Efficient Model Checking for LTL with Partial Order Snapshots

Peter Niebert¹ and Doron Peled²

¹ Laboratoire d'Informatique Fondamentale de Marseille,
CMI, 39, rue Joliot Curie, 13453 Marseille Cedex 13, France

² Department of Computer Science, University of Warwick,
Coventry CV4 7AL, United Kingdom

Abstract. Certain behavioral properties of distributed systems are difficult to express in interleaving semantics, whereas they are naturally expressed in terms of partial orders of events or, equivalently, Mazurkiewicz traces. Examples of such properties are serializability of a database or *snapshots*. Recently, a modest extension for LTL by an operator that expresses snapshots has been proposed. It combines the ease of linear (interleaving) specification with this useful partial order concept. The new construct allows one to assert that a global snapshot (also called a *slice* or a *cut*) was passed, perhaps not in the observed (interleaved) execution sequence, but possibly in a (trace) equivalent one. A model checking algorithm was suggested for a subset of this logic, with PSPACE complexity in the size of the system and the checked formula. For the whole logic, a solution that is in EXSPACE in the size of the system (PSPACE in the number of its global states) was given.

In this paper, we provide a model checking algorithm in PSPACE in the size of a system of communicating sequential processes when restricting snapshots to boolean combinations of local properties of each process. Concerning size of the formula, it is PSPACE for the case of snapshot properties expressed in DNF, and EXPSPACE where a translation to DNF is necessary.

1 Introduction

Automatic verification of concurrent systems is highly challenging. The exponential growth of the state space with the number of processes makes the problem intractable, as well as all the more important. Except for the intractability issue, one needs also to consider expressiveness, i.e., the ability to formulate the desired properties of the verified system. These two issues are not independent, as increasing the expressiveness may also increase complexity. The goal is therefore to find a formalism that can express common and useful specifications in a compact way, while still having a relatively efficient decision procedure.

Traditionally, concurrent systems are modeled using interleaving semantics, where occurrence of actions from different processes appear in either order in different execution sequences. Intuitively, sequences that differ only in the relative order of such occurrences can be considered as two representations for the same behavior. However, the interleaving model does not make any use of this

observation. The interleaving model is rather simple, and as such enjoys the benefit of using powerful mathematical tools for verification, e.g., based on finite automata [11]. The partial order model has been extensively studied within concurrency theory. It offers a more intuitive view than the interleaving semantics, as concurrent occurrences appear unordered, rather than one after another. However, formalisms that are based on that model and include global states have turned out to have highly intractable computational complexity. For example, EXSPACE-complete for the *until-free* version of ISTL [8, 1], and non-elementary for LTrL [1, 10].

This situation promoted a debate about the practicality of using the partial order model in automatic verification. Independent of this debate, practice shows that to a large extent, using the interleaving model for concurrency is sufficient in most cases. On the other hand, there are some cases where it is highly beneficial to exploit the connection between the interleaving and partial order models. In particular, one can consider all the linearizations (completions into total orders) of a partial order as equivalent; indistinguishable from each other for an observer that can simultaneously compare local states at multiple processes. Although not very common, such a distinction is made in some concurrent algorithms. One example of this is the calculation of global *snapshots* [2], i.e., a distributed global state of the system. Such a snapshot may not appear on an interleaving sequence in which it is accumulated, but rather on an equivalent one. Another example is database serializability, where transactions are designed to appear one after another in at least one representative out of every equivalence class of sequences.

The challenge here is therefore to design a specification formalism that allows using the traditional interleaving view, yet allow expressing some useful partial order aspects. Moreover, we would like to keep the complexity as low as possible. As learned from the high complexity of temporal logics with global states, we aim at a modest extension. We follow [4], where the logic LTL was expanded with the *snapshot operator* $[p]$, with p a propositional formula. The new operator asserts that a global snapshot satisfying p appears in the past on some interleaving sequence equivalent to the current one. This logic was titled in [4] *SLTL*, for Snapshot Linear Temporal Logic.

A model checking algorithm for SLTL was given in [4]. The complexity of this logic was rather high. Given a verified system B with set of actions Σ and an SLTL property φ , the space complexity was polynomial in $2^{|B|+|\Sigma|} + |\varphi|$, where $|B|$ is the size of the system calculated by adding the size of the local processes. For comparison, model checking for LTL can be done in space complexity polynomial in $\mathcal{O}(|B| + |\varphi|)$.

In this paper we provide a more efficient algorithm tailored for synchronous communication systems in the style of CSP [5]. This means that each action is either local to one process or synchronously involving groups (pairs) of processes (changing their local states simultaneously). Moreover, we restrict specifying snapshots by using Boolean combinations of propositional assertions expressing properties of local states. For example, we do not allow a proposition q that asserts that the sum of a variable x of process p_1 and y of process p_2 is even.

Instead, we may need to have a proposition q_1 representing that x is *even* and q_2 representing that y is even, and can replace q by $q_1 \leftrightarrow q_2$.

Under these definitions, we give a new model checking algorithm for SLTL with space complexity polynomial in $(|B| + 2^{|\varphi|})$. The substantial improvement over previous work is that the current algorithm is PSPACE in terms of the system description, which is typically the big part of the input. Moreover, the exponential complexity in the formula results from the requirement of the construction that occurrences of the snapshot operator $[p]$ are applied to propositional formulas in disjunctive normal form. In transforming arbitrary boolean properties to DNF, an exponential blowup may occur. When restricting occurrences of the snapshot operator to formulae in DNF, the space complexity is polynomial in $(|B| + |\varphi|)$.

Our construction introduces *freeze automata*, which are capable of determining the first state in the computation where we subsumed some global snapshot from an equivalent execution.

We believe that our model checking algorithm, whose complexity is not far worse than that for LTL, makes model checking using partial order with snapshot practical, as opposed to the high complexity of using previous such temporal logics.

The paper is structured as follows. In Section 2, we formalize communication sequential processes as a model of concurrent systems. In Section 3, we recall Snapshot LTL, slightly adapted to the model proposed in Section 2. In Section 4, we present the new model checking algorithm, prove its correctness and analyze its complexity.

2 Preliminaries

Communicating Sequential Processes

We first give a simple model of communicating sequential processes [5]. A system of communicating sequential processes is a structure B with the following components:

1. $Proc$ is a finite set of processes,
2. Σ_x an alphabet of actions for every $x \in Proc$.
3. P_x a set of local properties for every $x \in Proc$, such that $P_x \cap P_y = \emptyset$ for $x \neq y$.
4. S_x a set of local-states of the process x for every $x \in Proc$, $s_x^0 \in S_x$ the initial state.
5. $\lambda_x : S_x \longrightarrow 2^{P_x}$ an evaluation function of local properties for every $x \in Proc$.
6. $\delta_x : S_x \times \Sigma_x \longrightarrow S_x$ the deterministic transition function. The functions δ_x are *partial* in the sense that $\delta_x(s_x, a)$ does not have to be defined. If it is defined, we say that a is *enabled* at s_x , otherwise, a is *disabled*.

Note that we do not require the local alphabets to be disjoint. Common actions are used to synchronize local processes, giving rise to a global transition system.

The *transition system* for B is a structure $T_B = \langle S, \Sigma, \delta, s_0, P, \lambda \rangle$, where

1. $S = \prod_{x \in Proc} S_x$,
2. $\Sigma = \bigcup_{x \in Proc} \Sigma_x$ the global alphabet of actions,

3. $\delta : S \times \Sigma \longrightarrow S$ the global transition relation with $\delta(\Pi_{x \in Proc}(s_x), a) = \Pi_{x \in Proc}(t_x)$ iff for all x with $a \notin \Sigma_x$ it holds that $s_x = t_x$ (non participants do not change state), $t_x = \delta_x(s_x, a)$ and a is enabled from s_x for each process x such that $a \in \Sigma_x$.
4. $s_0 = \Pi_{x \in Proc}(s_x^0)$,
5. $P = \bigcup_{x \in Proc} P_x$.
6. $\lambda : S \longrightarrow 2^P$ such that for $p \in P_y$ it holds that $p \in \lambda(\Pi_{x \in Proc}(s_x))$ iff $p \in \lambda_y(s_y)$.

For convenience, we inductively extend the transition function δ to words: $\delta(s, \varepsilon) = s$ and $\delta(s, ua) = \delta(\delta(s, u), a)$. Denote $proc(a) = \{x | a \in \Sigma_x\}$, and extend it to $proc(v) = \bigcup_i \{proc(a_i) | v = a_1 a_2 \dots a_n\}$.

For simplicity, we assume that δ is total, i.e., for every state $s \in S$ there exists some $a \in \Sigma$ and $s' \in S$ such that $\delta(s, a) = s'$. An *execution* of B is an infinite sequence $v \in \Sigma^\omega$ such that $v = a_0 a_1 a_2 \dots$ and there exists an infinite sequence of states from S , $g_0 g_1 g_2 \dots$ where,

1. $g_0 = s_0$, i.e., the sequence starts with the initial state of B .
2. $s_{i+1} = \delta(s_i, a_i)$, for $i \geq 0$.

Trace Equivalence

Based on B , we define an independence relation on Σ , the irreflexive and symmetric relation $I \subseteq \Sigma \times \Sigma$, such that $a I b$ iff $proc(a) \cap proc(b) = \emptyset$. For two words $u, v \in \Sigma^*$, write $u \stackrel{1}{\equiv} v$ if there exist words w_1, w_2 and letters a, b such that $(a, b) \in I$, $u = w_1 a b w_2$ and $v = w_1 b a w_2$, i.e., if u is obtained from v by exchanging the order of two adjacent independent letters. Let $\equiv$ be the reflexive and transitive closure of the relation $\stackrel{1}{\equiv}$. We say that u and v are *trace equivalent* [6] over (Σ, I) if $u \equiv v$. That is, u is trace equivalent to v if u can be obtained from v by repeatedly commuting adjacent independent letters.

By simple induction, if u and v are trace equivalent, then $\delta(s, u) = \delta(s, v)$ and this holds in particular at $s = s_0$, the initial state. Let $a \in \Sigma$ and $p \in P_x$ for some x . Then we say that a is *visible w.r.t. p* if $a \in \Sigma_x$, *invisible* otherwise. If a is invisible w.r.t. p , then in particular $p \in \lambda(s)$ iff $p \in \lambda(\delta(s, a))$ for any state s .

A *global execution sequence* (or *interleaving sequence*) $\xi = g_0 a_0 g_1 a_1 g_2 \dots$ of B is an infinite (alternating) sequence in $(S \times \Sigma)^\omega$ such that $g_0 = s_0$, and for each $i \geq 0$ we have $g_{i+1} = \delta(g_i, a_i)$. Note, that this sequence is determined by the infinite sequence $a_0 a_1 \dots \in \Sigma^\omega$ because B is deterministic.

Denote by $u \prec v$ the fact that u is a finite prefix of v .

Definition 1. The limit extension $\approx^{\lim} \subseteq \Sigma^\omega \times \Sigma^\omega$ of an equivalence relation $\approx \subseteq \Sigma^* \times \Sigma^*$ is defined by $w_1 \approx^{\lim} w_2$ if and only if

- for every $u \in \Sigma^*$ such that $u \prec w_1$ there exist $v, v' \in \Sigma^*$ such that $v \prec w_2$ and $uv' \approx v$, and
- for every $u \in \Sigma^*$ such that $u \prec w_2$ there exist $v, v' \in \Sigma^*$ such that $v \prec w_1$ and $uv' \approx v$.

Two infinite words w_1 and w_2 are said to be *trace equivalent* if $w_1 \equiv^{\text{lim}} w_2$. We denote trace equivalence for both finite and infinite words by using $\equiv$. A *trace* is then defined to be such an equivalence class. We sometimes denote a trace by writing one representative of the equivalence class in square brackets, e.g., when aIb and aIc but $\neg bIc$, $[abac] = \{baac, abac, aabc, baca, abca\ bcaa\}$. The alphabet and independence relation should be clear from the context. We say that a trace $[u]$ *subsumes* $[v]$, denoted $[v] \sqsubseteq [u]$ if there exists some v' such that $u \equiv vv'$.

3 Snapshot Linear Temporal Logic

We recall the definition of Snapshot Linear Temporal Logic (SLTL) as an extension of LTL with a construct for dealing with snapshots. We call the new extension *Snapshot Linear Temporal Logic* or SLTL. Let P be the finite set of propositional formulas of a system of communicating sequential processes. Let $Bool(P)$ be the set of Boolean combinations with propositions over P .

$$\varphi ::= (\varphi) \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \bigcirc\varphi \mid \Box\varphi \mid \Diamond\varphi \mid \varphi\mathcal{U}\varphi \mid \varphi\mathcal{V}\varphi \mid q \mid [q]$$

where $q \in Bool(P)$. Note that the $[\]$ construct is applied only to a Boolean expression of local propositions, never to a formula with modalities. Note also that we use square brackets for two different (although related) notions: for trace equivalence classes, as in the previous section, and in the SLTL logic to denote that a Boolean combination holds in a subsumed snapshot.

The semantic interpretation of SLTL with respect to a system B is defined over a pair of sequences $u \in \Sigma^*$ and $v \in \Sigma^\omega$.

- $(u, v) \models \bigcirc\varphi$ iff for some $a \in \Sigma$ such that $v = av'$ it holds that $(ua, v') \models \varphi$.
- $(u, v) \models \varphi\mathcal{U}\psi$ iff there exists w, v' , where $v = wv'$, such that $(uw, v') \models \psi$ and for any decomposition $w = w_1w_2$ where w_2 is nonempty, $(uw_1, w_2v) \models \varphi$.
- $(u, v) \models \neg\varphi$ iff it is not the case that $(u, v) \models \varphi$.
- $(u, v) \models \varphi \vee \psi$ iff either $(u, v) \models \varphi$ or $(u, v) \models \psi$.
- $(u, v) \models p$ iff $p \in \lambda(\delta(s_0, u))$.
- $(u, v) \models [q]$ iff there exist a sequence $u_1, u_2 \in \Sigma^*$ such that $u \equiv u_1u_2$ and $(u_1, u_2v) \models q$. Note that since q is restricted to be a boolean combination of atomic propositions, this depends only on a subsumed trace $[u_1] \sqsubseteq [u]$ and the set of satisfied propositions $\lambda(\delta(s_0, u_1))$.

For the rest of the temporal operators, their semantics is implied by the following definitions as usual: $\varphi \wedge \psi = \neg((\neg\varphi) \vee (\neg\psi))$, $\varphi \rightarrow \psi = (\neg\varphi) \vee \psi$, $\varphi\mathcal{V}\psi = \neg((\neg\varphi)\mathcal{U}(\neg\psi))$, $true = p \vee \neg p$, $false = p \wedge \neg p$, $\Box\varphi = false\mathcal{V}\varphi$, and $\Diamond\varphi = true\mathcal{U}\varphi$.

We recall some properties of the snapshot operator:

- $\Box([p] \rightarrow \Box[p])$ Monotonicity of the snapshot operator.
- $\Box(p \rightarrow [p])$
- $\Box((p \wedge q) \rightarrow [p] \wedge [q])$
- $\Box((p \vee q) \leftrightarrow [p] \vee [q])$
- $\Box(\neg[p] \rightarrow [\neg p])$

A formula φ holds for a system (or program) B (written $B \models \varphi$) iff $(\varepsilon, u) \models \varphi$ for every execution u of B , $u \in \Sigma^\omega$. Given B and φ , the model checking problem is to verify whether $B \models \varphi$, or, equivalently, if there exists an execution u such that $(\varepsilon, u) \models \neg\varphi$.

We refer to each state subsumed by the execution as a *snapshot*. Formally, a snapshot of an infinite execution u of a system B is any state s such that there exists a finite prefix v of u with $u = vw$, $[v] = [v_1 v_2]$ and $s = \delta(s_0, v_1)$. Thus, the “S” in SLTL, which stands for “snapshot” refers to the ability of the logic to refer also to subsumed states, rather than only to the states appearing on a given execution sequence.

Consider the formula $\Box\varphi \wedge \Diamond[\psi]$, where φ and ψ are Boolean combinations of local properties of the involved processes. It asserts that φ holds in every state of an execution, while ψ holds on a snapshot of at least one equivalent sequence, but not necessarily the observed one, as opposed to the usual LTL formula $\Box\varphi \wedge \Diamond\psi$. The formula $(\neg[\varphi])\mathcal{U}[\psi]$ means that φ does not hold for any snapshot subsumed by states of the execution until the first state that subsumes a snapshot satisfying ψ . This can be compared to $(\neg\varphi)U\psi$, in which φ and ψ only refer to the states on the current execution sequence. For more motivation for the SLTL logic, refer to [4].

In [4], a model checking algorithm for Snapshot LTL was given. It is in EX-PSPACE with a space complexity of $\mathcal{O}(|S| \times 2^{|\Sigma|} \times |\varphi|)$. Note, that the complexity is actually EXPSPACE in $|B|$. This is due to the fact that the algorithm remembers sets of visited states. For a sublogic with only negative occurrences of the snapshot operator $[q]$, a PSPACE algorithm was given.

3.1 Elements of the Original Snapshot Construction

In [4], the algorithm for model checking SLTL uses particular components based on remembering past global states, which is the source of its EXPSPACE complexity.

The core of that construction is a deterministic automaton that remembers subsumed states. Formally, for a sequence u we keep track of all states s such that $[u] = [u_1 u_2]$ and $s = \delta(s_0, u_1)$, even when u_1 is a proper prefix of u . In order to find these global states, more information is required. As a first attempt, we could remember all pairs (s, u_2) for this decomposition. A state of such an automaton, which keeps track of subsumed states thus, consists of a set of such pairs. It always contains the pair $(\delta(u), \epsilon)$ with the current state $\delta(u)$ obtained from the initial state s_0 after executing the sequence u . When performing a transition and appending the action a to u (thus arriving at the prefix ua), this induces the substitution of (s, u_2) by (s, u_2a) . If a is independent of all of u_2 , another state is subsumed, as is expressed by the pair $(\delta(s, a), u_2)$. It can be shown that this approach actually maintains all subsumed states (with the information how to go “backwards” to them). However, because of keeping sequences of actions, the set of pairs maintained in this way may be infinite.

Now observe that the only relevant aspect of u_2 (as in $[u_1 u_2]$) is whether an appended label a is fully independent of u_2 or not. Hence, instead of having a sequence component u_2 in a pair, as above, u_2 can be abstracted to information

representing dependency. In [4] this is done by remembering the set of occurring labels, thus the “subsumed states tracker” contains, in addition to the current state, pairs (s, A) , where $s \in S$ and $A \subseteq \Sigma$.

- The number of pairs of such a tracker is $2^{S \times 2^\Sigma}$, and the memory required for representation of a tracker is thus $|S| \times 2^{|\Sigma|}$ bits. Note that typically $|S|$ is already exponential in the number of concurrent components.
- The initial tracker is $\{(s_0, \emptyset)\}$.
- For a tracker M and an action a , the successor tracker is the least set M' such that if $(s, A) \in M$,
 1. $(s, A \cup \{a\}) \in M'$,
 2. $(\delta(s, a), A) \in M'$ if for all $b \in A$ we have $a \not\sqsubseteq b$.

Then, given a snapshot formula $[q]$ the automaton has as accepting states those that include a pair (s, A) such that $\lambda(s)$ implies the satisfaction of q at s . It was shown in [4], that without assumptions on the properties that could be satisfied by arbitrary subsets of the set of states, no smaller deterministic automaton accepting $[q]$ prefixes exists.

In this paper, we follow a different path by looking in more detail on the structure of the properties in the snapshot operator.

4 A New Model Checking Algorithm

The construction given in this paper is based on a generic LTL model checking procedure, augmented with additional components for the snapshot formulas.

4.1 Freeze Automaton

Let $q = \bigwedge_{i \in Proc} q_i$ be a conjunction of local state properties (that is, each $q_i \in Bool(P_i)$). With respect to the full logic, this restriction excludes, for instance, disjunctions between propositions of distinct processes. We will see later how to treat the general case. For simplicity, we assume a conjunct for every process, hence this can include in particular $q_i = true$.

We present a construction of a deterministic *freeze automaton* that recognizes prefixes u of sequences uv such that $(u, v) \models [q]$.

- The *states* of the freeze automaton $freeze(\bigwedge_{i \in Proc} q_i)$ are pairs $(s, \{F_1, \dots, F_k\})$ of a system state s and a set of *freeze sets* $F_j \subseteq Proc$. Each such set F_j represents a subset of the processes where a collection of local states that satisfy a part of the above conjunction were found. Namely $\bigwedge_{i \in F_j} q_i$. Furthermore, this collection of local states can still be potentially completed after u into a full state that satisfies q .

We add a flag that is set to *true* when we have in the state a freeze set that contains all the processes. Subsequently, the flag remains *true* and thereafter we do not need to update the freeze sets.

- The *initial state* is the set of all subsets of $\{i \mid q_i \in \lambda(s_0)\}$.
- The *successor relation* is defined as follows. Let $a \in \Sigma$ and (s, G) with $G = \{F_1, \dots, F_k\}$ some set of freeze sets, s a state of the system and $s' = \delta(s, a)$

the successor system state and $\text{addproc}(s, a) = \{j \mid a \in \Sigma_j \wedge q_j \in \lambda(\delta(s, a))\}$ the set of processes participating in a such that their parts of the state property q are satisfied after the transition a . Then the successor state in the freeze set automaton is (s', G') where G' is the least set such that

extension For every $F_1 \subseteq \text{addproc}(s, a)$ and every $F_2 \in G$ such that $\text{proc}(a) \cap F_2 = \emptyset$ we have $F_1 \cup F_2 \in G'$. Note that F_1, F_2 can also include here the empty set.

This is the case where we can extend (or propagate, when $F_2 = \emptyset$) a freeze set from a previous state with some processes; these processes are not in the old freeze set and have just participated in the execution of the most recent action, and furthermore their new local states satisfy the corresponding literals of the conjunction q .

propagation For every $F \in G$ with $\text{proc}(a) \subseteq F$ also $F \in G'$.

This is the case where freeze sets propagate (without extension) from a previous state to the current one when the set of processes involved in the most recent action is included in the freeze set.

The crucial aspect of the freeze automata is given in the following lemma.

Lemma 1. *Let $u \in \Sigma^*$ and let (s, G) be the state of the freeze automaton reached after u (in particular $s = \delta(s_0, u)$), and let $F \subseteq \text{Proc}$ be some set of processes. Then $F \in G$ iff there exists a decomposition $[u] = [u_1 u_2]$ such that $\text{proc}(u_2) \subseteq F$ (actions in u_2 are limited to processes in F) and $q_j \in \lambda(\delta(s_0, u_1))$ for all $j \in F$.*

Proof. The lemma states an invariant, which we prove by induction on $|u|$.

In the case of $u = \varepsilon$ the obvious decomposition is $u_1 = u_2 = \varepsilon$ with $\text{proc}(u_1) = \emptyset \subseteq F$ for any $F \subseteq \text{Proc}$. The statement of the lemma then degenerates to the fact that $F \subseteq \{x \mid q_x \in \lambda(s_0)\}$, which corresponds precisely to the definition of the initial state of the freeze automaton.

Now let $u = u'a$, (s, G) be the state of the freeze automaton reached by $u'a$, and (s', G') be the state of the freeze automaton reached by u' .

$\Rightarrow$: Let $F \in G$. The inductive definition of the sets G allows for two cases (according to the two rules used in the transition relation of the freeze set automaton):

- By using the extension rule, there is some $F' \in G'$ with $F = F' \cup F''$ with $F'' \subseteq \text{addproc}(s', a) \subseteq \text{proc}(a)$ and moreover

$$F' \cap \text{proc}(a) = \emptyset. \quad (1)$$

Then apply by induction the lemma to u', F' . Let $u_1 = u_1'$ and $u_2 = u_2'$. We obtain a decomposition $[u'] = [u_1' u_2']$ with $[u_1' u_2' a] = [u' a]$, where

$$\text{proc}(u_2') \subseteq F' \quad (2)$$

and $q_x \in \lambda(\delta(s_0, u_1'))$ for each $x \in F'$. Because of (1) and (2), all actions in u_2' are independent of a and hence $[u_1' a u_2'] = [u_1' u_2' a]$. Thus, the actions in u_2 are also invisible for q_x for each $x \in F'' \subseteq \text{proc}(a)$, and we also obtain $q_x \in \lambda(\delta(s_0, u_1' a))$ for each $x \in F''$. Hence, the decomposition $[u' a] = [(u_1' a) u_2']$ satisfies the desired consequence.

- By using the **propagation** rule, $F \in G'$ and $\text{proc}(a) \subseteq F$. In this case, we apply the induction hypothesis to u' thus obtaining a decomposition $[u'] = [u'_1 u'_2]$ with $q_x \in \lambda(\delta(s_0, u'_1))$ for each $x \in F$ and $\text{proc}(u'_2) \subseteq F$ and we can extend it to a desired decomposition $[u] = [u_1 u_2]$ with $u_1 = u'_1$ and $u_2 = u'_2 a$.

$\Leftarrow$: Now let $[u'a] = [u_1 u_2]$ such that $\text{proc}(u_2) \subseteq F$ and $q_x \in \lambda(\delta(s_0, u_1))$ for all $x \in F$. We have to show that $F \in G$ for the set G reached by $u = u'a$. There are three cases concerning this decomposition:

- $\text{proc}(a) \subseteq F$. Then either (1a) a decomposition as required exists already for u' , i.e. $[u'] = [u'_1 u'_2]$, $\text{proc}(u'_2) \subseteq F$, $q_x \in \lambda(\delta(s_0, u'_1))$ with $x \in F$, in which case the induction hypothesis will yield $F \in G'$ where G' is the set of freeze sets reached by u' , which is preserved in G due to the **propagation** rule for freeze sets. Or (1b) a decomposition as required does not exist for u' . In this case, the decomposition $[u'a] = [u_1 u_2]$ with $q_x \in \lambda(\delta(s_0, u_1))$ for all $x \in F$ will not have a occur in u_2 and can be assumed to have it occur as last element of $u_1 = u'_1 a$, but then the decomposition $[u'] = [u'_1 u'_2]$ (we simply remove a from u_1 , where a is independent of u_2 , the price is that we may modify its visible properties) satisfies all the preconditions for $F' = F \setminus \text{proc}(a)$. By independence of a with u_2 , all the actions of u_2 are invisible for q_x , for each $x \in \text{proc}(a)$. Therefore, $q_x \in \lambda(\delta(s_0, u))$. By induction, $F' \in G'$ reached by u' . Thus, by the **extension** rule we obtain $F \in G$.
- $\text{proc}(a) \cap F = \emptyset$. Then we can assume for the decomposition $[u'a] = [u_1 u_2]$ that $u_1 = u'_1 a$, where a is independent of u_2 , and we obtain a decomposition $[u'] = [u'_1 u'_2]$ as required by the lemma and by induction we obtain $F \in G'$ and by the **extension** rule we obtain $F \in G$.
- There is a non trivial intersection $\text{proc}(a) \cap F \neq \emptyset$, but $\text{proc}(a) \not\subseteq F$. This case is quite similar to (1b) above, but here we partition F into $F' := F \setminus \text{proc}(a)$ and $F'' := F \cap \text{proc}(a)$. Thus, a cannot occur in u_2 and the decomposition of $[u] = [u_1 u_2]$ must have a occurring in $[u_1]$. Without loss of generality $u_1 = u'_1 a$, where $q_x \in \lambda(\delta(s_0, u'_1 a))$ for $q_x \in F$. Then we can apply the induction hypothesis to $[u'] = [u'_1 u'_2]$ and F' to find $F' \in G'$. On the other hand, due to the invisibility with respect to the propositions $\cup\{P_x | x \in F''\}$ of all of u_2 (because of independence of u_2 w.r.t. a), we also have that $q_x \in \lambda(\delta(s_0, u'a))$ for all $x \in F''$, thus by the **extension** rule we obtain $F \in G$. $\square$

Corollary 1. *Let uv be some infinite execution and (s, G) be a state of the freeze automaton reached by a word u . Then $(u, v) \models [\bigwedge_{i \in \text{Proc}} q_i]$ iff $\text{Proc} \in G$.*

Proof. The case of $F = \text{Proc}$ in Lemma 1 means that there exists a decomposition $[u] = [u_1 u_2]$ with $q_x \in \lambda(\delta(s_0, u_1))$ for all $x \in \text{Proc}$. This means that the decomposition $[u] = [u_1 u_2]$ corresponds to the existence of a previous snapshot satisfying the conjunction. $\square$

This completes the correctness proof of the freeze automaton. However, a naive “implementation” of the set of freeze sets would require space exponential in the number of processes. Next, we show that a polynomial memory representation with an efficient update operation is possible. The first step is a technical observation:

Lemma 2. *For any state (s, G) reachable in the freeze automaton, the set of sets G is closed under arbitrary union and intersection and thus forms a complete lattice.*

Proof. The lemma states an invariant that we prove by induction: We have to show that the property is preserved over the execution of transitions. First note that invariantly $\emptyset \in G$. Since G is finite, to show closure under arbitrary union and intersection it is sufficient to show that $F_1, F_2 \in G$ implies $F_1 \cup F_2, F_1 \cap F_2 \in G$.

Now let (s', G') satisfy the property and let (s, G) be a successor state reached from it by executing $a \in \Sigma$. Let $F_1, F_2 \in G$. For each F_i there are two cases: (1) $F'_i := F_i \setminus \text{proc}(a) \in G'$ and for all $i \in \text{proc}(a) \cap F_i$ it holds that $s \models q_i$ (this corresponds to the **extension** rule) and (2) $\text{proc}(a) \subseteq F_i$, where $F_i = F'_i \in G'$ (this is the case of the **propagation**).

By induction, we know that $F'_1 \cap F'_2, F'_1 \cup F'_2 \in G'$. Now, there are several cases of combinations for the two sets F_1, F_2 :

If both F_1, F_2 satisfy (1), then $F'_1 \cap F'_2 = (F_1 \setminus \text{proc}(a)) \cap (F_2 \setminus \text{proc}(a)) = (F_1 \cap F_2) \setminus \text{proc}(a) \in G'$, and for all $i \in (F_1 \cap \text{proc}(a)) \cap (F_2 \cap \text{proc}(a)) = (F_1 \cap F_2) \cap \text{proc}(a)$ we have $s \models q_i$. Hence, by the **extension** rule, we obtain $F_1 \cap F_2 \in G$. Similarly, $F'_1 \cup F'_2 = (F_1 \setminus \text{proc}(a)) \cup (F_2 \setminus \text{proc}(a)) = (F_1 \cup F_2) \setminus \text{proc}(a) \in G$ and for all $i \in (F_1 \cap \text{proc}(a)) \cup (F_2 \cap \text{proc}(a)) = (F_1 \cup F_2) \cap \text{proc}(a)$ we have $s \models q_i$, thus by rule **extension** we have $(F_1 \cup F_2) \in G$.

If both satisfy (2), then one immediately finds $(F_1 \cap F_2), (F_1 \cup F_2) \in G'$ and $\text{proc}(a) \subseteq (F_1 \cap F_2) \subseteq F_1 \cup F_2$, hence, by the **propagation** rule we obtain $(F_1 \cap F_2), (F_1 \cup F_2) \in G$.

Now let one (say F_1) satisfy (1) and the other (F_2) satisfy (2), then $F_1 \setminus \text{proc}(a), F_2 \in G'$. Then $(F_1 \setminus \text{proc}(a)) \cap F_2 = (F_1 \cap F_2) \setminus \text{proc}(a) \in G'$ and for all $i \in \text{proc}(a) \cap (F_1 \cap F_2) \subseteq \text{proc}(a) \cap F_1$ we have $s \models q_i$ and hence by the rule **extension** we obtain $F_1 \cap F_2 \in G$. On the other hand, $(F_1 \setminus \text{proc}(a)) \cup F_2 = F_1 \cup F_2 \in G'$ as $\text{proc}(a) \subseteq F_2 \subseteq F_1 \cup F_2$, hence by the **propagation** rule $F_1 \cup F_2 \in G$. $\square$

The closure properties of sets of freeze sets of Lemma 2 allow us to keep a compact representation, where we do not represent sets that are unions of other sets. In terms of lattice theory, we represent the sublattice of the powerset lattice by a *minimal basis*, which is very small, as expressed by the following constructive formulation of Birkhoff's theorem on distributive lattices.

Definition 2. *Let S be a finite set of elements and $\mathcal{T} \subseteq 2^S$ a set of subsets S , such that $\emptyset \in \mathcal{T}$ and for $T_1, T_2 \in \mathcal{T}$ also $T_1 \cap T_2, T_1 \cup T_2 \in \mathcal{T}$. Let $\mathcal{B} \subseteq \mathcal{T}$ be the set of nonempty elements of $\mathcal{T}$ such that each element is not a trivial (i.e., of size one) union of any subset of $\mathcal{T}$.*

Proposition 1. *For every set $T \in \mathcal{T}$, (a) $T = \bigcup \{B \in \mathcal{B} \mid B \subseteq T\}$, i.e. $\mathcal{B}$ is a basis of T and (b) $|\mathcal{B}| \leq |\bigcup \mathcal{B}| \leq |S|$.*

Proof. Let us first define $U_i := \bigcap \{V \in \mathcal{T} \mid i \in V\}$ and observe that $U_i \in \mathcal{T}$ because of closure under intersection. In fact $U_i \in \mathcal{B}$ because if $U_i = V_1 \cup V_2$ either $i \in V_1$ (and hence $U_i = V_1$) or $i \in V_2$ (and hence $U_i = V_2$).

For claim (a), it suffices to observe that $V = \bigcup\{U_i \mid i \in V\}$. For claim (b), we will show that for each $V \in \mathcal{B}$, there exists some $i \in V$ such that $V = U_i$. By contradiction, suppose that this is not the case. Let $\mathcal{K} = \{U_i \mid i \in V\}$. Then $V \notin \mathcal{K}$. But $V = \bigcup \mathcal{K}$, which contradicts the fact that $V \in \mathcal{B}$. This gives a surjective function $\bigcup \mathcal{T} \rightarrow \mathcal{B}$ (surjective since it can be that $U_i = U_j$ for $i \neq j$) and hence $|\mathcal{B}| \leq |\bigcup \mathcal{T}| \leq |S|$ as required. $\square$

Hence, we can represent a basis for freeze sets using a $|Proc|^2$ matrix where each row represents a basis set and each column represents an element. Unused rows can be filled with the empty set.

4.2 Updating Freeze Sets Bases

Let $\mathcal{B}_s$ be a basis for the freeze sets $\mathcal{G}_s$ at a state s , with action a occurring, producing a successor state s' with set of freeze sets $\mathcal{G}_{s'}$ with basis $\mathcal{B}_{s'}$. We describe now how to update $\mathcal{B}_s$ into $\mathcal{B}_{s'}$ using a polynomial amount of space and, in fact, polynomial time.

To understand the updating, we first construct a set C which will satisfy $\mathcal{B}_{s'} \subseteq C \subseteq \mathcal{G}_{s'}$. In a second step, C can be reduced by omitting elements that are non-trivial unions of other elements in the basis, so as to obtain $\mathcal{B}_{s'}$.

C is the least set such that:

1. If $F \in \mathcal{B}_s$ with $F \cap \text{proc}(a) = \emptyset$, then $F \in C$.
2. If $F \in \mathcal{B}_s$ with $\text{proc}(a) \subseteq F$, then $F \in C$.
3. If $\text{proc}(a) \subseteq \bigcup \mathcal{B}_s$, then for each $F' \in \mathcal{B}_s$ such that neither $F' \cap \text{proc}(a) = \emptyset$ nor $\text{proc}(a) \subseteq F'$, we have $F' \cup \bigcup_{x \in \text{proc}(a)} \bigcap \{F \in \mathcal{B}_s \mid x \in F\} \in C$.
4. If $x \in \text{proc}(a)$ such that $q_x \in \lambda(s')$, then $\{x\} \in C$.

We prove that $C \subseteq \mathcal{G}_{s'}$. It is sufficient to check that for each of the for cases, the corresponding set is added to $\mathcal{G}_{s'}$ either due to the **extension** rule or due to the **propagation** rule. Case 1 is added by extension with the empty set. Case 4 is added by extending the empty set. Case 2 is an obvious consequence of the **propagation** rule. To understand the slightly more complicated case 3, observe that $\mathcal{B}_s \subseteq \mathcal{G}_s$ and that due to the closure properties of $\mathcal{G}_s$ (union and intersection), the constructed sets are in $\mathcal{G}_s$ also; moreover by construction $\text{proc}(a) \subseteq F' \cup \bigcup_{x \in \text{proc}(a)} \bigcap \{F \in \mathcal{B}_s \mid x \in F\}$ and hence these sets are in $\mathcal{G}_{s'}$ by **propagation**.

Now we prove that $\mathcal{B}_{s'} \subseteq C$. For this, it is sufficient to check that every set in $\mathcal{G}_{s'}$ is a union of elements of C . First suppose some $F \in \mathcal{G}_{s'}$ is obtained by the **extension** rule. Then $F = F_1 \cup F_2$ with $F_1 \in \mathcal{G}_s$, $F_1 \cap \text{proc}(a) = \emptyset$ and $F_2 \subseteq \{x \in \text{proc}(a) \mid q_x \in \lambda(s')\}$. Now, F_2 is a union of the elements added to C under case 4, and F_1 is a union of sets F' in $\mathcal{B}_s$ which are added to C according to case 1. Suppose, on the other hand, that F is obtained by the **propagation** rule. To find that F is a union of sets in C let us consider a set $F' \in \mathcal{B}_s$ with $F' \subseteq F$. Now, either $F' \in C$ according to case 1 or 2 or F' is incomparable with $\text{proc}(a)$, therefore not propagated. But then, the preconditions of case 3 are satisfied, and $F' \cup \bigcup_{x \in \text{proc}(a)} \bigcap \{F_1 \in \mathcal{B}_s \mid x \in F_1\} \in C$. Moreover, for each $x \in \text{proc}(a)$ it holds that $\bigcap \{F_1 \in \mathcal{B}_s \mid x \in F_1\} \subseteq F$ (because for some F_1 with $x \in F_1$ it must

hold that $F_1 \subseteq F$). Thus, we can conclude that $F' \cup \bigcup_{x \in \text{proc}(a)} \bigcap \{F_1 \in \mathcal{B}_s \mid x \in F_1\} \subseteq F$. Hence, for each set $F' \in \mathcal{B}_s$ with $F' \subseteq F$ we find a set $F'' \in C$ with $F' \subseteq F'' \subseteq F$, hence F is a union of sets in C .

The size of C is limited by $|\mathcal{B}_s| + |\text{proc}(a)|$, because for each set in $\mathcal{B}_s$, at most one set is added to C according to the disjoint cases 1, 2 and 3. Additionally case 4 results in some singletons. More coarsely, $|C| \leq 2 \cdot |\text{Proc}|$ (see Proposition 1), and the update can be performed with $3 \times |\text{Proc}|$ sets, or $\mathcal{O}(|\text{Proc}|^2)$ space. The time requirement of computing C is easily seen to be equally $\mathcal{O}(|\text{Proc}|^2)$. The identification of a subset $\mathcal{B}_{s'}$ can be done on the same space by first ordering the vectors representing elements of C by set size (this can be done in $\mathcal{O}(|\text{Proc}|^2 \times \log_2(|\text{Proc}|))$ steps) and then searching for each $x \in \text{Proc}$ the first set F containing it (in $\mathcal{O}(|\text{Proc}| \times \log_2(|\text{Proc}|))$ steps), finally compressing the list by eliminating the sets that are not minimal sets of any element (in $\mathcal{O}(|\text{Proc}|^2)$ steps), all in essentially the space of memory containing C (additional $2 \times |\text{Proc}|$ bits are needed to mark elements in C).

4.3 Integration into a Model Checking Algorithm

We assume that in the negated property $\neg\varphi$, the (negative) occurrences of the snapshot operator only occur as conjunctions $[\bigwedge_{i \in \text{Proc}} q_i]$ as given in Section 4.1. In a more general setting, the observation that $[\varphi \vee \psi]$ is equivalent to $[\varphi] \vee [\psi]$ can be used to distribute the snapshot operator over disjunctions with an at most linear growth of the size of the formula (each disjunction potentially introducing an additional snapshot operator). In the case of arbitrary boolean properties, a translation to DNF has to be applied first, leading to a potentially exponential blowup of the formula.

The freeze automaton is deterministic, it thus can be used to determine both the satisfaction of positive and negative occurrences of $[q]$ provided that q is of the form indicated above. Pragmatically, the construction can be optimized using sharing of data structures on several levels (the system state space is needed just once, for instance).

In order to check that a concurrent system B satisfies a property φ , we intersect several automata components as follows. For optimizing on space complexity, we perform a binary search, hence do not fully construct the components. Instead, we only need to be able to enumerate pairs of successor states.

- The verified system B . This can be itself obtained as an asynchronous product of several subcomponents, one per each processor, as described in Section 2.
- For each snapshot formula $[\bigwedge_{i \in \text{Proc}} q_i]$ occurring in $\neg\varphi$ we construct a component $\text{freeze}(\bigwedge_{i \in \text{Proc}} q_i)$ as above (basis representation). Recall that each such component has states that include sets of subsets of processes, whereas they can share the access to B (the states S). The transition between two such states is marked by an action. If a state of $\text{freeze}(\bigwedge_{i \in \text{Proc}} q_i)$ contains a component that includes all the processes, then $[\bigwedge_{i \in \text{Proc}} q_i]$ holds thereafter. A special stable flag marks this situation.

- A component $\mathcal{A}_{\neg\varphi}$ for the Büchi translation of $\neg\varphi$. We keep the snapshot subformulas $[d]$ unprocessed as if they were special new propositions. According to the translation algorithm in [3], $\neg\varphi$ is written in a *negated normal form* first, and $[d]$ may appear negated or non-negated.

The product of these components enforces the following correspondences:

- Between a triple (s, a, s') with $\delta(s, a) = s'$ of the checked system B and a triple (t_i, b_i, t'_i) of $\text{freeze}(\bigwedge_{i \in \text{Proc}} q_i)$:
 - $a = b_i$.
 - The freeze sets of t'_i are updated from t_i according to a and the state s' , as explained above. In particular, it depends on whether the components of s that belong to the processes of a satisfy the local predicates of $\bigwedge_{i \in \text{Proc}} q_i$.
- Between the collection of triples (t_i, b_i, t'_i) of $\text{freeze}(\bigwedge_{i \in \text{Proc}} q_i)$ and a triple (r, c, r') of the property automaton $\mathcal{A}_{\neg\varphi}$. The node r contains propositions and snapshot subformulas negated or non-negated, according to the case where they need to hold or not in the current state. Accordingly, if r contains $[q]$ where $q = \bigwedge_{i \in \text{Proc}} q_i$, then we will have that t_i (and thus also t'_i) indicates that $[\bigwedge_{i \in \text{Proc}} q_i]$ was detected. Conversely, if r contains $\neg[\bigwedge_{i \in \text{Proc}} q_i]$ then for each t_i we must have not yet detected $\bigwedge_{i \in \text{Proc}} q_i$.
- Between triples (s, a, s') of B and (r, c, r') of $\mathcal{A}_{\neg\varphi}$ we have the usual correspondence, i.e., that r and s (and r' and s') agree on propositions.

The space needed for a single state of the constructed Büchi automaton is thus the sum of the space needed for a state of B (that is $\mathcal{O}(|B|)$), the space needed for the property automaton $\mathcal{A}_{\neg\varphi}$ which requires $\mathcal{O}(|\varphi|)$ bits, and the space required for the freeze automata (without the B component, which is shared). Each freeze automaton thus requires $|\text{Proc}|^2$ bits to store the basis of the set of freeze sets and there is one for every snapshot formula $[q]$, which we bound by $\mathcal{O}(|\varphi| \times |\text{Proc}|^2)$. A single state thus requires $\mathcal{O}(|B| + |\varphi| + |\varphi| \times |\text{Proc}|^2) = \mathcal{O}(|B| + |\varphi| \times |\text{Proc}|^2) \leq \mathcal{O}(|\varphi| \times |B|^2)$ bits, the algorithm (based on binary search) requiring $\mathcal{O}((|B| + |\varphi| \times |\text{Proc}|^2)^2)$ bits, which subsumes the $\mathcal{O}(|\text{Proc}|^2)$ space needed for the computation of successors on bases. This was based on the assumption that φ only contained occurrences of the snapshot operator with conjunctions of local properties as required in Section 4.1. To obtain them, a transformation to DNF of properties in some snapshot operators may be necessary, resulting in a potentially exponential blowup of the size of φ . Taking this step into account thus gives us $\mathcal{O}(|B| + 2^{|\varphi|} \times |\text{Proc}|^2)$ bits per state and $\mathcal{O}((|B| + 2^{|\varphi|} \times |\text{Proc}|^2)^2)$ bits for the algorithm.

4.4 Optimization for Negative Snapshots

In searching for a counter example, a negative occurrence of some $[q]$ in φ becomes a positive occurrence in $\neg\varphi$. On a given infinite sequence v , either $[q]$ never holds or there is a first prefix, i.e. a first decomposition $v = v_1 v_2$ such that $(v_1, v_2) \models [q]$, i.e. there exists a trace decomposition $[v_1] = [v'_1 v''_1]$ such that $(v'_1, v''') \models q$, where $v'_1 v''' = v_1 v_2$. Then choosing $q' := \bigwedge \lambda(\delta(s_0, v'_1))$ provides a

formula which implies q and such that on v the snapshot formula $[q]$ holds exactly where $[q']$ holds. Hence, a non-deterministic freeze automaton for $[q]$ first guesses a truth assignment implying q and then runs as normal. In terms of DNF, this automaton guesses a witness minterm of q transformed to DNF. On any run where this non-deterministic freeze automaton says that $[q']$ holds we know that $[q]$ holds and conversely for a sequence where $[q]$ holds there exists a guess q' such that $[q']$ is observed in the freeze automaton and this guess can be done in a coherent manner for all prefixes satisfying $[q]$.

Note that this optimization relies on a construction that does not force every state of the property automaton $\mathcal{A}_{\neg\varphi}$ to have each snapshot formula either negated or non-negated (i.e., a construction in the style of [3] rather than as in [11] needs to be used). This trick allows to enumerate the satisfying truth assignments of the different such positive subformulas separately. This may induce an exponential number of repetitions, but does not increase space. Unfortunately, this trick does not work for positive occurrences of formulas of the type $\neg[q]$ in φ . It is interesting to observe that in [4], the negated snapshots also make an easier case, treated by a separate construction called *lazy automata*.

5 Conclusion

We described in this paper a refined model checking algorithm for Snapshot Linear Temporal Logic. Due to a refined analysis we were able to obtain an algorithm with PSPACE-Complexity in terms of the system description, albeit EXPSpace in the size of the formula - unless the use of the Snapshot operator is restricted to, essentially, disjunctive normal form. An open problem remains to prove non-trivial lower bounds for the space complexity in terms of the properties.

Moreover, our analysis was based on boolean combinations of local properties, whereas certain uses of the snapshot operator might require global properties (like the sum of integer variables of the processes, etc), where the approach applied here cannot be applied directly. However, depending on the nature of the properties, a compromise between the state storing approach of [4] and the symbolic approach here may be possible: by freezing one process, one might symbolically induce a property remaining to be satisfied for the other processes. If for example the global property is $x_1 + x_2 > 5$ and x_1 of process 1 is frozen when $x_1 = 3$, the remaining property can be factored to $x_2 > 2$ without full knowledge of the state of process 1.

We believe that our construction is of value beyond model checking: The fact that the freeze automaton is deterministic with a polynomial time computable successor function suggests use in further applications, where one might actually want to implement and execute it, for instance in the domain of testing.

Acknowledgements

This work was done while the second author was on visit as invited professor at the Université de Provence, Marseille. The authors thank Blaise Genest for useful comments on a previous version of the paper.

References

1. Rajeev Alur, Ken McMillan, Doron Peled. Deciding Global Partial-Order Properties. In *ICALP 1998*, LNCS 1443, 41–52.
2. K. M. Chandy, L. Lamport, Distributed Snapshots: determining the global state of distributed systems, *ACM Transactions on Computer Systems* 3 (1985), 63–75.
3. R. Gerth, D. Peled, M. Vardi, P. Wolper, Simple on-the-fly automatic verification of linear temporal logic, *PSTV 1995*, 3–18.
4. B. Genest, D. Kuske, A. Muscholl, D. Peled, Snapshot Verification, *TACAS 2005*, LNCS 3440, 510–525.
5. C. A. R. Hoare. Communicating Sequential Processes *Communication of the ACM* 21 (8), 1978, 666–677.
6. A. Mazurkiewicz, Trace semantics In *Proceedings of Advances in Petri Nets 1986*, Bad Honnef, LNCS 255, pp. 279–324, 1987.
7. D. Peled. Specification and verification of Message Sequence Charts. In *FORTE/PSTV 2000*, pp.139–154.
8. D. Peled, A. Pnueli. Proving Partial Order Properties. *Theoretical Computer Science*, 126:143–182, 1994.
9. S. Stoller, Y.A. Liu, Efficient Symbolic Detection of Global Properties in Distributed Systems, *CAV 1998*, LNCS 1427, 357–368.
10. I. Walukiewicz. Difficult Configurations – On the Complexity of LTrL. In *ICALP 1998*, LNCS 1443, 140–151.
11. M.Y. Vardi, P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1994), 1–37.