\) there exists a global state \(s\) on \(x\) which (as a slice) contains \(e\). This means that each event from \(E\) appears eventually on each acceptable path. Let \(\text{ACCEPTABLE}(\text{POEX})\) be the set of acceptable paths on \(E\) from \(\text{PATHS}(\text{POEX})\). This will also be called an interleaving set.
The criterion of being acceptable (which is called just in [38]) differs from the usual fairness definitions [28,17] in that the partial orders are already assumed given from the underlying semantics, and these already may or may not be "fair" according to the other definitions.

The following proposition appears without a proof in [38].

**Proposition 2.14.** For PATHS((E, <, O)), given that (1) E is countable and (2) for each e \( \in \) E, the set \( \{ f : f \in E \land f < e \} \) is finite, it follows that the set of acceptable paths is not empty.

**Proof.** We first show that any finite path \( p \) can be always extended to include an event \( e \) which does not appear in any slice of \( p \). Consider the finite set \( S = \{ f : f \in E \land f < e \} \). This finite set is a slice by Definition 2.6. Consider the union of \( S \) with the last slice in the finite path \( p \). A union of two slices is a slice, and by Proposition 2.11 there is a finite path between the last slice in \( p \) and the union. This path can then be further extended to include \( e \).

Now, since the set of events is countable by (1), consider the path which is generated by always extending it in the following way: given any finite prefix, choose the next smallest event (according to some enumeration) which is not yet in any slice in the current prefix. Then extend the last slice in the prefix to contain this event. This path will be acceptable. □

**Definition 2.15.** A set of sequences is suffix closed [1, 14] if every suffix of a sequence which appears in the set is also in the set.

**Definition 2.16.** A set of sequences is fusion closed if whenever \( x_1 y_1 \) and \( x_2 y_2 \) are sequences in the set (where \( x_i \) is a prefix of a sequence, \( s \) is a state and \( y_i \) is a suffix of a sequence) then \( x_1 y_2 \) is also a sequence in the set.

A structure which allows the sequence quantifiers to range over a semantically defined set of sequences which is suffix and fusion closed is called an Abrahamson structure [1, 10].

**Proposition 2.17.** The set of paths that constitute an interleaving set is suffix closed and fusion closed.

**Proof.** Follows easily from the fact that if a path is acceptable, then all of its suffixes are also acceptable. □

### 3. Towards the new TL

In this section, a formal definition of the syntax and the semantics of ISTL is given and a few possible extended logics are briefly described. The modals are those
of CTL [15] but we build the interleaving sets into the semantics. For simplicity, we define a propositional ISTL.

**Definition 3.1 (Syntax of ISTL).** Define \( P \), a set of atomic propositions:

1. For each proposition \( p \in P \), \( p \) is in ISTL.
2. If \( Q, W \) are in ISTL then \( Q \land W, Q \lor W \) and \( \neg Q \) are in ISTL.
3. If \( Q \) is in ISTL then \( AGQ, AFQ, EGQ, EFQ, AXQ \) and \( EXQ \) are in ISTL.
4. If \( Q \) and \( W \) are in ISTL then \( E[Q \cup W], A[Q \cup W] \) are in ISTL.

We call the following symbols sequence modalities: \( G = \text{"always"}, F = \text{"sometimes"}, X = \text{"next"} \) and \( U = \text{"until"}. \) The sequence quantifiers are: \( A = \text{"on every sequence"}, E = \text{"there exists a sequence"}. \)

Define \( \text{first}(x) \), the first state in a sequence \( x \). \( x^i \), the suffix of the sequence \( x \) starting from its \( i \)th state (\( x \) begins with \( s_0 \)).

We first define the semantics for a single substructure and then define the semantics for the collection of substructures.

**Definition 3.2.** A substructure \( M \) has the form \( (\text{POEX}, L) \) where \( \text{POEX} = (E, <, \theta) \) is a partial order execution whose set of slices is \( \text{SLICES(POEX)} \) and \( L \) is an assignment function \( L: \text{SLICES(POEX)} \times P \rightarrow \{\text{true, false}\} \).

Let \( IM = \text{ACCEPTABLE(POEX)} \) and \( SM = \text{SLICES(POEX)} \) denote the interleaving set and the set of slices, respectively, associated with the substructure \( M = (\text{POEX}, L) \).

Now the satisfaction relation is defined for a single substructure.

**Definition 3.3.** A substructure \( M = (\text{POEX}^M, L^M) \) and a slice \( s \in SM \) satisfy a formula \( f \) (written \( s \models_M f \)) iff:

1. \( s \models_M p \) for \( p \in P \), iff \( L^M(s, p) = \text{true} \);
2. \( s \models_M Q \land W \) iff \( s \models_M Q \) and \( s \models_M W \);
3. \( s \models_M Q \lor W \) iff \( \neg s \models_M Q \lor \neg s \models_M W \);
4. \( s \models_M \neg Q \) iff \( s \models_M Q \);
5. \( s \models_M AGQ \) iff for each sequence \( x \in IM \) with \( \text{first}(x) = s \), for each \( i \geq 0 \), \( \text{first}(x^i) \models_M Q \);
6. \( s \models_M AFQ \) iff for each sequence \( x \in IM \) with \( \text{first}(x) = s \), there exists \( i \geq 0 \) such that \( \text{first}(x^i) \models_M Q \);
7. \( s \models_M EGQ \) iff there exists a sequence \( x \in IM \) with \( \text{first}(x) = s \) and for each \( i \geq 0 \), \( \text{first}(x^i) \models_M Q \);
8. \( s \models_M EFQ \) iff there exists a sequence \( x \in IM \) with \( \text{first}(x) = s \) and there exists \( i \geq 0 \) such that \( \text{first}(x^i) \models_M Q \);
9. \( s \models_M EXQ \) iff there exists a sequence \( x \in IM \) with \( \text{first}(x) = s \) and \( \text{first}(x^i) \models_M Q \).
10. \( s \models_M AXQ \) iff for each sequence \( x \in IM \) with \( \text{first}(x) = s \), \( \text{first}(x^i) \models_M Q \).
11. \( s \models_M E[Q \cup W] \) iff there exists a sequence \( x \in IM \) with \( \text{first}(x) = s \) and there exists \( i \geq 0 \) such that \( \text{first}(x^i) \models_M W \) and for each \( 0 \leq j < i \), \( \text{first}(x^j) \models_M Q \).
(4b) \( s \models_{M} A[Q \cup W] \) iff for each sequence \( x \in \text{T}^M \) with first\((x) = s \), there exists \( i \geq 0 \) such that first\((x^i) \models_{M} W \) and for each \( 0 < j < i \), first\((x^j) \models_{M} Q \).

**Definition 3.4.** \( \models_{M} f \) iff \( s \models_{M} f \) for each \( s \in S^M \). (Many typical formulas will be trivially true for all but initial states by adding \( \text{at}(\text{START}_i) \) as a conjunct for each process \( i \) to the left of an implication.)

**Definition 3.5 (Semantics).** An ISTL structure is a triple \((V, W, F)\) where \( V \) is a set of worlds, \( W \) is a set of pairs \( M = (P \cdot O \cdot E \cdot X^M, L^M) \), each of which is a substructure as defined above, and \( F \) is a function \( F : V \rightarrow W \). Let \( \mathcal{A} \) be an ISTL structure \((V, W, F)\). An ISTL structure \( \mathcal{A} \) satisfies a formula \( f \) (written \( \mathcal{A} \models f \)) iff \( \models_{F(v)} f \) for each \( v \in V \).

The semantic implication \( \Sigma \models f \), where \( \Sigma \) is a set of ISTL formulas, \( f \) is a single ISTL formula and \( \mathcal{A} \) is an ISTL structure, is as follows.

**Definition 3.6.** \( \Sigma \models f \) iff for every structure \( \mathcal{A} \), whenever \( \mathcal{A} \models \Phi \) for each \( \Phi \in \Sigma \), then \( \mathcal{A} \models f \).

We do not need all sequence modalities \( G, F, U \) and \( X \) as there is a simple translation which uses semantical equivalences and shows that one needs only \( U \) and \( X \) [9].

The assignment function \( L \) for each substructure assigns values to any propositional variable in any slice. It is common to be interested only in the global snapshot which characterizes each slice (Definition 2.7). In many cases, \( L \) actually assigns to each slice its characterizing snapshot. Nevertheless, the definition of \( L \) is general and allows a predicate to be dependent upon the entire slice (for example, a predicate which is true when the number of events in a slice is even). When a fixed number of processes is considered, the extension to a first order logic can be done just as for linear temporal logic [30]. Of course, for a first order logic, the assignment function assigns values to each of the program variables. This contrasts with PETL where the extension to first order is more difficult because of the nonavailability of some variables in local states.

**Example 3.7.** Let \( \mathcal{A} \) be an ISTL structure with set of substructures which are the partial orders and event sets obtained from the execution of a distributed program \( PR = [P_1 \parallel P_2] \). Assume that \( Q \) is a proposition on global states. The predicate \( \text{at}(l) \) which will be used in this and the following examples is true in global states in which the program's control in the process containing the label \( l \) is just before executing the commands associated with \( l \). That is, a command which semantically precedes the command labeled with \( l \) has completed, and the one labeled with \( l \) has not yet started. This can also be seen as if \( L \) assigns the value \( l \) to the program counter of the process which contains the label \( l \). In the sequel, we assume that the
command to be executed first on a process numbered \(i\) is labeled with \(\text{START}_i\). Thus, \(\text{at}(\text{START}_1) \land \text{at}(\text{START}_2)\) means that the controls of \(P_1\) and \(P_2\) are at the beginning of the process. (That is, the global state is the empty slice.) Let \(f\) be the formula \((\text{at}(\text{START}_1) \land \text{at}(\text{START}_2)) \rightarrow \text{EF}Q\). The notation \(\text{at}(S)\) for a program section \(S\) is defined as if there were an implicit label preceding \(S\). The notation \(\text{after}(S)\) is defined similarly.

The meaning of \(\mathcal{A} \models f\) is: for every substructure \(M \in W\) and for every slice \(s\) appearing on some path of \(I^M\), it holds that \(s \models_M f\). According to Definition 3.3, \(s \models_M f\) means: if \(s\) is a state that satisfies \((\text{at}(\text{START}_1) \land \text{at}(\text{START}_2))\) then there exists a path in \(I^M\) starting with \(s\) on which there is a state which satisfies \(Q\). Thus, the meaning of \(\mathcal{A} \models f\) is “for every partial order execution, there exists a path from the global initial state that reaches a state which satisfies \(Q\)”.

The expressive power of ISTL depends on the set of modalities chosen. Restricting the set of modalities from the previous definition to various subsets generates a hierarchy of logics similar to that in [15]. ISTL* may be defined with the syntax of CTL* [15] but again building the interleaving sets into the semantics. The difference between the logics is that ISTL forces path quantifiers \(\exists\) and \(\forall\) to be followed by exactly one of the path modalities \(F\), \(G\), \(X\) and \(U\) (which are the same as the LTL modalities \(\Diamond\), \(\Box\), \(\Diamond\) and \(\text{Until}\)) while in ISTL* no such restriction is required. In later sections, some examples from ISTL* are given.

By Definition 3.5, a set of structures \(\mathcal{A}\) will satisfy a formula \(f\), if each one of the structures in \(\mathcal{A}\) satisfies \(f\). Therefore, if a set of structures represents all the executions of a program, a formula here will be considered valid in a program if it is satisfied by each of its partial order executions (i.e., by each interleaving set). This is a property of “always talking about all the executions”, similar to LTL, and opposed to UB or CTL which can talk about the existence of an execution. Therefore, we cannot, for example, express in ISTL the existence of a single path with some property, and instead must say that each interleaving set has a path with the property.

A possible extension which makes it possible to express the existence of a partial order execution is considered below in a version called QISTL (QISTL*). A new level of quantifiers which allows both expressing “for every partial order” and “there exists a partial order” is added. The Q stands for “quantified” because this logic allows quantification of ISTL formulas over the set of all partial order executions.

**Definition 3.8 (Syntax of QISTL).** (1) If \(Q\) is in ISTL, then \(\exists Q\) and \(\forall Q \in \text{QISTL}\).
(2) If \(Q, W\) are in QISTL then \(Q \land \bar{W}, Q \lor W\) and \(\neg Q\) are in QISTL.

**Definition 3.9 (Semantics).** Let \(\mathcal{A}\) be a structure \((V, W, F)\).
(1a) \(\mathcal{A} \models \exists Q\) iff there exists \(v \in V\) such that \(\models_{F(v)} Q\).
(1b) \(\mathcal{A} \models \forall Q\) iff for each \(v \in V\) it holds that \(\models_{F(v)} Q\). (By the previous semantics, this was written as \(\mathcal{A} \models Q\).
(2a) \(\mathcal{A} \models Q \land W\) iff \(\mathcal{A} \models Q\) and \(\mathcal{A} \models W\).
(2b) $A \models Q \lor W$ iff $A \models Q$ or $A \models W$.
(2c) $A \models \neg Q$ iff not $A \models Q$.

Now we can write assertions about the existence of a partial order and event set in which a temporal property holds. For example, to express the fact that there exists an execution in which eventually $x = y$, we may write $\exists (\mathit{at}(\mathit{START}) \rightarrow \mathit{EF}x = y)$, which means "there exists a (partial order) execution in which one of the paths leads to a global state in which $x = y$". As another example, consider the QISTL formula $\exists \mathit{EfTerminate} \land \exists \mathit{AG} \neg \mathit{Terminate}$ which asserts that the program has executions that terminate as well as executions that do not terminate.

4. Connection with other logics

In this section, an example of constructions of Kripke structures for various logics are presented and compared. It is clear that grouping the paths into interleaving sets allows us to assert new properties which are based on the underlying partial order. We will demonstrate this in a pair of examples.

Example 4.1. Consider the CCS [34] expression "a ; b + b ; a" which means that the system chooses nondeterministically either to execute sequentially a and then b or to execute b and then a, and the expression "a || b" which means that a and b are executed concurrently. With respect to linear and branching structures these expressions are indistinguishable, as depicted in Fig. 1. Recently, it has been argued [6,11], that one should distinguish between nondeterminism and concurrency. If we assume the usual partial order with a total ordering of sequential events, as depicted in Fig. 2, a ; b + b ; a has two singleton interleaving sets. On the other hand, a || b has a single interleaving set consisting of two interleavings.

Assume that the predicate $a$ asserts that the event occurred and similarly $b$ asserts that $b$ occurred. The formula $\neg a \land \neg b \rightarrow \mathit{EX}(\neg a \land b)$ asserts that in every partial order execution, the event $b$ is immediately executed after the global state in which none of the events has been executed. This formula is satisfied by the single interleaving set of $a || b$, but not by one of the interleaving sets of $a ; b + b ; a$. Therefore, the ISTL formula distinguishes between these CCS expressions.

![Branching and linear structures for a ; b + b ; a and a || b.](image)
Obviously, any logic based on the linear or branching structure cannot have a formula distinguishing between the expressions.

**Example 4.2.** It is impossible to assert in POTL directly on the global states [21]. On the other hand, sometimes it is possible, though awkward, to express global properties in POTL by combining local assertions. However, we will see that propositional ISTL suffices to express many properties, while for the same properties it is necessary to transfer into first order POTL.

Assume that the two variables $x$ and $y$ belong to different processes and range over the natural numbers. Using ISTL, one can define a proposition $R$ which is true exactly in states where $x > y$. However, in POTL that collection of global states must be identified by using boolean and temporal operators to combine assertions about the local states. It is sufficient to show that there is no set of locally definable propositions which can be combined using boolean operators to assert that $x > y$. Assume to the contrary that there is such a propositional formula $Q$ which is true exactly when $x > y$. Using a boolean combination of a finite set of propositional variables, each describing a property of $x$ (or $y$), it is only possible to classify the set of possible local values of $x$ (or $y$) into a finite set of classes. Thus, $Q$ cannot distinguish between global states having different values of the variable $x$ (or $y$) that belong to the same class. Since the domain of the variables $x$ and $y$ is the natural numbers, there exist at least two infinite classes: $S_1$ of $x$ values, and $S_2$ of $y$ values. Pick any value $a$ in $S_2$. Since $S_1$ is infinite, there must be some value $b$ in $S_1$ greater than $a$. By the same reasoning, there is some $c$ in $S_2$ which is greater than $b$ in $S_1$. By construction, $Q$ must give the same result for the relation between $a$ and $b$, and the relation between $b$ and $c$. This contradicts the fact that $Q$ expresses the property $x > y$.

In order to show that ISTL* strictly includes linear temporal logic, we first show how to embed a linear assertion in ISTL*.

**Definition 4.3.** A set LPF of linear path formulas is a subset of the ISTL* formulas defined as:

1. For each atomic proposition, $p \in P$, $p \in$ LPF;
2. If $Q \in$ LPF then $GQ$, $FQ$, $XQ$, $\neg Q \in$ LPF;
3. If $Q$, $W \in$ LPF then $Q \land W$, $Q \lor W$, $Q \lor W \in$ LPF.
We will note by $L'$ the LPF $L$ transformed into an LTL formula where every $G$ is changed into $\square$, every $F$ is changed into $\Diamond$ and every $X$ is changed into $\bigcirc$.

To compare LTL with ISTL*, notice that in LTL a formula is checked against a set of all the linear structures generated by a program. Consider then a set of LTL structures $B$ which includes all the paths from the interleaving sets of an ISTL* structure $A = (V, W, F)$. That is, $B = \bigcup_{M \in F(V)} I^M$. Satisfaction of an LTL formula $L$ over $B$ is defined such that $B \models L$ iff each of the sequences (paths) in $B$, taken as a linear structure, satisfies $L$. It is immediate from the semantic definition of LTL and ISTL* that the following connection holds.

**Proposition 4.4.** $B \models L' \iff A \models AL$.

(A similar proposition referring to CTL* is proven in [15].) Therefore, it is implied that any verification method used for LTL (e.g. [31, 32]) is applicable also to ISTL*, although of course it will not be complete for ISTL*.

5. Deduction in the new framework

In order to use the variants of ISTL for proving correctness of programs, one needs a sound (and preferably complete) deductive system. The axioms and consequence rules used for verifying a program in the logic can be divided into several categories [32]:

1. those axioms and consequence rules that stem directly from the definition of the logic and set of models;
2. those axioms and consequence rules that follow from the syntactical structures of a specific program;
3. the axioms that are particular to the domain of interest (e.g., integers, strings, trees).

It is convenient to look at each part of the deductive system as an “increment” to the consequence rules and axioms.

First we consider the rules from the basic definition of the logic. From Proposition 2.17 it follows that the ISTL substructures are a subset of Abrahamson structures. Therefore, any valid CTL formula which is valid over Abrahamson structures is also valid for ISTL. However, since ISTL substructures are a proper subset of Abrahamson structures, a sound and complete axiom system for CTL under Abrahamson structures is sound for ISTL but not complete. In our context, the additions which follow from the special structures must be considered. The same connection holds between CTL* and ISTL*. In [10] it is proven that a CTL* formula is satisfiable over the class of Abrahamson structures if it is satisfiable over probabilistic structures [29]. Therefore, the sound and complete deductive system for probabilistic structures in [29] is also sound for ISTL*.

Most deductive systems [4, 16] are given for an R-generable definition of the structures rather than for Abrahamson structures. The R-generable structures are
also a subset of Abrahamson structures but differ from the set of interleaving sets. The R-generable structures satisfy a semantical property called limit closure [14] in addition to the suffix and fusion closure that the Abrahamson structures satisfy. Therefore, the set of tautologies for R-generable structures is a superset of those for Abrahamson structures. Extending the set of axioms from [29] into a complete deductive system for CTL* over the R-generable structures is an open problem. In much the same way, one might like to add to the deductive system of [29] enough axioms and consequence rules in order to obtain a sound and complete system over the class of interleaving sets. For example, the consequence rule $p \rightarrow AGp \vdash EFp \rightarrow AFp$ should be added to the deductive system. This will be justified semantically using the next theorem.

**Theorem 5.1.** The following formula is a semantic implication both in ISTL and in ISTL*:

$$(p \rightarrow AGp) \vdash (EFp \rightarrow AFp).$$

**Proof.** For each ISTL substructure $M$, assume that for each slice in each interleaving set it holds that $p \rightarrow AGp$. Such a property $p$ is called a stable property [8, 7]. Choose a slice $s \in S^M$ such that $s \vdash_{M} EFp$ holds. Then, one semantically concludes that $s \vdash_{M} EF(AGp)$. Given that "there is a path (call it $p1$) in $I^M$, starting from $s$, on which there exists a slice (call it $t$) such that any descendant slice of $t$ satisfies $p$", it must be proven that "for each path in $I^M$, starting from $s$, there is a slice satisfying $p$”. Take any path $p2$ in $I^M$ which also starts at $s$. Since the set of paths satisfy the acceptability property from Definition 2.13, it follows that there is a slice $r$ on $p2$ which contains all the events of $t$ (in addition it may contain some more events). From Proposition 2.11 it follows that $r$ is a descendant of $t$. Since by the stability of $p$, any descendant of $t$ satisfies $p$, then $r$ satisfies $p$. That is, $s \vdash_{M} AFp$. □

A second category of rules should be added. Clearly, in verifying programs over a given programming language, it is necessary to have a set of consequence rules and axioms which relate the program syntax with the logic. Part of these rules can be given in a “generic” way, not referring to specific variables or labels of a program. An example is the proof rule to be seen in the example in Section 7.1. Another source for deduction rules comes from a specific program, by giving a formula which specifies some properties of the program (which is called the program part in [32]).

Finally, the third source for deduction rules is the domain of interest. When defining a first order version of a logic over a specific domain (the integer numbers for example), one must include (nontemporal) consequence rules and axioms which define the semantics of the relations and functions of this domain. These issues, which are orthogonal to the particular features of ISTL, will not be considered further here.
6. Distributed specification

One of the features of the new TL is in expressing properties which make use of the ability to choose one path out of all the paths that correspond to a single execution. Recall that there is no way to externally determine which of these paths, if any, actually occurred, since a global clock is not used. Thus, we are free to choose one that satisfies our specification. A criterion for distributed correctness under a temporal specification might therefore be the existence of a path in every execution which satisfies the desired property. For some properties, for example the total correctness of a distributed program, this criterion is sufficient as will be proven in the sequel. For other purposes, the use of such an approach depends on the way the program is abstracted.

Theorem 5.1 can be used in concluding that a property is satisfied by all the (acceptable) paths of an execution of a given program even though the property is proven only for a single path of that execution. It explains why it is common (as will be demonstrated in the next section) to reason about a possible execution sequence out of several “equivalent” ones [27] (generated from the same partial order) or to use techniques such as “flipping” between independent events [12] and “atomizing” together sequential events [3]. These proof techniques follow a uniform pattern: first prove that a property P is stable, then show that there is an interleaving sequence out of every partial order execution in which P is eventually satisfiable and finally deduce that P occurs eventually on every interleaving sequence. The example of Section 7.1 demonstrates how a temporal deductive system (similar to [3]) uses this property. A proof rule which is based on proving a total correctness property for one path is used to conclude total correctness for all the paths.

Although the use of Theorem 5.1 provides an important tool for verification, there are other ways to exploit the properties of ISTL with formulas of special types. Combining Definition 4.3 dealing with linear path formulas and the next proposition is helpful when dealing with properties about a subset of the program variables which are local to a certain process:

**Proposition 6.1.** For any linear path formula $L \in \text{LPF}$, if (a) $L$ does not contain the $\text{X(NEXT-TIME)}$ modal and (b) all of the events which can affect the logical values of the atomic propositions (or the variables in a first order extension) of $L$ are totally ordered (e.g., events that occur in the same process or events that change the value of a single physical location), then $E L \rightarrow A L$.

**Proof.** With respect to propositions or variables that can be changed only by a subset of the events which are totally ordered, the different paths in the interleaving set are identical except for stuttering [23] (i.e., the repeating of identical states a finite number of times). The absence of the NEXTTIME operator in the formula $L$ prevents forming a path formula that is satisfied by one such path and not by all others. □
In order to exploit this property, notice that in the common partial order \([22, 26]\) those events local to the same process are totally ordered in every generated interleaving set. Therefore, by using the program's text to check that a set of variables is local to a process and making an assertion \(L\) only about them, one may conclude \(AL\) by proving that \(EL\) holds.

The following discussion deals with alternative ways of giving specifications. Let us compare two approaches to correctness: one that says that any property must be tested for all the possible interleaving sets of any execution (this approach is taken in LTL) and the approach that it is sufficient to show that a property is true for at least one path from each execution. Below, it is argued that the latter approach sometimes is suitable even in cases where \(\neg(EL \rightarrow AL)\).

The specification of a distributed system is very sensitive to the inclusion or absence of external observers (sometimes called the environment) from the model. An external observer is evidence of the order among events which are otherwise unordered according to the partial order semantics of the program. Such an observer can be for example a shared database, a shared resource, or a human who can watch two printers of two different processes at the same time.

Assume that \(EL\) (for a linear path formula \(L\)) holds in a structure representing a certain program, but \(E \neg L\) holds too. (That is, there is a path that satisfies \(L\) and another path that does not satisfy \(L\).) If the execution satisfying \(E \neg L\) is undesirable, it is due to some improper interaction between the program and an external observer. When all such observers are made part of the system, it imposes an additional ordering upon some events that compose the two different paths. Then, two such paths belong to different executions and the undesirable one is now ruled out by the implied universal quantification over all executions rather than by the local universal quantifier \(A\) over paths in a single interleaving set.

We call a system in which all possible observers are included by means of adding their interface events to the partial order and event set a completely abstracted system. (Lamport \([24, 25]\) discusses the issue of "interface specification" as a method to force a specification to be correct under each possible interaction with the user.) A criterion for such a distributed system to satisfy a temporal specification which is of an interleaved sequence is that for every execution there exists a path that satisfies it.

The related notion of "linearization" \([12, 19, 37]\) plays an important role in reasoning about parallel and distributed algorithms. Linearization means completing the partial order to a total order which contains it. In reasoning about parallel and distributed algorithms it is common to take each of the sequences (which represent computations in the interleaving model) and interchange unrelated events \([12, 19, 27]\). A similar argument appears when using what is called "logical variables" in a program that has shared variables. Here, an assignment to local logical variables is sometimes said to be attached to the preceding or successive event \([18]\). Here, too, a program may be specified by a (relatively weak) formula \(EL\) for a linear path formula \(L\), even when \(AL\) does not follow logically from it.
Example 6.2. Assume two users $P_1$ and $P_2$ are using a shared database, and for consistency, it is required that the transactions they make are non-overlapping. The safety (mutual exclusion) property

$$(\text{at}(START_1) \land \text{at}(START_2)) \rightarrow \Box \neg (\text{in}(CS_1) \land \text{in}(CS_2))$$

is equivalent to the ISTL* property

$$\text{A}[ (\text{at}(START_1) \land \text{at}(START_2)) \rightarrow G \neg (\text{in}(CS_1) \land \text{in}(CS_2))]$$

But this enforces a strict requirement which is needed only when the database is not part of the system itself and acts as an outside observer. If we embed the database as a process in the system to which different processes may refer by read and write communication commands, we may use the weaker property

$$E( (\text{at}(START_1) \land \text{at}(START_2)) \rightarrow G \neg (\text{in}(CS_1) \land \text{in}(CS_2)))$$

which says that it is sufficient to linearize the events in such a way that they behave as if mutual exclusion occurs in each execution. This may be useful in case some transactions or part of them can overlap in time without both causing concurrent change to the database. If indeed the entire critical section uses the database and the database is abstracted as a sequence of events then EL and AL are equivalent requirements by Proposition 6.1 since the executions of the critical sections are totally ordered.

This does not prevent using mixed assertions with both existential and universal sequence quantifiers on a distributed system if one wishes to, as shown in the next example.

Example 6.3. In a recent paper on properties that a fairness definition must fulfill [2], one of the requirements is that if one of the paths in a single partial order computation satisfies the fairness condition then so do the rest of the paths in that execution, and if one does not, then none of the paths satisfies the fairness condition. This is equivalent to the assertion \( (EL \rightarrow AL) \) where \( L \) is the fairness constraint. This requirement is reasonable since if a fairness constraint is used in a termination proof to rule out certain executions, it is not possible that a single partial order execution is "partially ruled out".

Comment: An ISTL* formula of the form $E(p \rightarrow q)$ where $p$ is a state formula (contains no path modalities or path quantifiers) and $q$ is any ISTL formula can be easily translated into an ISTL equivalent formula $p \rightarrow Eq$. This applies to the first three examples of the next section where ISTL is sufficient.

7. Applications of the logic

Theorem 5.1 formalized the justification for an approach to correctness which says that "a program is correct under a temporal specification" iff there exists at
least one path for each execution that satisfies the specification. This was also discussed in Section 6. It will be shown through several examples that such an interpretation is useful and the new logic provides a convenient specification tool.

7.1. Temporal semantics for a distributed language

The Hoare-like axioms and rules of inference for CSP [3] have an interesting approach to proving partial correctness. We may give TL axioms and rules which will be similar. (The rule here deals with total correctness instead of partial correctness. There is no problem in defining a partial correctness version for temporal logic). We can see that the rules in [3] choose a distinguished path in which the events occur in a convenient order. Consider the rule (which is somewhat simplified, but equivalent with respect to soundness and completeness of the resulting deductive system, as noted in [3]):

\[
\{ p \} a \| a' \{ t1 \}, \{ t1 \} S1 \{ t2 \}, \{ t2 \} S2 \{ q \} \\
\{ p \} (a, S1) \| (a', S2) \{ q \}
\]

in which \( a \) and \( a' \) are a pair of communication commands, and \( S1 \) and \( S2 \) are noncommunication program segments. We call \((a; S1)\) and \((a'; S2)\) 

bracketed sections because as a consequence of this rule of inference they behave as a single atomic event.

The rule states that we can choose a path in which we first execute the communication event \( a \| a' \), then \( S1 \) in the first process, and then \( S2 \) in the second. If the antecedents of the rule can be proven for that situation, the conclusion will hold no matter which interleaving (if any) actually occurred. We can look at it as if we "bend" time according to our needs and use the fact that no one can tell us we are wrong without any evidence (the missing global clock!).

In terms of ISTL the analogous consequence rule depends on the syntax of the verified program which must satisfy the above conditions on \( a, a', S, \) and \( S \).

\[
(\text{at}(a) \land \text{at}(a') \land p) \rightarrow \text{EF}(\text{after}(a) \land \text{after}(a') \land t1), \quad /*\{ p \} a \| a' \{ t1 \} */
\]

\[
(\text{at}(S1) \land \text{at}(S2) \land t1) \rightarrow \text{EF}(\text{after}(S1) \land \text{at}(S2) \land t2), \quad /*\{ t1 \} S1 \{ t2 \} */
\]

\[
(\text{after}(S1) \land \text{at}(S2) \land t2) \rightarrow \text{EF}(\text{after}(S1) \land \text{after}(S2) \land q) \quad /*\{ t2 \} S2 \{ q \} */
\]

\[
(\text{at}(a) \land \text{at}(a') \land p) \rightarrow \text{EF}(\text{after}(S1) \land \text{after}(S2) \land q) \quad /*\{ p \}(a; S1) \| (a'; S2) \{ q \} */
\]

Suppose that using the rule we have proven a property of the form

\[
(\text{at}(P_1) \land \cdots \land \text{at}(P_n) \land \Phi) \rightarrow \text{EF}(\text{after}(P_1) \land \cdots \land \text{after}(P_n) \land \Psi).
\]

Observe that the property

\[
\text{after}(P_1) \land \cdots \land \text{after}(P_n) \land \Psi
\]

is stable because a terminating global state is stable. That is,

\[
(\text{after}(P_1) \land \cdots \land \text{after}(P_n) \land \Psi) \rightarrow \text{AG}(\text{after}(P_1) \land \cdots \land \text{after}(P_n) \land \Psi).
\]

Using Theorem 5.1 it follows that

\[
(\text{at}(P_1) \land \cdots \land \text{at}(P_n) \land \Phi) \rightarrow \text{AF}(\text{after}(P_1) \land \cdots \land \text{after}(P_n) \land \Psi)).
\]
7.2. The decomposition of programs into layers [13]

Instead of decomposing programs into processes, we can decompose programs into communication-closed layers. A single layer is made of several program segments, each one in a different process, which communicate with each other in order to complete a common task. A (distributed) program may be composed of several layers executed sequentially—first the code in each process for the first layer, then the code for the second, etc.

If there are no communications across layer boundaries (i.e., from a segment of one layer in one process to a segment of another in a different process), then the program executes as if there is a synchronization at the time that each layer begins. This is a pleasant property, since a process can begin executing a new layer while the other processes are still in the first layer, and the program will behave as if the processes all began executing the new layer together.

Looking at properties that all the paths satisfy does not seem to give us any insight into how to reason about this phenomenon. However if we look at the partial order model, we can easily prove the following.

**Proposition 7.1.** In any terminating execution of a layered program, there exists a slice $S$ which defines a global state $G$ in which all the processes have finished the first layer, and are about to enter the second one.

**Proof.** Take the set of events $S$ that are executed according to the code of the first layer. We have to show that these events form a slice. It is sufficient to show that there cannot be two adjacent events $e_1 < e_2$ such that $e_1 \notin S$ and $e_2 \in S$. By the choice of $S$, it is evident that such two events cannot belong to the same process, and because there is no communication across layer boundaries, two such events are ruled out. $\Box$

This means, by the previously mentioned connection between the partial order model and the interleaving model, that for each one of the executions, there exists a path which behaves as if there was a synchronization. Because we cannot say which of the interleavings satisfying the same partial order "really occurred" without a global clock, this distinguished path is as good as any other.

A layered program $PR = [S_1; Q_1] || S_2; Q_2$ will satisfy the ISTL formula:

$$(at(S_1) \land at(S_2)) \rightarrow [EF(at(Q_1) \land at(Q_2)) \lor AG(\neg \text{after}(S_1)) \lor AG(\neg \text{after}(S_2))].$$

There are two layers $S = [S_1 || S_2]$ and $Q = [Q_1 || Q_2]$. We start at labels $S_1$ and $S_2$; that is, at the beginning of the program. Then in each execution which eventually completes the two segments $S_1$ and $S_2$, there is a corresponding path with a global state in which we are exactly before entering both $Q_1$ and $Q_2$. Formally, the expression means that for each (partial order) execution, from a state that starts when the processes are at the beginning of the code, there is an interleaved path constructed from the partial order such that the layers do synchronize.
Here again, as in the previous example, one may use Theorem 5.1 to prove total correctness for a layered program by showing the correctness only for those paths with a state that synchronizes the beginnings of the layers.

7.3. The Chandy and Lamport snapshot algorithm [8]

After the execution of the superimposition of this algorithm on another (basic) algorithm, a global state of the combined superimposed algorithm is recorded. The recorded global state is used for detection of properties which are stable. The global state recorded does not necessarily appear on every interleaved path defined by the program. The important property is that for each (partial order) execution there exists a path that contains the global state which is eventually recorded.

This aspect of the correctness of the snapshot algorithm can be stated:

\[(\text{at}(\text{START}_1) \land \text{at}(\text{START}_2) \land \cdots \land \text{at}(\text{START}_n) \land \text{EF}(\text{finished} \land rf)) \rightarrow \text{EF}f\]

where \(f\) is a formula describing some global property, and \(rf\) is a formula that says "property \(f\) is recorded". The predicate \(\text{finished}\) is true when the snapshot part has been completed. In terms of ISTL this means: for every (partial order) execution, if there is an interleaving sequence which reaches a state in which the snapshot part is completed and \(rf\) is true, then for the same execution there exists an interleaved sequence with a state in which \(f\) actually was true.

Other aspects of the correctness of the snapshot algorithm [7] include metatheorems on the correspondence between the ISTL structures which correspond to the set of executions before and after the superimposition of the snapshot part.

One might like to deduce from the correctness claim of the snapshot algorithm and the fact that the property detected is stable (i.e., \(f \rightarrow \text{AG}f\)) that if the property was detected, then on any path in the set that represents the same execution, eventually \(f\) will hold forever. From the correctness condition, the stability of \(f\) and Theorem 5.1, we may deduce

\[(\text{at}(\text{START}_1) \land \text{at}(\text{START}_2) \land \cdots \land \text{at}(\text{START}_n) \land \text{EF}(\text{finished} \land rf)) \rightarrow \text{AF}f\]

Again using stability

\[(\text{at}(\text{START}_1) \land \text{at}(\text{START}_2) \land \cdots \land \text{at}(\text{START}_n) \land \text{EF}(\text{finished} \land rf)) \rightarrow \text{AF}(\text{AG}f)\]

7.4. Concurrency

It is convenient that the logic can express the potential concurrency of independent events or operations. These can then be executed on independent processors, potentially increasing the efficiency of execution. In order to express that the two local operations \(e_1\) and \(e_2\) which occur only once can run concurrently the following assertion may be used:

\[(\text{at}(\text{START}_1) \land \text{at}(\text{START}_2)) \rightarrow \text{EF}[\{(\text{at}(e_1) \land \text{at}(e_2)) \land \{\text{FX}(\text{at}(e_1) \land \text{after}(e_2)) \land \text{EX}(\text{at}(e_2) \land \text{after}(e_1))\}\]
Proposition 7.2. In terms of slices, \(e_1\) and \(e_2\) are concurrent (unrelated by the partial order) iff there exist three slices \(S\), \(S_1\) and \(S_2\) such that \(e_1, e_2 \notin S\), \(S_1 = S \cup \{e_1\}\) and \(S_2 = S \cup \{e_2\}\).

Proof. (\(\Rightarrow\)) Let \(S_1^*\) be the set of events preceding \(e_1\) and \(S_2^*\) the set of events preceding \(e_2\). From Definition 2.6 it follows that \(S_1^*\) and \(S_2^*\) are slices. The union of two slices is a slice. Since \(e_1\) does not precede \(e_2\), \(e_1 \notin S_2^*\) and similarly \(e_2 \notin S_1^*\). Therefore, assign

\[
S = S_1^* \cup S_2^*, \quad S_1 = S \cup \{e_1\}, \quad S_2 = S \cup \{e_2\}.
\]

(\(\Leftarrow\)) Given \(S\), \(S_1\) and \(S_2\) as above, without loss of generality, assume to the contrary of the proposition that \(e_1\) precedes \(e_2\). Then according to Definition 2.6, \(S_2\) (which does not include \(e_1\)) cannot be a slice. \(\square\)

In the above example we interpret \(e_1\) as the operation that is executed from the point that \(at(e_1)\) is true until \(after(e_1)\) and similarly for \(e_2\). \(S\) satisfies \(at(e_1) \land after(e_2)\). \(S_2\) satisfies \(at(e_1) \land after(e_2)\) and \(S_1\) satisfies \(at(e_2) \land after(e_1)\).

8. Conclusions

A temporal framework for reasoning about global states which are constructed from partial orders is suggested. The major motivation for the new logic and interpretation is to rigorously express and prove within a uniform formalism properties which were previously explained informally. The novel property of ISTL and its extensions is that, whenever convenient, it allows us to use global states in proofs and specifications while thinking in terms of partial orders.

A correctness criterion which is very natural for dealing with partial orders and very natural to ISTL is linearization: for each partial order, choose a single total order which contains it to represent the computation. Various aspects of this correctness criterion appear in [12, 19, 18] and are defined rigorously in the new framework.

It is evident that the following property, which was formally proven in Section 5, is important to the understanding of many phenomena which are explained using the partial order model: a stable property which occurs on one interleaving sequence will eventually hold on any interleaving sequence which is a completion of the same partial order.

Even if interleaving sets are not directly needed to express a property (for example, total correctness is traditionally expressed in LTL) it might be convenient to move into ISTL. Properties which stem from the partial order are sometimes not interesting in themselves (like, for example, a possible global state in an execution in Sections 7.2, 7.3) but are helpful as an intermediate stage in proving other, much more common, properties such as total correctness. A deductive system which makes use
The ability to reason about linearizations of partial orders can be used here (Section 7.1).

The logics and in particular the strongest version QISTL* are strong enough to allow using the syntactic proof systems of [31, 32, 3, 13]. The additional semantic structure suggests the addition of new proof rules which take advantage of the underlying partial orders, thus exploiting the convenience of global reasoning, without losing information about closely related execution sequences.

Finding sufficient consequence rules and axioms to guarantee completeness over our interpretation is an open problem. Another interesting issue here is that the underlying partial order semantics might further restrict the class of structures (for example, by allowing only a fixed number of processes). For each such restriction it might be interesting to find the corresponding "increment" of the deductive system.

Acknowledgment

We appreciate the extremely helpful conversations we have had on this subject with Allen Emerson, Nissim Francez, Janos Makowsky, Shlomit Pinter, Amir Pnueli, Wolfgang Reisig and Pierre Wolper.

References


