Snapshot Verification

Blaise Genest[3], Dietrich Kuske[2], Anca Muscholl[1], and Doron Peled[3]

[1]LIAFA, Université Paris 7, 2 Place Jussieu
75251 Paris Cedex 05 France
[2]Institut für Algebra, Technische Universität Dresden
D-01062 Dresden, Germany
[3]Department of Computer Science, University of Warwick
Coventry CV4 7AL, United Kingdom

Abstract. The classical model for concurrent systems is based on observing execution sequences of global states, separated from each other by atomic transitions. This model is intuitively simple and enjoys a variety of mathematical tools, e.g., finite automata and linear temporal logic, and algorithms that can be applied in order to test and verify concurrent systems. Although this model is sufficient for most frequently used validation tasks, some phenomena of concurrent systems are difficult to express using its related formalisms. In particular, not all the global states (snapshots) related to an execution appear on a particular execution sequence; some appear on equivalent sequences. Previous attempts to move into formalisms that are based on a more detailed model of execution, e.g,. the causality based model, resulted in specification formalisms with inherently high complexity verification algorithms. We study here verification problems that involve allowing the execution sequences model to observe past global states from equivalent executions. We show various algorithms and complexity results related to our extension of the interleaving model.

1 Introduction

Several temporal logics are tailored to reason about partial order executions. With such logics, we are interested in local states of actions occurred according to the partial order, or in the global states compatible with a partial order. A partial order among events can be completed into multiple total orders that are consistent with it, forming a set of *equivalent* execution sequences. It is sometimes argued that executions that are equivalent in the sense that they are indistinguishable from the point of view of an observer not capable of monitor instantaneously concurrent processes, and hence it is unnatural to distinguish between them. A specification that permits some interleaving sequence but forbids another that is equivalent to it is possibly ambiguous. Such logics include ISTL [16, 21], LTrL with [25] or without past [13] and others. These are global temporal logics, since they model global states of the partial order executions. There are also local partial order logics, e.g., TLC [5], LocTL [8] which are not discussed here. The main problem of the global logics is their very high complexity. Even a reduced version of ISTL that has no *Until* operator turns out to be EXPSPACE complete [4, 24]. For local logics the complexity is within PSPACE [12], but they seem harder to use in practice. For instance, the precise expressivity of TLC is not clear, and LocTL needs more elaborate mechanisms for ensuring FO-completeness.

As a partial order execution model we select Mazurkiewicz traces. Namely, equivalence classes of sequences over some finite alphabet generated using a (fixed) independence relation over the alphabet. Two sequences are equivalent exactly when we can obtain one from the other by commuting adjacent independent occurrences of letters. Thus, if the alphabet includes a, b and c, where a and b are independent, but both interdependent with c, then we have $cabba \equiv caabb$, with $[cabba]$ denoting the trace (equivalence class) that includes the denoted sequence. A trace $[v]$ subsumes a trace $[u]$ if there is a sequence u' such that $v \equiv uu'$. In concurrency theory, this represents the fact that $[u]$ is in a (possible) past of $[u]$. For example, $[cab]$ subsumes itself, $[ca]$, $[cb]$, $[c]$ and $[\epsilon]$ (the empty trace). Informally, we also say that (the word cab subsumes cb).

We describe the Snapshot Linear Temporal Logic, a new temporal logic with propositions $[p]$, expressing that a state satisfying p has to be subsumed. Together with that logic, we give a model-checking algorithm, which is EXPSPACE only in the size of the alphabet. We further identify a fragment of the logic which is PSPACE-complete only in the size of the formula, extending the model-checking for LTL. In order to gain further insight of the model checking problem (as we do not have a tight lower bound for it), we study the model-checking of snapshots of a word. The corresponding language theoretic problem is given a word (which can represent an execution), we want to check whether it subsumes a word that is in some language $\mathcal{L}$ (where $\mathcal{L}$ can represent some property). To formalize the problem, we consider that the property is as follows: given by a trace-closed automaton. Hence, checking whether the snapshot of a word satisfies the property is equivalent to test whether $w \in [\mathcal{L}\Sigma^\star]$, which is somehow related to pattern matching in traces (trace notation will be explained later). We later use a construction for the word problem for giving more efficient model checking algorithm for a subset of our temporal logic.

Model-checking snapshots of a word can be seen as an extension of model-checking a word [19], which is an important task that has not received enough attention. For instance, model-checking a word is the core of runtime verification, but is also needed for DNA algorithms, or checking for a spurious counter-example in an abstracted model. We study this word problem for various cases. We study variations of the problem, namely relaxing the dependencies and considering very long words. If the dependencies are not too complicated, then it is the case the trace alphabet is series-parallel [9], that is, built on serial and parallel composition of letters. This kind of alphabet is often used to facilitate the algorithms [17, 7]. To produce more efficient algorithms on very long words, we will follow the work of several papers [22, 15, 19] which consider that the word is given as a compressed word, by means of Straight Line Programs [22].

Related works deal with checking whether snapshots of an execution of a distributed system satisfies a given propositional predicates. Solutions for this appear e.g., in [11, 23]. In our work, we study the problem of checking whether such a word, or a finite state system, satisfy a given *temporal* property that also deals with snapshots.

2 Preliminaries

Let Σ be a finite alphabet. An *independence relation* is a irreflexive and symmetric relation $I \subseteq \Sigma \times \Sigma$. The alphabet (Σ, I) is called a *concurrency alphabet*.

For two words $u, v \in \Sigma^*$, write $u \overset{1}{\equiv} v$ if there exist words w_1, w_2 and letters a, b such that $(a, b) \in I$, $u = w_1 ab w_2$ and $v = w_1 ba w_2$, 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 $\overset{1}{\equiv}$. We say that u and v are *trace equivalent* [18] 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.

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

We denote trace equivalence for both finite and infinite words by using $\equiv$.

Two infinite words w_1 and w_2 are said to be *trace equivalent* if $w_1 \equiv^{\lim} w_2$.

We sometimes denote a trace by writing one representative of the equivalence class in square brackets, e.g., $[abaac]$. 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 $u \equiv vv'$.

We say that an automaton is trace-closed , if $[\mathcal{L}(\mathcal{A})] = \mathcal{L}(\mathcal{A})$. We denote trace concatenation $[u][v] = [uv]$ for a finite word u and a finite or infinite word v. Then $[u] \sqsubseteq [u][v]$. We extend this to languages

$$[L_1 L_2] = [L_1][L_2] = \bigcup_{u \in L_1, v \in L_2} [u][v]$$

Let $\mathcal{A} = (S, E, \iota, \Sigma)$ be a transition system (a finite automaton, but without accepting states, and with labels on the edges) with

- Set of states S,
- Labeled transitions $E \subseteq S \times \Sigma \times S$ a deterministic transition relation, i.e., if $s \xrightarrow{\alpha} t$ and $s \xrightarrow{\alpha} t'$ then $t = t'$.
- An initial state $\iota \in S$.
- The alphabet Σ.

The following are standard assumptions when $\alpha I \beta$.

- If $s \xrightarrow{\alpha} t_1$ and $s \xrightarrow{\beta} t_2$ then there exists r such that $t_1 \xrightarrow{\beta} r$ and $t_2 \xrightarrow{\alpha} r$.
- If $s \xrightarrow{\alpha} t_1$ and $t_1 \xrightarrow{\beta} r$ then there exists t_2 such that $s \xrightarrow{\beta} t_2$ and $t_2 \xrightarrow{\alpha} r$.

Assume that there exists $s_1, \ldots, s_n$ such that $\iota \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \ldots \xrightarrow{\alpha_n} s_n$ for $w = \alpha_1\alpha_2 \ldots \alpha_n$. Let $s_n = s$. Denote this by $\iota \xrightarrow{w} s$. Then, by the above assumption, and a simple induction over the number of times we commute adjacent letter, we have $\iota \xrightarrow{v} s$ for every $v \equiv w$. The set of *snapshots* of an infinite word w are all the states s such that $\iota \xrightarrow{v} s$ for $[v] \sqsubseteq [w]$. That is, all the states that we can reach with sequences subsumed by w.

3 Snapshot Linear Temporal Logic

We extend the definition of Linear Temporal Logic (LTL) by adding a construct for dealing with snapshots. We call the new extension *Snapshot Linear Temporal Logic* or SLTL. Let $\mathcal{P}$ be a finite set of propositional formulas. Let $Bool(\mathcal{P})$ be the set of Boolean combinations with propositions over $\mathcal{P}$.

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

where $p \in Bool(\mathcal{P})$. Note that the '[ ]' construct is applied only to a Boolean expression, 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 logic to denote that a Boolean combination holds in a subsumed snapshot.

We assume that there is a unique initial state s_0, and that letters represent deterministic transitions. Hence given a finite sequence w executed from s_0, there is a unique state $state(s_0, w)$ that is obtained by applying the actions labeled according to w to the initial state s_0. The semantic interpretation of SLTL is defined over a pair of sequences $u \in \Sigma^\star$ and $v \in \Sigma^\omega$.

- $(u, v) \models \bigcirc\varphi$ exactly when for some $\alpha \in \Sigma$ such that $v = \alpha v'$ it holds that $(u\alpha, v') \models \varphi$.
- $(u, v) \models \varphi\mathcal{U}\psi$ exactly when we can write $v = wv'$ such that $(uw, v') \models \psi$ and for any decomposition $w = w_1w_2$ where w_2 is nonempty, $(uw_1, w_2v) \models \varphi$.
- $(u, v) \models \neg\varphi$ exactly when it is not the case that $(u, v) \models \varphi$.
- $(u, v) \models \varphi \vee \psi$ exactly when either $(u, v) \models \varphi$ or $(u, v) \models \psi$.
- $(u, v) \models p$ exactly when $state(s_0, u) \models p$ according to propositional logic semantics of '$\models$'.
- $(u, v) \models [p]$ exactly when there exists a sequence $u' \in \Sigma^\star$ such that $[u'] \sqsubseteq [u]$ such that $state(s_0, u') \models p$ according to propositional logic.

Note that we distinguish between the connections '$\vee$' and '$\wedge$' over temporal formulas (as interpreted below) and over Boolean formulas (interpreted using the standard propositional logic semantics). In both cases, these operators are denoted in the same way.

The rest of the temporal operators can be defined using the above operators. In particular, $\varphi \wedge \psi = \neg((\neg\varphi) \vee (\neg\psi))$, $\varphi\mathcal{V}\psi = \neg((\neg\varphi)\mathcal{U}(\neg\psi))$, $true = p \vee \neg p$, $false = p \wedge \neg p$, $\Box\varphi = false\mathcal{V}\varphi$, and $\Diamond\varphi = true\mathcal{U}\varphi$.

Some properties of the new snapshot operator are as follows:

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

To motivate such a logic, we describe a situation where an execution is a partial order of events, and a global state is a collection of local states of the different system components that are *history closed*. History closedness means that if the past or history (basically, a set of events) includes some event, then it must also include any event that happened previously. This notion of global states coincides with Chandy and Lamport's snapshot algorithm [6]. In an equivalent way, such a global state is related to snapshots as defined in Section 2. Thus, unlike interleaving semantics, we do not just have a simple sequence of global states. Snapshots are in particular important for achieving fault tolerance, where occasional snapshots of a distributed system are saved in order to allow the system to recover in a consistent way.

In our example, consider a bank with several branches, in different states. The bank operation hours in the different states accord with the local time. There is no global observation of the bank operation. Different branches can update each other by making phone calls. The bank employees are working according to some code of conduct that dictates what action to take in different situations. The customers can make various interactions with their branches (or even visit other branches), including deposits withdrawals and balance enquiries. Phone calls between branches are also actions of the system.

The bank analysts prepared a finite state model of the bank, and have written a specification of allowed behaviors of the bank system, using our specification formalism. The marked nodes in the graph correspond to some bank targets, e.g., having a certain balance, which is defined as the sum of money over the different branches in some global state. The bank lawyers can use histories of the executed actions to show that a balance existed. The bank does not stop everything in all branches to take frequent global and synchronized snapshots, e.g., printing the balance in each branch at exactly every hour. Hence snapshots are a good notion of balance that they can have.

We can express the fact that the bank has a positive balance snapshot for every (execution) sequence by $\Diamond[p]$, where p is a predicate denoting positive balance. This does not mean that p holds for some state in every execution sequence. We can also express the fact that q holds for the minimal state that has a subsumed snapshot satisfying p by $(\neg q \wedge \neg[p])\mathcal{U}(q \wedge [p])$. We can extend the logic with related operators. For example, under the current definitions, $\Box\Diamond[p]$ does not mean under our semantics that there are infinitely many subsumed snapshots satisfying p, since $[p]$ is monotonic. Thus, we can add an appropriate construct to capture such a property.

3.1 A Deterministic Snapshot Construction

We first construct from the state space $\mathcal{A} = (S, E, \iota, F, \Sigma)$ a state space $\mathcal{B}$ that includes the 'memory' that is required for deciding snapshot properties.

Observing the state space $\mathcal{A} = (S, E, \iota, F, \Sigma)$, we derive a state space $\mathcal{B} = (S^{\mathcal{B}}, E^{\mathcal{B}}, \iota^{\mathcal{B}}, F^{B}, \Sigma)$ as follows:

- The state set $S^{\mathcal{B}}$ equals $2^{S\times 2^{\Sigma}} \times S$. We denote the singleton state from $\mathcal{A}$ in a state s of $\mathcal{B}$ as *current*(s), and the set of all states of $\mathcal{A}$ in s as *past*(s).
- There is an a-labeled edge in $E^{\mathcal{B}}$ from $(\mathcal{X}, s)$ to $(\mathcal{Y}, t)$ if
  1. $(s, a, t) \in E$, and
  2. $\mathcal{Y}$ is the set of all pairs $(t', B) \in S \times 2^{\Sigma}$ for which there is $(s', A) \in \mathcal{X}$ satisfying either
     (a) $s' = t'$ and $B = A \cup \{a\}$, or
     (b) $(s', a, t') \in E$, $A = B$, and $\{a\} \times A \subseteq I$.
- the initial state $\iota^{\mathcal{B}}$ is $(\{(\iota, \emptyset)\}, \iota)$
- a state $(\mathcal{X}, s)$ is accepting (i.e., belongs to $F^{\mathcal{B}}$) if $\mathcal{X}$ contains some pair (t, A) where $t \in F$.

Intuitively, we keep in every state of $\mathcal{B}$ the current state s on $\mathcal{A}$ given the same sequence of letters from the initial state. In addition, we keep with s the set of states of subsumed traces. If t is a state of a subsumed trace, then we keep with it also the set of letters A (but not the actual sequence) that are executed between t and s. Let $s \xrightarrow{\beta} r$ in E. Given a pair $\langle t, A\rangle$ kept as a past of s we generate the following pairs as a past of r: We add β to A and remain in state t, obtaining $\langle t, A \cup \{\beta\}\rangle$. This is because the set of subsumed traces is just extended, and if $t \xrightarrow{v} s$ then $t \xrightarrow{v\beta} r$. Another pair is formed is when β is independent of the letters in A. In this case we can then also progress from t to r according to the transition relation of E obtaining $\langle r, A\rangle$. This is because if $u \equiv vv'$ (and hence $[v]$ is a prefix of $[u]$, and β is independent of the letters in v', then $[v\beta]$ is a prefix of $[u\beta]$ and $t \xrightarrow{v'} r$.

A model checking algorithm for SLTL uses the structure $\mathcal{B}$ instead of $\mathcal{A}$. In every state s of $\mathcal{B}$, if p holds in *current*(s) then p also holds in s. If p holds in any of the components of *past*(s) then $[p]$ holds for s. We can translate $\neg\varphi$ into an automaton $\mathcal{C}$ [27]. When a subformula of the form $[p]$ occurs, we keep it as one entity, even if p is a Boolean combination. Thus, in the translation, a node of $\mathcal{C}$ can be labeled with $[p]$ or with $\neg[p]$.

Then we check the intersection of $\mathcal{C}$ with $\mathcal{B}$. This gives an algorithm of time complexity $\mathcal{O}(2^{|\mathcal{A}|\times 2^{|\Sigma|}} \times 2^{|\varphi|})$. However, note that we do not need to construct the automata for $\mathcal{C}$ or $\mathcal{B}$. Instead, we can perform a binary search on the state space, where all we need to decide is whether a given state belongs to $\mathcal{C}$ or $\mathcal{B}$, and whether a pair of such states is consecutive according to these automata. This gives an algorithm with space complexity $\mathcal{O}(|\mathcal{A}| \times 2^{|\Sigma|} \times |\varphi|)$. In comparison, note that the usual space efficient algorithm for model checking is logarithmic in the number of states and polynomial in the size of the property φ.

Theorem 1 *Let $\mathcal{S}$ be an automaton describing the system, and φ be a formula of the Snapshot Linear Temporal Logic. Then one can check whether $\mathcal{S} \models \varphi$ in EXPSPACE, with a space complexity of $\mathcal{O}(|\mathcal{S}| \times 2^{|\Sigma|} \times |\varphi|)$, that is, in space complexity exponential only in the size of the alphabet.*

4 Model-Checking Snapshots of a Word

We are given a language $L = [L]$ i.e., a language closed under trace equivalence w.r.t. some given alphabet Σ and independence relation I. We are also given an automaton $\mathcal{A}$ such that $L = \mathcal{L}(\mathcal{A})$, and a word $w \in \Sigma^\star$. We want to check whether some snapshot of w fulfils the property given by $\mathcal{A}$, that is, whether $w \in [L\Sigma^\star]$. Note that the language $[L\Sigma^\star]$ consists of words from L where arbitrary letters are appended to them, and then shuffled to the left according to the independence relation I.

4.1 A Nondeterministic Construction for the Membership Problem

Let $\mathcal{A} = (S, E, \iota, F)$ be an automaton for some regular language $\mathcal{L}(\mathcal{A})$, over the alphabet Σ, labeling the edges E. Let $I \subset \Sigma \times \Sigma$ be the symmetric, irreflexive independence relation. The automaton $\mathcal{A}$ is deterministic and closed under trace equivalence over I.

For checking emptiness or inclusion of a word w in the language $[\mathcal{L}(\mathcal{A})\Sigma^\star]$, we can make the following construction.
Construction: a new automaton $\mathcal{D} = (S \times 2^\Sigma, E', \iota \times 2^\Sigma, F \times \emptyset)$. This automaton guesses the letters that we still need to use in order to complete w. In addition to the original state s of $\mathcal{A}$, we save in the state memory the aforementioned set of letters A. Thus, there are $|S| \times 2^{|\Sigma|}$ states. Given a letter $\alpha \in \Sigma$, we continue from such a state $\langle s, A\rangle$ according to the transition relation of $\mathcal{A}$ to a state $\langle s, B\rangle$, provided that $\alpha \in A$. We need to update the guess of the letters of w still to be used. If β will not occur anymore, then $B = A \setminus \{\alpha\}$. Otherwise $B = A$. Also, if $\alpha \notin A$ and α is independent of every letter in A, then we may loop on the state $\langle s, A\rangle$ with α. Thus, $\mathcal{L}(\mathcal{D}) = [\mathcal{L}(\mathcal{A})\Sigma^\star]$.

Formally, $\mathcal{D}$ has the following transitions:

- $\langle s, X\rangle \xrightarrow{\alpha} \langle s, X\rangle$, when α is independent of the letters in X.
- $\langle s, X\rangle \xrightarrow{\beta} \langle t, X\rangle$, when $\beta \in X$ and $s \xrightarrow{\beta} t \in E$.
- $\langle s, X\rangle \xrightarrow{\beta} \langle t, X \setminus \{\beta\}\rangle$, when $\beta \in X$ and $s \xrightarrow{\beta} t \in E$.

Thus, the automaton $\mathcal{D}$, guesses what letters it would use on a word from $L(\mathcal{A})$ (opposite to previous automaton, which kept what it used from any point, hence had to keep many histories). When progressing on such a word, it may update that subset (but doesn't have to, since it may use the same latter again). It also use some letter independent of this guess, making a self loop. It accepts only with an empty guess (all letters were subsumed).

Basically, $\mathcal{D}$ is built on $2^{|\Sigma|}$ copies of $\mathcal{A}$, that is, it is of size $\mathcal{O}(|S| \times 2^{|\Sigma|})$. Now, consider that if a word belongs to $\mathcal{L}(\mathcal{D})$, then it will pass through at most $|\Sigma|$ of these copies. That is, these Σ automata can be non-deterministically guessed, together with the positions of w where the transition from one automaton to another is made. Then, it suffices to test whether each factor of w between two consecutive positions belongs to the automaton that was guessed, which can be easily performed in polynomial time.

Theorem 2 *Let (Σ, I) be a trace alphabet. Let $\mathcal{A}$ be a trace-closed automaton and w a word of $\Sigma^\star$. Then one can test in NP whether $w \in [\mathcal{L}(\mathcal{A})\Sigma^\star]$. If the alphabet is fixed (not part of the input), then the problem is NLOGSPACE.*

4.2 Lower Bound

We consider the trace monoid $(\Sigma, I) = \{a,b\}^* \times \{c,d\}^*$, that is $(x,y) \in I$ exactly when $x \in \{a,b\}$ and $y \in \{c,d\}$. For $p \in \mathbb{N}$, let $L_p = \{(ua, vc) \in \Sigma^\star : |u| \equiv |v| \mod p\}$. To accept $[L_p]$, it suffices to recall the number of letter from $\{a,b\}$ modulo p, plus the last letter from $\{a,b\}$. Symmetrically, we do the same for $\{c,d\}$, yielding a deterministic automaton of size $(2p)^2$.

The language $[L_p \cdot \Sigma^\star]$ accepts every word $(u_1 \cdots u_n, v_1 \cdots v_m)$ such that there exists $i = j \pmod p$ with $u_i = a$ and $v_j = c$. We want to show that in order to accept $L_p \cdot \Sigma^\star$ with a deterministic automaton, we need exponentially many states (exponential in p): For each subset $X = \{x_1, x_2, \ldots, x_n\} \subseteq \{0, 1, \ldots, p-1\}$ let $u_X = (b^{x_1}ab^{p-x_1-1})(b^{x_2}ab^{p-x_2-1})\ldots(b^{x_n}ab^{p-x_n-1})$. Then, for $m < p$ we have $(u_X, d^m c) \in [L_p \cdot \Sigma^\star] \iff \exists i, m = x_i \pmod p \iff m \in X$.

We show now that in any deterministic automaton accepting $[L_p \cdot \Sigma^\star]$, the traces (u_X, ε) lead to mutually different states when executed in the initial state. Assume by contradiction that u_X and $u_{X'}$ lead to the same state. Let us take $x \in X \setminus X'$. If such an x does not exist, we take $x \in X' \setminus X$. Then $(u_X, d^x c)$ and $(u_{X'}, d^x c)$ both lead to the same state s. The state s should be final because $(u_X, d^x c) \in [L_p \cdot \Sigma^\star]$, a contradiction since it shouldn't be accepting since $(u_{X'}, d^x c) \notin [L_p \cdot \Sigma^\star]$. Hence the deterministic automaton has at least 2^p states.

Notice that we cannot derive a lower bound for the non deterministic automaton since $(u_X, d^x c)$ and $(u_{X'}, d^x c)$ can both lead to a final state when $x \in X \cap X'$. Moreover, theorem 2 shows that a non deterministic automaton can be constructed of size exponential only in the size of the alphabet. That is, since the alphabet is fixed for this example, one could write a non-deterministic automaton accepting $[L_p \cdot \Sigma^\star]$ polynomial in the size of L_p.

4.3 Series-Parallel Alphabets

We want to show now that as soon as the dependencies are not too complicated, the complexity of the membership problem is PTIME. The trace alphabets we consider here are the so-called series-parallel alphabets, that is alphabets built from processes using serial and parallel composition.

The serial composition $(\Sigma_1, I_1) \cdot (\Sigma_2, I_2)$ two trace alphabets (Σ_1, I_1), (Σ_2, I_2) is $(\Sigma_1 \cup \Sigma_2, I)$ where $(a,b) \in I$ exactly when $(a,b) \in I_i$ for $i = 1, 2$.

The parallel composition $(\Sigma_1, I_1) \mathbin{/\!\!/} (\Sigma_2, I_2)$ of two trace alphabets (Σ_1, I_1), (Σ_2, I_2) is $(\Sigma_1 \cup \Sigma_2, I)$ where $(a,b) \in I$ exactly when $(a,b) \in I_i$ for $i = 1, 2$ or $a \in \Sigma_j, b \in \Sigma_k$ with $j \neq k$.

That is, a series-parallel alphabet is built with the grammar $\Sigma = a \mid \Sigma \cdot \Sigma \mid \Sigma \mathbin{/\!\!/} \Sigma$ where a is a letter (the independence relation I is entirely defined by this grammar). Every alphabet used in the grammar of Σ is called a *component alphabet*.

We assume in the following that there is a strict alternation between each level, and that it begins by a parallel composition. That is if $\Sigma = \Sigma_1 \mathbin{/\!/} \Sigma_2$ is parallel, then both Σ_1 and Σ_2 are serial. If it is not the case, then we just need to include some empty alphabets, at most doubling the number of components.

A characterization of series-parallel alphabets is that the graph of the relation (Σ, I) is N-free. More precisely, a trace alphabet (Σ, I) is series-parallel exactly when there do not exist four letters a, b, c, d such that the only commutation is $(b, d) \in I$. The size $\Sigma|$of a series-parallel alphabet is its number of component.

Note that expressing communication by messages is not possible with such alphabets, since calling the send a and the receive c would yield an N provided there are other events on the same process as a and c. However, we can express for instance a hierarchical decision structure. That is, let us take a tree T. All workers on the same level are mutually independent. Whenever one worker talks, all its descendants stop to listen to him (that is, they are dependent on his actions). Notice also that the alphabet used in the section "Lower Bound" is series-parallel, that is, computing a trace-closed deterministic automaton accepting $[L\mathbb{M}]$ has no chance to lead to a polynomial algorithm, even in the series-parallel case.

The idea is that in case of series-parallel alphabet, we can choose a special execution of the trace, as was suggested in [20] for another problem. We then build a (non trace-closed) automaton accepting this particular execution if and only if $w \in [\mathcal{L}(\mathcal{A})\mathcal{L}(\mathcal{B})]$. This automaton is polynomial in the size of the automata $\mathcal{A}, \mathcal{B}$, that are both trace-closed. The execution that we choose is the one which follows the lexicographic normal form (LNF), see [9]. We define a total order $\prec$ on Σ, such that if $\Sigma_1 \mathbin{/\!/} \Sigma_2$ is a component of Σ, either $\Sigma_1 \prec \Sigma_2$ or $\Sigma_2 \prec \Sigma_1$, that is every letter from an alphabet is before any letter from the other alphabet. We extend it by the following recursive definition to a relation between words, with $x_1 \cdots x_n \prec y_1 \cdots y_n$ exactly when $x_1 \prec y_1$, or $x_1 = y_1$ and $x_2 \cdots x_n \prec y_2 \cdots y_n$. Note that $\prec$ is a total order restricted to equivalent words. A word $x_1 \cdots x_n$ is in LNF exactly when it is minimal among its set of equivalent words. Then LNF(w) denotes the unique word in LNF that is trace-equiavlent to w.

4.4 Automaton Construction for Series Parallel Alphabets

Instead of checking that $w \in [\mathcal{L}(\mathcal{A})\Sigma^\star]$, we will check whether $w \in [\mathcal{L}(\mathcal{A})\mathcal{L}(\mathcal{B})]$. Obviously, the latter is more general than the former since we can choose $\mathcal{B}$ an trace closed deterministic automaton recognizing $\Sigma^\star$ with $|\mathcal{B}| = 1$.

We build a (non-deterministic) automaton accepting LNF(w) iff $w \in [\mathcal{L}(\mathcal{A})\mathcal{L}(\mathcal{B})]$ in a recursive way, following the series-parallel structure. The base states of this automaton are pairs (v, w) for every state v of $\mathcal{A}$ and w of $\mathcal{B}$. The global idea is as follows. Suppose that $x, y \in \Sigma^\star$. Then the automaton we construct for Σ has a path from (v, w) to (v', w') labeled by LNF(xy) iff there is a path from v to v' in $\mathcal{A}$ labeled by x, and one from w to w' in $\mathcal{B}$ labeled by y. Moreover, any transition comes from a transition of the asynchronous product of $\mathcal{A}, \mathcal{B}$, that is if $(v, w) \xrightarrow{a} (v', w')$, then either $w' = w$ and $v \xrightarrow{a}_{\mathcal{A}} v'$, or $v' = v$ and $w \xrightarrow{a}_{\mathcal{B}} w'$.

The basic step is easy. If Σ is fully sequential, then $\mathcal{A} \cdot \mathcal{B}$ is made of two copies of $\mathcal{A} \times \mathcal{B}$. The transitions in the first copy are $(v, w) \xrightarrow{a} (v', w')$ exactly when $a \in \Sigma$, $w' = w$ and $v \xrightarrow{a}_{\mathcal{A}} v'$. The transitions leading to the second copy (coming from either copy) are $(v, w) \xrightarrow{a} (v', w')$ exactly when $a \in \Sigma$, $v' = v$ and $w \xrightarrow{a}_{\mathcal{B}} w'$.

Assume that $\Sigma = a_1 \mathbin{/\!\!/} a_2$, and $a_1 \prec a_2$. The automaton $\mathcal{A} \mathbin{/\!\!/} \mathcal{B}$ is made of two copies of $\mathcal{A} \times \mathcal{B}$. The transitions in the first copy are labeled by $a = a_1$ and come only from the first copy. The transitions leading to the second copy are labeled by $a = a_j$ and come from either copy.

Let T^X_Σ be the number of transitions labeled by a letter of Σ in the automaton X. Then $|\mathcal{A} \cdot \mathcal{B}| \leq |\mathcal{A}| T^{\mathcal{B}}_\Sigma + |\mathcal{B}| T^{\mathcal{A}}_\Sigma$. In the same way, $|\mathcal{A} \mathbin{/\!\!/} \mathcal{B}| \leq \sum_i (|\mathcal{B}| T^{\mathcal{A}}_{a_i} + |\mathcal{A}| T^{\mathcal{B}}_{a_i}) \leq |\mathcal{A}| T^{\mathcal{B}}_\Sigma + |\mathcal{B}| T^{\mathcal{A}}_\Sigma$.

We describe now the inductive step. For an alphabet Σ with component Σ_1, Σ_2, we denote by X_i the automaton obtained recursively for Σ_i. Assume that $\Sigma = \Sigma_1 \cdot \Sigma_2$. Then we define the automaton $\mathcal{A} \cdot \mathcal{B}$ with set of states $V = V_{(\mathcal{A}, \Sigma)} \cup V_{(\mathcal{B}, \Sigma)} \cup V_{\mathcal{A}_1 /\!\!/ \mathcal{B}_1} \cup V_{\mathcal{A}_2 /\!\!/ \mathcal{B}_2}$, where V_X is the state space of automaton X. There are transitions $(v, w) \xrightarrow{a} (v', w')$ from $(\mathcal{A}, \Sigma)$ to either $(\mathcal{A}, \Sigma)$ or $(\mathcal{B}, \Sigma)$ iff $a \in \Sigma$, $w' = w$ and $v \xrightarrow{a}_{\mathcal{A}} v'$ and $(v, w) \in \mathcal{A}$. Symmetrically for transition reaching $(\mathcal{B}, \Sigma)$, except that the transitions come either from $(\mathcal{B}, \Sigma)$ or from $(\mathcal{A}, \Sigma)$. Moreover, for every pair $j \neq i$, there are transitions labeled by $a \in \Sigma_j$ from $\mathcal{A}$ to $\mathcal{A}_i \mathbin{/\!\!/} \mathcal{B}_i$, and from $\mathcal{A}_i \mathbin{/\!\!/} \mathcal{B}_i$ to $\mathcal{B}$.

If $\Sigma = \Sigma_1 \mathbin{/\!\!/} \Sigma_2$, then we define the automaton $\mathcal{A} \mathbin{/\!\!/} \mathcal{B}$ with set of states $V = V_{\mathcal{A}_1 \cdot \mathcal{B}_1} \cup V_{\mathcal{A}_2 \cdot \mathcal{B}_2}$. Let's say that $\Sigma_1 \prec \Sigma_2$. Then there are transitions labeled by $a \in \Sigma_2$ from $\mathcal{A}_1 \cdot \mathcal{B}_1$ to $\mathcal{A}_2 \cdot \mathcal{B}_2$.

The initial (resp. final) states (v, w) of these automata are those with v initial (resp. final) in $\mathcal{A}$ and w initial (resp. final) in $\mathcal{B}$.

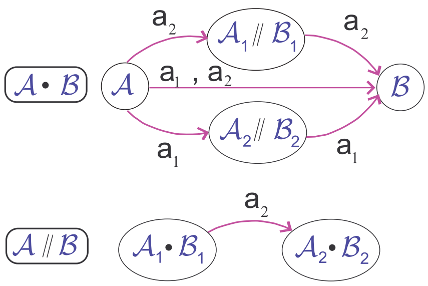

Fig. 1. Automaton construction over series-parallel alphabet.

We show that the size of $\mathcal{A} \mathbin{/\!\!/} \mathcal{B}$ remains polynomial. Since each different alphabet can increase the size of the whole system only by $2|\mathcal{A}||\mathcal{B}|$ (in the serial case, since the parallel case does not increase the size of the system), we get that the number of state of $\mathcal{A} \mathbin{/\!\!/} \mathcal{B}$ is at most $2|\Sigma| \times |\mathcal{A}| \times |\mathcal{B}|$.

Proposition 1 *Assume that Σ is a series-parallel trace alphabet. Moreover, assume that if $\mathcal{A}$ (or $\mathcal{B}$) accepts w, then it also accepts $LNF(w)$. Then $w \in [\mathcal{L}(\mathcal{A})\mathcal{L}(\mathcal{B})]$ if and only if $LNF(w) \in \mathcal{L}(\mathcal{A} \mathbin{/\!/} \mathcal{B})$.*

Notice that if the automata $\mathcal{A}$ or $\mathcal{B}$ are not trace-closed, then the problem is NP-complete. Actually, a slightly easier problem, deciding whether there exists $v \equiv_I w$ such that $v \in \mathcal{L}(\mathcal{A})$, is already NP-complete [2]. The fact is that it remains NP-complete even if the alphabet is fully parallel, in particular even if it is series-parallel.

5 Compression

Usually, when one model-checks a word, either this word comes from a very long log file (or DNA encoding), or it can be a looping run of some system: it is often very long. Having the most succinct representation for this word is then a big advantage, since it can severely decrease the runtime of the algorithm. We present here the idea of using word given in a compressed way, by means of macros, or straight-line programs. If the word is not already compressed, then the tool from [1] can be used.
Straight-line programs. A straight-line program (SLP for short) over the alphabet Σ is a context-free grammar with variables $V = \{X_1, \ldots, X_k\}$, initial variable X_1 and rules from $V \times (V^* \cup \Sigma)$. The rules are such that there is exactly one rule for each left-hand side variable, and if $X_i \longrightarrow \alpha$, then each X_j in α satisfies $j > i$.

The constraint on the rules makes that any variable X_i generates a unique word. For convenience, we denote the word generated by the variable X_i also as X_i. Without lack of generality, we can assume that rules are of size 2, that is of the form $X \longrightarrow YZ$ with $Y, Z \in (V \cup \{\epsilon\}) \cup \Sigma$. The size of an SLP is its number of variables. Lately, algorithms on SLP-compressed objects have been intensively studied [22, 15, 1, 19]. We will used two such results, namely:

Proposition 2 *Let w be an SLP, $\mathcal{A}$ an automaton and (Σ, I) a trace alphabet.*

- *The problem whether $w \in \mathcal{L}(\mathcal{A})$ is PTIME-complete [22, 15, 19], and solvable in time $\mathcal{O}(|w| \times |\mathcal{A}|^3)$.*
- *The problem whether there exists $v \equiv_I w$ with $v \in \mathcal{L}(\mathcal{A})$ is PSPACE-complete [15].*

Using the first result of proposition 2 and the theorem 2, we can easily show that:

Proposition 3 *Let w be an SLP and $\mathcal{A}, \mathcal{B}$ two trace-closed automata. The test whether $w \in [\mathcal{L}(\mathcal{A})\mathcal{L}(\mathcal{B})]$ is of complexity NP in the size of the alphabet (i.e., polynomial-time for a fixed alphabet).*

We can restate the second result of proposition 2, showing that testing whether $w \in [\mathcal{L}(\mathcal{A})\mathcal{L}(\mathcal{B})]$ is PSPACE-complete if w is an SLP and $\mathcal{A}$ or $\mathcal{B}$ is not trace-closed. With more work, we can show that the complexity remains PSPACE-complete if the alphabet is series-parallel.

The interesting question is what happens when $\mathcal{A}, \mathcal{B}$ are both trace-closed and the alphabet is series-parallel. Actually, we manage to show that the problem is PTIME-complete in this case. That is, unlike the case where $\mathcal{A}$ or $\mathcal{B}$ is not trace-closed, compression does not increase the complexity.

Theorem 3 *Let w be an SLP, Σ be a series-parallel alphabet, and $\mathcal{A}, \mathcal{B}$ be two trace-closed automata. Then testing whether $w \in [\mathcal{L}(\mathcal{A})\mathcal{L}(\mathcal{B})]$ is PTIME-complete, and solvable in time $\mathcal{O}(|w| \times (|\Sigma|^2 \times |\mathcal{A}| \times |\mathcal{B}|)^3)$.*

Proof. Let w be an SLP. We use proposition 1 to obtain an automaton C that recognizes $\mathrm{LNF}(w)$ exactly when $w \in [\mathcal{L}(A)\mathcal{L}(B)]$. As soon as we obtain a polynomial-size SLP representation of $\mathrm{LNF}(w)$, we can use proposition 2 to have a PTIME algorithm for testing whether $\mathrm{LNF}(w) \in \mathcal{L}(C)$. It is PTIME-hard using [19].

The blow-up for obtaining an SLP representation of $\mathrm{LNF}(w)$ from an SLP w is likely to be exponential in general. Anyway, we are in the special case where the alphabet is series-parallel, for which we show now how to compute a polynomial SLP for $\mathrm{LNF}(w)$.

We first need to describe the SLP variables differently than for the SLP w. For any component alphabet Σ_i and rule $X = YZ$, we define the projection (X, Σ_i) of X on Σ_i by the rules:

- $(a, \Sigma_i) = a$ if $a \in \Sigma_i$, else $(a, \Sigma_i) = \epsilon$,
- $(X, \Sigma_i) = (Y, \Sigma_i)(Z, \Sigma_i)$.

Then, for $\Sigma_i = \Sigma_j \cdot \Sigma_k$ and $X = YZ$, we describe the longest prefix and suffix in $\Sigma_j^\star$ of the projection on Σ_i of X by $\mathrm{Pref}(X, \Sigma_j), \mathrm{Suf}(X, \Sigma_j)$.

The rules associated with $\mathrm{Pref}(X, \Sigma_j)$ are as follows:

- If $a \in \Sigma_j$, then $\mathrm{Pref}(a, \Sigma_j) = a$, else $\mathrm{Pref}(a, \Sigma_j) = \epsilon$.
- If $\mathrm{Pref}(Y, \Sigma_j)$ does not contain any letter from Σ_k, then $\mathrm{Pref}(X, \Sigma_j) = (Y, \Sigma_j)\mathrm{Pref}(Z, \Sigma_j)$, else $\mathrm{Pref}(X, \Sigma_j) = \mathrm{Pref}(Y, \Sigma_j)$.

Doing so gives an SLP (w, Σ). For every rule $X = YZ$ of (w, Σ), we add the variable (XY) with the rule $(XY) = XY$. In particular, X, Y, Z can be a suffix or a prefix of variables of w. This gives an SLP of size $\mathcal{O}(2|\Sigma||w|)$.

We can now describe the SLP for $\mathrm{LNF}(w, \Sigma)$. For each component alphabet $\Sigma_i = \Sigma_j \cdot \Sigma_k$ and each variable $X = YZ$ of (w, Σ), we introduce new variables $\mathrm{Fact}(Y, \Sigma_h, \Sigma_l)$ for $h, l \in \{j, k\}$ representing the factor of $\mathrm{LNF}(Y, \Sigma_i)$ obtained by deleting the longest prefix of Y in $\Sigma_h^\star$ and the longest suffix of Y in $\Sigma_l^\star$. In particular, $\mathrm{Fact}(Y, \emptyset, \Sigma_j) LNF(\mathrm{Suf}(Y, \Sigma_j), \Sigma_j) = LNF(Y, \Sigma_i)$.

Let $X = YZ$, $\Sigma_i = \Sigma_j \cdot \Sigma_k$ and $h, l, t \in \{j, k\}$. The rule associated with $\mathrm{Fact}(X, \Sigma_h, \Sigma_l)$ is (for $\mathrm{Suf}(Y, \Sigma_t) \neq \epsilon$):

$$\mathrm{Fact}(X, \Sigma_h, \Sigma_l) = \mathrm{Fact}(Y, \Sigma_h, \Sigma_t)\mathrm{LNF}((\mathrm{Suf}(Y, \Sigma_t)(\mathrm{Pref}(Z, \Sigma_t)), \Sigma_t)\mathrm{Fact}(Z, \Sigma_t, \Sigma_l)$$

The rules associated with $\mathrm{LNF}(X, \Sigma_i)$ are

- $\mathrm{LNF}(a, \Sigma_i) = a$ if $a \in \Sigma_i$, else $\mathrm{LNF}(a, \Sigma_i) = \epsilon$,

- If $\Sigma_i = \Sigma_j \mathbin{/\!/} \Sigma_k$ and $\Sigma_j \prec \Sigma_k$, then $\mathrm{LNF}(X, \Sigma_i) = \mathrm{LNF}(X, \Sigma_j)\mathrm{LNF}(X, \Sigma_k)$,
- If $\Sigma_i = \Sigma_j \cdot \Sigma_k$ and $\mathrm{Suf}(Y, \Sigma_j) \neq \epsilon$, then $\mathrm{LNF}(X, \Sigma_i) = \mathrm{Fact}(Y, \emptyset, \Sigma_j)$ $\mathrm{LNF}(\mathrm{Suf}(Y, \Sigma_j)\mathrm{Pref}(Z, \Sigma_j), \Sigma_j)\mathrm{Fact}(Z, \Sigma_j, \emptyset)$.

Notice that it might be the case that we use recursively $\mathrm{Suf}(\mathrm{Suf}(Y, \Sigma_l), \Sigma_j)$, because of the additional variables for prefixes and suffixes (or a prefix of a prefix, but neither a prefix of a suffix nor a suffix of a prefix). This is not a variable of (w, Σ). However, we can express $\mathrm{Suf}(\mathrm{Suf}(Y, \Sigma_l), \Sigma_j)$ with only one suffix. Assume that $\Sigma_i = \Sigma_j \cdot \Sigma_k$. If $\mathrm{Suf}(Y, \Sigma_l)$ contains a letter of Σ_k, then $\mathrm{Suf}(\mathrm{Suf}(Y, \Sigma_l), \Sigma_j) = \mathrm{Suf}(Y, \Sigma_j)$. Else, $\mathrm{Suf}(\mathrm{Suf}(Y, \Sigma_l), \Sigma_j) = \mathrm{Suf}(Y, \Sigma_l)$.

It is easy to show by induction that $\mathrm{LNF}(w, \Sigma)$ computes exactly the lexicographic normal form of the projection of w on Σ. Moreover, the size of the SLP $\mathrm{LNF}(w, \Sigma)$ is at most $\mathcal{O}(|\Sigma|^3|w|)$.

Since the time complexity of checking whether an SLP S belongs to $\mathcal{L}(A)$ is $\mathcal{O}(|S| \cdot |\mathcal{A}|^3)$, we get a complexity of $\mathcal{O}(|w| \times (|\Sigma|^2 \times |\mathcal{A}| \times |\mathcal{B}|)^3)$. $\square$

6 A PSPACE-complete Fragment of SLTL

We define now the 'negative' fragment of SLTL. Let us look at the usual normal form of LTL, that is when only the expression $\{p, [p]\}$ can use negation (negation is pushed the deepest possible). Then we say that a formula φ is a *negative* formula of SLTL if in the normal form of $\neg\varphi$, the negation is used only over Boolean combinations. That is, the snapshot expressions $[p]$ only appear in a positive form (in the negation of the formula). Note that negation may appear inside the '[ ]' operator. For instance, the property $\varphi = \Box\neg[p] \vee \Box\neg[q]$ is a negative formula of SLTL since if it holds exactly when $\neg\varphi = (\Diamond[p] \wedge \Diamond[q])$ *does not* hold.

Notice that every LTL formula is a negative formula of SLTL since it does not use $[p]$. Hence, the model checking of negative SLTL formula is PSPACE-hard.

We show now how to do model-checking in PSPACE for such a formula. We use the LTL translation [14], except that subformulas of the form $[p]$ are kept as a whole (as in the construction for Theorem 1). This construction does not introduce new negations to propositional letters or snapshot subsexpressions (as opposed e.g., to the construction in [27]). We know that there exists an automaton $\mathcal{B}_{\neg\varphi}$ accepting $\mathcal{L}(\neg\varphi)$, labeled by $\{p, \neg p\}$ and by positive $\{[p]\}$, whose size is exponential in $|\varphi|$.

For each subformula $[p]$ we check on a separate automaton copy, computing the snapshot (traces prefixes) of $\mathcal{S}$, that is $[\mathcal{L}(\mathcal{S})\Sigma^\star]$, as constructed in Section 4.1. Note however that for two different $[p], [q]$, the prefix need not be the same, and thus we need two different copies of the trace prefix automaton. Notice that once $[p]$ holds, it holds forever, so we need not to have several copies for the same subformula $[p]$.

For every subformula $[p]$ (there are at most $|\varphi|$ such propositions), we create a copy $\mathcal{S}_p$ of the automaton $\mathcal{S}$ describing the system. There exists an automaton $\mathcal{A}_p$ of size exponential only in $|\Sigma|$ accepting $[\mathcal{L}(\mathcal{S})\Sigma^\star]$ (see section 4.1). We just need to check whether there exists accepting paths $\rho_{\neg\varphi}, \rho, \rho_{p_1}, \cdots, \rho_{p_n}$ of $\mathcal{A}_{\neg\varphi}, S, \mathcal{A}_{p_1}, \cdots, \mathcal{A}_{p_n}$ labeled by the same word w (in the different copies), such that for every prefix u of w, the state $v_{\neg\varphi}, v, v_{p_1}, \cdots, v_{p_n}$ reached on u satisfy: if $v_{\neg\varphi} \models (\neg)p$, then $v \models (\neg)p$ and if $v_{\neg\varphi} \models [p]$, then $v_p \models p$. Hence,

Theorem 4 *Let $\mathcal{S}$ be an automaton describing the system, and φ be a negative formula of SLTL. Then the model-checking of $\mathcal{S} \models \varphi$ is PSPACE-complete, with a space complexity of $\mathcal{O}(\log(|\mathcal{S}|) \times |\varphi| \times |\Sigma|)$.*

7 Conclusion

We described in this paper a new Linear Temporal Logic, the Snapshot LTL, which captures some properties of global logics on traces without the inherently high complexity. We proposed an EXPSPACE algorithm to do model-checking against this logic, based on a deterministic automaton construction. It would be interesting to compare the properties expressed by our logic with the one expressed by the EXPSPACE-complete fragment of LTrL [25] and of ISTL [21]. Moreover, the 'negative' fragment of SLTL is PSPACE-complete, yet more general than LTL.

We already considered model-checking for snapshots of a word, which is not easy to tackle either. For instance, the precise complexity is still unknown (but neither is known the precise complexity of model-checking a word [19]).

$w \in [\mathcal{L}(A)\Sigma^\star]$	$\mathcal{A}$ trace-closed	$\mathcal{A}$
Normal case	NP	NP-complete
w compressed	NP	PSPACE-complete
Series-parallel	$2\|\Sigma\| \times \|\mathcal{A}\|$	NP-complete
Series-parallel + compression	PTIME-complete	PSPACE-complete

Fig. 2. Complexity of the snapshot verification of a word.

We studied the complexity when we vary slightly the problem (see figure 2), to understand the limit of our algorithms. We show that the problem is quickly in polynomial time, for instance when the alphabet is not too complicated (series-parallel). Moreover, we show that the algorithms we proposed were pretty robust, since the complexity remains the same even in case that we use compressed words. On series-parallel alphabet and using compression, we succeeded in showing a PTIME-algorithm, which is in strong contrast with the PSPSACE-complete complexity as soon as $\mathcal{A}$ is not trace closed.

This work shows that pattern matching a trace is PTIME in $|\Sigma|$ and NLOG in the size of the word in case the alphabet is series parallel, unlike in the general case. In this case, it also simplifies greatly the pattern matching for compressed traces [15].

Notice that this problem is somehow not complicated enough to have an NP-complete result: the problem whether $w \in [\mathcal{L}(\mathcal{A}_1) \cdots \mathcal{L}(\mathcal{A}_n)]$ is NP-complete.

Acknowledgement: We would like to thanks to Christof Löding and Lenore Zuck for fruitful discussion.

References

1. R. Alur, S. Chaudhuri, K. Etessami, S. Guha, and M. Yannakakis. Compression of partially ordered strings. In *CONCUR 2003*, LNCS 2761 pp. 42-56.
2. R. Alur, K. Etessami, and M. Yannakakis. Realizability and verification of MSC graphs. In *ICALP'01*, LNCS 2076, pp.797-808.

3. Rajeev Alur, Thomas A. Henzinger, Orna Kupferman. Alternating-time temporal logic. JACM 49(5): 672-713 (2002).
4. Rajeev Alur, Ken McMillan, Doron Peled. Deciding Global Partial-Order Properties. In *ICALP 1998*, LNCS 1443, 41–52.
5. Rajeev Alur, Doron Peled, Wojciech Penczek. Model-Checking of Causality Properties. In *LICS 1995*, pp. 90-100.
6. K. M. Chandy, L. Lamport, Distributed Snapshots: determining the global state of distributed systems, ACM Transactions on Computer Systems 3 (1985), 63–75.
7. V. Diekert and P. Gastin. Local Temporal Logic is Expressively Complete for Cograph Dependence Alphabets. In *LPAR 2001*, LNAI 2250, p. 55-69.
8. V. Diekert and P. Gastin. Pure future local temporal logics are expressively complete for Mazurkiewicz traces. In *LATIN 2004*, LNCS 2976, pp. 232-241.
9. V. Diekert and G. Rozenberg. *The Book of Traces.* World Scientific, Singapore, 1995.
10. E. A. Emerson and C. S. Jutla. The complexity of tree automata and logics of programs. In *29th FOCS* 1988.
11. V.K. Garg, B. Waldecker, Detecting of Weak Unstable Predicates in Distributed Programs, IEEE Transactions on Parallel and Distributed Systems, 5(3), 299–307, 1994.
12. P. Gastin and D. Kuske. Satisfiability and Model Checking for MSO-definable Temporal Logics are in PSPACE. In *CONCUR 2003*, LNCS 2761, pp. 222-236.
13. P. Gastin and M. Mukund. An Elementary Expressively Complete Temporal Logic for Mazurkiewicz Traces. In *ICALP 2002*, LNCS 2380, pp. 938-949.
14. R. Gerth, D. Peled, M. Vardi, P. Wolper, Simple on-the-fly automatic verification of linear temporal logic, PSTV 1995, 3–18.
15. B. Genest, A. Muscholl. Pattern Matching and Membership for Hierarchical Message Sequence Charts. In *LATIN 2002*, LNCS 2286, pp. 326-340.
16. Doron Peled, Amir Pnueli. Proving Partial Order Liveness Properties. In *ICALP 1990*, pp. 553-571.
17. D. Kuske. Infinite series-parallel pomsets: logic and languages. In *ICALP 2000*, LNCS 1853, pp. 648-662.
18. A. Mazurkiewicz, Trace semantics In Proceedings of Advances in Petri Nets 1986, Bad Honnef, LNCS 255, pp. 279–324, 1987.
19. N. Markey and Ph. Schnoebelen Model Checking a Path *Concur 2003*, LNCS 2761, pp. 251-265.
20. D. Peled. Specification and verification of Message Sequence Charts. In *FORTE/PSTV 2000*, pp.139-154.
21. D. Peled, A. Pnueli. Proving Partial Order Properties. Theoretical Computer Science, 126:143–182, 1994.
22. W. Plandowski and W. Rytter. Complexity of Language recognition problems for compressed words. In Jewel are forever, Springer, pp. 262–272, 1999.
23. S. Stoller, Y.A. Liu, Efficient Symbolic Detection of Global Properties in Distributed Systems, CAV 1998, LNCS 1427, 357–368.
24. I. Walukiewicz. Difficult Configurations – On the Complexity of LTrL. In *ICALP 1998*, LNCS 1443, 140–151.
25. P.S. Thiagarajan, I. Walukiewicz. An Expressively Complete Linear Time Temporal Logic for Mazurkiewicz Traces. *IC* 179(2): 230-249 (2002)
26. W. Thomas. On the Ehrenfeucht-Fraïssè game in theoretical computer science. LNCS 668, 1993.
27. M.Y. Vardi, P. Wolper. Reasoning about infinite computations. Information and Computation, 115(1994), 1–37.