Snapshot Verification

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

[1]Department of Computer Science, University of Warwick,
Coventry CV4 7AL, United Kingdom
[2]Institut für Informatik, Universität Leipzig,
D-04009 Leipzig, Germany
[3]LIAFA, Université Paris 7, 2 place Jussieu,
75251 Paris Cedex 05, France

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 that 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. As these equivalent sequences cannot be distinguished by an observer not capable of monitoring instantaneously concurrent processes, it is unnatural to distinguish between them. A specification that permits some interleaving sequence but forbids another equivalent one is possibly ambiguous. Local temporal logics like TLC [5], LocTL [8] and all MSO-definable temporal logics [12], do not distinguish equivalent sequences and allow model-checking in PSPACE. But expressing global properties of the system is notoriously hard in most of these formalisms. Alternatively, one can use global temporal logics that are tailored

N. Halbwachs and L. Zuck (Eds.): TACAS 2005, LNCS 3440, pp. 510–525, 2005.
© Springer-Verlag Berlin Heidelberg 2005

to express global properties. The inherent problem of these logics is the very high computational complexity (e.g., EXSPACE-complete for the *UNTIL*-free fragment of ISTL [4] and non-elementary for LTrL [24, 26]).

In this paper, we consider a global temporal logic whose capability to talk about partial-order properties is restricted to expressing elementary properties of snapshots.

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 a (possible) past of $[v]$. 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, and has the same complexity as the model-checking of LTL otherwise. We further identify a fragment of the logic which is PSPACE-complete only in the size of the formula, extending the model-checking algorithm 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 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. We later use a construction for the word problem for giving a 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 variations of the problem, namely relaxing the dependencies and considering very long words. A case where the dependencies are not too complicated, is when the trace alphabet is series-parallel [9], that is, built on serial and parallel composition of letters. This kind of alphabets is often used to facilitate algorithms [7, 17]. To produce more efficient algorithms on very long words, we follow several papers [15, 19, 22] that con-

sider that the word is given in a compressed way, by means of Straight Line Programs [22].

Related works deal with checking whether snapshots of an execution of a distributed system satisfy a given propositional predicate. 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 an irreflexive and symmetric relation $I \subseteq \Sigma \times \Sigma$. The pair (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 $(a, b) \in I$, $u = w_1 abw_2$ and $v = w_1 baw_2$, 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.

We next want to extend this equivalence to infinite words. Denote by $u \prec v$ the fact that u is a finite prefix of v. For two infinite words $w_1, w_2 \in \Sigma^\omega$ over Σ, we write $w_1 \equiv^{\lim} w_2$ iff

- 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 $uv' \equiv v$, 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 $uv' \equiv v$.

Since no confusion can arise, we abbreviate $w_1 \equiv^{\lim} w_2$ by $w_1 \equiv w_2$, i.e., we consider the *trace equivalence* $\equiv$ as an equivalence relation on the set of finite and infinite words. A *trace* is an equivalence class w.r.t. $\equiv$. It is usually denoted 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. Note that $u \equiv u'$ and $v \equiv v'$ imply $uv \equiv u'v'$ (for u, u' finite and v, v' possibly infinite). Thus, we can define a concatenation of traces simply by $[u][v] = [uv]$ for a finite word u and a finite or infinite word v. A trace $[u]$ *subsumes* $[v]$, denoted $[v] \sqsubseteq [u]$ if there exists some v' such that $u \equiv vv'$ (equivalently, if $[u] = [v][v']$). For a language L of finite and infinite words we write $[L]$ for the set $\{u \mid u \equiv v, v \in L\}$.

A (labeled) *transition system over* Σ is a tuple $\mathcal{A} = (S, E, \iota, \Sigma)$ with set of states S, transitions $E \subseteq S \times \Sigma \times S$, and initial state $\iota \in S$. An *automaton* is a transition system extended by a set of accepting states F. It is *trace-closed* if $[\mathcal{L}(\mathcal{A})] = \mathcal{L}(\mathcal{A})$, i.e., if its language is closed under the trace equivalence $\equiv$.

3 Snapshot Linear Temporal Logic

3.1 Syntax, Semantics, and Motivation

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 and $Bool(\mathcal{P})$ be the set of Boolean combinations of propositions over $\mathcal{P}$.

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

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.

A *Kripke structure* $\mathcal{S} = (S, E, \iota, \Sigma, val)$ is a deterministic transition system (S, E, ι, Σ) together with a valuation function $val : S \to 2^{\mathcal{P}}$ assigning to a state s those atomic propositions that hold in this state. We now fix a Kripke structure $\mathcal{S}$. For a word $w \in \Sigma^*$, let $state(\iota, w)$ denote the unique state that is obtained by applying the actions of w to the initial state ι. The interpretation of SLTL-formulas is defined over a pair of sequences $u \in \Sigma^\star$ and $v \in \Sigma^\omega$.

- $(u, v) \models p$ iff $p \in val(state(\iota, u))$ for $p \in \mathcal{P}$.
- $(u, v) \models [p]$ iff there exists a sequence $u' \in \Sigma^\star$ such that $[u'] \sqsubseteq [u]$ and $state(\iota, u') \models p$ for $p \in Bool(\mathcal{P})$ (according to propositional logic).
- $(u, v) \models \neg\varphi$ iff $(u, v) \not\models \varphi$.
- $(u, v) \models \varphi \vee \psi$ iff $(u, v) \models \varphi$ or $(u, v) \models \psi$.
- $(u, v) \models \bigcirc\varphi$ iff $(ua, v') \models \varphi$ where $a \in \Sigma$ and $v' \in \Sigma^\omega$ with $v = av'$.
- $(u, v) \models \varphi\mathcal{U}\psi$ iff we can write $v = wv'$ such that $(uw, v') \models \psi$ and for any decomposition $w = w_1 w_2$ where w_2 is nonempty, $(uw_1, w_2 v) \models \varphi$.

Based on these temporal operators, we can define (as usual) several other ones. 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 $\Diamond\varphi = true\mathcal{U}\varphi$.

It is not hard to verify that the following are tautologies involving the new snapshot operator:

$$\Box([p] \to \Box[p]) \qquad \Box(p \to [p]) \qquad \Box(([p \wedge q]) \to [p] \wedge [q])$$
$$\Box(([p] \vee [q]) \leftrightarrow [p \vee q]) \qquad \Box(\neg[p] \to [\neg p])$$

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 closeness 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 starts to hold 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 one snapshot satisfying p suffices. Therefore, we can add an appropriate construct to capture such a property.

3.2 Model-Checking SLTL

In this section, we outline an algorithm that decides whether an SLTL-formula φ holds true for all words in a given Kripke structure $\mathcal{S} = (S, E, \iota, \Sigma, val)$. The idea is to construct a second Kripke structure $\mathcal{B}$ that includes the 'memory' which is required for deciding snapshot properties. While $\mathcal{P}$ is the set of atomic propositions of $\mathcal{S}$, we allow in $\mathcal{B}$ atomic propositions of the form p for $p \in \mathcal{P}$ as well as $[p]$ for $p \in Bool(\mathcal{P})$.

- The state set $S^{\mathcal{B}}$ equals $2^{S \times 2^{\Sigma}} \times S$. For $s = (\mathcal{X}, t) \in S^{\mathcal{B}}$, we write $current(s) = t \in S$ and $past(s) = \{s' \in S \mid \exists A \subseteq \Sigma : (s', A) \in \mathcal{X}\}$.
- The valuation function $val^{\mathcal{B}}$ is given by $val^{\mathcal{B}}(\mathcal{X}, s) = val(s) \cup \{[p] \mid \exists t \in past(\mathcal{X}, s) : t \models p\}$.
- 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 $(\{(\iota, \emptyset)\}, \iota)$

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 A of letters (but not the actual sequence) that belong to the difference between the actual sequence and the subsumed one. Let $s \xrightarrow{a} r$ 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 a to A and remain in state t, obtaining $\langle t, A \cup \{a\}\rangle$. This is because the set of subsumed traces is just extended, and if $t \xrightarrow{v} s$ then $t \xrightarrow{va} r$. Another pair is formed when a is independent of all letters in A. In this case we can 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 a is independent of the letters in v') then $[va]$ is a prefix of $[ua]$ and $t \xrightarrow{v'} r$.

A model-checking algorithm for SLTL uses the structure $\mathcal{B}$ instead of $\mathcal{S}$. Let φ be some SLTL-formula whose validity over $\mathcal{S}$ we want to check. Recall that the atomic propositions of $\mathcal{B}$ are of the form p for $p \in \mathcal{P}$ and $[p]$ for p a Boolean combination of elements from $\mathcal{P}$. Hence φ can be seen as a classical LTL-formula speaking about paths in the structure $\mathcal{B}$. Because of the construction of $\mathcal{B}$ from $\mathcal{S}$, we get for any infinite word v: $(\varepsilon, v) \models_{\mathcal{S}} \varphi$ (seen as SLTL-formula) iff $(\varepsilon, v) \models_{\mathcal{B}} \varphi$ (seen as LTL-formula). Now the well-known model-checking algorithm (i.e., translating the LTL-formula $\neg\varphi$ into an automaton and checking emptiness of the intersection of this automaton and $\mathcal{B}$) yields the following result

Theorem 1. Let $\mathcal{S}$ be a Kripke structure describing the system, and φ be an SLTL-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 concurrency alphabet (Σ, 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 fulfills 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 Non deterministic Construction for the Membership Problem

Let $I \subseteq \Sigma^2$ be an independence relation and $\mathcal{A} = (S, E, \iota, \Sigma, F)$ be a trace-closed automaton.

For checking emptiness or inclusion of a word $w \equiv xy$ in the language $[\mathcal{L}(\mathcal{A})\Sigma^\star]$ with $x \in \mathcal{L}(\mathcal{A})$, we can make the following construction. The idea is to guess the set of letters of the suffix of x that can still appear (as opposed to the previous automaton which kept what is used before from any point, hence had to keep many histories). Thus, the set of states is $S \times 2^\Sigma$ and a state $\langle s, A\rangle$ is accepting iff $s \in F$. The initial state is $\langle \iota, \Sigma\rangle$. Given a letter $a \in \Sigma$, there is a transition from state $\langle s, A\rangle$ labeled by a to $\langle s, B\rangle$, where $B \subseteq A$ excludes any letter from A that depends on a. If $a \in A$, we continue from a state $\langle s, A\rangle$ according to the transition relation of $\mathcal{A}$ to a state $\langle t, A\rangle$.

Formally, we define an automaton $\mathcal{D}$ with the following transitions:

- $\langle s, A\rangle \xrightarrow{a} \langle s, B\rangle$, when $B = A \setminus \{b \in A \mid (b, a) \in D\}$.
- $\langle s, A\rangle \xrightarrow{a} \langle t, A\rangle$, when $a \in A$ and $s \xrightarrow{a} t \in E$.

Basically, $\mathcal{D}$ is built on $2^{|\Sigma|}$ copies of $\mathcal{A}$. 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 concurrency 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 in the Deterministic Case

In this section, we show that the minimal deterministic automaton accepting $[L\Sigma^*]$ is exponential in the size of the automaton accepting L. To this aim, let $\Sigma = \{a, b, c, d\}$ with $I = \{a, b\} \times \{c, d\}$. We consider, for $p \in \mathbb{N}$, the language $L_p = [\{uavc \mid u \in \{a, b\}^*, v \in \{c, d\}^*, |u| \equiv |v| \mod p\}]$. Because of the special form of the independence relation I, a word $w \in \Sigma^*$ belongs to L_p iff its projection to $\{a, b\}$ ends with an a, its projection to $\{c, d\}$ ends with a c, and these two projections have the same length modulo p. Thus, in order to accept L_p, we need to count modulo p the occurrences of letters from $\{a, b\}$ and remember the last one of them, and similarly for $\{c, d\}$. Thus, we need $4 \cdot p^2$ many states.

Now let $u_1, \cdots, u_n \in \{a, b\}$ and $v_1, \cdots, v_m \in \{c, d\}$. Then the words $u_1 \ldots u_n$ $v_1 \ldots v_m$ and $u_1 \ldots u_i\, v_1 \ldots v_j\, u_{i+1} \ldots u_n\, v_{j+1} \ldots v_m$ are equivalent for any i and j. Hence the former belongs to $[L_p\, \Sigma^*]$ iff there are $1 \leq i \leq n$ and $1 \leq j \leq m$ with $i = j \pmod p$, $u_i = a$ and $v_j = c$.

We want to show that in order to accept $[L_p\, \Sigma^\star]$ with a deterministic automaton, we need exponentially many states (exponential in p): For a set $X =$

$\{x_1, x_2, \ldots, x_n\} \subseteq \{0, 1, \ldots, p-1\}$ let $u_X = (b^{x_1} a b^{p-x_1-1})(b^{x_2} a b^{p-x_2-1}) \ldots (b^{x_n} a b^{p-x_n-1})$. Then, by the observation in the previous paragraph, for $m < p$ we have $u_X d^m c \in [L_p \cdot \Sigma^\star] \iff \exists i : m = x_i \pmod p \iff m \in X$.

Now suppose $\mathcal{A}$ is a deterministic automaton accepting $[L_p\, \Sigma^*]$ and let ι be its initial state. Furthermore, let X and X' be two distinct subsets of $\{0, 1, \ldots, p-1\}$ and suppose that they both lead to the same state when executed in the initial state of $\mathcal{A}$. Then there is (without loss of generality) $x \in S \setminus X'$. Since u_X and $u_{X'}$ lead to the same state, so do $u_X\, d^x c$ and $u_{X'}\, d^x c$. Since $u_X\, d^x c \in [L_p\, \Sigma^*]$ this state is accepting, implying that $u_{X'}\, d^x c$ is accepted by $\mathcal{A}$. Since this contradicts our assumption on $\mathcal{A}$ to accept $[L_p\, \Sigma^*]$, two distinct words of the form u_X cannot lead to the same state of $\mathcal{A}$ when executed in the initial state ι. Since there are exponentially many words u_X, the automaton $\mathcal{A}$ has exponentially many states.

Note that, in contrast to the exponential lower bound for a deterministic automaton, Theorem 2 gives a polynomial non deterministic automaton accepting $[L_p\, \Sigma^*]$ (since the alphabet is fixed).

5 A PSPACE-Complete Fragment of SLTL

We define now the 'negative' fragment of SLTL, whose model-checking exploits the model-checking of snapshots of a word (see section 4). 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 $\neg\varphi = (\Diamond[p] \wedge \Diamond[q])$.

Notice that every LTL formula is a negative formula of SLTL since it does not use $[p]$. Hence, model-checking of negative SLTL formulas is already 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 subexpressions (as opposed e.g., to the construction in [25]). 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 $|\varphi|$.

We check each subformula $[p]$ on a separate automaton copy, computing the snapshots (trace 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 prefixes 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 have several copies for the same subformula $[p]$.

For every subformula $[p]$ (there are at most $|\varphi|$ such propositions), we create an automaton $\mathcal{S}_p$ from the Kripke structure $\mathcal{S}$ as follows: states, initial states,

and transitions are those from $\mathcal{S}$ and a state is accepting iff it satisfies p; i.e., $\mathcal{S}_p$ accepts all those words that lead to a state in $\mathcal{S}$ satisfying p. There exists an automaton $\mathcal{A}_p$ of size exponential only in $|\Sigma|$ accepting $[\mathcal{L}(\mathcal{S}_p)\Sigma^\star]$ (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 3. Let $\mathcal{S}$ be a Kripke structure 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 $\mathcal{O}(\log(|\mathcal{S}|) \cdot |\varphi| \cdot |\Sigma|)$.

6 Efficient Model-Checking of a Word

Since model-checking snapshots of a word is important (see section 4 and 5), we propose here some variations to improve its efficiency.

6.1 Series-Parallel Alphabets

We show that the membership problem in $[\mathcal{L}(\mathcal{A})\Sigma^*]$ is in PTIME provided the independence alphabet is series-parallel (see also [7, 17] for algorithms on series-parallel alphabet). Actually, we consider the more general case of deciding membership in $[\mathcal{L}(\mathcal{A})\mathcal{L}(\mathcal{B})]$ in polynomial time provided $\mathcal{A}$ and $\mathcal{B}$ are trace-closed automata over a series-parallel independence alphabet (Σ, I).

In this section, we consider independence alphabets together with a chosen total order on the letters. Let $(\Sigma_1, I_1, \leq_1)$ and $(\Sigma_2, I_2, \leq_2)$ be disjoint independence alphabets where $\leq_1$ and $\leq_2$ are linear orders on Σ_1 and Σ_2, resp. A linear order $\leq$ is defined on $\Sigma_1 \cup \Sigma_2$ by $a \leq b$ iff $a \leq_1 b$ or $a \leq_2 b$ or $a \in \Sigma_1$ and $b \in \Sigma_2$. Then the serial composition $(\Sigma_1, I_1, \leq_1) \cdot (\Sigma_2, I_2, \leq_2)$ is $(\Sigma_1 \cup \Sigma_2, I_1 \cup I_2, \leq)$. The parallel composition $(\Sigma_1, I_1, \leq_1) \mathbin{/\!\!/} (\Sigma_2, I_2, \leq_2)$ is defined to be $(\Sigma_1 \cup \Sigma_2, I_1 \cup I_2 \cup (\Sigma_1 \times \Sigma_2) \cup (\Sigma_2 \times \Sigma_1), \leq)$. A *series-parallel independence alphabet* is a tuple $(\Sigma, I, \leq)$ that can be constructed from ordered independence alphabets of the form $(\{\alpha\}, \emptyset, \leq)$. A *component* of $(\Sigma, I, \leq)$ is a set $\Gamma \subseteq \Sigma$ that occurs in this inductive construction. Note that any series-parallel independence alphabet has at most $|\Sigma|$ many components.

The linear order $\leq$ on Σ can be extended to words setting $x_1 x_2 \ldots x_m \leq y_1 y_2 \ldots y_n$ if $m < n$ or $m = n$ and $x_1 < y_1$ or $m = n$, $x_1 = y_1$ and $x_2 x_3 \ldots x_m \leq y_2 y_3 \ldots y_n$. Since this length-lexicographic order is a well order on Σ^*, any trace (i.e., any equivalence class of words) contains a minimal element. We call the minimal element of $[u]$ the lexicographic normal form $\mathrm{LNF}(u)$ of u.

From now on, we fix some series-parallel independence alphabet $(\Sigma, I, \leq)$ and two trace-closed automata $\mathcal{A} = (S^{\mathcal{A}}, E^{\mathcal{A}}, \iota^{\mathcal{A}}, \Sigma, F^{\mathcal{A}})$ and $\mathcal{B} = (S^{\mathcal{B}}, E^{\mathcal{B}}, \iota^{\mathcal{B}}, \Sigma, F^{\mathcal{B}})$. We will construct an automaton $\mathcal{C}(\Sigma)$ with states of the form (s, t, Γ, X) where $s \in S^{\mathcal{A}}$, $t \in S^{\mathcal{B}}$, $\Gamma \subseteq \Sigma$ is a component of $(\Sigma, I, \leq)$, and X is A or B with the following property:

Let $\Gamma \subseteq \Sigma$ be a component and $u \in \Gamma^*$ be in lexicographic normal form. Then $(s,t,\Gamma,X) \xrightarrow{u}_{\mathcal{C}} (s',t',\Gamma,Y)$ iff there exist $u_{\mathcal{A}}, u_{\mathcal{B}} \in \Gamma^*$ with $u \equiv u_{\mathcal{A}} u_{\mathcal{B}}$, $s \xrightarrow{u_{\mathcal{A}}}_{\mathcal{A}} s'$, $t \xrightarrow{u_{\mathcal{B}}}_{\mathcal{B}} t'$, and $u_{\mathcal{B}} = \varepsilon$ provided $X = Y = A$ and $u_{\mathcal{A}} = \varepsilon$ provided $X = Y = B$.

This automaton $\mathcal{C}$ will be constructed inductively following the inductive construction of the series-parallel independence alphabet $(\Sigma, I, \leq)$, i.e., we will have automata $\mathcal{C}(\Delta)$ for any component Δ such that the above invariant holds for all components $\Gamma \subseteq \Delta$ (and $\mathcal{C}(\Delta)$ does not have transitions labeled by letters outside of Δ).

In this construction, we will use the following automata $\mathcal{A}_\Gamma$ and $\mathcal{B}_\Gamma$ for Γ a component: The set of states of $\mathcal{A}_\Gamma$ is $S^{\mathcal{A}} \times S^{\mathcal{B}} \times \{\Gamma\} \times \{A\}$. There is transition $(s,t,\Gamma,A) \xrightarrow{a}_{\mathcal{A}_\Gamma} (s',t',\Gamma,A)$ iff $a \in \Gamma$, $s \xrightarrow{a}_{\mathcal{A}} s'$ and $t = t'$. Symmetrically, the set of states of $\mathcal{B}_\Gamma$ is $S^{\mathcal{A}} \times S^{\mathcal{B}} \times \{\Gamma\} \times \{B\}$ and there is a transition $(s,t,\Gamma,B) \xrightarrow{a}_{\mathcal{B}_\Gamma} (s',t',\Gamma,B)$ in $\mathcal{B}$ iff $a \in \Gamma$, $s = s'$, and $t \xrightarrow{a}_{\mathcal{B}} t'$.

The base case is simple: if $|\Gamma| = 1$, then $\mathcal{C}(\Gamma)$ is the union of $\mathcal{A}_\Gamma$ and $\mathcal{B}_\Gamma$, plus transitions from $\mathcal{A}_\Gamma$ to $\mathcal{B}_\Gamma$. That is, the set of states of $\mathcal{C}(\Gamma)$ equals $S^{\mathcal{A}} \times S^{\mathcal{B}} \times \{\Gamma\} \times \{A, B\}$ and there is a transition $(s,t,\Gamma,X) \xrightarrow{a}_{\mathcal{C}(\Gamma)} (s',t',\Gamma,Y)$ iff $a \in \Gamma$, $s \xrightarrow{a}_{\mathcal{A}} s'$, $t = t'$, and $X = A$ or $s = s'$, $t \xrightarrow{a}_{\mathcal{B}} t'$ and $X = Y = B$.

Now suppose that Γ is a component that is built as the parallel product of the components Γ_1 and Γ_2. Then we take as $\mathcal{C}(\Gamma)$ the union of the automata $\mathcal{C}(\Gamma_1)$ and $\mathcal{C}(\Gamma_2)$ together with transitions of the form $(s,t,\Gamma_1,X) \xrightarrow{a}_{\mathcal{C}(\Gamma)} (s',t',\Gamma_2,Y)$ provided $a \in \Gamma$, $s \xrightarrow{a}_{\mathcal{A}} s'$ and $t = t'$ or $s = s'$ and $t \xrightarrow{a}_{\mathcal{B}} t'$. Note that words over Γ in lexicographic normal form belong to $\Gamma_1^* \, \Gamma_2^*$. This allows to prove the invariant for $\mathcal{C}(\Gamma)$.

Finally, let Γ be a component that is built as the serial product of the components Γ_1 and Γ_2. Then $\mathcal{C}(\Gamma)$ is the union of the automata $\mathcal{A}_\Gamma$, $\mathcal{C}(\Gamma_1)$, $\mathcal{C}(\Gamma_2)$, and $\mathcal{B}_\Gamma$ together with transitions of the form $(s,t,\Delta_1,X) \xrightarrow{a}_{\mathcal{C}(\Gamma)} (s',t',\Delta_2,Y)$ provided Δ_1, Δ_2 are components of Γ (already seen by induction) and one of the following holds

(1) (s,t,Δ_1,X) is a state of $\mathcal{A}_\Gamma$ and
   - (s',t',Δ_2,Y) is a state of $\mathcal{C}(\Gamma_1)$ or of $\mathcal{B}_\Gamma$ and $a \in \Gamma_2$, or
   - (s',t',Δ_2,Y) is a state of $\mathcal{C}(\Gamma_2)$ or of $\mathcal{B}_\Gamma$ and $a \in \Gamma_1$

(2) (s',t',Δ_2,Y) is a state of $\mathcal{B}_\Gamma$ and
   - (s,t,Δ_1,X) is a state of $\mathcal{C}(\Gamma_1)$ and $a \in \Gamma_2$, or
   - (s,t,Δ_1,X) is a state of $\mathcal{C}(\Gamma_2)$ and $a \in \Gamma_1$.

This construction is visualized in Figure 1. To prove the invariant for $\mathcal{C}(\Gamma)$, let $u \in \Gamma^*$ be in lexicographic normal form. Write u as an alternating sequence of nonempty words u_i from Γ_1^+ and of Γ_2^+. For any trace equivalent factorization $v\,w \equiv u_1 u_2 \ldots u_n$, there exists i and a trace equivalent factorization $v'\,w' \equiv u_i$ with $v \equiv u_1 u_2 \ldots u_{i-1} v'$ and $w \equiv w' u_{i+1} \ldots u_n$. If $v' = \varepsilon$ or $w' = \varepsilon$, we go directly from $\mathcal{A}_\Gamma$ to $\mathcal{B}_\Gamma$. Otherwise, we go from $\mathcal{A}_\Gamma$ to $\mathcal{B}_\Gamma$ via $\mathcal{C}(\Gamma_k)$ with $u_i \in \Gamma_k^+$.

Proposition 1. Let $(\Sigma, I, \leq)$ be a series-parallel independence alphabet. Moreover, let $\mathcal{A}$ and $\mathcal{B}$ be automata such that $\mathrm{LNF}(w)$ is accepted as soon as w is

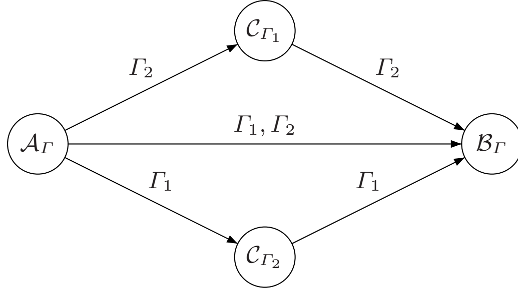

Fig. 1. Construction of $\mathcal{C}(\Gamma_1 \mathbin{/\!\!/} \Gamma_2)$

accepted for any $w \in \Sigma^*$.[1] Then $u \in [\mathcal{L}(\mathcal{A})\mathcal{L}(\mathcal{B})]$ if and only if $\mathrm{LNF}(u) \in \mathcal{L}(\mathcal{C}(\Sigma))$ for any $u \in \Sigma^*$.

Note that $\mathcal{C}(\Gamma)$ is polynomial in $\mathcal{A}$ and $\mathcal{B}$ since there are only linearly many components of $(\Sigma, I, \leq)$. Since, in addition, $\mathrm{LNF}(u)$ can be constructed in polynomial time from u, we get the following improvement of Theorem 2:

Theorem 4. Let $(\Sigma, I, \leq)$ be a series-parallel independence alphabet and $\mathcal{A}$, $\mathcal{B}$ be trace-closed automata. Then we can test in polynomial time whether $w \in [\mathcal{L}(\mathcal{A})\, \Sigma^*]$.

Notice that if the automata $\mathcal{A}$ or $\mathcal{B}$ are not trace-closed, then the membership 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.

6.2 Compression

Usually, when one model-checks a word (see section 4), 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 words compressed by means of 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 $j > i$.

[1] This holds in particular if $\mathcal{A}$ and $\mathcal{B}$ are trace-closed.

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 loss 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 $|X|$ of an SLP X is its number of variables. Lately, algorithms on SLP-compressed objects have been intensively studied [1, 15, 19, 22]. We will used two known results, namely:

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

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

Using the first part of proposition 2 and 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 5. Let w be an SLP, (Σ, I) 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 $\mathcal{O}(|w| \cdot (|\Sigma|^2 \cdot |\mathcal{A}| \cdot |\mathcal{B}|)^3)$.

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

The blow-up for obtaining an SLP representation of 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 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 $\mathrm{Pref}(X, \Sigma_j) = \mathrm{Pref}(Y, \Sigma_j)$.

The SLP (w, Σ) is defined using variables $(X, \Sigma_i), \mathrm{Pref}(X, \Sigma_i), \mathrm{Suf}(X, \Sigma_j)$. Then, 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 $\mathcal{O}(2|\Sigma| \cdot |w|)$.

We can now describe the SLP $\mathrm{LNF}(w, \Sigma)$ computing $\mathrm{LNF}(w)$. 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, $\mathrm{Fact}(Y, \emptyset, \Sigma_j)\mathrm{LNF}(\mathrm{Suf}(Y, \Sigma_j), \Sigma_j) = \mathrm{LNF}(Y, \Sigma_i)$.

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

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

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 $\Sigma_i = \Sigma_j \cdot \Sigma_k$ with $\mathrm{Suf}(Y, \Sigma_j) \neq \epsilon$, 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 in the description above, it might be the case that $\mathrm{Suf}(X, \Sigma_j)$ stands for $\mathrm{Suf}(\mathrm{Suf}(Y, \Sigma_l), \Sigma_j)$, because $X = \mathrm{Suf}(Y, \Sigma_l)$ is a variable of (w, Σ). Even though $\mathrm{Suf}(\mathrm{Suf}(Y, \Sigma_l), \Sigma_j)$ is not a variable of (w, Σ), we can express it with only one suffix, that is with variables of (w, Σ). 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, $\mathrm{Suf}(\mathrm{Suf}(Y, \Sigma_l), \Sigma_j) = \mathrm{Suf}(Y, \Sigma_l)$. The case with two (or more) nested prefixes is symmetric. Notice moreover that a prefix of a suffix or a suffix of a prefix is not possible.

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 $\mathcal{O}(|\Sigma|^3 \cdot |w|)$.

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 $\mathcal{O}(|w| \cdot (|\Sigma|^2 \cdot |\mathcal{A}| \cdot |\mathcal{B}|)^3)$. □

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 [24] and of ISTL [21]. Moreover, the 'negative' fragment of SLTL is PSPACE-complete, yet more general than LTL.

We also 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 against LTL properties [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 limits of our algorithms. We show that the time complexity becomes quickly polynomial, for instance when the alphabet is not too complex (series-parallel). Moreover, we show that the algorithms we proposed are pretty robust, since the complexity remains the same even in case where we use compressed words. On series-parallel alphabets and using compression, we obtained a PTIME-algorithm, which contrasts with the PSPACE-complete complexity as soon as $\mathcal{A}$ is not trace-closed.

Notice that this problem is somehow not complicated enough to get an NP-completeness result, since we would need for this more than a fixed number of automata. Indeed, the problem whether $w \in [\mathcal{L}(\mathcal{A}_1) \cdots \mathcal{L}(\mathcal{A}_n)]$ is NP-complete.

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

Acknowledgement. We would like to thank Christof Löding and Lenore Zuck for fruitful discussions and comments on this paper.

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 2001*, LNCS 2076, pp. 797-808.
3. R. Alur, Th. A. Henzinger, and O. Kupferman. Alternating-time Temporal Logic. JACM 49(5):672-713 (2002).
4. R. Alur, K. McMillan, and D. Peled. Deciding Global Partial-Order Properties. In *ICALP 1998*, LNCS 1443, pp. 41-52.
5. R. Alur, D. Peled, and W. 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 3:63-75 (1985).
7. V. Diekert and P. Gastin. Local Temporal Logic is Expressively Complete for Cograph Dependence Alphabets. In *LPAR 2001*, LNAI 2250, pp. 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 *FOCS 1988*.
11. V.K. Garg, B. Waldecker. Detecting 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, and P. Wolper. Simple on-the-fly Automatic Verification of Linear Temporal Logic. In *PSTV 1995*, pp. 3-18.
15. B. Genest and A. Muscholl. Pattern Matching and Membership for Hierarchical Message Sequence Charts. In *LATIN 2002*, LNCS 2286, pp. 326-340.
16. D. Peled and A. 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 *Advances in Petri Nets 1986*, LNCS 255, pp. 279-324, 1987.
19. N. Markey and Ph. Schnoebelen. Model-checking a Path. In *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. Jewels are Forever, Springer, pp. 262-272, 1999.
23. S. Stoller and Y.A. Liu. Efficient Symbolic Detection of Global Properties in Distributed Systems. In *CAV 1998*, LNCS 1427, pp. 357-368.

24. P.S. Thiagarajan and I. Walukiewicz. An Expressively Complete Linear Time Temporal Logic for Mazurkiewicz Traces. Information and Computation 179(2):230-249 (2002)
25. M.Y. Vardi and P. Wolper. Reasoning About Infinite Computations. Information and Computation, 115:1-37 (1994).
26. I. Walukiewicz. Difficult Configurations – On the Complexity of LTrL. In *ICALP 1998*, LNCS 1443, pp. 140-151.