Testing Distributed Systems through Symbolic Model Checking of Traces

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 trace. In practice, this trace can be obtained through a 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 k-uples 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, 2] 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. [3–5]) can be used to validate parts of models. For instance, such tools can be used to check that, in the system, 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.

[⋆] Supported by the Belgian National Science Foundation (FNRS) under a FRIA grant.
[⋆⋆] Research fellow supported by the Belgian National Science Foundation (FNRS).

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 [6, 7] or symbolic model checking [8–10].

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 [11], can be obtained through correct code instrumentation using, for instance, vector clocks [11, 12].

Hence in this case, an execution is a finite *trace*, i.e. a partially ordered set of events. 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* ϕ amounts to verifying that every sequential execution, *compatible* with the partial order, satisfies ϕ or, in other terms, model checking ϕ on the corresponding *trace*. Unfortunately, this problem is hard [13], 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 [14], A. Sen and Garg present the temporal logic RCTL (for *regular*-CTL), which is a subset of the branching time temporal logic CTL [15] and shows that the compact symbolic data structure called *computation slice* [16], can be used to efficiently compute all global states which satisfy a RCTL formula.

However, RCTL does not include such simple CTL property as $\mathsf{AG}(p \implies \mathsf{AF}(q \wedge r))$, 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) [17, 18]. 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 [10] and implemented in the tool NuSMV [5].

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, future works are given in section 7. **Note** : all proofs from sec. 5 are included for the reviewers 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 [19], 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* [20, 21] and *global* [22–24] trace logics have been defined. However, in our case, the *trace* is an abstraction of a distributed execution (or of a scenario) and models a set of possible interleaving 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 ϕ. Garg and Chase showed in [13] 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 [25], *independent* predicates by Charron-Bost *et al* [26], *conjunctive* predicates by Garg and Waldecker [27, 28], *linear* and *semi-linear* predicates by Chase and Garg [13], *regular* predicates by Garg and Mittal [29] and predicates expressed by a finite automata that can be checked online by Jard *et al* [30]. 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 [16]. This structure in used by A. Sen and Garg in their work on the temporal logic RCtl [14]. In [31, 32] 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 [33], 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 (shared variable, message passing, ...), other causal dependencies are added. 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 $\mathbf{T} = \langle E, \alpha, \preceq\rangle$ where:*

- *$E = P_1 \cup P_2 \cup ... \cup P_k$ is a finite set of events partitioned into k disjoint non empty subsets P_i, called processes; $\mathsf{pid}(e)$ denotes the process of event e belongs to ($\mathsf{pid}(e) = i$ iff $e \in P_i$);*
- *$\alpha : E \mapsto \mathbb{V} \times \mathbb{Q}$ is a labeling function mapping each event to an assignment, i.e. $\alpha(e) = (x, v)$ associates the assignment $x := v$ to e; if $\alpha(e) = (x := v)$, $\mathsf{var}(e)$ denotes x and $\mathsf{val}(e)$ denotes v;*
- *$\preceq \subseteq E \times E$ is a partial order relation on E such that $\forall e, e' \in E$:*
  (i) $\mathsf{pid}(e) = \mathsf{pid}(e') \Rightarrow (e \preceq e') \vee (e' \preceq e)$
  (ii) $\mathsf{var}(e) = \mathsf{var}(e') \Rightarrow (e \preceq e') \vee (e' \preceq e)$.

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 $\downarrow e = \{e' \in E \mid e' \preceq e\}$, the past of e (including itself), and $\mathsf{pos}(e) = |\downarrow e \cap P_{\mathsf{pid}(e)}|$ (where $|\cdot|$ denotes the size of sets), the position

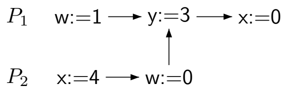

Fig. 1. Example of partial order trace

of e in its process. A *cut* is a subset $C \subseteq E$ such that $\forall e \in C : \downarrow e \subseteq C$. $\mathsf{cuts}(\mathbf{T}) = \{C \subseteq E \mid \forall e \in C : \downarrow e \subseteq C\}$ is the set of all cuts in $\mathbf{T}$. In the remainder of this paper, we always consider the set of variables $\mathbb{V}$ and the partial order trace of k processes $\mathbf{T} = \langle E, \alpha, \preceq \rangle$.

Semantics. Given a cut $C \in \mathsf{cuts}(\mathbf{T})$, we define $\mathsf{enabled}(C) = \{e \in E \setminus C \mid (\downarrow e \setminus \{e\}) \subseteq C\}$ 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 $C \in \mathsf{cuts}(\mathbf{T})$, so is $C \cup \{e\}$ for all $e \in \mathsf{enabled}(C)$. Given a set of cuts $X \subseteq \mathsf{cuts}(\mathbf{T})$, $\mathsf{pre}^{\exists}(X) = \{C \in \mathsf{cuts}(\mathbf{T}) \mid \exists e \in \mathsf{enabled}(C) : C \cup \{e\} \in X\}$ is the set of existential predecessors of X, i.e. the set of cuts having at least one successor in X, and $\mathsf{pre}^{\forall}(X) = \{C \in \mathsf{cuts}(\mathbf{T}) \mid \forall e \in \mathsf{enabled}(C) : C \cup \{e\} \in X\}$ 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 $\sigma = C_0, C_1, ..., C_n$, σ_i denotes C_i, the i^{th} element of σ, and $|\sigma| = n$ denotes the size of σ. A *run from a cut* C is a sequence $\sigma \in \mathsf{cuts}(\mathbf{T})^*$ such that (i) $\sigma_0 = C$, (ii) $\sigma_{|\sigma|} = E$, and (iii) $\forall 0 \leq i < |\sigma| : \sigma_i \in \mathsf{pre}^{\exists}(\{\sigma_{i+1}\})$, i.e. a sequence of cuts (i) starting in C, (ii) ending in E, and (iii) σ_{i+1} is a successor of σ_i for any i. The set of runs starting in $C \in \mathsf{cuts}(\mathbf{T})$ is denoted by $\mathsf{runs}(C)$. Finally, $\mathsf{runs}(\emptyset)$ is the set of runs of the trace $\mathbf{T}$.

Practical representation. A trace $\mathbf{T} = \langle E, \alpha, \preceq \rangle$ 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. $\neg \exists e'' \in E : e \prec e'' \prec e'$ where $e_1 \prec e_2$ 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-system. 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 $\mathsf{pid}(e) \neq \mathsf{pid}(e')$ 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 $\mathbf{T}$.

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, $x \in \mathbb{V}$ and where $\bullet \in \{<, \leq, >, \geq, =, \neq\}$. A formula in the Ctl logic is built on predicates using classical boolean operators, and temporal modalities. If p denotes a predicate and ϕ, ϕ_1, ϕ_2 denote Ctl formulae, then the set of Ctl formulae is defined as follows:

$$\phi ::= \top \mid p \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \mathsf{EX}\phi \mid \mathsf{AX}\phi \mid \mathsf{EG}\phi \mid \mathsf{AG}\phi \mid \mathsf{E}[\phi_1 \mathsf{U} \phi_2] \mid \mathsf{A}[\phi_1 \mathsf{U} \phi_2]$$

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: $\mathsf{EF}\phi \equiv \mathsf{E}[\top \mathsf{U} \phi]$ and $\mathsf{AF}\phi \equiv \mathsf{A}[\top \mathsf{U} \phi]$.

Semantics. Basic formulae are constraints over one variables in $\mathbb{V}$. Since all assignments to a particular variable are ordered, each cut $C \in \mathsf{cuts}(\mathbf{T})$ induces a unique valuation on the variables in $\mathbb{V}$ 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 $\forall x \in \mathbb{V},\ v_C(x) = 0$,
- if $C = C' \cup \{e\}$ with $C' \in \mathsf{cuts}(\mathbf{T})$ then $\forall x \in \mathbb{V} : v_C = \begin{cases} \mathsf{val}(e) & \text{if } \mathsf{var}(e) = x \\ v_{C'}(x) & \text{otherwise} \end{cases}$

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

$$\begin{array}{lll}
C \models \top & & \\
C \models p & \text{iff} & v_C(p) \text{ is true} \\
C \models \neg\phi & \text{iff} & C \not\models \phi \\
C \models \phi_1 \vee \phi_2 & \text{iff} & (C \models \phi_1) \vee (C \models \phi_2) \\
C \models \phi_1 \wedge \phi_2 & \text{iff} & (C \models \phi_1) \wedge (C \models \phi_2) \\
C \models \mathsf{EX}\phi & \text{iff} & \exists e \in \mathsf{enabled}(C) : C \cup \{e\} \models \phi \\
C \models \mathsf{AX}\phi & \text{iff} & \forall e \in \mathsf{enabled}(C) : C \cup \{e\} \models \phi \\
C \models \mathsf{EG}\phi & \text{iff} & \exists \sigma \in \mathsf{runs}(C), \forall i \in [0, |\sigma|] \ : \sigma_i \models \phi \\
C \models \mathsf{AG}\phi & \text{iff} & \forall \sigma \in \mathsf{runs}(C), \forall i \in [0, |\sigma|] \ : \sigma_i \models \phi \\
C \models \mathsf{E}[\phi_1 \mathsf{U} \phi_2] & \text{iff} & \exists \sigma \in \mathsf{runs}(C), \exists i \in [0, |\sigma|] \ : \\
 & & \qquad (\sigma_i \models \phi_2) \wedge (\forall j \in [0, i) \ : \sigma_j \models \phi_1) \\
C \models \mathsf{A}[\phi_1 \mathsf{U} \phi_2] & \text{iff} & \forall \sigma \in \mathsf{runs}(C), \exists i \in [0, |\sigma|] \ : \\
 & & \qquad (\sigma_i \models \phi_2) \wedge (\forall j \in [0, i) \ : \sigma_j \models \phi_1)
\end{array}$$

Note that according to this semantics, when the execution of $\mathbf{T}$ is finished (when the cut E is reached), for any Ctl formula ϕ, we have that $E \not\models \mathsf{EX}\phi$ and $E \models \mathsf{AX}\phi$.

$[\![\phi]\!]$ denotes the set $\{C \in \mathsf{cuts}(\mathbf{T}) \mid C \models \phi\}$ of cuts that satisfy formula ϕ. 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 $\mathsf{cuts}(\mathbf{T})$, is in general exponential in the size of $\mathbf{T}$. 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-uple $\overrightarrow{x}$ of naturals where the i^{th} component of $\overrightarrow{x}$ gives the number of events of the i^{th} process that already occured. For example, if a trace $\mathbf{T}$ is composed of 3 processes, the 3-uple $\langle 1, 2, 0\rangle$ represents the cut where process P_0 has executed its first event, i.e. $e \in P_1$ with $\mathsf{pos}(e) = 1$, process P_2 has executed its first 2 events, i.e. $e_1, e_2 \in P_2$ with $\mathsf{pos}(e_i) = i$ $(i \in \{1, 2\})$, and process P_3 has executed no events. The successor (predecessor) relation between cuts can be lifted to their vector representation: an event $e \in P_i$ is enabled in $\overrightarrow{x} = \langle x_1, \ldots, x_k\rangle$ if $x_{\mathsf{pid}(e)} < \mathsf{pos}(e) \wedge \forall e' \in\downarrow e \setminus \{e\} : \mathsf{pos}(e') \leq x_{\mathsf{pid}(e)}$ and the successor of $\overrightarrow{x}$ for e is $\langle x_1, \ldots, x_i + 1, \ldots, x_k\rangle$. Note that a vector $\overrightarrow{x}$ is not necessarily a representation for a cut. Indeed, if $\exists i \neq j \in [1, k], \exists e \in P_i, \exists e' \in\downarrow e \cap P_j : (\mathsf{pos}(e) \leq x_i) \wedge (\mathsf{pos}(e') > x_j)$ then $\overrightarrow{x}$ does not represent a cut, otherwise it does. Given a subset $X \subseteq \mathbb{N}^k$, we note $\mathsf{sets}(X) = \{C \subseteq E \mid \exists \overrightarrow{x} \in X, \forall 1 \leq i \leq k : |C \cap P_i| = x_i\}$ the set of subsets of events represented by the set X. Moreover, $\overrightarrow{x} \leq \overrightarrow{x}'$ 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-uples $\langle x_1, \ldots, x_k\rangle$ over naturals such that $\forall 1 \leq i \leq k : x_i$ is 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-uples. 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 $e_{i,1}, e_{i,2} ..., e_{i,m_i}$ of P_i $(i \in \{1, 2\})$ occurring sequentially without any restrictions on the events of P_{3-i} and such that $\forall j \in [1, m_i] : \mathsf{pos}(e_{i,j}) = j$. Then, the set of cuts where P_1 and P_2 have executed some of those 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 [18] is a compact data structure for representing sets of k-uples. An IST is basically a sharing tree [34], 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 to obtain 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 $\langle N, \iota, \mathsf{succ}\rangle$ where:*

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

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*), $\iota(n)$ is interpreted as an interval of integers. We note $x \in \iota(n)$ if an integer value x belongs to that interval. Figure 2 illustrates some IST. A path of an IST $\mathcal{I}$ is a sequence of node $root, n_1, n_2, ..., n_k, end$ such that $n_1 \in \mathsf{succ}(root), end \in \mathsf{succ}(n_k)$ and $\forall i \in [1, k) : n_{i+1} \in \mathsf{succ}(n_i)$. A k-uple $\overrightarrow{x} = \langle x_1, x_2, ..., x_k\rangle$ is accepted by an IST $\mathcal{I}$ if and only if there exists a path $root, n_1, n_2, ..., n_k, end$ in $\mathcal{I}$ such that $\forall i \in [1, k] : x_i \in \iota(n_i)$. The set of k-uples accepted by $\mathcal{I}$ is denoted by $\mathsf{tuple}(\mathcal{I})$ and if $\mathsf{tuple}(\mathcal{I}) \subseteq \mathbb{N}^k$, then $\mathsf{sets}(\mathcal{I}) = \mathsf{sets}(\mathsf{tuple}(\mathcal{I}))$. 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 $\mathcal{I}_1 \cup \mathcal{I}_2$, intersection, noted $\mathcal{I}_1 \cap \mathcal{I}_2$, set difference, noted $\mathcal{I}_1 \setminus \mathcal{I}_2$ and complementation, noted $\overline{\mathcal{I}}$. Other operations have been defined like downward closure, noted $\downarrow\mathcal{I}$, such that $\mathsf{tuple}(\downarrow\mathcal{I}) = \{\overrightarrow{x} \in \mathbb{N}^k \mid \exists \overrightarrow{x}' \in \mathsf{tuple}(\mathcal{I}) : \overrightarrow{x} \leq \overrightarrow{x}'\}$, 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 $\mathcal{I}^{[x_i \leftarrow x_i + \delta]}$. Formally, $\mathsf{tuple}(\mathcal{I}^{[x_i \leftarrow x_i + \delta]}) = \{\langle x_1, ..., x_i + \delta, ..., x_k\rangle \mid \overrightarrow{x} \in \mathsf{tuple}(\mathcal{I})\}$. Symbolic algorithm, 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 counter-part 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 [17] 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 [35].

5 Using IST for Ctl Model Checking

A basic approach to solve the Ctl model checking problem over partial order traces consists to flatten the trace 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 practicable since the resulting graph is in general exponential in the size of the trace. To overcome that problem, we propose to build $[\![\phi]\!]$ without flattening the partial order trace but working directly on it. Our method builds $[\![\phi]\!]$ inductively on the structure of ϕ. Since $[\![\phi]\!]$ 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 $\phi \equiv \top, \mathcal{I}_\top$ is an IST representing all possible cuts of the trace $\mathbf{T}$. The principle to build $\mathcal{I}_\top$ is to start from the very simple IST $\mathcal{I}_0$ where $\mathsf{sets}(\mathcal{I}_0)$ 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 $\mathcal{I}_0, \mathcal{I}_1, \mathcal{I}_2, \ldots$ where $\mathcal{I}_i$ is built from $\mathcal{I}_{i-1}$ $(i > 0)$ 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 $\mathsf{sets}(\mathcal{I}_{i-1})$ 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, $\mathsf{sets}(\mathcal{I}_0) \supseteq \mathsf{sets}(\mathcal{I}_1) \supseteq \ldots \supseteq \mathsf{sets}(\mathcal{I}_v) = [\![\top]\!]$. $\mathcal{I}_0$ is defined as follows:

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

To take into account a communication $e \rightarrow_c e'$, we need to remove from $\mathsf{sets}(\mathcal{I}_0)$ 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 $\mathcal{B}(e)$ representing all the sets of events that do not contain e (and have a vector representation). In other words, $\mathcal{B}(e)$ is the same as $\mathcal{I}_0$ except for the layer $\mathsf{pid}(e)$ where $\iota(n_{\mathsf{pid}(e)}) = [0, \mathsf{pos}(e) - 1]$. Then, we build an IST $\mathcal{A}(e')$ representing all the sets of events that contain e' (having a vector representation), i.e. $\mathcal{A}(e')$ is the same as $\mathcal{I}_0$ except for $\iota(n_{\mathsf{pid}(e')}) = [\mathsf{pos}(e'), |P_{\mathsf{pid}(e')}|]$. The events to remove from $\mathsf{sets}(\mathcal{I}_0)$ are in the intersection of $\mathsf{sets}(\mathcal{A}(e'))$ and $\mathsf{sets}(\mathcal{B}(e))$. Hence, to remove them we compute $\mathcal{I}_1 = \mathcal{I}_0 \setminus (\mathcal{A}(e') \cap \mathcal{B}(e))$. Then, we iterate the treatment from $\mathcal{I}_1$ to build $\mathcal{I}_2, \ldots, \mathcal{I}_v$. 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* $\mathbf{T} = \langle E, \alpha, \preceq \rangle$*, we have that* $\mathsf{sets}(\mathcal{I}_\top) = [\![\top]\!]$

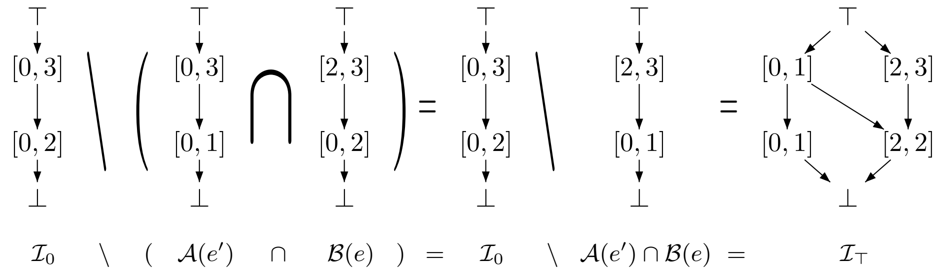
⊤
[0, 3]
[0, 2]
⊥
[0, 1]
[2, 3]
[2, 2]
=
$\mathcal{I}_0 \setminus (\mathcal{A}(e') \cap \mathcal{B}(e)) = \mathcal{I}_0 \setminus \mathcal{A}(e') \cap \mathcal{B}(e) = \mathcal{I}_\top$

Fig. 2. Computation of $\mathcal{I}_\top$

5.2 Predicates

If $\phi \equiv p$, where p is a predicate $x \bullet c$, we proceed as follows. First, we collect all events that can potentially modify the truth value of p. Let $E_p = \{e \in E \mid \mathsf{var}(e) = x\}$ 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 $\rho = e_1, e_2, ..., e_m$ be the linearization of E_p, i.e. $\forall i \in [1, m] : e_i \in E_p$, $|E_p| = m$ and $\forall i \in [1, m) : e_i \prec e_{i+1}$. This sequence can be used to determine *slices* of $\mathbf{T}$ where p is true. Indeed, let $s_1, s_2, ..., s_\ell$ be the sequence of indices splitting ρ into $\ell - 1$ contiguous blocks $\underline{e_{s_1}, ..., e_{s_2-1}}$, $\underline{e_{s_2}, ..., e_{s_3-1}}$, ...,$\underline{e_{s_\ell}, ..., e_m}$ 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) $\forall i \in [1, \ell], \forall j_1, j_2 \in [s_i, s_{i+1}) : (\downarrow e_{j_1} \models p) \iff (\downarrow e_{j_2} \models p)$
(iii) $\forall i \in [1, \ell) : (\downarrow e_{s_i} \models p) \iff (\downarrow e_{s_{i+1}} \not\models p)$

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 $\mathcal{A}(e_{s_i}) \cap \mathcal{B}(e_{s_{i+1}})$, as described above. Thus, for all block $i \in [1, \ell]$ such that $\downarrow e_{s_i} \models p$, we add $\mathcal{A}(e_{s_i}) \cap \mathcal{B}(e_{s_{i+1}})$ to $\mathcal{I}_p$ initially empty. Additionally, we must take into account the cuts at the beginning and at the end of $\mathbf{T}$. If p is satisfied at the beginning of $\mathbf{T}$ ($\emptyset \models p$), we must add $\mathcal{B}(e_{s_1})$ to $\mathcal{I}_p$, and similarly, if p is true at the end of $\mathbf{T}$ ($E \models p$), we add $\mathcal{A}(e_{s_m})$ to $\mathcal{I}_p$. Finally, in order to keep only cuts, we take the intersection with $\mathcal{I}_\top$.

Lemma 2. *Given a trace* $\mathbf{T} = \langle E, \alpha, \preceq \rangle$ *and a predicate* p, *we have that* $\mathsf{sets}(\mathcal{I}_p) = [\![p]\!]$.

5.3 Boolean Operators

In order to deal with boolean operators $\phi_1 \vee \phi_2$ (resp. $\phi_1 \wedge \phi_2$, $\neg\phi_1$), we can use standard operation on IST [17] and compute $\mathcal{I}_\phi = \mathcal{I}_{\phi_1} \cup \mathcal{I}_{\phi_2}$ (resp. $\mathcal{I}_\phi = \mathcal{I}_{\phi_1} \cap \mathcal{I}_{\phi_2}$, $\mathcal{I}_\phi = \overline{\mathcal{I}_{\phi_1}} \cap \mathcal{I}_\top$).

Lemma 3. *Given a trace* $\mathbf{T} = \langle E, \alpha, \preceq \rangle$ *and* CTL *formulae* ϕ, ϕ_1 *and* ϕ_2, *we have that* $\mathsf{sets}(\mathcal{I}_{\phi_1 \vee \phi_2}) = [\![\phi_1 \cup \phi_2]\!]$, $\mathsf{sets}(\mathcal{I}_{\phi_1 \wedge \phi_2}) = [\![\phi_1 \cap \phi_2]\!]$ *and* $\mathsf{sets}(\mathcal{I}_{\neg\phi}) = [\![\neg\phi]\!]$.

5.4 Existential Modalities

The treatment of existential modalities can be computed through the use of the $\mathsf{pre}^\exists$ operator, greatest and least fixed point (as explained e.g. in [8]):

$$\begin{aligned}
[\![\mathsf{EX}\phi]\!] &= \mathsf{pre}^\exists([\![\phi]\!]) \\
[\![\mathsf{EG}\phi]\!] &= \mathbf{gfp}\ \lambda X \cdot [\![\phi]\!] \cap \mathsf{pre}^\exists(X) \\
[\![\mathsf{E}[\phi_1 \mathsf{U} \phi_2]]\!] &= \mathbf{lfp}\ \lambda X \cdot [\![\phi_2]\!] \cup ([\![\phi_1]\!] \cap \mathsf{pre}^\exists(X))
\end{aligned}$$

In order to compute ISTs corresponding to those temporal formulae, we only need an algorithm for computing symbolically the $\mathsf{pre}^\exists(\cdot)$ operation. For that, we decompose $\mathsf{pre}^\exists(\cdot)$ into a function of $\mathsf{pre}_i^\exists(\cdot)$, where $\mathsf{pre}_i^\exists(X) = \{C \in \mathsf{cuts}(\mathbf{T}) \mid \exists e \in \mathsf{enabled}(C) \cap P_i : C \cup \{e\} \in X\}$ denotes the set of existential predecessors of X only for process P_i. This decomposition is provided by the following lemma.

Lemma 4. *Given a trace* $\mathbf{T} = \langle E, \alpha, \preceq \rangle$ *and a subset* $X \subseteq \mathsf{cuts}(\mathbf{T})$*, we have that* $\mathsf{pre}^\exists(X) = \bigcup_{i \in [1,k]} \mathsf{pre}_i^\exists(X)$*.*

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

Lemma 5. *Given a trace* $\mathbf{T} = \langle E, \alpha, \preceq \rangle$*, and an IST* $\mathcal{I}$ *such that* $\mathsf{sets}(\mathcal{I}) \subseteq \mathsf{cuts}(\mathbf{T})$*, we have that* $\mathsf{pre}_i^\exists(\mathsf{sets}(\mathcal{I})) = \mathsf{sets}(\mathcal{I}^{[x_i \leftarrow x_i - 1]} \cap \mathcal{I}_\top)$*.*

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

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

$$\mathsf{spre}^\exists(\mathcal{I}) = \bigcup_{i \in [1,k]} \left(\mathcal{I}^{[x_i \leftarrow x_i - 1]} \cap \mathcal{I}_\top \right)$$

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

Theorem 1 (Correctness $\mathsf{spre}^\exists(\cdot)$). *Given a trace* $\mathbf{T} = \langle E, \alpha, \preceq \rangle$*, and an IST* $\mathcal{I}$ *such that* $\mathsf{sets}(\mathcal{I}) \subseteq \mathsf{cuts}(\mathbf{T})$*, we have that* $\mathsf{pre}^\exists(\mathsf{sets}(\mathcal{I})) = \mathsf{sets}(\mathsf{spre}^\exists(\mathcal{I}))$*.*

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 [9, sec. 2.4]):

$$\begin{aligned}
[\![\mathsf{AX}\phi]\!] &= \mathsf{pre}^\forall([\![\phi]\!]) \\
[\![\mathsf{AG}\phi]\!] &= \mathbf{gfp}\ \lambda X \cdot [\![\phi]\!] \cap \mathsf{pre}^\forall(X) \\
[\![\mathsf{A}[\phi_1 \mathsf{U} \phi_2]]\!] &= \mathbf{lfp}\ \lambda X \cdot [\![\phi_2]\!] \cup ([\![\phi_1]\!] \cap \mathsf{pre}^\forall(X))
\end{aligned}$$

Computing ISTs corresponding to universal formulae amounts to defining a symbolical version of the $\mathsf{pre}^\forall(\cdot)$ operator on sets of cuts.

$\mathsf{pre}^{\forall}(\cdot)$ can be computed through the equivalence $\mathsf{pre}^{\forall}(\llbracket\phi\rrbracket) = \llbracket\mathsf{AX}\phi\rrbracket = \llbracket\neg\mathsf{EX}\neg\phi\rrbracket = \mathsf{cuts}(\mathbf{T})\backslash\mathsf{pre}^{\exists}(\llbracket\neg\phi\rrbracket)$. On the other hand, we may compute $\mathsf{pre}^{\forall}(\cdot)$ in an alternate way, similarly to what we did for the $\mathsf{pre}^{\exists}(\cdot)$. We can decompose $\mathsf{pre}^{\forall}(\cdot)$ as a function of $\mathsf{pre}^{\forall}_i(\cdot)$, where $\mathsf{pre}^{\forall}_i(X) = \{C \in \mathsf{cuts}(\mathbf{T}) \mid \forall e \in \mathsf{enabled}(C) \cap P_i : C \cup \{e\} \in X\}$ 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* $\mathbf{T} = \langle E, \alpha, \preceq\rangle$*, and an subset* $X \subseteq \mathsf{cuts}(\mathbf{T})$*, we have that* $\mathsf{pre}^{\forall}(X) = \bigcap_{i\in[1,k]} \mathsf{pre}^{\forall}_i(X)$*.*

To compute symbolically $\mathsf{pre}^{\forall}_i(\cdot)$, we need to characterize exactly which cuts are in $\mathsf{pre}^{\forall}_i(X)$. By definition, $\mathsf{pre}^{\forall}_i(X)$ denotes the set of cuts from which all enabled events of process P_i lead to a cut in X. $\mathsf{pre}^{\forall}_i(X)$ is composed of two classes of cuts: (i) $\mathsf{blocked}_i = \{C \in \mathsf{cuts}(\mathbf{T}) \mid \mathsf{enabled}(C) \cap P_i = \emptyset\}$, 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. $\mathsf{pre}^{\exists}_i(X)$.

Lemma 7. *Given a trace* $\mathbf{T} = \langle E, \alpha, \preceq\rangle$*, and an subset* $X \subseteq \mathsf{cuts}(\mathbf{T})$*, we have that* $\mathsf{pre}^{\forall}_i(X) = \mathsf{pre}^{\exists}_i(X) \cup \mathsf{blocked}_i$*.*

We already have a way to compute $\mathsf{pre}^{\exists}_i(X)$ symbolically (see lem.5). The following lemma characterized $\mathsf{blocked}_i$.

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

This result can be used to define an IST $\mathcal{I}_{\mathsf{blocked}_i}$ for $\mathsf{blocked}_i$. Indeed, from Lemma 8, we can see that $\mathsf{blocked}_i$ 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 $\mathcal{I}_{blocked_i}$ starts with an IST $\mathcal{I}_F$ representing the set of sets C of events where process P_i has finished its execution, i.e. where $|C \cap P_i| = |P_i|$. $\mathcal{I}_F$ is the same as $\mathcal{I}_0$ (c.f. sec. 5.1) except for layer i, where $\iota(n_i) = [|P_i|, |P_i|]$. 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 $P_{\mathsf{pid}(e)}$ has not executed e yet. This IST is the same as $\mathcal{I}_0$, except for layer i, where $\iota(n_i) = [\mathsf{pos}(e') - 1, \mathsf{pos}(e') - 1]$ and for layer $\mathsf{pid}(e)$, where $\iota(n_{\mathsf{pid}(e)}) = [0, \mathsf{pos}(e) - 1]$. The IST representing the sets of events where P_i is blocked is obtained by making the union between $\mathcal{I}_F$ 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 $\mathcal{I}_{\top}$. It is then easy to see, that $\mathcal{I}_{\mathsf{blocked}_i}$ contains exactly those cuts satisfying the condition of lem. 8. This leads us to the following symbolic characterization of $\mathsf{pre}^{\forall}_i(\cdot)$.

Lemma 9. *Given a trace* $\mathbf{T} = \langle E, \alpha, \preceq\rangle$*, and an IST* $\mathcal{I}$ *such that* $\mathsf{sets}(\mathcal{I}) \subseteq \mathsf{cuts}(\mathbf{T})$*, we have that* $\mathsf{pre}^{\forall}_i(\mathsf{sets}(\mathcal{I})) = \mathsf{sets}((\mathcal{I}^{[x_i \leftarrow x_i - 1]} \cap \mathcal{I}_{\top}) \cup \mathcal{I}_{\mathsf{blocked}_i})$*.*

We can now define the symbolic universal predecessors.

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

$$\mathsf{spre}^{\forall}(\mathcal{I}) = \bigcap_{i \in [1,k]} \Big((\mathcal{I}^{[x_i \leftarrow x_i - 1]} \cap \mathcal{I}_{\top}) \cup \mathcal{I}_{\mathsf{blocked}_i} \Big)$$

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

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

5.6 Improving the computation of $[\![\mathsf{EF}\phi]\!]$ and $[\![\mathsf{AG}\phi]\!]$

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

Lemma 10. *Given a trace* $\mathbf{T} = \langle E, \alpha, \preceq \rangle$ *of* k *processes and a* CTL *formula* ϕ*, we have that* $\mathsf{sets}(\downarrow\mathcal{I}_{\phi} \cap \mathcal{I}_{\top}) = [\![\mathsf{EF}\phi]\!]$.

Moreover, the quickest way to compute $[\![\mathsf{AG}\phi]\!]$ is generally through the translation $\mathsf{AG}\phi \equiv \neg\mathsf{EF}\neg\phi$ which avoids the fixpoint computation.

6 Experimental results

In this section, we experimentally validate our method. We compare our symbolic approach using ISTwith a state-of-the-art symbolic model checking (of the trace) using the tool NuSMV [5]. We considered several 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. Those results are presented in table 1. More detailed results are included for the reviewers 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: $\mathsf{AG}(\texttt{ncrit} < 2)$. On this property, we experimented two ways of computing AG. The first using the downward closure on IST, as explained in sec. 5.6, and the second using the fixed point on the $\mathsf{spre}^{\forall}(\cdot)$ operator, as explained in sec. 5.5. As expected the downward closure method is quicker (with the fixpoint 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

Model	#proc	#events	IST	NuSMV	Model	#proc	#events	IST	NuSMV
			(in sec.)	(in sec.)				(in sec.)	(in sec.)
Pet	2	2000	0.46	349.57	ABP	2	1000	13.60	297.28
	2	5000	7.53	↑↑		2	2000	27.56	↑↑
	2	15000	189.65	↑↑		2	5000	257.29	↑↑
PetN	2	2000	0.20	294.46	Phil	3	100	0.15	6.36
	2	5000	6.44	↑↑		3	200	1.11	↑↑
	2	20000	390.90	↑↑		3	2000	366.22	↑↑
	5	1000	2.04	13.74		5	100	0.25	↑↑
	5	1500	6.82	↑↑		5	200	27.05	↑↑
	5	5000	176.62	↑↑		5	500	125.56	↑↑
	10	1500	7.53	150.23		10	100	1.67	↑↑
	10	2000	27.01	↑↑		10	200	26.94	↑↑
	10	5000	147.89	↑↑		10	500	↑↑	↑↑

Table 1. Experimental results; ↑↑ indicates (> 10 min.).

detailed comparison is included for the reviewers 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 out tool can handle 15000 events in the allotted time. We also considered a generalization of this protocol for n processes (*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 *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: $\mathsf{AG}((\mathtt{sent\_msg} = 0) \implies \mathsf{AF}(\mathtt{received\_msg} = 0))$. 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 Philosopher* problem (*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 $\mathsf{AG}((\mathtt{state1} = \mathit{eat}) \implies (\mathsf{AG}(\mathtt{state1} = \mathit{eat}) \mid\mid \mathsf{A}[(\mathtt{state0} \neq \mathit{eat})\ \mathsf{U}\ (\mathtt{state1} \neq \mathit{eat})]))$. 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 interleaving.

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 Future works

As future works, our symbolic method using IST will be intergrated in our tool TraX [1] and will be fully interfaced with our distributed controllers design environment $_{\mathrm{d}}$SL [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 [14]. 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

1. De Wachter, B., Massart, T., Meuter, C.: dsl : An environment with automatic code distribution for industrial control systems. In: Lecture Notes in Computer Sciences. Volume 3144. Springer (2004) 132–145 (14 pages).
2. De Wachter, B., Genon, A., Massart, T., Meuter, C.: The formal design of distributed controllers with dsl and spin. Formal Aspects of Computing **17**(2) (2005) 177–200 (24 pages).
3. Holzmann, G.J.: The model checker spin. IEEE Trans. Software Eng. **23**(5) (1997) 279–295
4. McMillan, K.: The smv system. Technical Report CMU-CS-92-131, Carnegie Mellon University (1992)
5. Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: Nusmv 2: An opensource tool for symbolic model checking. In: CAV. (2002) 359–364
6. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem. Volume 1032 of Lecture Notes in Computer Science. Springer (1996)
7. Valmari, A.: On-the-fly verification with stubborn sets. In: CAV. (1993) 397–408
8. Clarke, E., Grumberg, O., Peled, D.: Model Checking. The MIT Press (1999)
9. McMillan, K.L.: Symbolic model checking: an approach to the state explosion problem. Carnegie Mellon University (1992)
10. Bryant, R.E.: Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Comput. Surv. **24**(3) (1992) 293–318
11. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7) (1978) 558–565
12. Mattern, F.: Virtual time and global states of distributed systems. In et al., C.M., ed.: Proc. Workshop on Parallel and Distributed Algorithms, North-Holland / Elsevier (1989) 215–226
13. Chase, C.M., Garg, V.K.: Detection of global predicates: Techniques and their limitations. Distributed Computing **11**(4) (1998) 191–201

[1] http://www.ulb.ac.be/di/ssd/cmeuter/trax/

14. Sen, A., Garg, V.K.: Detecting temporal logic predicates in distributed programs using computation slicing. In: OPODIS. (2003) 171–183
15. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Logic of Programs. (1981) 52–71
16. Mittal, N., Garg, V.K.: Computation slicing: Techniques and theory. In: DISC. (2001) 78–92
17. Ganty, P., Meuter, C., Begin, L.V., Kalyon, G., Raskin, J.F., Delzanno, G.: Symbolic data structure for sets of k-uples of integers. Technical report, Université Libre de Bruxelles (2006)
18. Ganty, P.: Algorithmes et structures de données efficaces pour la manipulation de contraintes sur les intervalles. Master's thesis, Université Libre de Bruxelles (2002)
19. Mazurkiewicz, A.W.: Trace theory. In: Advances in Petri Nets. (1986) 279–324
20. Thiagarajan, P.S.: A trace based extension of linear time temporal logic. In Abramsky, S., ed.: Proceedings of the Ninth Annual IEEE Symp. on Logic in Computer Science, LICS 1994, IEEE Computer Society Press (1994) 438–447
21. Alur, R., Peled, D., Penczek, W.: Model checking of causality properties. In: Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science (LICS'95), San Diego, California (1995) 90–100
22. Niebert, P., Peled, D.: Efficient model checking for ltl with partial order snapshots. In Hermanns, H., Palsberg, J., eds.: TACAS. Volume 3920 of Lecture Notes in Computer Science., Springer (2006) 272–286
23. Thiagarajan, P.S., Walukiewicz, I.: An expressively complete linear time temporal logic for mazurkiewicz traces. Inf. Comput. **179**(2) (2002) 230–249
24. Diekert, V., Gastin, P.: LTL is expressively complete for Mazurkiewicz traces. Journal of Computer and System Sciences **64**(2) (2002) 396–418
25. Chandy, K.M., Lamport, L.: Distributed snapshots: Determining global states of distributed systems. ACM Trans. Comput. Syst. **3**(1) (1985) 63–75
26. Charron-Bost, B., Delporte-Gallet, C., Fauconnier, H.: Local and temporal predicates in distributed systems. ACM Trans. Program. Lang. Syst. **17**(1) (1995)
27. Garg, V.K., Waldecker, B.: Detection of weak unstable predicates in distributed programs. IEEE Trans. Parallel Distrib. Syst. **5**(3) (1994) 299–307
28. Garg, V.K., Waldecker, B.: Detection of strong unstable predicates in distributed programs. IEEE Trans. Parallel Distrib. Syst. **7**(12) (1996) 1323–1333
29. Garg, V.K., Mittal, N.: On slicing a distributed computation. In: ICDCS. (2001) 322–329
30. Jard, C., Jéron, T., Jourdan, G.V., Rampon, J.X.: A general approach to trace-checking in distributed computing systems. In: ICDCS. (1994) 396–403
31. Sen, K., Rosu, G., Agha, G.: Online efficient predictive safety analysis of multithreaded programs. In: TACAS. (2004) 123–138
32. Sen, K., Rosu, G., Agha, G.: Detecting errors in multithreaded programs by generalized predictive analysis of executions. In: FMOODS. (2005) 211–226
33. Genon, A., Massart, T., Meuter, C.: Monitoring distributed controllers: When an efficient ltl algorithm on sequences is needed to model-check traces. In Misra, J., Nipkow, T., Sekerinski, E., eds.: FM. Volume 4085 of Lecture Notes in Computer Science., Springer (2006) 557–572
34. Zampunieris, D., Le Charlier, B.: Efficient handling of large sets of tuples with sharing trees. In: Proceedings of the 5th Data Compression Conference (DCC'95), IEEE Computer Society Press (1995) 428
35. Ammirati, P., Delzanno, G., Ganty, P., Geeraerts, G., Raskin, J.F., Van Begin, L.: Babylon: An integrated toolkit for the specification and verification of parameterized systems. In: 2nd workshop on Specification, Analysis and Validation for Emerging technologies (SAVE02). (2002)

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-uple $\overrightarrow{x}$, we note $C_{\overrightarrow{x}} = \{e \in E \mid \mathsf{pos}(e) \leq x_{\mathsf{pid}(e)}\}$ the subset of events represented by $\overrightarrow{x}$ represents. We will also need the following results.

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

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

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

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 $|C' \setminus C| = n$, we have that $\exists e \in (C' \setminus C) : e \in \mathsf{enabled}(C)$. Indeed, since $\preceq$ 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 $\nexists e' \in (C' \setminus C) : e' \prec e$. Since C' is a cut and $e \in C'$, we have that $\downarrow e \in C'$ and since e is minimal in $C' \setminus C$, that $\downarrow e \setminus \{e\} \in C$. It follows directly that $e \in \mathsf{enabled}(C)$. We know therefore that $C \cup \{e\} \in \mathsf{cuts}(\mathbf{T})$ and since $e \in C' \setminus C$ that $C \cup \{e\} \subseteq C'$. By induction there exists a sequence $C_0, ..., C_{n-1}$ of cuts between $C \cup \{e\}$ and C'. The sequence for C, C' is then given by $C, C_0, ..., C_{n-1}$.*

A.2 Proof of lemma 1

Given a trace $\mathbf{T} = \langle E, \alpha, \preceq\rangle$, we have that:

$$\mathsf{sets}(\mathcal{I}_\top) = [\![\top]\!]$$

Proof. First, note that $[\![\top]\!] = \{C \in \mathsf{cuts}(\mathbf{T}) \mid C \models \top\} = \mathsf{cuts}(\mathbf{T})$. Therefore, proving $\mathsf{sets}(\mathcal{I}_\top) = [\![\top]\!]$ amounts to proving that $\mathsf{sets}(\mathcal{I}_\top) = \mathsf{cuts}(\mathbf{T})$. Consequently, we prove the inclusion in both ways:

- For $\mathsf{cuts}(\mathbf{T}) \subseteq \mathsf{sets}(\mathcal{I}_\top)$, we prove that $\mathsf{cuts}(\mathbf{T}) \subseteq \mathsf{sets}(\mathcal{I})$ is an invariant of the algorithm. At the initial step, we start with $\mathcal{I} = \mathcal{I}_0$. It can be easily proven that $\mathsf{sets}(\mathcal{I}_0) = \{C \subseteq E \mid \forall e \in C : \downarrow e \cap P_{\mathsf{pid}(e)} \subseteq C\}$. Condition (i) of definition 1 then implies that every $\preceq$-closed subset of E belongs to $\mathsf{sets}(\mathcal{I}_0)$. Therefore, we have that $\mathsf{cuts}(\mathbf{T}) \subseteq \mathsf{sets}(\mathcal{I}_0)$. Then, at each step of the algorithm, we remove cuts from $\mathcal{I}$ to take into account a communication $e \rightarrow_c e'$. For that we compute two ISTs $\mathcal{B}(e)$ and $\mathcal{A}(e')$. It can be easily proven that $\mathsf{sets}(\mathcal{B}(e)) = \{C \in \mathsf{sets}(\mathcal{I}_0) \mid e \notin C\}$ and that $\mathsf{sets}(\mathcal{A}(e')) = \{C \in \mathsf{sets}(\mathcal{I}_0) \mid e' \in C\}$. Thus, $\mathsf{sets}(\mathcal{A}(e') \cap \mathcal{B}(e)) = \mathsf{sets}(\mathcal{A}(e')) \cap \mathsf{sets}(\mathcal{B}(e)) = \{C \in \mathsf{sets}(\mathcal{I}_0) \mid (e' \in C) \wedge (e \notin C)\}$. Therefore, since $e \preceq e'$, any $C \in \mathsf{sets}(\mathcal{A}(e') \cap \mathcal{B}(e))$ is not $\preceq$-closed. It follows directly that $\mathsf{sets}(\mathcal{A}(e') \cap \mathcal{B}(e)) \cap \mathsf{cuts}(\mathbf{T}) = \emptyset$, and that $\mathsf{cuts}(\mathbf{T}) \subseteq \mathsf{sets}(\mathcal{I} \setminus (\mathcal{A}(e') \cap \mathcal{B}(e)))$. We can therefore conclude that $\mathsf{cuts}(\mathbf{T}) \subseteq \mathsf{sets}(\mathcal{I})$ is an invariant and finally that $\mathsf{cuts}(\mathbf{T}) \subseteq \mathsf{sets}(\mathcal{I}_\top)$.

- For $\mathsf{sets}(\mathcal{I}_\top) \subseteq \mathsf{cuts}(\mathbf{T})$, we equivalently prove that $\forall C \subseteq E : (C \notin \mathsf{cuts}(\mathbf{T})) \Rightarrow (C \notin \mathsf{sets}(\mathcal{I}_\top))$. If $C \notin \mathsf{cuts}(\mathbf{T})$, then $\exists e \in C, e' \in \downarrow e : e' \notin C$. In this case, since $e' \in \downarrow e$, there exists in $\mathbf{T}$ a sequence of event $e' = e_1, e_2, ..., e_\ell = e$ and $\forall i \in [1, \ell) : e_i \rightarrow e_{i+1}$. Moreover, since $e' \notin C$, $\exists i \in [1, \ell) : (e_{i+1} \in C) \wedge (e_i \notin C)$. From there, we have two cases:
  (i) if $\mathsf{pid}(e_i) = \mathsf{pid}(e_{i+1})$, then C cannot be represented using a k-uple. Indeed, let $(x_1, x_2, ..., x_k)$ be such an hypothetical k-uple. If $e_{i+1} \in C$, then $\mathsf{pos}(e_{i+1}) \leq x_{\mathsf{pid}(e_{i+1})}$. But since $e_i \rightarrow e_{i+1}$, $\mathsf{pos}(e_i) \leq \mathsf{pos}(e_{i+1})$ and e_i is also in C. In this case, we therefore know that $C \notin \mathsf{sets}(\mathcal{I}_\top)$.
  (ii) on the other hand, if $\mathsf{pid}(e_i) \neq \mathsf{pid}(e_{i+1})$, then there is a communication $e_i \rightarrow_c e_{i+1}$. Thus, in some step of the algorithm, we will compute $\mathcal{B}(e_i)$ and $\mathcal{A}(e_{i+1})$ such that $\mathsf{sets}(\mathcal{A}(e_{i+1}) \cap \mathcal{B}(e_i)) = \{C \in \mathsf{sets}(\mathcal{I}_0) \mid (e_{i+1} \in C) \wedge (e_i \notin C)\}$. Therefore, C will be removed at this step.

  In both case, $C \notin \mathsf{sets}(\mathcal{I}_\top)$ and we can conclude that $\mathsf{sets}(\mathcal{I}_\top) \subseteq \mathsf{cuts}(\mathbf{T})$

A.3 Proof of lemma 2

Given a trace $\mathbf{T} = \langle E, \alpha, \preceq \rangle$ and a predicate p, we have that

$$\mathsf{sets}(\mathcal{I}_p) = [\![p]\!]$$

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

- For $\mathsf{sets}(\mathcal{I}_p) \subseteq [\![p]\!]$, we prove equivalently that $\forall C \in \mathsf{sets}(\mathcal{I}_p) : C \models p$. First note that $\forall C \in \mathsf{sets}(\mathcal{I}_p) : C \in \mathsf{sets}(\mathcal{I}_\top) = \mathsf{cuts}(\mathbf{T})$. Therefore, C is $\preceq$-closed. Then, we have three possibilities:
  (i) $C \in \mathsf{sets}(\mathcal{B}(e_{s_1}))$: in this case, since $e_{s_1} = e_1 \notin C$ and since C is $\preceq$-closed, we have that $v_C(p) = v_\emptyset(p)$. However, since C was added to $\mathcal{I}_p$, by construction we have that $\emptyset \models p$ which in turn implies that $C \models p$.
  (ii) $\exists i \in [1, \ell) : C \in \mathsf{sets}(\mathcal{A}(e_{s_i}) \cap \mathcal{B}(e_{s_{i+1}}))$: in this case, since $e_{s_i} \in C$, $e_{s_{i+1}} \notin C$ and since C is $\preceq$-closed, we have that $\{e_j \mid j \in [s_i, s_{i+1})\} \subseteq C$. However, by construction, $\forall j \in [s_i, s_{i+1}) : \downarrow e_j \models p$. Thus, if C_i denotes $\downarrow e_{s_i}$, it follows that $v_C(p) = v_{C_i}(p)$, which in turn implies $C \models p$.
  (iii) $C \in \mathsf{sets}(\mathcal{A}(e_{s_\ell}))$: in this case since $e_{s_\ell} = e_m \in C$ and since C $\preceq$-closed, $v_C(p) = v_E(p)$. 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 $[\![p]\!] \subseteq \mathsf{sets}(\mathcal{I}_p)$, we prove equivalently that $\forall C \in \mathsf{cuts}(\mathbf{T}) : (C \models p) \Rightarrow (C \in \mathsf{sets}(\mathcal{I}_p))$. Again, we have three possibilities:

(i) $C \cap E_p = \emptyset$: in this case, we have that $v_C(p) = v_\emptyset(p)$ and $C \models p$, that $\emptyset \models p$. Therefore, since $C \in \mathcal{B}(e_{s_1})$ and since $C \in \mathsf{cuts}(\mathbf{T}) = \mathsf{sets}(\mathcal{I}_\top)$, we can conclude that $C \in \mathsf{sets}(\mathcal{I}_p)$.

(ii) $\emptyset \neq C \cap E_p \neq E_p$: in this case, let $i = \mathbf{max}_\leq(\{j \in [1, \ell) \mid e_{s_j} \in C\})$. Since $e_{s_j} \in C$, we have that $C \in \mathsf{sets}(\mathcal{A}(e_{s_j}))$. Moreover, we know that $e_{s_{i+1}} \notin C$, otherwise, i would not be maximal. It follows that $C \in \mathsf{sets}(\mathcal{B}(e_{s_{i+1}}))$ and that $C \in \mathsf{sets}(\mathcal{A}(e_{s_i}) \cap \mathcal{B}(e_{s_{i+1}}))$. However, if C_i denotes $\downarrow e_{s_i}$, since $C \models p$ and $v_C(p) = v_{C_i}(p)$, we have that $C_i \models p$. Therefore, $\mathcal{A}(e_{s_i}) \cap \mathcal{B}(e_{s_{i+1}})$ will be added to $\mathcal{I}_p$ in the construction. Finally, since $C \in \mathsf{cuts}(\mathbf{T}) = \mathsf{sets}(\mathcal{I}_\top)$, we can conclude that $C \in \mathsf{sets}(\mathcal{I}_p)$.

(iii) $C \cap E_p = E_p$: in this case, we have that $v_C(p) = v_E(p)$ and since $C \models p$, that $E \models p$. Therefore, since $C \in \mathcal{A}(e_{s_\ell})$ and since $C \in \mathsf{cuts}(\mathbf{T}) = \mathsf{sets}(\mathcal{I}_\top)$, we can conclude that $C \in \mathsf{sets}(\mathcal{I}_p)$.

A.4 Proof of lemma 3

Given a trace $\mathbf{T} = \langle E, \alpha, \preceq \rangle$ and CTL formulae ϕ, ϕ_1 and ϕ_2, we have that:

$$\begin{aligned}
\mathsf{sets}(\mathcal{I}_{\phi_1 \vee \phi_2}) &= [\![\phi_1 \cup \phi_2]\!] \\
\mathsf{sets}(\mathcal{I}_{\phi_1 \wedge \phi_2}) &= [\![\phi_1 \cap \phi_2]\!] \\
\mathsf{sets}(\mathcal{I}_{\neg\phi}) &= [\![\neg\phi]\!]
\end{aligned}$$

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

$$\begin{aligned}
\mathsf{sets}(\mathcal{I}_{\neg\phi}) &= \mathsf{sets}(\overline{\mathcal{I}_\phi} \cap \mathcal{I}_\top) \\
&= \mathsf{sets}(\overline{\mathcal{I}_\phi}) \cap \mathsf{sets}(\mathcal{I}_\top) \\
&= (\mathsf{sets}(\mathbb{N}^k) \setminus \mathsf{sets}(\mathcal{I}_\phi)) \cap \mathsf{sets}(\mathcal{I}_\top) \\
&= (\mathsf{sets}(\mathbb{N}^k) \cap \mathsf{sets}(\mathcal{I}_\top)) \setminus (\mathsf{sets}(\mathcal{I}_\phi) \cap \mathsf{sets}(\mathcal{I}_\top)) \\
&= \mathsf{sets}(\mathcal{I}_\top) \setminus \mathsf{sets}(\mathcal{I}_\phi) \\
&= [\![\top]\!] \setminus [\![\phi]\!] \\
&= [\![\neg\phi]\!]
\end{aligned}$$

$$\begin{aligned}
\mathsf{sets}(\mathcal{I}_{\phi_1 \vee \phi_2}) &= \mathsf{sets}(\mathcal{I}_{\phi_1} \cup \mathcal{I}_{\phi_2}) \\
&= \mathsf{sets}(\mathcal{I}_{\phi_1}) \cup \mathsf{sets}(\mathcal{I}_{\phi_2}) \\
&= [\![\phi_1]\!] \cup [\![\phi_2]\!] \\
&= [\![\phi_1 \vee \phi_2]\!]
\end{aligned}$$

$$\begin{aligned}
\mathsf{sets}(\mathcal{I}_{\phi_1 \wedge \phi_2}) &= \mathsf{sets}(\mathcal{I}_{\phi_1} \cap \mathcal{I}_{\phi_2}) \\
&= \mathsf{sets}(\mathcal{I}_{\phi_1}) \cap \mathsf{sets}(\mathcal{I}_{\phi_2}) \\
&= [\![\phi_1]\!] \cap [\![\phi_2]\!] \\
&= [\![\phi_1 \wedge \phi_2]\!]
\end{aligned}$$

A.5 Proof of lemma 4

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

$$\mathsf{pre}^\exists(X) = \bigcup_{i \in [1,k]} \mathsf{pre}^\exists_i(X)$$

Proof. This is direct consequence of the following equalities:

$$\begin{aligned}
\mathsf{pre}^{\exists}(X) &= \{C \in \mathsf{cuts}(\mathbf{T}) \mid \exists e \in \mathsf{enabled}(C) : C \cup \{e\} \in X\} \\
&= \{C \in \mathsf{cuts}(\mathbf{T}) \mid \exists e \in \mathsf{enabled}(C) \cap E : C \cup \{e\} \in X\} \\
&= \{C \in \mathsf{cuts}(\mathbf{T}) \mid \exists e \in \mathsf{enabled}(C) \cap (\textstyle\bigcup_{i\in[1,k]} P_i) : C \cup \{e\} \in X\} \\
&= \{C \in \mathsf{cuts}(\mathbf{T}) \mid \exists e \in \textstyle\bigcup_{i\in[1,k]}(\mathsf{enabled}(C) \cap (P_i) : C \cup \{e\} \in X\} \\
&= \textstyle\bigcup_{i\in[1,k]}\{C \in \mathsf{cuts}(\mathbf{T}) \mid \exists e \in \mathsf{enabled}(C) \cap P_i : C \cup \{e\} \in X\} \\
&= \textstyle\bigcup_{i\in[1,k]} \mathsf{pre}_i^{\exists}(X)
\end{aligned}$$

A.6 Proof of lemma 5

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

$$\mathsf{pre}_i^{\exists}(\mathsf{sets}(\mathcal{I})) = \mathsf{sets}(\mathcal{I}^{[x_i \leftarrow x_i - 1]} \cap \mathcal{I}_{\top})$$

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

$$\begin{aligned}
&\mathsf{sets}(\mathcal{I}^{[x_i \leftarrow x_i - 1]} \cap \mathcal{I}_{\top}) \\
&= \{C \in \mathsf{cuts}(\mathbf{T}) \mid \exists \overrightarrow{x} \in \mathsf{tuple}(\mathcal{I}^{[x_i \leftarrow x_i - 1]}) : C = C_{\overrightarrow{x}}\} \\
&= \{C \in \mathsf{cuts}(\mathbf{T}) \mid \exists \overrightarrow{x}' \in \mathsf{tuple}(\mathcal{I}) : C = C_{\overrightarrow{x}'} \setminus \{e \in P_i \mid \mathsf{pos}(e) = x'_i\}\} \\
&= \{C \in \mathsf{cuts}(\mathbf{T}) \mid \exists \overrightarrow{x}' \in \mathsf{tuple}(\mathcal{I}) : C = C_{\overrightarrow{x}'} \setminus \{e \in P_i \mid \mathsf{pos}(e) = |C_{\overrightarrow{x}'} \cap P_i|\}\} \\
&= \{C \in \mathsf{cuts}(\mathbf{T}) \mid \exists \overrightarrow{x}' \in \mathsf{tuple}(\mathcal{I}) : C = C_{\overrightarrow{x}'} \setminus \{e \in P_i \mid \mathsf{pos}(e) = |C \cap P_i| + 1\}\} \\
&= \{C \in \mathsf{cuts}(\mathbf{T}) \mid \exists \overrightarrow{x}' \in \mathsf{tuple}(\mathcal{I}) : C_{\overrightarrow{x}'} = C \cup \{e \in P_i \mid \mathsf{pos}(e) = |C \cap P_i| + 1\}\} \\
&= \{C \in \mathsf{cuts}(\mathbf{T}) \mid \exists C' \in \mathsf{sets}(\mathcal{I}) : C' = C \cup \{e \in P_i \mid \mathsf{pos}(e) = |C \cap P_i| + 1\}\} \\
&\hfill \text{(by lem. 11)} \\
&= \{C \in \mathsf{cuts}(\mathbf{T}) \mid \exists e \in \mathsf{enabled}(C) \cap P_i, \exists C' \in \mathsf{sets}(\mathcal{I}) : C' = C \cup \{e\}\} \\
&= \{C \in \mathsf{cuts}(\mathbf{T}) \mid \exists e \in \mathsf{enabled}(C) \cap P_i : C \cup \{e\} \in \mathsf{sets}(\mathcal{I})\} \\
&= \mathsf{pre}_i^{\exists}(\mathsf{sets}(\mathcal{I}))
\end{aligned}$$

A.7 Proof of lemma 6

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

$$\mathsf{pre}^{\forall}(X) = \bigcap_{i\in[1,k]} \mathsf{pre}_i^{\forall}(X)$$

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

$$
\begin{aligned}
\mathsf{pre}^{\forall}(X) &= \{C \in \mathsf{cuts}(\mathbf{T}) \mid \forall e \in \mathsf{enabled}(C) : C \cup \{e\} \in X\} \\
&= \{C \in \mathsf{cuts}(\mathbf{T}) \mid \neg(\exists e \in \mathsf{enabled}(C) : C \cup \{e\} \notin X)\} \\
&= \{C \in \mathsf{cuts}(\mathbf{T}) \mid \neg(\exists e \in \mathsf{enabled}(C) : C \cup \{e\} \in (\mathsf{cuts}(\mathbf{T}) \setminus X))\} \\
&= \{C \in \mathsf{cuts}(\mathbf{T}) \mid \neg(C \in \mathsf{pre}^{\exists}(\mathsf{cuts}(\mathbf{T}) \setminus X))\} \\
&= \mathsf{cuts}(\mathbf{T}) \setminus \mathsf{pre}^{\exists}(\mathsf{cuts}(\mathbf{T}) \setminus X) \\
&= \mathsf{cuts}(\mathbf{T}) \setminus \Big(\bigcup\nolimits_{i\in[1,k]} \mathsf{pre}_i^{\exists}(\mathsf{cuts}(\mathbf{T}) \setminus X)\Big) \\
&= \bigcap\nolimits_{i\in[1,k]} \big(\mathsf{cuts}(\mathbf{T}) \setminus \mathsf{pre}_i^{\exists}(\mathsf{cuts}(\mathbf{T}) \setminus X)\big) \\
&= \bigcap\nolimits_{i\in[1,k]} (\mathsf{cuts}(\mathbf{T}) \setminus \{C \in \mathsf{cuts}(\mathbf{T}) \mid \exists e \in \mathsf{enabled}(C) \cap P_i : C \cup \{e\} \in \mathsf{cuts}(\mathbf{T}) \setminus X\}) \\
&= \bigcap\nolimits_{i\in[1,k]} (\mathsf{cuts}(\mathbf{T}) \setminus \{C \in \mathsf{cuts}(\mathbf{T}) \mid \exists e \in \mathsf{enabled}(C) \cap P_i : C \cup \{e\} \notin X\}) \\
&= \bigcap\nolimits_{i\in[1,k]} (\mathsf{cuts}(\mathbf{T}) \setminus \{C \in \mathsf{cuts}(\mathbf{T}) \mid \neg(\forall e \in \mathsf{enabled}(C) \cap P_i : C \cup \{e\} \in X\})) \\
&= \bigcap\nolimits_{i\in[1,k]} \big(\mathsf{cuts}(\mathbf{T}) \setminus \{C \in \mathsf{cuts}(\mathbf{T}) \mid \neg(C \in \mathsf{pre}_i^{\forall}(X)\}\big) \\
&= \bigcap\nolimits_{i\in[1,k]} \big(\mathsf{cuts}(\mathbf{T}) \setminus (\mathsf{cuts}(\mathbf{T}) \setminus \mathsf{pre}_i^{\forall}(X))\big) \\
&= \bigcap\nolimits_{i\in[1,k]} \mathsf{pre}_i^{\forall}(X)
\end{aligned}
$$

A.8 Proof of lemma 7

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

$$\mathsf{pre}_i^{\forall}(X) = \mathsf{pre}_i^{\exists}(X) \cup \mathsf{blocked}_i$$

Proof. We proceed by proving inclusion in both ways.

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

A.9 Proof of lemma 8

Given a trace $\mathbf{T} = \langle E, \alpha, \preceq \rangle$ and a process $P_i \subseteq E$, we have that $C \in \mathsf{blocked}_i$ if and only if:

$$\forall e \in E \cap P_i : (\mathsf{pos}(e) = |C \cap P_i| + 1) \implies (\exists e' \in E \setminus C : e' \rightarrow_c e)$$

Proof. We prove the equivalence in both ways.

- ($\Leftarrow$) Let us examine a cut C such that $\forall e \in E \cap P_i : (\mathsf{pos}(e) = |C \cap P_i| + 1) \implies (\exists e' \in E \setminus C : e' \rightarrow_c e)$. By lem. 11, we know that $\mathsf{enabled}(C) \cap P_i \subseteq \{e\}$ where e is such that $\mathsf{pos}(e) = |C \cap P_i| + 1$. We have then that $\exists e' \in E \setminus C : e' \rightarrow_c e$. Since $e' \notin C$ and since $e' \rightarrow_c e$ implies that $e' \preceq e$, we have that $e \notin \mathsf{enabled}(C)$ hence $\mathsf{enabled}(C) \cap P_i = \emptyset$ and $C \in \mathsf{blocked}_i$.
- ($\Rightarrow$) We proceed by proving the contraposition. Let us examine a cut C such that $\exists e \in E \cap P_i : (\mathsf{pos}(e) = |C \cap P_i| + 1) \wedge (\forall e' \in E \setminus C : e' \not\rightarrow_c e)$. Since $\mathsf{pos}(e) > |C \cap P_i|$, we can conclude that $e \notin C$. Moreover, since $C \in \mathsf{cuts}(\mathbf{T})$, we have that $\forall e' \in P_i : (\mathsf{pos}(e') \leq |C \cap P_i|) \implies (e' \in C)$. Finally, $\forall e' \in E \setminus C : e' \not\rightarrow_c e$ implies that $\nexists e' \in E \setminus P_i : e' \preceq e$. We can deduce that $e \in \mathsf{enabled}(C)$ and that $\mathsf{enabled}(C) \cap P_i \neq \emptyset$, which implies that $C \notin \mathsf{blocked}_i$.

A.10 Proof of lemma 9

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

$$\mathsf{pre}_i^{\forall}(\mathsf{sets}(\mathcal{I})) = \mathsf{sets}((\mathcal{I}^{[x_i \leftarrow x_i - 1]} \cap \mathcal{I}_{\top}) \cup \mathcal{I}_{\mathsf{blocked}_i})$$

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

$$\begin{aligned}
\mathsf{pre}^{\forall}(\mathsf{sets}(\mathcal{I})) &= \mathsf{pre}_i^{\exists}(\mathsf{sets}(\mathcal{I})) \cup \mathsf{blocked}_i && \text{(by lem. 7)}\\
&= \mathsf{sets}(\mathcal{I}^{[x_i \leftarrow x_i - 1]} \cap \mathcal{I}_{\top}) \cup \mathsf{blocked}_i && \text{(by lem. 5)}\\
&= \mathsf{sets}(\mathcal{I}^{[x_i \leftarrow x_i - 1]} \cap \mathcal{I}_{\top}) \cup \mathsf{sets}(\mathcal{I}_{\mathsf{blocked}_i}) && \text{(by constr. of } \mathcal{I}_{\mathsf{blocked}_i}\text{)}\\
&= \mathsf{sets}((\mathcal{I}^{[x_i \leftarrow x_i - 1]} \cap \mathcal{I}_{\top}) \cup \mathcal{I}_{\mathsf{blocked}_i})
\end{aligned}$$

A.11 Proof of lemma 10

Given a trace $\mathbf{T} = \langle E, \alpha, \preceq \rangle$ of k processes and a CTL formula ϕ, we have that:

$$\mathsf{sets}(\mathcal{I}_{\mathsf{EF}\phi}) = [\![\mathsf{EF}\phi]\!]$$

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

$$\begin{aligned}
&\mathsf{sets}(\downarrow\mathcal{I}_{\phi} \cap \mathcal{I}_{\top})\\
&= \mathsf{sets}(\downarrow\mathcal{I}_{\phi}) \cap \mathsf{sets}(\mathcal{I}_{\top})\\
&= \mathsf{sets}(\{\overrightarrow{x} \in \mathbb{N}^k \mid \exists \overrightarrow{x}' \in \mathsf{tuple}(\mathcal{I}_{\phi}) : \overrightarrow{x} \leq \overrightarrow{x}'\}) \cap \mathsf{sets}(\mathcal{I}_{\top})\\
&= \{C \subseteq E \mid \exists C' \in \mathsf{sets}(\mathcal{I}_{\phi}) : C \subseteq C'\} \cap \mathsf{cuts}(\mathbf{T})\\
&= \{C \in \mathsf{cuts}(\mathbf{T}) \mid \exists C' \in \mathsf{sets}(\mathcal{I}_{\phi}) : C \subseteq C'\}\\
&= \{C \in \mathsf{cuts}(\mathbf{T}) \mid \exists C' \in [\![\phi]\!] : C \subseteq C'\}\\
&\hspace{20em} \text{(by lem. 12)}\\
&= \left\{ C \in \mathsf{cuts}(\mathbf{T}) \;\middle|\; \begin{array}{c} \exists C' \in [\![\phi]\!], \exists C_0, ..., C_m : (C_0 = C) \wedge (C_m = C') \wedge \\ (\forall i \in [1, m] : (C_{i-1} \in \mathsf{pre}^{\exists}(\{C_i\})) \wedge (C_i \in \mathsf{cuts}(\mathbf{T}))) \end{array} \right\}\\
&= [\![\mathsf{EF}\phi]\!]
\end{aligned}$$

B Details of the experimental results

#events	#comm.	#cuts	TraX (f.p.)	TraX (d.c.)	NuSMV
100	7	290	0.01 sec.	0.01 sec.	0.06 sec.
200	15	600	0.02 sec.	0.01 sec.	0.17 sec.
500	36	1380	0.30 sec.	0.02 sec.	0.88 sec.
1000	74	2860	0.70 sec.	0.07 sec.	3.43 sec.
2000	147	5570	1.45 sec.	0.46 sec.	349.57 sec.
5000	366	14030	15.2 sec.	7.53 sec.	↑↑
10000	729	27995	96.51 sec.	58.12 sec.	↑↑
15000	1093	41980	323.59 sec.	189.65 sec.	↑↑
20000	1451	55848	↑↑	528.74 sec.	↑↑

Table 2. Peterson mutual exclusion protocol for two process; the property is $\mathsf{AG}(\texttt{ncrit} < 2)$ where $\texttt{ncrit}$ is the number of process in their critical section; ↑↑ indicates that the execution did not terminate in under 10 min; f.p. (resp. d.c.) indicates that AG was computed using a fixed point (downward closure) .

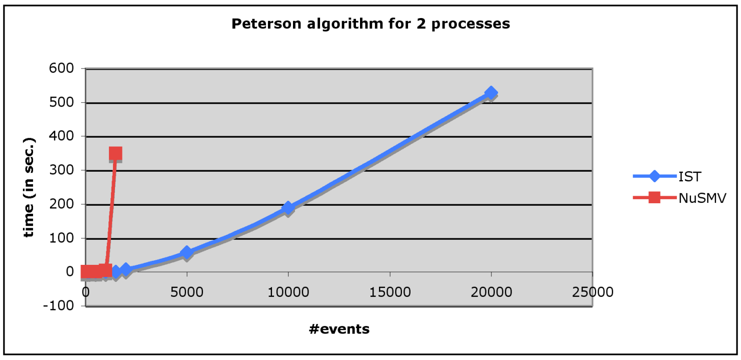

Fig. 3. Graphical results for the simple Peterson with 2 processes

#events	#comm.	#cuts	TraX	NuSMV
100	22	1653	0.32 sec.	1.44 sec.
200	32	2319	0.45 sec.	1.67 sec.
500	75	5278	1.82 sec.	14.79 sec.
1000	149	11314	13.60 sec.	297.28 sec.
2000	276	21075	27.56 sec.	↑↑
5000	682	51531	257.29 sec.	↑↑
10000	1360	102714	↑↑	↑↑

Table 3. Alternating Bit protocol with one sender and one receiver; the property is $\mathsf{AG}((\texttt{sent\_msg} = 0) \implies \mathsf{AF}(\texttt{received\_msg} = 0))$; ↑↑ indicates that the execution did not terminate in under 10 min.

#processes	#events	#comm	#cuts	TraX	NuSMV
2	100	7	107	0.00 sec.	0.03 sec.
2	200	14	206	0.00 sec.	0.09 sec.
2	500	34	506	0.01 sec.	0.42 sec.
2	1000	68	1016	0.04 sec.	2.80 sec.
2	2000	134	2039	0.20 sec.	294.46 sec.
2	5000	334	5106	6.44 sec.	↑↑
2	10000	667	10047	48.10 sec.	↑↑
2	20000	1335	20267	390.90 sec.	↑↑
5	100	56	404	0.03 sec.	0.09 sec.
5	200	97	1009	0.07 sec.	0.16 sec.
5	500	202	2618	0.33 sec.	0.85 sec.
5	1000	402	5393	2.04 sec.	13.74 sec.
5	1500	602	11732	6.82 sec.	↑↑
5	2000	773	13885	12.61 sec.	↑↑
5	5000	1926	27835	176.62 sec.	↑↑
5	10000	3801	55535	↑↑	↑↑
10	100	328	14072	0.82 sec.	0.01 sec.
10	200	314	22173	0.79 sec.	0.46 sec.
10	500	493	43908	2.12 sec.	1.60 sec.
10	1000	796	72567	5.42 sec.	4.20 sec.
10	1500	1024	92340	7.53 sec.	150.23 sec.
10	2000	1405	147219	27.01 sec.	↑↑
10	5000	3255	203002	147.89 sec.	↑↑
10	10000	6361	452897	↑↑	↑↑

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

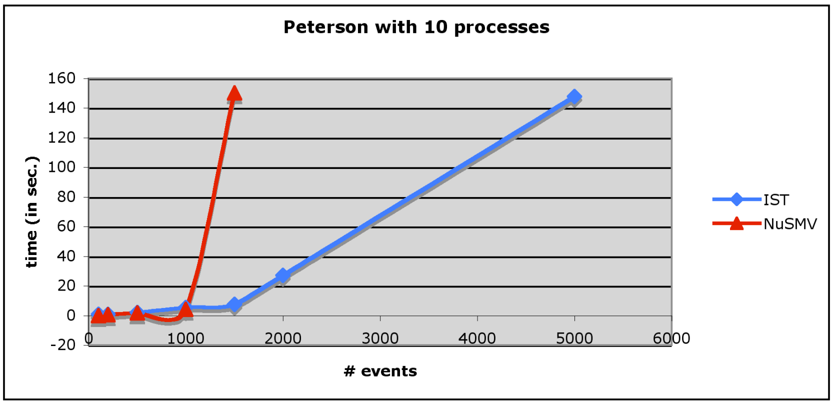

Fig. 4. Graphical results for Peterson with 10 processes

#processes	#events	#comm	#cuts	TraX	NuSMV
3	100	31	2060	0.15	6.36
3	200	61	5879	1.11	↑↑
3	500	160	11587	6.24	↑↑
3	1000	324	20780	28.90	↑↑
3	2000	613	55680	366.22	↑↑
3	5000	1654	104591	↑↑	↑↑
5	100	65	41334	0.25	↑↑
5	200	101	405858	27.05	↑↑
5	500	229	1021052	125.56	↑↑
5	1000	547	1342108	↑↑	↑↑
10	100	130	377853293	1.67	↑↑
10	200	189	797010724	26.94	↑↑
10	500	474	1478286661	↑↑	↑↑

Table 5. Dining philosophers using shared variables for the forks; the property is $\mathsf{AG}((\texttt{state1} = \mathit{eat}) \implies (\mathsf{AG}(\texttt{state1} = \mathit{eat}) \mid\mid \mathsf{A}[(\texttt{state0} \neq \mathit{eat})\ \mathsf{U}\ (\texttt{state1} \neq \mathit{eat})]))$ where $\texttt{state}i$ is the state of philosopher i (*eat*, *hungry* or *idle*); ↑↑ indicates that the execution did not terminate in under 10 min.