Testing Distributed Systems through Symbolic Model Checking

Gabriel Kalyon, Thierry Massart, Cédric Meuter, Laurent Van Begin

This work was partially supported by a FRFC grant: 2.4530.02

http://www.ulb.ac.be/di/ssl/cfv
Testing Distributed Systems through Symbolic Model Checking

Gabriel Kalyon*, Thierry Massart**, Cédric Meuter, and Laurent Van Begin***

Université Libre de Bruxelles (U.L.B.),
Boulevard du Triomphe, CP-212, 1050 Bruxelles, BELGIUM
{gkalyon,tmassart,cmeuter,lvbegin}@ulb.ac.be

Abstract. The observation of a distributed system's finite execution can be abstracted as a partial ordered set of events generally called finite (partial order) trace. In practice, this trace can be obtained through standard code instrumentation, which takes advantage of existing communications between processes to partially order events of different processes. We show that testing that such a distributed execution satisfies some global property amounts therefore to model check the corresponding trace. This work can be time consuming; we therefore provide an efficient symbolic CTL model-checking algorithm for traces. This method is based on a symbolic data structure, called Interval Sharing Trees, allowing to efficiently represent and manipulate sets of $\omega$-tuples of naturals. Efficient symbolic operations are defined on this data structure in order to deal with all CTL modalities. We show that in practice this data structure is well adapted for CTL model checking of traces.

Keywords: testing, asynchronous distributed systems, global property, model checking of traces, trace checking

1 Introduction

A distributed system is typically a set of distributed hardware equipments which run concurrent processes, communicating through some network. The design of such system is known to be a difficult task. When the purpose of such a system is to perform some control of critical equipment like an industrial plant, a plane, or a satellite, its correctness is extremely important. The designer can ease her work by various techniques [1–3] including validation and debugging. In particular, traditional model-based approaches abstract the action the system can do into events which change the system’s global state. Validation works therefore on a labelled directed graph called a Kripke structure which describes the possible system’s behaviours. Verification tools (e.g., [4–6]) can be used to validate parts of models. For instance, such tools can be used to check that, in the system,

* Supported by the Belgian National Science Foundation (FNRS) under a FRIA grant.
** Supported by the Belgian Science Policy IAP-Phase VI: MoVES and Centre Fédéré en Vérification (FNRS-FRFC n 2.4530.02)
*** Research fellow supported by the Belgian National Science Foundation (FNRS).
every time the system goes in a state where a condition $p$ holds, it is followed by a state where $q$ and $r$ holds. $p$ can for instance be an abstraction for some alarm detected through some given sensor, while $q$ and $r$ may correspond to, possibly distributed, values assignment on some actuators.

Unfortunately in practice, even with this abstraction, the state-explosion problem generally prevents the designer from exhaustively verifying the whole system, even with efficient exploration techniques such as partial order reduction [7,8] or symbolic model checking [9–11].

In such cases, the designer generally falls back to testing which cannot guarantee that a system is completely bug-free, but if achieved on a large number of test-cases (e.g., covering all the functionalities of the system), can give a reasonable confidence that the system is correct. In this context, a test-case defines the model of the part of the system which corresponds to a particular execution. Testing may therefore be seen as the validation of this smaller model.

To extract this smaller model from a system, the implementation is instrumented to record only relevant events. A special process, called the monitor, records this model (the events of the system), that we can just call execution here, and then checks that it satisfies some desired property.

Notice that an execution can also be extracted from a design model. In particular scenarios of executions, modelled as MSC (Message Sequence Charts) is a particular form of such execution and can also be validated. Hence, at both the design and implementation levels, it is an important activity for which efficient methods must be provided.

In the centralized case, an execution of the system is a sequence of events. Determining if such an execution satisfies a property is in general simple. In the distributed case, if the system to control is slow enough, one can assume that all processes of the system are synchronized using a global discrete clock. This so-called synchrony hypothesis allows to see such distributed execution as a sequence of set of events where all events in a set are seen as simultaneous. This hypothesis allows a relatively simple validation of such a distributed execution. Unfortunately, if the system to control is too fast compared to the synchronization mechanism offered by the implementation, the synchrony hypothesis cannot be made and the asynchronism between distributed processes must be taken into account in the analysis. In this case, the exact order in which two concurrent events occur in the execution is, in general, not always known or guaranteed. By taking into account the communications between processes, only a partial order on the events of the execution can be obtained. In practice, this partial order relation, often called the happened-before relation [12], can be obtained through correct code instrumentation using, for instance, vector clocks [12,13].

Hence in this case, an execution is a partially ordered set of events often called partial order trace or simply trace. Since the order in which the events of this (partial order) trace are interleaved is generally relevant to the safety of the system, testing that a distributed execution satisfies a global property $\phi$ amounts to verifying that every sequential execution, compatible with the partial order, satisfies $\phi$ or, in other terms, model checking $\phi$ on the corresponding trace. Un-
fortunately, this problem is hard [14], since the number of compatible sequential executions and the size of the Kripke structure which models an execution may be exponential in the number of concurrent processes. Therefore, to tackle this complexity, instead of working on the underlying Kripke structure, efficient techniques have been developed to work directly on the partial order itself, which is, in general, exponentially more compact. In this line, in [15], A. Sen and Gang present the temporal logic RCTL (for regular-Ctrl), which is a subset of the branching time temporal logic Ctl. [16] and shows that the compact symbolic data structure called computation slice [17], can be used to efficiently compute all global states which satisfy a RCTL formula.

However, RCTL does not include such simple Ctl property as AG(p \implies AF(q \land r)), i.e., every p is eventually followed by a state where q and r hold true; formula that may be very useful during validation. In general, a computation slice is too restrictive to represent any arbitrary set of global states of a finite trace.

This motivates our work; in this paper, we introduce an efficient symbolic method using Interval Sharing Trees (IST) [18, 19]. This data structure allows to represent any set of global states of a finite trace. We define how to use IST to provide a full Ctl model checking of finite traces. We show that intervals of naturals can be used, in practice, to have a compact representation for sets of global states of the trace satisfying the desired formula and hence, to provide an efficient algorithm for Ctl model checking of finite traces. Moreover, we show that our algorithms perform very well compared to standard symbolic model checking using BDDs [11] and implemented in the tool NuSMV [6].

This paper is organized as follows. In section 2, we detail related works. In section 3, we introduce our model for traces and define the Ctl over this model. In section 4, we explain how sets of configurations can be represented compactly using intervals and interval sharing trees. In section 5, we show how Ctl model checking on traces can be solved using this symbolic representation. Next, in section 6, we experimentally validate our method on various examples compared to Ctl model checking with the NuSMV tool. Finally, conclusion and future works are given in section 7. Note: all proofs from sec. 5 are included in app. A.

2 Related Works

Testing and monitoring the global behaviours of distributed systems can be categorized in two classes: trace model-checking and global predicate detection.

Trace model checking has been studied mainly theoretically through the definition of several linear temporal logic for Mazurkiewicz traces. A Mazurkiewicz trace [20], over an alphabet Σ with a independence relation I, can be defined as a Σ-labelled partial order set of events with special properties not explained here. For Mazurkiewicz traces, local [21, 22] and global [23–25] trace logics have been defined. However, in our case, the trace is an abstraction of a distributed

3 Our trace can be seen as a prime event structure with an empty conflict relation [26]
execution (or of a scenario) and models a set of possible interleavings of events the distributed system may have had. Since we do not suppose to have information about independence between actions, none of these actions are independent a priori; testing must then check that all these possible orderings of events are correct. Since the independence relation is not a data that trace temporal logics may exploit, we do not use these logics to model-check our executions and stick to simple sequence (interleaving) semantics.

Global predicate detection initially aims at answering reachability questions, i.e. does there exist a possible global configuration of the system, that satisfies a given global predicate $\phi$. Garg and Chase showed in [14] that this problem is NP-complete for an arbitrary predicate, even when there is no inter-process communication. Efficient (polynomial) methods have been proposed for various classes of predicates, such as stable predicates proposed by Chandy and Lamport [27], independent predicates by Charron-Bost et al [28], conjunctive predicates by Garg and Waldecker [29, 30], linear and semi-linear predicates by Chase and Garg [14], regular predicates by Garg and Mittal [31] and predicates expressed by a finite automata that can be checked online by Jard et al [32]. Garg and Mittal implicitly use a symbolic data structure called computation slice, to compute efficiently all global states, compatible with a given execution satisfying a given regular predicate [17]. This structure is used by A. Sen and Garg in their work on the temporal logic RCTL [15]. In [33, 34] K. Sen et al. use an automaton to specify the system’s monitor. The authors provide an explicit exploration of the state space and to limit this exploration a window is used. In a previous work [35], we have used this technique to provide an efficient LTL tester of distributed executions.

3 Framework

In this section, we detail our framework. We start by formally introducing our model for traces of distributed systems, i.e. finite partial order trace. Then, we define the branching time temporal logic CTL* over such finite traces.

3.1 Partial Order Trace

Our executions are obtained by a fixed numbers of concurrent processes, each executing a finite sequence of assignments. Moreover, due to inter-process communications, other causal dependencies are added. These communications will usually be done by message passing, but if some processes are not distributed, can be done by other means such as shared variable. An execution is modeled as a finite partial order trace, i.e. a finite partially ordered set of events, where each event belongs to some process and is labeled by the assignment which took place during this event.

Definition 1 (Partial order trace). A partial order trace of $k$ processes and over a set of variables $\mathbb{V}$ is a tuple $T = (E, \alpha, \preceq)$ where:
- $E = P_1 \cup P_2 \cup \ldots \cup P_k$ is a finite set of events partitioned into $k$ disjoint non-empty subsets $P_i$, called processes; $\text{pid}(e) = i$ if $e \in P_i$;
- $\alpha : E \rightarrow \mathbb{V} \times \mathbb{Q}$ is a labeling function mapping each event to an assignment, i.e., $\alpha(e) = (x, v)$ associates the assignment $x := v$ to $e$; if $\alpha(e) = (x := v)$, $\text{var}(e)$ denotes $x$ and $\text{val}(e)$ denotes $v$;
- $\preceq \subseteq E \times E$ is a partial order relation on $E$ such that $\forall e, e' \in E$:
  (i) $\text{pid}(e) = \text{pid}(e') \Rightarrow (e \preceq e') \lor (e' \preceq e)$
  (ii) $\text{var}(e) = \text{var}(e') \Rightarrow (e \preceq e') \lor (e' \preceq e)$.

Condition (i) on $\preceq$ ensures that all events from the same process are ordered and condition (ii) enforces that all events assigning the same variable are ordered. Given an event $e \in E$, we define $|e| = \{e' \in E \mid e' \preceq e\}$, the past of $e$ (including itself), and $\text{pos}(e) = \{(e | e \in P_\text{pid}(e))\}$ (where $\cdot | \cdot$ denotes the size of sets), the position of $e$ in its process. A cut is a subset $C \subseteq E$ such that $\forall e \in C : |e| \subseteq C$. $\text{cuts}(T) = \{C \subseteq E \mid \forall e \in C : |e| \subseteq C\}$ is the set of all cuts in $T$. In the remainder of this paper, we always consider the set of variables $\mathbb{V}$ and the partial order trace of $k$ processes $T = \langle E, \alpha, \preceq \rangle$.

**Semantics.** Given a cut $C \in \text{cuts}(T)$, we define $\text{enabled}(C) = \{e \in E \mid |C| \subseteq C\}$ the set of events enabled in $C$. If $e$ is enabled in the cut $C$, then it can be fired from $C$ leading to $C \cup \{e\}$, the successor of $C$ for $e$. Note that if $C \in \text{cuts}(T)$, so is $C \cup \{e\}$ for all $e \in \text{enabled}(C)$. Given a set of cuts $X \subseteq \text{cuts}(T)$, $\text{pre}^3(X) = \{C \in \text{cuts}(T) \mid \exists e \in \text{enabled}(C) : C \cup \{e\} \in X\}$ is the set of existential predecessors of $X$, i.e., the set of cuts having at least one successor in $X$, and $\text{pre}^1(X) = \{C \in \text{cuts}(T) \mid \forall e \in \text{enabled}(C) : C \cup \{e\} \in X\}$ is the set of universal predecessors of $X$, i.e., the set of cuts having all their successors in $X$. Additionally, given a sequence of cuts $\sigma = C_0, C_1, \ldots, C_n$, $\sigma_i$ denotes $C_i$, the $i^{th}$ element of $\sigma$, and $|\sigma| = n$ denotes the size of $\sigma$. A run from a cut $C$ is a sequence $\sigma \in \text{cuts}(T)^*$ such that (i) $\sigma_0 = C$, (ii) $\sigma_0 | \sigma = E$, and (iii) $\forall 0 \leq i < |\sigma| : \sigma_i \in \text{pre}^3(|\sigma_{i+1}|)$, i.e., a sequence of cuts (i) starting in $C$, (ii) ending in $E$, and (iii) $\sigma_{i+1}$ is a successor of $\sigma_i$ for any $i$. The set of runs starting in $C \in \text{cuts}(T)$ is denoted by $\text{runs}(C)$. Finally, $\text{runs}(\emptyset)$ is the set of runs of the trace $T$.

**Practical representation.** A trace $T = \langle E, \alpha, \preceq \rangle$ can be represented using a directed acyclic graph $(E, \rightarrow)$ called Hasse diagram. In this graph, there is an edge from event $e$ to event $e'$ if and only if they are ordered, i.e., $e \preceq e'$, and if their order is not imposed by transitivity, i.e., $\neg \exists e'' : e \preceq e'' \preceq e'$ where $e_1 \preceq e_2$ denotes $e_1 \preceq e_2$ and $e_1 \neq e_2$. As an example, Fig. 1 depicts such a graph for a partial order trace with two processes. That trace describes an execution of a distributed system with two concurrent sub-systems. During that execution, the first process makes three assignments to variables $w, y, x$ and the second one makes two assignments to $x$ and $w$. An edge between two events $e$ and $e'$ in the Hasse graph such that $\text{pid}(e) \neq \text{pid}(e')$ models a communication between processes (noted $e \rightarrow_C e'$). Communication edges model either message
passing between processes or the fact that the event \( e \) assigns a value to a shared variable used in \( e' \). Note that \( v \) in event \( x := v \) can be obtained by evaluating an expression involving the variable appearing in \( e \). For instance, the arrow between \( w := 0 \) and \( y := 3 \) in Fig. 1 can model that value 3 is obtained at run time by evaluating an expression where \( w \) appears and its value is given by the first assignment. In the following, we always consider that we have the Hasse diagram corresponding to \( T \).

### 3.2 CTL over Finite Partial Order Trace

We define in this section a version of the logic CTL (computational tree logic) evaluated on partial order traces.

**Syntax.** A predicate \( p \) is a constraint \( x \bullet c \) where \( c \) is a rational constant, \( x \in V \) and where \( \bullet \in \{<,\leq,>,\geq,=,\neq\} \). A formula in the CnT logic is built on predicates using classical Boolean operators, and temporal modalities. If \( \phi \) denotes a predicate and \( \phi_1,\phi_2 \) denote CTL formulae, then the set of CnT formulae is defined as follows:

\[
\phi ::= T \mid p \mid \neg \phi \mid \phi_1 \lor \phi_2 \mid \phi_1 \land \phi_2 \mid EX\phi \mid AX\phi \mid EG\phi \mid AG\phi \mid E[\phi_1 U\phi_2] \mid A[\phi_1 U\phi_2]
\]

where \( A \) stands for for all runs, \( E \) for exists a run, \( X \) for next, \( G \) for globally and \( U \) for until. Two other temporal modalities, \( EF \) and \( AF \), where \( F \) stands for finally, are derived syntactically as follows: \( EF\phi \equiv E[T U\phi] \) and \( AF\phi \equiv A[T U\phi] \).

**Semantics.** Basic formulae are constraints over one variables in \( V \). Since all assignments to a particular variable are ordered, each cut \( C \in \text{cuts}(T) \) induces a unique valuation on the variables in \( V \) no matter the order in which the events are executed. Formally, given a cut \( C \), we can define inductively the valuation induced by \( C \), noted \( v_C \), as follows:

- if \( C = \emptyset \) then \( \forall x \in V, \ v_C(x) = 0 \),
- if \( C = C' \cup \{e\} \) with \( C' \in \text{cuts}(T) \) then \( \forall x \in V : v_C = \begin{cases} \text{val}(e) & \text{if } \text{var}(e) = x \\ v_{C'}(x) & \text{otherwise} \end{cases} \)

Hence, we forget variables in \( V \) and only consider cuts of \( T \) when defining the semantics of CTL formula. More precisely, the semantics of a CTL formula
is given by the satisfaction relation $\models$ defined hereafter.

\[
\begin{align*}
C \models \top & & \text{iff} & & C \models \phi & & \text{is true,} \\
C \models p & & \text{iff} & & C \models \phi & & \phi \text{ is } p \\
C \models \neg \phi & & \text{iff} & & C \models \phi & & \phi \text{ is false} \\
C \models \phi_1 \lor \phi_2 & & \text{iff} & & (C \models \phi_1) \lor (C \models \phi_2) \\
C \models \phi_1 \land \phi_2 & & \text{iff} & & (C \models \phi_1) \land (C \models \phi_2) \\
C \models \exists x \phi & & \text{iff} & & \exists x \in \text{enabled}(C) : C \cup \{x\} \models \phi \\
C \models A_{\exists} \phi & & \text{iff} & & \forall x \in \text{enabled}(C) : C \cup \{x\} \models \phi \\
C \models \forall x \exists \sigma \in \text{runs}(C), \forall i \in [0, \sigma] : \sigma_i \models \phi & & \text{iff} & & \exists \sigma \in \text{runs}(C), \exists i \in [0, \sigma] : \sigma_i \models \phi \\
C \models E_{\exists} \phi & & \text{iff} & & \exists \sigma \in \text{runs}(C), \exists i \in [0, \sigma] : \\
& & & & (\sigma_i \models \phi_2) \land (\forall j \in [0, i) : \sigma_j \models \phi_1) \\
C \models A_{\exists} \phi & & \text{iff} & & \forall \sigma \in \text{runs}(C), \exists i \in [0, \sigma] : \\
& & & & (\sigma_i \models \phi_2) \land (\forall j \in [0, i) : \sigma_j \models \phi_1)
\end{align*}
\]

Note that according to this semantics, when the execution of $T$ is finished (when the cut $E$ is reached), for any CTL formula $\phi$, we have that $E \models \exists x \forall \sigma \models \phi$. $[[\phi]]$ denotes the set $\{C \in \text{cuts}(T) \mid C \models \phi\}$ of cuts that satisfy formula $\phi$. In the remainder of the paper, we will present an efficient method to build $[[\phi]]$.

4 Symbolic Representation for Sets of Cuts

The number of cuts, i.e., the size of $\text{cuts}(T)$, is in general exponential in the size of $T$. Hence, efficient representations for large sets of cuts are needed. Our proposal is based on the following observation: a cut can be represented by a $k$-tuple $\overline{x}$ of naturals where the $i$th component of $\overline{x}$ gives the number of events of the $i$th process that already occurred. For example, if a trace $T$ is composed of 3 processes, the 3-tuple $(1, 2, 0)$ represents the cut where process $P_1$ has executed its first event, i.e., $e \in P_1$ with $\text{pos}(e) = 1$, process $P_2$ has executed its first 2 events, i.e., $e_1, e_2 \in P_2$ with $\text{pos}(e_i) = i$ ($i \in \{1, 2\}$), and process $P_3$ has executed no events. The successor (predecessor) relation between cuts can be lifted to their vector representation: an event $e \in P_1$ is enabled in $\overline{x} = (x_1, \ldots, x_k)$ if $x_{\text{pos}(e)} < \text{pos}(e) \land \forall e' \in e \setminus \{e\} : \text{pos}(e') \leq x_{\text{pos}(e')}$ and the successor of $\overline{x}$ for $e$ is $(x_1, \ldots, x_i + 1, \ldots, x_k)$. Note that a vector $\overline{x}$ is not necessarily a representation for a cut. Indeed, if $i \neq j \in [1, k]$, $x \in P_j$, $x' \in e \cap P_j : (\text{pos}(e) \leq x_i) \land (\text{pos}(e') > x_j)$ then $\overline{x}$ does not represent a cut, otherwise it does. Given a subset $X \subseteq \mathbb{N}^k$, we denote $\text{sets}(X) = \{C \subseteq E \mid \exists \overline{x} \in X, \forall 1 \leq i \leq k : |C \cap P_i| = x_i\}$ the set of subsets of events represented by the set $X$. Moreover, $\overline{x} \preceq \overline{x}'$ denotes that $\forall i \in [1, k] : x_i \leq x'_i$ which in terms of cuts corresponds to inclusion. In conclusion, in order to represent sets of cuts, we show how to efficiently represent large set of tuples of naturals.
4.1 Multi-rectangles: a Compact Representation for Sets of Cuts

A $k$-multi-rectangle $M$ is a tuple of intervals over natural values of dimension $k$. $M$ defines the set of $k$-tuples $(x_1, \ldots, x_k)$ over naturals such that $\forall 1 \leq i \leq k : x_i \in$ in the interval corresponding to the $i^{th}$ dimension of $M$. Assuming that each interval contains $n$ values, $M$ represents a set of $n^k$ $k$-tuples. Hence, it is a compact representation for the set it represents. Moreover, $k$-multi-rectangles correspond to a natural class of sets of cuts. Indeed, suppose $k = 2$ and the events $e_{k1}, e_{k2}, \ldots, e_{km}$ of $P_1$ ($i \in \{1, 2\}$) occurring sequentially without any restrictions on the events of $P_2$ and such that $\forall j \in [1, m] : \text{pos}(e_{kj}) = j$. Then, the set of cuts where $P_1$ and $P_2$ have executed some of these events corresponds to the multi-rectangle $\langle [1, m_1], [1, m_2] \rangle$. This multi-rectangle represents succinctly the result of all possible interleavings of $P_1, P_2$. However, due to communications between processes, sets of cuts are not represented in general by one $k$-multi-rectangle, but a set thereof. Hence, to prevent a symbolic state explosion, we use a data structure, called Interval Sharing Tree (IST), to represent efficiently large sets of $k$-multi-rectangles.

4.2 Interval Sharing Tree

Interval Sharing Trees [19] is a compact data structure for representing sets of $k$-tuples. An IST is basically a sharing tree [36], i.e., a directed acyclic graph, where each node is labelled with an interval of integers. Each path in such a graph represents a $k$-multi-rectangle. The sharing of common prefixes and suffixes of $k$-multi-rectangles allows obtaining a compact representation for sets of $k$-multi-rectangles. Interval sharing tree are defined as follows.

**Definition 2 (Interval Sharing Tree (IST)).** An interval sharing tree $\mathcal{I}$, is a labelled directed acyclic graph $(N, \ell, \text{succ})$ where:

- $N = N_0 \cup N_1 \cup N_2 \cup \ldots \cup N_k \cup N_{k+1}$ is the finite set of nodes, partitioned into $k + 2$ disjoint subsets $N_i$ called layers with $N_0 = \{\text{root}\}$ and $N_{k+1} = \{\text{end}\}$;
- $\ell : N \mapsto \{\top, \bot, \text{end}, \text{root} \}$ is the labelling function such that $\ell(n) = \top$ (resp. $\bot$) if and only if $n = \text{root}$ (resp. $\text{end}$);
- $\text{succ} : N \mapsto 2^N$ is the successor function such that:
  
  (i) $\text{succ}(\text{end}) = \emptyset$;
  
  (ii) $\forall i \in [0,k], \forall n \in N_i : \text{succ}(n) \subseteq N_{i+1} \wedge \text{succ}(n) \neq \emptyset$;
  
  (iii) $\forall n \in N, \forall n_1, n_2 \in \text{succ}(n) : (n_1 \neq n_2) \Rightarrow (\ell(n_1) \neq \ell(n_2))$;
  
  (iv) $\forall i \in [0,k], \forall n_1, n_2 \in N_i : (\ell(n_1) = \ell(n_2)) \Rightarrow (\text{succ}(n_1) \neq \text{succ}(n_2))$.

In other words, an IST is a directed acyclic graph where each nodes are labelled with couples of integers except for two special nodes (root and end), such that (i) the end node has no successors, (ii) all nodes from layer $i$ have their successors in layer $i + 1$, (iii) a node cannot have two successors with the same label, (iv) two nodes with the same label in the same layer do not have the same successors. For a node $n$ (except root and end), $\ell(n)$ is interpreted as an interval of integers. We note $x \in \ell(n)$ if an integer value $x$ belongs to
that interval. Figure 2 illustrates some IST. A path of an IST $I$ is a sequence of node root, $n_1, n_2, ..., n_k, end$ such that $n_1 \in \text{succ(root)}, end \in \text{succ}(n_k)$ and $\forall i \in [1, k]: n_{i+1} \in \text{succ}(n_i)$. A $k$-tuple $\pi = \langle x_1, x_2, ..., x_k \rangle$ is accepted by an IST $I$ if and only if there exists a path root, $n_1, n_2, ..., n_k, end$ in $I$ such that $\forall i \in [1, k]: x_i = v(n_i)$. The set of $k$-tuples accepted by $I$ is denoted by $\text{tuple}(I)$ and if $\text{tuple}(I) \subseteq \mathbb{N}^k$, then $\text{sets}(I) = \text{sets}(\text{tuple}(I))$. In practice, sharing of prefixes (iii) and suffixes (iv) in IST allow a non-negligible memory saving, which can be exponential in the best cases (there exists IST whose number of nodes and edges is logarithmic in the number of $k$-multi rectangles it represents).

**Manipulation of sets represented with IST.** Standard set operations have been defined symbolically over IST's, namely, union, noted $I_1 \cup I_2$, intersection, noted $I_1 \cap I_2$, set difference, noted $I_1 \setminus I_2$ and complementation, noted $\overline{I}$. Other operations have been defined like downward closure, noted $\downarrow I$, such that $\text{tuple}(\downarrow I) = \{ \pi' \in \mathbb{N}^k \mid \exists \pi \in \text{tuple}(I) : \pi \leq \pi' \}$, and shift of a variable, i.e., replace $x_i$ by $x_i + \delta$ for $i \in [1, k]$ and $\delta \in \mathbb{Z}$, noted $I^{[x_i \leftarrow x_i + \delta]}$. Formally, $\text{tuple}(I^{[x_i \leftarrow x_i + \delta]}) = \{ \langle x_1, ..., x_i + \delta, ..., x_k \rangle \mid \pi \in \text{tuple}(I) \}$. Symbolic algorithms, i.e., algorithms that do not enumerate all the paths of IST, for those operations have been defined. Since the number of paths is in general larger than the size of the IST, symbolic algorithms allow efficient manipulation of $k$-multi-rectangles sets taking into account their prefix and suffix sharing. Note that the counterpart of the compactness of IST is that most of their operations cannot be computed in polynomial time in general. Hence, (most of) the symbolic algorithms to manipulate IST are exponential in their worst case (see [18] for more details). However, those algorithms are in general far from their worst case in practice and IST have been shown to be more efficient than other known data-structure (to represent subsets of $\mathbb{N}^k$) both in execution time and memory saving [37].

## 5 Using IST for CTL Model Checking

A basic approach to solve the CTL model checking problem over partial order traces consists in flattening the trace by building a graph where nodes are cuts and edge corresponds to the successor relation and then solve the classical CTL model checking on Kripke structures. Unfortunately, that method is not practical since the resulting graph is in general exponential in the size of the trace. To overcome that problem, we propose to build $\downarrow [\phi]$ without flattening the partial order trace but working directly on it. Our method builds $\downarrow [\phi]$ inductively on the structure of $\phi$. Since $\downarrow [\phi]$ can be large, we use IST to efficiently represent and manipulate sets of cuts. We now present in details the construction.

### 5.1 Tautology

If $\phi \equiv \top$, $I_T$ is an IST representing all possible cuts of the trace $T$. The principle to build $I_T$ is to start from the very simple IST $I_0$ where $\text{sets}(I_0)$ is the set of cuts
if we do not consider communication edges of the Hasse diagram. Then, we consider communication edges one by one, i.e., we build the IST $I_0, I_1, I_2, \ldots$ where $I_i$ is built from $I_{i-1}$ ($i > 0$) by taking into account one more communication edge until we have considered all of them. To take into account a communication edge, we remove from $\text{sets}(I_{i-1})$ the sets of events that do not satisfy the definition of cuts because of that edge. Hence, assuming the Hasse diagram has $v$ communication edges, $\text{sets}(I_0) \supseteq \text{sets}(I_1) \supseteq \ldots \supseteq \text{sets}(I_v) = \{\top\}$. $I_0$ is defined as follows:

- $N = \{\text{root}\} \cup \{n_1\} \cup \{n_2\} \cup \ldots \cup \{n_k\} \cup \{\text{end}\}$
- $\forall i \in [1,k] : u(n_i) = [0, |P_i|]$
- $\text{succ}(\text{root}) = \{n_1\}$, $\text{succ}(n_k) = \{\text{end}\}$, and $\forall i \in [1,k] : \text{succ}(n_i) = \{n_{i+1}\}$.

To take into account a communication $e \rightarrow_c e'$, we need to remove from $\text{sets}(I_i)$ all the sets of events that do not satisfy the definition of cuts, i.e., the sets that contain $e'$ but not $e$. To achieve that goals, we first build an IST $B(e)$ representing all the sets of events that do not contain $e$ (and have a vector representation). In other words, $B(e)$ is the same as $I_0$ except for the layer $\text{pid}(e)$ where $u(\text{pid}(e)) = [0, |\text{pos}(e)| - 1]$. Then, we build an IST $A(e')$ representing all the sets of events that contain $e'$ (having a vector representation), i.e., $A(e')$ is the same as $I_0$ except for $u(\text{pid}(e')) = [\text{pos}(e'), |\text{pid}(e')|]$. The events to remove from $\text{sets}(I_i)$ are in the intersection of $\text{sets}(A(e'))$ and $\text{sets}(B(e))$. Hence, to remove them we compute $I_i = I_{i-1} \setminus (A(e') \cap B(e))$. We iterate this construct until all communication edges are taken into account. Figure 2 illustrates the method by computing the IST corresponding to the set of cuts satisfying $\top$ in the trace from Fig. 1.

**Lemma 1.** Given a trace $T = \langle E, \alpha, \preceq \rangle$, we have that $\text{sets}(I_T) = \{\top\}$

5.2 Predicates

If $\phi \equiv p$, where $p$ is a predicate $x \cdot c$, we proceed as follows. First, we collect all events that can potentially modify the truth value of $p$. Let $E_p = \{ e \in E \mid \text{var}(e) = x \}$ be the set of those events. All events in $E_p$ assign the same variable, and by condition (ii) of definition 1, they are totally ordered. Let $p = e_1, e_2, \ldots, e_m$
be the linearization of $E_p$, i.e., $\forall i \in [1, m] : e_i \in E_p$ and $|E_p| = m$ and $\forall i \in [1, m] : e_i \prec e_{i+1}$. This sequence can be used to determine slices of $T$ where $p$ is true. Indeed, let $s_1, s_2, ..., s_\ell$ be the sequence of indices splitting $p$ into $\ell - 1$ contiguous blocks $e_{s_1}, ..., e_{s_\ell - 1}, e_{s_\ell}, ..., e_m$ such that the value of $p$ remains the same inside each block and changes in the following block. Formally, this is the sequence satisfying the following constraints ($m = s_{\ell+1} - 1$):

(i) $1 = s_1 < s_2 < ... < s_\ell$
(ii) $\forall i \in [1, \ell], \forall j_1, j_2 \in [s_i, s_i + 1) : (\mid e_{j_1} \models p\mid \iff \mid e_{j_2} \models p\mid)$
(iii) $\forall i \in [1, \ell) : (\mid e_{s_i} \models p\mid \iff \mid e_{s_i + 1} \not\models p\mid)$

Note that, given a block $i \in [1, \ell]$, the value of $p$ in any cuts between $e_{s_i}$ and $e_{s_{i+1} - 1}$ is determined by $e_{s_i}$. This set of cuts can be represented using $A(e_{s_i}) \cap B(e_{s_{i+1} - 1})$, as described above. Thus, for all block $i \in [1, \ell]$ such that $\mid e_{s_i} \models p\mid$, we add $A(e_{s_i}) \cap B(e_{s_{i+1} - 1})$ to $I_p$ initially empty. Additionally, we must take into account the cuts at the beginning and at the end of $T$. If $p$ is satisfied at the beginning of $T$ ($\emptyset \models p\mid$), we must add $B(e_{s_1})$ to $I_p$ and similarly, if $p$ is true at the end of $T$ ($E \models p\mid$), we add $A(e_m)$ to $I_p$. Finally, in order to keep only cuts, we take the intersection with $I_T$.

Lemma 2. Given a trace $T = \langle E, \alpha, \preceq \rangle$ and a predicate $p$, we have that $\text{sets}(I_p) = \{p\}$.

5.3 Boolean Operators

In order to deal with Boolean operators $\phi_1 \lor \phi_2$ (resp. $\phi_1 \land \phi_2$, $\neg \phi_1$), we can use standard operation on IST [18] and compute $I_\phi = I_{\phi_1} \cup I_{\phi_2}$ (resp. $I_\phi = I_{\phi_1} \cap I_{\phi_2}$, $I_\phi = I_{\phi_1} \cap I_{T}$).

Lemma 3. Given a trace $T = \langle E, \alpha, \preceq \rangle$ and CTL formulae $\phi$, $\phi_1$ and $\phi_2$, we have that $\text{sets}(I_{\phi_1 \lor \phi_2}) = \{\phi_1 \lor \phi_2\}$, $\text{sets}(I_{\phi_1 \land \phi_2}) = \{\phi_1 \land \phi_2\}$ and $\text{sets}(I_{\neg \phi}) = \{\neg \phi\}$.

5.4 Existential Modalities

The treatment of existential modalities can be computed through the use of the $\text{pre}^3(\cdot)$ operator, greatest and least fixed point (as explained $e_{s_i}$ in [9]):

\[
\begin{align*}
\text{EX}\phi & = \text{pre}^3(\phi) \\
\text{EG}\phi & = \text{gfp} \lambda X : [\phi] \cap \text{pre}^3(X) \\
\text{E}[\phi_1 U \phi_2] & = \text{lfp} \lambda X : [\phi_2] \cup ([\phi_1] \cap \text{pre}^3(X))
\end{align*}
\]

In order to compute ISTs corresponding to those temporal formulæ, we only need an algorithm for computing symbolically the $\text{pre}^3(\cdot)$ operation. For that, we decompose $\text{pre}^3(\cdot)$ into a function of $\text{pre}^3(\cdot)$, where $\text{pre}^3(X) = \{C \in \text{cuts}(T) \mid \exists \psi \in \text{enabled}(C) \cap P_C : C \cup \{e\} \in X\}$ denotes the set of existential predecessors of $X$ only for process $P_C$. This decomposition is provided by the following lemma.
Lemma 4. Given a trace $T = \langle E, \alpha, \preceq \rangle$ and a subset $X \subseteq \text{cuts}(T)$, we have that $\text{pre}^\gamma(X) = \bigcup_{i \in [1,4]} \text{pre}^\gamma_i(X)$.

The only remaining step is to characterize symbolically $\text{pre}^\gamma(X)$. This characterization is given by the following lemma.

Lemma 5. Given a trace $T = \langle E, \alpha, \preceq \rangle$, and an IST $I$ such that $\text{sets}(I) \subseteq \text{cuts}(T)$, we have that $\text{pre}^\gamma(\text{sets}(I)) = \text{sets}(I^{|X|\times X -1} \cap I^\gamma)$.

Finally, we can define the symbolic existential predecessors on IST.

Definition 3 (Symbolic existential predecessors). Given a trace $T = \langle E, \alpha, \preceq \rangle$ and an IST $I$ such that $\text{sets}(I) \subseteq \text{cuts}(T)$, the symbolic existential predecessors of $I$, noted $\text{spe}^\gamma(I)$, is defined as follows:

$$\text{spe}^\gamma(I) = \bigcup_{i \in [1,4]} \left( I^{|X|\times X -1} \cap I^\gamma \right)$$

As a direct consequence of Lem. 4 and Lem. 5, we get the next theorem.

Theorem 1 (Correctness $\text{spe}^\gamma(\cdot)$). Given a trace $T = \langle E, \alpha, \preceq \rangle$, and an IST $I$ such that $\text{sets}(I) \subseteq \text{cuts}(T)$, we have that $\text{spe}^\gamma(\text{sets}(I)) = \text{sets}(\text{spe}^\gamma(I))$.

5.5 Universal modalities

Universal modalities are treated in a similar way then existential ones. For these, we can use the following equivalence (taken from [10, sec. 2.4]):

$$[\text{AX} \phi] = \text{pre}^\gamma([\phi])$$
$$[\text{AG} \phi] = \text{gfp} \lambda X \cdot [\phi] \cap \text{pre}^\gamma(X)$$
$$[\text{A}[\phi_1 \cup \phi_2]] = \text{lfp} \lambda X \cdot [\phi_1] \cup ([\phi_2] \cap \text{pre}^\gamma(X))$$

Computing ISTs corresponding to universal formulae amounts to defining a symbolical version of the $\text{pre}^\gamma(\cdot)$ operator on sets of cuts. The $\text{pre}^\gamma(\cdot)$ operation can be computed through the equivalence $\text{pre}^\gamma([\phi]) = [\text{AX} \phi] = [\text{AX} \neg \psi] = \text{cuts}(T) \setminus \text{pre}^\gamma([\neg \phi])$. On the other hand, we may compute $\text{pre}^\gamma(\cdot)$ in an alternate way, similarly to what we did for the $\text{pre}^\gamma(\cdot)$. We can decompose $\text{pre}^\gamma(\cdot)$ as a function of $\text{pre}^\gamma(\cdot)$, where $\text{pre}^\gamma(X) = \{C \in \text{cuts}(T) \mid \forall e \in \text{enabled}(C) \cap P_i : C \cup \{e\} \subseteq X\}$ denotes the set of universal predecessors of $X$ only for process $P_i$.

This decomposition is given by the following lemma.

Lemma 6. Given a trace $T = \langle E, \alpha, \preceq \rangle$, and an subset $X \subseteq \text{cuts}(T)$, we have that $\text{pre}^\gamma(X) = \bigcap_{i \in [1,4]} \text{pre}^\gamma_i(X)$.

To compute symbolically $\text{pre}^\gamma(\cdot)$, we need to characterize exactly which cuts are in $\text{pre}^\gamma(X)$. By definition, $\text{pre}^\gamma(X)$ denotes the set of cuts from which all enabled events of process $P_i$ lead to a cut in $X$. $\text{pre}^\gamma(X)$ is composed of two classes of cuts: (i) $\text{blocked}_i = \{C \in \text{cuts}(T) \mid \text{enabled}(C) \cap P_i = \emptyset\}$, the class of cuts in $X$ where process $P_i$ is blocked; and (ii) the class of cuts where the next event of $P_i$ is enabled and leads to a cut in $X$, i.e. $\text{pre}^\gamma(X)$. 
Lemma 7. Given a trace $T = \langle E, \alpha, \preceq \rangle$, and an subset $X \subseteq \text{cuts}(T)$, we have that $\text{pre}_1^2(X) = \text{pre}_2^2(X) \cup \text{blocked}_1$.

We already have a way to compute $\text{pre}_1^2(X)$ symbolically (see Lem.5). The following lemma characterized $\text{blocked}_1$.

Lemma 8. Given a trace $T = \langle E, \alpha, \preceq \rangle$ and a process $P_i \subseteq E$, we have that $C \in \text{blocked}_1$ holds if and only if $\forall e \in E \cap P_i : (\text{pos}(e) = |C \cap P_i| + 1) \implies (\exists e' \in E \setminus C : e' \rightarrow_c e)$.

This result can be used to define an IST $I_{\text{blocked}_1}$ for $\text{blocked}_1$. Indeed, from Lemma 8, we can see that $\text{blocked}_1$ is composed of the set of all the cuts including all events of $P_i$ and the set of all the cuts where the next event to be triggered by $P_i$ is waiting for an incoming communication. Therefore, the computation of $I_{\text{blocked}_1}$ starts with an IST $I_F$ representing the set of sets $C$ of events where process $P_i$ has finished its execution, i.e, where $|C \cap P_i| = |P_i|$. $I_F$ is the same as $I_0$ (cf. sec. 5.1) except for layer $i$, where $u(n_i) = [\text{pos}(P_i) + 1]$. Then, for each incoming communication $e \rightarrow_c e'$ with $e' \in P_i$, we build an IST where process $P_i$ is ready to execute $e'$ and where process $P_i$ has not executed $e$ yet. This IST is the same as $I_0$, except for layer $i$, where $u(n_i) = [\text{pos}(e') - 1, \text{pos}(e') - |P_i|]$ and for layer $\text{pid}(e)$, where $u(\text{pid}(e)) = [0, \text{pos}(e) - 1]$. The IST representing the sets of events where $P_i$ is blocked is obtained by making the union between $I_F$ and all the IST built for the communication edges. Finally, in order to keep only valid cuts, we simply take the intersection of the resulting IST with $I_T$. It is then easy to see, that $I_{\text{blocked}_1}$ contains exactly those cuts satisfying the condition of lem. 8. This leads us to the following symbolic characterization of $\text{pre}_1^2(\cdot)$.

Lemma 9. Given a trace $T = \langle E, \alpha, \preceq \rangle$, and an IST $I$ such that $\text{sets}(I) \subseteq \text{cuts}(T)$, we have that $\text{pre}_1^2(\text{sets}(I)) = \text{sets}(I \cup I_{\text{blocked}_1})$.

We can now define the symbolic universal predecessors.

Definition 4 (Symbolic universal predecessor). Given a trace $T = \langle E, \alpha, \preceq \rangle$ and an IST $I$ such that $\text{sets}(I) \subseteq \text{cuts}(T)$, the symbolic universal predecessors of $I$, noted $\text{pre}^\forall(I)$, is defined as follows:

$$\text{pre}^\forall(I) = \bigcap_{i \in [1, k]} \left( I_{\text{reserved}_i} \cap I_T \right) \cup I_{\text{locked}_i}$$

As a direct consequence of lem. 6 and 9, we get the next theorem.

Theorem 2 (Correctness $\text{pre}^\forall(\cdot)$). Given a trace $T = \langle E, \alpha, \preceq \rangle$, and an IST $I$ such that $\text{sets}(I) \subseteq \text{cuts}(T)$, we have that $\text{pre}^\forall(\text{sets}(I)) = \text{sets}(\text{pre}^\forall(I))$.

Note that it is possible to reduce an *always until* formula to an exist *until* formulae. However, using this translation might explode the size of the formula, and is therefore rejected in favor of a fixed point computation using $\text{pre}^\forall(\cdot)$. 


5.6 Improving the computation of \([\mathsf{EF}_\phi]\) and \([\mathsf{AG}_\phi]\)

To compute \(I_{\mathsf{EF}_\phi}\), one can simply use the equivalence \(\|\mathsf{EF}_\phi\| = \|\mathsf{E}[\uparrow \mathsf{U}_\phi]\| = \mathsf{fp \lambda X} \cdot \|\phi\| \cup (\mathsf{I}\cup \mathsf{pre}^2(X))\), and compute the fix point using the \(\mathsf{pre}^2()\) operator. But, in this particular case, since \(\mathsf{pre}^2(X) \subseteq \mathsf{I}\), this fix point can be reduced to \(\mathsf{fp \lambda X} \cdot \|\phi\| \cup \mathsf{pre}^2(X)\). Using IST, we can directly obtain the result of this fix point symbolically, in one operation using the downward closure. Indeed, we have that \(I_{\mathsf{EF}_\phi} = I_\phi \cap I^-\).

**Lemma 10.** Given a trace \(T = (E, \alpha, \prec)\) of \(k\) processes and a C11 formula \(\phi\), we have that \(\text{sets}(I_\phi \cap I^-) = \|\mathsf{EF}_\phi\|\).

Moreover, the quickest way to compute \([\mathsf{AG}_\phi]\) is generally through the translation \(\mathsf{AG}_\phi \equiv \neg \mathsf{EF}^{-\phi}\) which avoids the fixpoint computation.

6 Experimental results

In this section, we experimentally validate our method. We compare our symbolic approach using IST with a state-of-the-art symbolic model checking (of the trace) using the tool NuSMV [6]. We considered several classical academic examples and compared the running time of our early prototype against NuSMV. Running time was limited to 10 minutes. This seems to be a reasonable assumption considering that the testing should be achieved on a large number of traces. On all the examples we considered, memory consumption was not an issue. The IST manipulated in these examples contains no more than 7000 nodes. These results are presented in table 1. More detailed results are included in app. B.

The first example we considered was the Peterson mutual exclusion protocol with two processes (Pet), where communication is done through shared variables. We used a monitor to check mutual exclusion: \(\mathsf{AG}(n_{crit} < 2)\). On this
property, we experimented two ways of computing $\text{AG}$. The first using the downward closure on IST, as explained in sec. 5.6, and the second using the fixed point on the $\text{spre}^\omega(\cdot)$ operator, as explained in sec. 5.5. As expected the downward closure method is quicker (with the fixedpoint methods the results recorded for 2000, 5000 and 15000 events were 1.45 sec, 15.2 sec and 323.59 sec). We therefore decided to keep only the downward closure method for the remaining experiments (a detailed comparison is included in app. B). Even on this relatively small example, we can already see a big difference in running time: NuSMV runs out of time after 2000 events, whereas our tool can handle 15000 events in the allotted time. We also considered a generalization of this protocol for $n$ processes ($\text{PetN}$) using the same mutual exclusion property. We experimented on 2, 5 and 10 processes. Again, we can see that our approach using IST outperforms the traditional symbolic approach using BDD.

The third model we considered was the alternating-bit protocol between two process $\text{ABP}$, i.e. a sender and a receiver. This time the communication is achieved using asynchronous channel. We verified that every message tagged with a 0 is followed by one with the same tag, which translates in Ctl. as follows: $\text{AG}((\text{sent.msg} = 0) \implies \text{AF(\text{received.msg} = 0)})$. This formula is a bit more complicated. Nonetheless, our method is still scalable up to 5000 events, whereas NuSMV stops after 1000.

The last example we considered was the Dining Philosophers problem ($\text{Phil}$). We considered 3, 5 and 10 philosophers. We verified that whenever philosopher 1 is eating, either he keeps eating until the end of the trace or his left neighbour cannot eat until he stops. In Ctl., this property is expressed as $\text{AG}((\text{state1} = \text{eat}) \implies (\text{AG(state1} = \text{eat}) \lor \text{AG(state0} \neq \text{eat)} \land \text{state1} \neq \text{eat}))$. We deliberately chose a complex formula to test the robustness of our approach. On this example, NuSMV can only handle 3 philosophers with 100 events, with the (too complex) property in the allotted time whereas we can still manage to terminate the analysis on some instances of respectable size. This can be explained by the fact that, in this models, the processes are more independant, thus leading to more interleavings.

For each example, we have computed the size of the lattice of cuts. In the 10 minutes of allotted times, our prototype is capable of handling instances of up to $10^{10}$ cuts, whereas NuSMV stops at $10^5$. This leads us to conclude that our approach is more scalable for this problem.

7 Conclusion and Future Works

In this paper, we have presented a new symbolic technique for the testing of distributed systems, that seems to work well in practice. We still need to validate our approach on more realistic examples. For that purpose, our method will be integrated shortly in our tool Trax\footnote{http://www.ulb.ac.be/di/ssd/cmenter/trax/} and fully interfaced with our distributed controllers design environment $\text{	ext{qSL}}\ [1, 2]$ to allow efficient testing of real industrial distributed controllers. We will also continue to investigate possible further
improvements of our technique, as the one inspired on the RCTL model checking with computation slicing described in [15]. We also intend to investigate the use of our method in different frameworks. A first candidate is the validation of Message Sequence Charts (MSC). We must study how our method can improve the efficiency of existing MSC validation methods.

Finally, from a theoretical point of view, the exact complexity class of CTL over partial order trace is not known. We plan to determine that full CTL and some interesting fragments (like RCTL).

References

A Proofs of section 5

A.1 Preliminary results

Before presenting the proofs of sec. 5, we need to establish a few preliminaries. First, for a $k$-tuple $\mathbf{r}$, we note $C_{\mathbf{r}} = \{ e \in E \mid \text{pos}(e) \leq x_{\text{id}(e)} \}$ the subset of events represented by $\mathbf{r}$ represents. We will also need the following results.

Lemma 11. Given a trace $T = \langle E, \alpha, \leq \rangle$, and a process $P_i \subseteq E$, we have that
\[
\forall C \in \text{cuts}(T) : \text{enabled}(C) \cap P_i \subseteq \{ e \} \text{ where } e \in E \text{ is such that } \text{pos}(e) = |C \cap P_i| + 1.
\]

Proof. First, note that $\text{enabled}(C) \cap P_i \subseteq P_i$. We therefore only consider events of $P_i$. Let $e$ be such an event. If $\text{pos}(e) < |C \cap P_i| + 1$, we have that $e \in C$ which implies $e \notin \text{enabled}(C)$. On the other hand $\text{pos}(e) > |C \cap P_i| + 1$, the event $e'$ such that $\text{pos}(e') = |C \cap P_i| + 1$ does not belong to $C$ but is in $\{ e \setminus \{ e \} \}$ and again $e \notin \text{enabled}(C)$. The only remaining possibility is that $\text{pos}(e) = |C \cap P_i| + 1$.

Lemma 12. Given a trace $T = \langle E, \alpha, \leq \rangle$ of $k$ processes and $C, C' \in \text{cuts}(T)$, if $C \subseteq C'$ then there exists a sequence $C_0, C_1, C_2, \ldots, C_m$ such that $(C = C_0) \land (C_m = C') \land (\forall i \in [1, m] : (C_{i-1} \in \text{pre}^2(\{ C_i \})) \land (C_i \in \text{cuts}(T)))$.

Proof. We proceed by induction on $|C' \setminus C|

- **base case:** if $|C' \setminus C| = 0$, then $C = C'$ and the proof is immediate. The sequence is simply $C = C_0 = C_m = C'$.
- **induction step:** if $|C' \setminus C| > 0$, we have that $\exists e \in (C' \setminus C) : e \in \text{enabled}(C)$. Indeed, since $\leq$ is a partial order, and since $|C' \setminus C|$ is non-empty, we know that $(C' \setminus C)$ has at least one minimal element, i.e., an event $e$ such that $\forall e' \in (C' \setminus C) : e' \prec e$. Since $C'$ is a cut and $e \in C'$, we have that $\{ e \setminus \{ e \} \} \subseteq C$. Since $e$ is minimal in $C' \setminus C$, that $\{ e \setminus \{ e \} \} \subseteq C$. It follows directly that $e \in \text{enabled}(C)$. We know therefore that $C \cup \{ e \} \in \text{cuts}(T)$ and since $e \in C'$ then $C \cup \{ e \} \subseteq C'$. By induction there exists a sequence $C_0, \ldots, C_{m-1}$ of cuts between $C \cup \{ e \}$ and $C'$. The sequence for $C$, $C'$ is then given by $C, C_0, \ldots, C_{m-1}$.

A.2 Proof of lemma 1

Given a trace $T = \langle E, \alpha, \leq \rangle$, we have that:

\[
\text{sets}(I_T) = [\top]
\]

Proof. First, note that $[\top] = \{ C \in \text{cuts}(T) \mid C \models \top \} = \text{cuts}(T)$. Therefore, proving $\text{sets}(I_T) = [\top]$ amounts to proving that $\text{sets}(I_T) = \text{cuts}(T)$. Consequently, we prove the inclusion in both ways.
For \( \text{cuts}(T) \subseteq \text{sets}(I_T) \), we prove that \( \text{cuts}(T) \subseteq \text{sets}(I) \) is an invariant of the algorithm. At the initial step, we start with \( I = I_0 \). It can be easily proven that \( \text{sets}(I_0) = \{ C \subseteq E \mid \forall e \in C : e \in F_{\text{pid}}(e) \subseteq C \} \). Condition (i) of definition 1 then implies that every \( \subseteq \)-closed subset of \( E \) belongs to \( \text{sets}(I_0) \). Therefore, we have that \( \text{cuts}(T) \subseteq \text{sets}(I_0) \). Then, at each step of the algorithm, we remove \( \text{cuts} \) from \( I \) to take into account a communication \( e \rightarrow_c e' \). For that we compute two BSTs \( B(e) \) and \( A(e') \). It can be easily proven that \( \text{sets}(B(e)) = \{ C \subseteq \text{sets}(I_0) \mid e \notin C \} \) and that \( \text{sets}(A(e')) = \{ C \subseteq \text{sets}(I_0) \mid e' \subseteq C \} \). Thus, \( \text{sets}(A(e') \cap B(e)) = \text{sets}(A(e')) \cap \text{sets}(B(e)) = \{ C \subseteq \text{sets}(I_0) \mid (e' \subseteq C) \land (e \notin C) \} \). Therefore, since \( e \not\subseteq e' \), any \( C \subseteq \text{sets}(A(e') \cap B(e)) \) is not \( \subseteq \)-closed. It follows directly that \( \text{sets}(A(e') \cap B(e)) \cap \text{cuts}(T) = \emptyset \), and that \( \text{cuts}(T) \subseteq \text{sets}(I \setminus (A(e') \cap B(e))) \). We can therefore conclude that \( \text{cuts}(T) \subseteq \text{sets}(I) \) is an invariant and finally that \( \text{cuts}(T) \subseteq \text{sets}(I_T) \).

For \( \text{sets}(I_T) \subseteq \text{cuts}(T) \), we equivalently prove that \( \forall C \subseteq E : (C \not\subseteq \text{cuts}(T)) \Rightarrow (C \not\subseteq \text{sets}(I_T)) \). If \( C \not\subseteq \text{cuts}(T) \), then \( \exists e \in C, e' \in e : e' \not\subseteq C \). In this case, since \( e' \in e \), there exists in \( T \) a sequence of events \( e' = e_1, e_2, \ldots, e_i = e \) and \( \forall i \in [1, l] : e_i \rightarrow e_{i+1} \). Moreover, since \( e' \not\subseteq C \), \( \exists i \in [1, l] : (e_{i+1} \in C) \land (e_i \not\subseteq C) \). From there, we have two cases:

(i) if \( \text{pid}(e_i) = \text{pid}(e_{i+1}) \), then \( C \) cannot be represented using a \( k \)-uple.

Indeed, let \( (x_1, x_2, \ldots, x_k) \) be such an hypothetical \( k \)-uple. If \( e_{i+1} \in C \), then \( \text{pos}(e_{i+1}) \leq \text{pos}(e_i) \). But since \( e_i \rightarrow e_{i+1} \), \( \text{pos}(e_i) \leq \text{pos}(e_{i+1}) \) and \( e_i \) is also in \( C \). In this case, we therefore know that \( C \not\subseteq \text{sets}(I_T) \).

(ii) on the other hand, if \( \text{pid}(e_i) \neq \text{pid}(e_{i+1}) \), then there is a communication \( e_i \rightarrow_c e_{i+1} \). Thus, in some step of the algorithm, we will compute \( B(e_i) \) and \( A(e_{i+1}) \) such that \( \text{sets}(A(e_{i+1}) \cap B(e_i)) = \{ C \subseteq \text{sets}(I_0) \mid (e_{i+1} \in C) \land (e_i \not\subseteq C) \} \). Therefore, \( C \) will be removed at this step.

In both case, \( C \not\subseteq \text{sets}(I_T) \) and we can conclude that \( \text{sets}(I_T) \subseteq \text{cuts}(T) \).

A.3 Proof of lemma 2

Given a trace \( T = (E, \alpha, \preceq) \) and a predicate \( p \), we have that

\[
\text{sets}(I_p) = [p]
\]

Proof. We need to prove the inclusion in both ways:

- For \( \text{sets}(I_p) \subseteq [p] \), we prove equivalently that \( \forall C \in \text{sets}(I_p) : C \models p \). First note that \( \forall C \in \text{sets}(I_p) : C \subseteq \text{sets}(I_T) = \text{cuts}(T) \). Therefore, \( C \) is \( \subseteq \)-closed. Then, we have three possibilities:

(i) \( C \in \text{sets}(B(e_n)) \); in this case, since \( e_n = e_1 \not\subseteq C \) and since \( C \) is \( \subseteq \)-closed, we have that \( \nu_C(p) = \nu_B(p) \). However, since \( C \) was added to \( I_p \), by construction we have that \( \emptyset \models p \) which in turn implies that \( C \models p \).

(ii) \( \exists i \in [1, l] : C \in \text{sets}(A(e_i) \cap B(e_{i+1})) \); in this case, since \( e_i \in C, e_{i+1} \not\subseteq C \) and since \( C \) is \( \subseteq \)-closed, we have that \( \{ e_j \mid j \in [s_i, s_{i+1}) \} \subseteq C \). However, by construction, \( \forall j \in [s_i, s_{i+1}) : e_j \models p \). Thus, if \( C_i \) denotes \( \{ e_n \} \), it follows that \( \nu_C(p) = \nu_{C_i}(p) \), which in turn implies \( C \models p \).
(iii) $C \in \text{sets}(A(c_{\mathcal{E}}))$: in this case since $c_{\mathcal{E}} = c_{\mathcal{M}} \in C$ and since $C$ is closed, $v_C(p) = v_{\mathcal{E}}(p)$. However, since $C$ was added to $\mathcal{I}_p$, we know by construction that $E \models p$, which in turn implies that $C \models p$.

- For $[p] \subseteq \text{sets}(\mathcal{I}_p)$, we prove equivalently that $\forall C \in \text{cuts}(\mathcal{T}) : (C \models p) \Rightarrow (C \in \text{sets}(\mathcal{I}_p))$. Again, we have three possibilities:
  
  (i) $C \cap E_p = \emptyset$: in this case, we have that $v_C(p) = v_{\mathcal{E}}(p)$ and $C \models p$, that $\emptyset \models p$. Therefore, since $C \in B(c_{\mathcal{E}})$ and since $C \in \text{cuts}(\mathcal{T}) = \text{sets}(\mathcal{T}_p)$, we can conclude that $C \in \text{sets}(\mathcal{I}_p)$.

  (ii) $\emptyset \neq C \cap E_p \neq E_p$: in this case, let $i = \max\{\{j \in [1, l] \mid c_{\mathcal{E}_j} \subseteq C\}\}$. Since $c_{\mathcal{E}_j} \subseteq C$, we have that $C \in \text{sets}(A(c_{\mathcal{E}_j}))$. Moreover, we know that $c_{\mathcal{E}_{i+1}} \notin C$, otherwise, $i$ would not be maximal. It follows that $C \in \text{sets}(B(c_{\mathcal{E}_{i+1}}))$ and that $C \in \text{sets}(A(c_{\mathcal{E}_{i+1}}) \cap B(c_{\mathcal{E}_{i+1}}))$. However, if $C_i$ denotes $c_{\mathcal{E}_i}$, since $C \models p$ and $v_{C_i}(p) = v_{\mathcal{E}}(p)$, we have that $C_i \models p$. Therefore, $A(c_{\mathcal{E}_i}) \cap B(c_{\mathcal{E}_{i+1}})$ will be added to $\mathcal{I}_p$ in the construction. Finally, since $C \in \text{cuts}(\mathcal{T}) = \text{sets}(\mathcal{T}_p)$, we can conclude that $C \in \text{sets}(\mathcal{I}_p)$.

(iii) $C \cap E_p = E_p$: in this case, we have that $v_C(p) = v_{\mathcal{E}}(p)$ and since $C \models p$, that $E \models p$. Therefore, since $C \in A(c_{\mathcal{E}})$ and since $C \in \text{cuts}(\mathcal{T}) = \text{sets}(\mathcal{T}_p)$, we can conclude that $C \in \text{sets}(\mathcal{I}_p)$.

### A.4 Proof of lemma 3

Given a trace $\mathcal{T} = (E, \alpha, \preceq)$ and CTL formulae $\phi$, $\phi_1$ and $\phi_2$, we have that:

\[
\begin{align*}
\text{sets}(\mathcal{I}_{\phi_1 \lor \phi_2}) &= \left[ \phi_1 \cup \phi_2 \right] \\
\text{sets}(\mathcal{I}_{\phi_1 \land \phi_2}) &= \left[ \phi_1 \cap \phi_2 \right] \\
\text{sets}(\mathcal{I}_{-\phi}) &= \left[ \neg \phi \right]
\end{align*}
\]

**Proof.** This is a direct consequence of the following equalities:

\[
\begin{align*}
\text{sets}(\mathcal{I}_{-\phi}) &= \text{sets}(\mathcal{I}_\emptyset \cap \mathcal{T}) \\
&= \text{sets}(\mathcal{I}_\emptyset) \cap \text{sets}(\mathcal{T}) \\
&= \left( \text{sets}(\mathcal{N}_\emptyset) \setminus \text{sets}(\mathcal{I}_\emptyset) \right) \cap \text{sets}(\mathcal{T}) \\
&= \left( \text{sets}(\mathcal{N}_\emptyset) \cap \text{sets}(\mathcal{T}) \right) \setminus \left( \text{sets}(\mathcal{I}_\emptyset) \cap \text{sets}(\mathcal{T}) \right) \\
&= \text{sets}(\mathcal{T}) \setminus \text{sets}(\mathcal{I}_\emptyset) \\
&= \left[ \mathcal{T} \right] \setminus \left[ \phi \right] \\
&= \left[ \neg \phi \right] \\
\text{sets}(\mathcal{I}_{\phi_1 \lor \phi_2}) &= \text{sets}(\mathcal{I}_{\phi_1} \cup \mathcal{I}_{\phi_2}) \\
&= \text{sets}(\mathcal{I}_{\phi_1}) \cup \text{sets}(\mathcal{I}_{\phi_2}) \\
&= \left[ \phi_1 \right] \cup \left[ \phi_2 \right] \\
&= \left[ \phi_1 \lor \phi_2 \right] \\
\text{sets}(\mathcal{I}_{\phi_1 \land \phi_2}) &= \text{sets}(\mathcal{I}_{\phi_1} \cap \mathcal{I}_{\phi_2}) \\
&= \text{sets}(\mathcal{I}_{\phi_1}) \cap \text{sets}(\mathcal{I}_{\phi_2}) \\
&= \left[ \phi_1 \right] \cap \left[ \phi_2 \right] \\
&= \left[ \phi_1 \land \phi_2 \right]
\end{align*}
\]
A.5 Proof of lemma 4

Given a trace $T = \langle E, \alpha, \preceq \rangle$ and a subset $X \subseteq \text{cuts}(T)$, we have that:

$$\text{pre}^3(X) = \bigcup_{i \in [1,k]} \text{pre}^3_i(X)$$

Proof. This is a direct consequence of the following equalities:

$$\text{pre}^3(X) = \{C \in \text{cuts}(T) | \exists e \in \text{enabled}(C) : C \cup \{e\} \in X\}$$
$$= \{C \in \text{cuts}(T) | \exists e \in \text{enabled}(C) \cap E : C \cup \{e\} \in X\}$$
$$= \{C \in \text{cuts}(T) | \exists e \in \text{enabled}(C) \cap (\bigcup_{i \in [1,k]} P_i) : C \cup \{e\} \in X\}$$
$$= \{C \in \text{cuts}(T) | \exists e \in \bigcup_{i \in [1,k]} \text{enabled}(C) \cap (P_i) : C \cup \{e\} \in X\}$$
$$= \bigcup_{i \in [1,k]} \{C \in \text{cuts}(T) | \exists e \in \text{enabled}(C) \cap P_i : C \cup \{e\} \in X\}$$
$$= \bigcup_{i \in [1,k]} \text{pre}^3_i(X)$$

A.6 Proof of lemma 5

Given a trace $T = \langle E, \alpha, \preceq \rangle$, and an IST $I$ such that $\text{sets}I \subseteq \text{cuts}(T)$, we have that:

$$\text{pre}^3_\preceq(\text{sets}(I)) = \text{sets}(I^{[v_i=v_i-1]} \cap I_\preceq)$$

Proof. This is a direct consequence of the following equivalences:

$$\text{sets}(I^{[v_i=v_i-1]} \cap I_\preceq)$$
$$= \{C \in \text{cuts}(T) | \exists x_i \in \text{tuple}(I^{[v_i=v_i-1]}), C = C_{-\text{I}_\preceq}\}$$
$$= \{C \in \text{cuts}(T) | \exists x_i' \in \text{tuple}(I), C = C_{-\text{I}_\preceq} \setminus \{e \in P_i \mid \text{pos}(e) = x_i'\}\}$$
$$= \{C \in \text{cuts}(T) | \exists x_i' \in \text{tuple}(I), C = C_{-\text{I}_\preceq} \setminus \{e \in P_i \mid \text{pos}(e) = |C \cap P_i|\}\}$$
$$= \{C \in \text{cuts}(T) | \exists x_i' \in \text{tuple}(I), C = C_{-\text{I}_\preceq} \setminus \{e \in P_i \mid \text{pos}(e) = |C \cap P_i| + 1\}\}$$
$$= \{C \in \text{cuts}(T) | \exists e \in \text{enabled}(C) \cap P_i, C' \in \text{sets}(I), C' = C \cup \{e\}\}$$
$$= \{C \in \text{cuts}(T) | \exists e \in \text{enabled}(C) \cap P_i, C' \in \text{sets}(I), C' = C \cup \{e\}\}$$
$$= \text{pre}^3_\preceq(\text{sets}(I))$$

A.7 Proof of lemma 6

Given a trace $T = \langle E, \alpha, \preceq \rangle$, and an subset $X \subseteq \text{cuts}(T)$, we have that:

$$\text{pre}^\gamma(X) = \bigcap_{i \in [1,k]} \text{pre}^\gamma_i(X)$$
Proof. This is a direct consequence of the following equalities:

\[
\begin{align*}
\text{pre}^\gamma(X) &= \{ C \in \text{cuts}(T) \mid \forall e \in enabled(C) : C \cup \{ e \} \in X \} \\
&= \{ C \in \text{cuts}(T) \mid \neg(\exists e \in enabled(C) : C \cup \{ e \} \not\in X) \} \\
&= \{ C \in \text{cuts}(T) \mid \neg(\exists e \in enabled(C) : C \cup \{ e \} \in \text{cuts}(T) \setminus X) \} \\
&= \{ C \in \text{cuts}(T) \mid \neg C \in \text{pre}^\gamma(\text{cuts}(T) \setminus X) \} \\
&= \text{cuts}(T) \setminus \bigcup_{i \in [1,\ldots,\lambda]} \text{pre}^\gamma(\text{cuts}(T) \setminus X) \\
&= \bigcap_{i \in [1,\ldots,\lambda]} (\text{cuts}(T) \setminus \text{pre}^\gamma(\text{cuts}(T) \setminus X)) \\
&= \bigcap_{i \in [1,\ldots,\lambda]} (\text{cuts}(T) \setminus \{ C \in \text{cuts}(T) \mid \exists e \in enabled(C) \cap P_i : C \cup \{ e \} \in \text{cuts}(T) \setminus X \}) \\
&= \bigcap_{i \in [1,\ldots,\lambda]} (\text{cuts}(T) \setminus \{ C \in \text{cuts}(T) \mid \exists e \in enabled(C) \cap P_i : C \cup \{ e \} \not\in X \}) \\
&= \bigcap_{i \in [1,\ldots,\lambda]} (\text{cuts}(T) \setminus \{ C \in \text{cuts}(T) \mid \neg(\forall e \in enabled(C) \cap P_i : C \cup \{ e \} \in X) \}) \\
&= \bigcap_{i \in [1,\ldots,\lambda]} (\text{cuts}(T) \setminus \{ C \in \text{cuts}(T) \mid \neg C \in \text{pre}^\gamma(X) \}) \\
&= \bigcap_{i \in [1,\ldots,\lambda]} (\text{cuts}(T) \setminus \{ C \in \text{cuts}(T) \mid \text{pre}^\gamma(X) \}) \\
&= \bigcap_{i \in [1,\ldots,\lambda]} \text{pre}^\gamma(X)
\end{align*}
\]

A.8 Proof of lemma 7

Given a trace \( T = (E, \alpha, \leq) \), and an subset \( X \subseteq \text{cuts}(T) \), we have that

\[
\text{pre}^\gamma(X) = \text{pre}^\gamma(X) \cup \text{blocked}_i
\]

Proof. We proceed by proving inclusion in both ways.

- For \( \text{pre}^\gamma(X) \subseteq \text{pre}^\gamma(X) \cup \text{blocked}_i \), let us examine a cut \( C \in \text{pre}^\gamma(X) \). Either, \( enabled(C) \cap P_i = \emptyset \), in which case, \( C \in \text{blocked}_i \), or \( enabled(C) \cap P_i \neq \emptyset \), in which case \( \exists e \in enabled(C) \cap P_i : C \cup \{ e \} \in X \) holds, which, in turn implies that \( C \in \text{pre}^\gamma(X) \).

- For \( \text{pre}^\gamma(X) \cup \text{blocked}_i \subseteq \text{pre}^\gamma(X) \) and that \( \text{pre}^\gamma(X) \subseteq \text{pre}^\gamma(X) \) independently. The proof for \( \text{blocked}_i \) is straightforward. Indeed, for a cut \( C \in \text{blocked}_i \), we know that \( enabled(C) \cap P_i = \emptyset \). The universal quantification over \( enabled(C) \cap P_i \) in \( \text{pre}^\gamma(X) \) is therefore trivially satisfied, and \( C \in \text{pre}^\gamma(X) \). The proof for \( \text{pre}^\gamma(X) \) is also quite simple. For a cut \( C \in \text{pre}^\gamma(X) \), we know that \( \exists e \in enabled(C) \cap P_i : C \cup \{ e \} \in X \). This implies that \( e \in enabled(C) \cap P_i \neq \emptyset \). The only possibility for \( enabled(C) \cap P_i \) is a singleton containing the next event of \( P_i \), by lem.11. Therefore we have that \( \exists e \in enabled(C) \cap P_i : C \cup \{ e \} \in X \) implies \( \forall e \in enabled(C) \cap P_i : C \cup \{ e \} \in X \), which, in turn implies \( C \in \text{pre}^\gamma(X) \).

A.9 Proof of lemma 8

Given a trace \( T = (E, \alpha, \leq) \) and a process \( P_i \subseteq E \), we have that \( C \in \text{blocked}_i \) if and only if:

\[
\forall e \in E \cap P_i : (\text{pos}(e) = |C \cap P_i| + 1) \implies (\exists e' \in E \setminus C : e' \rightarrow_C e)
\]
Proof. We prove the equivalence in both ways.

- (⇒) Let us examine a cut C such that ∀e ∈ E ∩ P1 : (pos(e) = |C ∩ P1| + 1) ⇒ (∃e′ ∈ E \ C : e′ → e). By lem. 11, we know that enabled(C) ∩ P1 ⊆ {e} where e is such that pos(e) = |C ∩ P1| + 1. We have then that ∃e′ ∈ E \ C : e′ → e. Since e′ ∉ C and since e′ → e implies that e′ ≤ e, we have that e ∉ enabled(C) hence enabled(C) ∩ P1 = ∅ and C ∈ blocked.

- (⇒) We proceed by proving the contraposition. Let us examine a cut C such that ∃e ∈ E ∩ P1 : (pos(e) = |C ∩ P1| + 1) ∧ (∀e′ ∈ E \ C : e′ ∉ e). Since pos(e) > |C ∩ P1|, we can conclude that e ∉ C. Moreover, since C ∈ cuts(T), we have that ∀e′ ∈ P1 : (pos(e′) ≤ |C ∩ P1|) ⇒ (e′ ∈ C). Finally, ∀e′ ∈ E \ C : e′ ∉ e implies that ∃e′ ∈ E \ P1 : e′ ≤ e. We can deduce that e ∈ enabled(C) and that enabled(C) ∩ P1 = ∅, which implies that C ∉ blocked.

A.10 Proof of lemma 9

Given a trace T = ⟨E, α, ≤⟩, and an IST I such that I ⊆ cuts(T), we have that:

\[
\text{pre}^2_\sigma(\text{sets}(I)) = \text{sets}(\bigcup_{i=0}^{\text{blocks}(I) - 1} \cap I_T) \cup I_{\text{blocked}}.
\]

Proof. This is a direct consequence of the following equalities:

\[
\begin{align*}
\text{pre}^2_\sigma(\text{sets}(I)) &= \text{pre}^2_\sigma(\text{sets}(I)) \cup \text{blocks} \\
&= \text{sets}(\bigcup_{i=0}^{\text{blocks}(I) - 1} \cap I_T) \cup \text{blocks} \quad \text{(by lem. 7)} \\
&= \text{sets}(\bigcup_{i=0}^{\text{blocks}(I) - 1} \cap I_T) \cup \text{sets}(I_{\text{blocked}}) \quad \text{(by constr. of I_{\text{blocked}})} \\
&= \text{sets}(\bigcup_{i=0}^{\text{blocks}(I) - 1} \cap I_T) \cup I_{\text{blocked}}.
\end{align*}
\]

A.11 Proof of lemma 10

Given a trace T = ⟨E, α, ≤⟩ of k processes and a Cnt. formula φ, we have that:

\[
\text{sets}(I_{\phi}) = [[EFφ]]
\]

Proof. This is a direct consequence of the following equalities:

\[
\begin{align*}
\text{sets}(I_{\phi}) &= \text{sets}(I_T) \\
&= \text{sets}(I_T) \cap \text{sets}(I_T) \\
&= \text{sets} \left( \{ \overline{e} \in [^k] \mid \exists \overline{e}' \in \text{tuple}(I_T) : \overline{e} \leq \overline{e}' \} \cap \text{sets}(I_T) \right) \\
&= \{ C \subseteq E \mid \exists C' \in \text{sets}(I_T) : C \subseteq C' \} \cap \text{cuts}(T) \\
&= \{ C \in \text{cuts}(T) \mid \exists C' \in \text{sets}(I_T) : C \subseteq C' \} \\
&= \{ C \in \text{cuts}(T) \mid \exists C' \in \phi : C \subseteq C' \} \\
&= \{ C \in \text{cuts}(T) \mid \exists C' \in \phi : (C_0 = C) \land (C_m = C') \} \\
&= \{ C \in \text{cuts}(T) \mid (\forall i \in [1, m] : (C_{i-1} \in \text{pre}^i(\{C_i\})) \land (C_i \in \text{cuts}(T))) \} \\
&= [[EFφ]]
\end{align*}
\]
B Details of the experimental results

<table>
<thead>
<tr>
<th>Server</th>
<th>Comm. Scal.</th>
<th>TrX (loc.)</th>
<th>TrX (e.o.)</th>
<th>TrX (loc.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>7</td>
<td>280</td>
<td>0.01 sec.</td>
<td>0.01 sec.</td>
</tr>
<tr>
<td>2000</td>
<td>15</td>
<td>600</td>
<td>0.02 sec.</td>
<td>0.03 sec.</td>
</tr>
<tr>
<td>5000</td>
<td>28</td>
<td>1200</td>
<td>0.30 sec.</td>
<td>0.02 sec.</td>
</tr>
<tr>
<td>10000</td>
<td>75</td>
<td>2600</td>
<td>0.70 sec.</td>
<td>0.07 sec.</td>
</tr>
<tr>
<td>20000</td>
<td>147</td>
<td>5200</td>
<td>1.16 sec.</td>
<td>0.16 sec.</td>
</tr>
<tr>
<td>50000</td>
<td>286</td>
<td>14000</td>
<td>15.3 sec.</td>
<td>7.33 sec.</td>
</tr>
<tr>
<td>100000</td>
<td>529</td>
<td>27995</td>
<td>36.54 sec.</td>
<td>58.12 sec.</td>
</tr>
<tr>
<td>150000</td>
<td>1063</td>
<td>49930</td>
<td>302.59 sec.</td>
<td>198.15 sec.</td>
</tr>
<tr>
<td>200000</td>
<td>1951</td>
<td>58848</td>
<td>&quot;</td>
<td>578.74 sec.</td>
</tr>
</tbody>
</table>

Table 2. Peterson mutual exclusion protocol for two processes; the property is AG(merit < 2), where merit is the number of process in their critical section. ‘‘’ indicates that the assertion did not terminate in under 10 min. E.g. (e.o.) indicates that AG was computed using a fixed point (downward closure).

![Peterson algorithm for 2 processes](image)

Fig. 3. Graphical results for the simple Peterson with 2 processes
### Table 3. Alternating Bit protocol with one sender and one receiver; the property is $AG(\text{sent}.msg = 0) \Rightarrow AF(\text{received}.msg = 0)$; †† indicates that the execution did not terminate in under 10 min.

<table>
<thead>
<tr>
<th>#events</th>
<th>#comm.</th>
<th>#cuts</th>
<th>Trax</th>
<th>NuSMV</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>22</td>
<td>1653</td>
<td>0.32 sec.</td>
<td>1.44 sec.</td>
</tr>
<tr>
<td>200</td>
<td>32</td>
<td>2319</td>
<td>0.46 sec.</td>
<td>1.67 sec.</td>
</tr>
<tr>
<td>500</td>
<td>75</td>
<td>5278</td>
<td>1.82 sec.</td>
<td>14.79 sec.</td>
</tr>
<tr>
<td>1000</td>
<td>149</td>
<td>11314</td>
<td>13.00 sec.</td>
<td>207728 sec.</td>
</tr>
<tr>
<td>2000</td>
<td>276</td>
<td>21075</td>
<td>27.56 sec. ††</td>
<td></td>
</tr>
<tr>
<td>5000</td>
<td>682</td>
<td>51351</td>
<td>257.29 sec. ††</td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td>1360</td>
<td>102714 ††</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 4. Peterson mutual exclusion protocol generalized for $n$ processes; the property is $AG(\text{ncrit} < 2)$ where $\text{ncrit}$ is the number of process in their critical section; †† indicates that the execution did not terminate in under 10 min.

<table>
<thead>
<tr>
<th>#processes</th>
<th>#events</th>
<th>#comm.</th>
<th>#cuts</th>
<th>Trax</th>
<th>NuSMV</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>100</td>
<td>7</td>
<td>107</td>
<td>0.00 sec.</td>
<td>0.03 sec.</td>
</tr>
<tr>
<td>2</td>
<td>200</td>
<td>14</td>
<td>206</td>
<td>0.00 sec.</td>
<td>0.09 sec.</td>
</tr>
<tr>
<td>2</td>
<td>500</td>
<td>34</td>
<td>506</td>
<td>0.01 sec.</td>
<td>0.42 sec.</td>
</tr>
<tr>
<td>2</td>
<td>1000</td>
<td>68</td>
<td>1016</td>
<td>0.04 sec.</td>
<td>2.80 sec.</td>
</tr>
<tr>
<td>2</td>
<td>2000</td>
<td>134</td>
<td>2039</td>
<td>0.20 sec.</td>
<td>294.46 sec.</td>
</tr>
<tr>
<td>2</td>
<td>5000</td>
<td>334</td>
<td>5106</td>
<td>6.44 sec. ††</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>10000</td>
<td>667</td>
<td>10047</td>
<td>48.10 sec. ††</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>20000</td>
<td>1335</td>
<td>20237</td>
<td>300.00 sec. ††</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>56</td>
<td>401</td>
<td>0.03 sec.</td>
<td>0.03 sec.</td>
</tr>
<tr>
<td>5</td>
<td>200</td>
<td>97</td>
<td>1009</td>
<td>0.07 sec.</td>
<td>0.16 sec.</td>
</tr>
<tr>
<td>5</td>
<td>500</td>
<td>202</td>
<td>2618</td>
<td>0.33 sec.</td>
<td>0.85 sec.</td>
</tr>
<tr>
<td>5</td>
<td>1000</td>
<td>402</td>
<td>5393</td>
<td>2.04 sec.</td>
<td>13.74 sec.</td>
</tr>
<tr>
<td>5</td>
<td>1500</td>
<td>602</td>
<td>11732</td>
<td>6.82 sec. ††</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2000</td>
<td>773</td>
<td>13885</td>
<td>12.61 sec. ††</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5000</td>
<td>1926</td>
<td>27835</td>
<td>176.62 sec. ††</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>10000</td>
<td>3801</td>
<td>55535 ††</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>328</td>
<td>14072</td>
<td>0.82 sec.</td>
<td>0.01 sec.</td>
</tr>
<tr>
<td>10</td>
<td>200</td>
<td>314</td>
<td>22173</td>
<td>0.79 sec.</td>
<td>0.46 sec.</td>
</tr>
<tr>
<td>10</td>
<td>500</td>
<td>493</td>
<td>43808</td>
<td>2.12 sec.</td>
<td>1.60 sec.</td>
</tr>
<tr>
<td>10</td>
<td>1000</td>
<td>706</td>
<td>7267</td>
<td>5.42 sec.</td>
<td>4.20 sec.</td>
</tr>
<tr>
<td>10</td>
<td>1500</td>
<td>1024</td>
<td>93440</td>
<td>7.53 sec.</td>
<td>150.23 sec.</td>
</tr>
<tr>
<td>10</td>
<td>2000</td>
<td>1405</td>
<td>147219</td>
<td>27.01 sec. ††</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>5000</td>
<td>3255</td>
<td>203002</td>
<td>147.89 sec. ††</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10000</td>
<td>6361</td>
<td>452897 ††</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 4. Graphical result for Peterson with 10 processes

<table>
<thead>
<tr>
<th>N processes</th>
<th>Qmax</th>
<th>Qtrue</th>
<th>Time (sec.)</th>
<th>ThrA</th>
<th>NvNSAV</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>200</td>
<td>30</td>
<td>2869</td>
<td>0.15</td>
<td>6.33</td>
</tr>
<tr>
<td>3</td>
<td>200</td>
<td>61</td>
<td>5379</td>
<td>1.11</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>500</td>
<td>100</td>
<td>14587</td>
<td>8.24</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1000</td>
<td>321</td>
<td>207789</td>
<td>28.99</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2000</td>
<td>653</td>
<td>530681</td>
<td>306.22</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5000</td>
<td>1384</td>
<td>164301</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>68</td>
<td>41337</td>
<td>0.25</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>200</td>
<td>101</td>
<td>466088</td>
<td>21.05</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>500</td>
<td>229</td>
<td>1210522</td>
<td>125.56</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1000</td>
<td>517</td>
<td>3121068</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>700</td>
<td>130</td>
<td>3778522961</td>
<td>1.67</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>2000</td>
<td>199</td>
<td>790916772</td>
<td>26.94</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>5000</td>
<td>474</td>
<td>128266861</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5. Dining philosophers using showed logic for the fails: the property is $\forall G(s. s.t. = \text{eat}) \rightarrow (\exists G(s. s.t. = \text{eat}) \land (s. s.t. \neq \text{eat}) \lor (s. s.t. \neq \text{eat})$, where status is the state of philosopher $x$ (eat, hungry is false); $\uparrow$ indicates that the execution did not terminate in under 10 min.