LTL under reductions with weaker conditions than stutter invariance

Emmanuel Paviot-Adet[1,2], Denis Poitrenaud[1,2], Etienne Renault[3], Yann Thierry-Mieg[1]

[1] Sorbonne Université, CNRS, LIP6, F-75005 Paris, France
first.last@lip6.fr
[2] Université de Paris, F-75006 Paris, France
[3] EPITA, LRDE, Kremlin-Bicêtre, France
renault@lrde.epita.fr

Abstract. Verification of properties expressed as ω-regular languages such as LTL can benefit hugely from stutter insensitivity, using a diverse set of reduction strategies. However properties that are not stutter invariant, for instance due to the use of the neXt operator of LTL or to some form of counting in the logic, are not covered by these techniques in general.
We propose in this paper to study a weaker property than stutter insensitivity. In a stutter insensitive language both adding and removing stutter to a word does not change its acceptance, any stuttering can be abstracted away; by decomposing this equivalence relation into two implications we obtain weaker conditions. We define a shortening insensitive language where any word that stutters less than a word in the language must also belong to the language. A lengthening insensitive language has the dual property. A semi-decision procedure is then introduced to reliably prove shortening insensitive properties or deny lengthening insensitive properties while working with a *reduction* of a system. A reduction has the property that it can only shorten runs. Lipton's transaction reductions or Petri net agglomerations are examples of eligible structural reduction strategies.
An implementation and experimental evidence is provided showing most non-random properties sensitive to stutter are actually shortening or lengthening insensitive. Performance of experiments on a large (random) benchmark from the model-checking competition indicate that despite being a semi-decision procedure, the approach can still improve state of the art verification tools.

1 Introduction

Model checking is an automatic verification technique for proving the correctness of systems that have finite state abstractions. Properties can be expressed using the popular Linear-time Temporal Logic (LTL). To verify LTL properties, the automata-theoretic approach [25] builds a product between a Büchi automaton representing the negation of the LTL formula and the reachable state graph of the system (seen as a set of infinite runs). This approach has been used successfully to verify both hardware and software components, but it suffers from the so called "state explosion problem": as the number of state variables in the system increases, the size of the system state space grows exponentially.

One way to tackle this issue is to consider *structural reductions*. Structural reductions take their roots in the work of Lipton [15] and Berthelot [1]. Nowadays, these reductions are still considered as an attractive way to alleviate the state explosion problem [14, 2]. Structural reductions strive to fuse structurally "adjacent" events into a single atomic step, leading to less interleaving of independent events and less observable behaviors in the resulting system. An example of such a structural reduction is shown on Figure 1a where actions are progressively grouped (see Section 3.1 for a more detailed presentation). It can be observed that the Kripke structure representing the state space of the program is significantly simplified.

Traditionally structural reductions construct a smaller system that preserves properties such as deadlock freedom, liveness, reachability [10], and *stutter insensitive* temporal logic [20] such as $\mathrm{LTL}_{\setminus X}$. The verification of a *stutter insensitive* property on a given system does not depend on whether non observable events (i.e. that do not update atomic propositions) are abstracted or not. On Fig 1a both instructions "$z = 40$;" and "$chan.send(z)$" of thread β are non observable.

This paper shows that structural reductions can in fact be used even for fragments of LTL that are *not* stutter insensitive. We identify two fragments that we call *shortening insensitive* (if a word is in the language, any version that stutters less also) or *lengthening insensitive* (if a word is in the language, any version that stutters more also). Based on this classification we introduce two semi-decision procedures that provide a reliable verdict only in one direction: e.g. presence of counter examples is reliable for lengthening insensitive properties, but absence is not.

The paper is structured as follows, Section 2 presents the definitions and notations relevant to our setting in an abstract manner, focusing on the level of description of a language. Section 3 instantiates these definitions in the more concrete setting of LTL verification. Section 4 provides experimental evidence supporting the claim that the method is both applicable to many formulae and can significantly improve state of the art model-checkers. Some related work is presented in Section 5 before concluding.

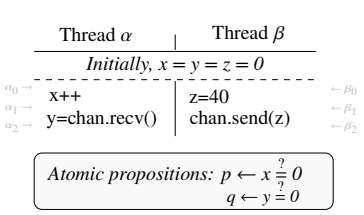
2 Definitions

In this section we first introduce in Section 2.1 a "shorter than" partial order relation on infinite words, based on the number of repetitions or stutter in the word. This partial order gives us in Section 2.2 the notions of shortening and lengthening insensitive language, which are shown to be weaker versions of classical stutter insensitivity in Section 2.3. We then define in Section 2.4 the *reduction of a language* which contains a shorter representative of each word in the original language. Finally we show that we can use a semi-decision procedure to verify shortening or lengthening insensitive properties using a reduction of a system.

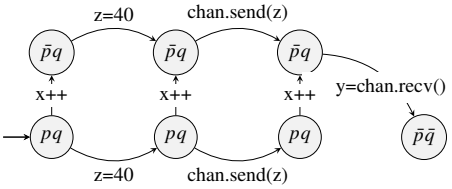
2.1 A "Shorter than" relation for infinite words

Definition 1 (Word). *: A word over a finite alphabet Σ is an infinite sequence of symbols in Σ. We canonically denote a word r using one of the two forms:*

- *(plain word) $r = a_0^{n_0} a_1^{n_1} a_2^{n_2} \dots$ with for all $i \in \mathbb{N}$, $a_i \in \Sigma$, $n_i \in \mathbb{N}^+$ and $a_i \neq a_{i+1}$, or*

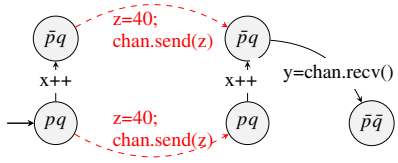


(1) Program



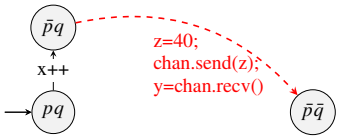
$\mathscr{L}_A = \{\ pq^3\ \bar{p}q\ \bar{p}\bar{q}^\omega,\ pq^2\ \bar{p}q^2\ \bar{p}\bar{q}^\omega,\ pq\ \bar{p}q^3\ \bar{p}\bar{q}^\omega\ \}$

(2) State space if each statement of the program is atomic



$\mathscr{L}_B = \{\ pq\ \bar{p}q^2\ \bar{p}\bar{q}^\omega,\ pq^2\ \bar{p}q\ \bar{p}\bar{q}^\omega\ \}$

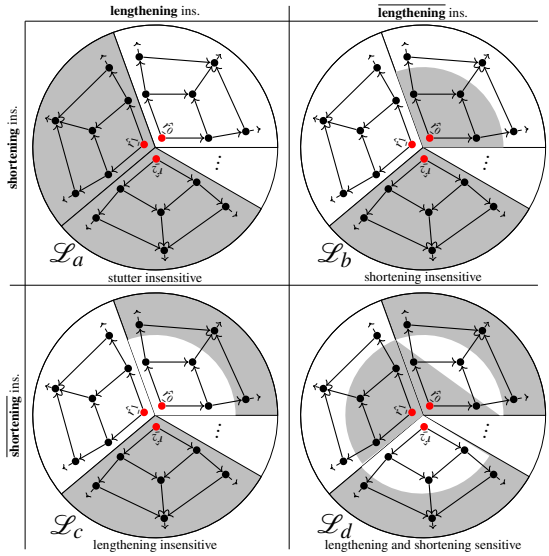
(3) State space if "z=40;send(z)" is atomic.



$\mathscr{L}_C = \{\ pq\ \bar{p}q\ \bar{p}\bar{q}^\omega\ \}$

(4) State space if "z=40;send(z);y=chan.recv()" is atomic.

(a) Example of reductions. (1) is a program with two threads and 3 variables. *chan* is a communication channel where *send(int)* insert a message and *int recv()* waits until a message is available and then consumes it. We consider that the logic only observes whether x or y is zero denoted p and q. (2) depicts the state-space represented as a Kripke structure. Each node is labelled by the value of atomic propositions p and q. When an instruction is executed the value of these propositions *may* evolve. (3) represents the state-space of a structurally reduced version of the program where actions of thread β "z=40;chan.send(z)" are fused into a single atomic operation. (4) represents the state-space of a program where the three actions of the original program "z=40;chan.send(z);y=chan.recv()" are now a single atomic step.



(b) Σ^ω is represented as a circle that is partitioned into equivalence classes of words $(\hat{r_0}, \hat{r_1} \ldots)$. Each point in the space is a word, and some of the $\preccurlyeq$ relations are represented as arrows ; the red point is the shortest word $\underline{\hat{r}}$ in the equivalence class. Gray areas are inside the language, white are outside of it. Four languages are depicted : $\mathscr{L}_a$: equivalence classes are entirely inside or outside **a stutter insensitive language**, $\mathscr{L}_b$: the "bottom" of an equivalence class may belong to **a shortening insensitive language**, $\mathscr{L}_c$: the "top" of an equivalence class may belong to **a lengthening insensitive language**, $\mathscr{L}_d$: some languages are neither lengthening insensitive nor shortening insensitive.

– *(ω-word) $r = a_0^{n_0} a_1^{n_1} \dots a_k^{\omega}$ with $k \in \mathbb{N}$ and for all $0 \leq i \leq k$, $a_i \in \Sigma$, and for $i < k$, $n_i \in \mathbb{N}^+$ and $a_i \neq a_{i+1}$. a_k^{ω} represents an infinite stutter on the final symbol a_k of the word.*

The set of all words over alphabet Σ is denoted Σ^{ω}.

These notations using a power notation for repetitions of a symbol in a word are introduced to highlight stuttering. We force the symbols to alternate to ensure we have a canonical representation: with σ a suffix (not starting by symbol b), the word $aab\sigma$ must be represented as $a^2b^1\sigma$ and not $a^1a^1b^1\sigma$. To represent a word of the form $aabbcccccc\dots$ we use an ω-word: $a^2b^2c^{\omega}$.

Definition 2 (Shorter than). *: A plain word $r = a_0^{n_0} a_1^{n_1} a_2^{n_2} \dots$ is* shorter *than a plain word $r' = a_0^{n'_0} a_1^{n'_1} a_2^{n'_2} \dots$ if and only if for all $i \in \mathbb{N}$, $0 < n_i \leq n'_i$. For two ω-words $r = a_0^{n_0} \dots a_k^{\omega}$ and $r' = a_0^{n'_0} \dots a_k^{\omega}$, r is* shorter *than r' if and only if for all $i < k$, $n_i \leq n'_i$.*
We denote this relation on words as $r \preccurlyeq r'$.

For instance, for any given suffix σ, $ab\sigma \preccurlyeq a^2b\sigma$. Note that $ab\sigma \preccurlyeq ab^2\sigma$ as well, but that $a^2b\sigma$ and $ab^2\sigma$ are incomparable. ω-words are incomparable with plain words.

Property 1. The $\preccurlyeq$ relation is a partial order on words.

Proof. The relation is clearly reflexive ($\forall r \in \Sigma^{\omega}, r \preccurlyeq r$), anti-symmetric ($\forall r, r' \in \Sigma^{\omega}$, $r \preccurlyeq r' \wedge r' \preccurlyeq r \Rightarrow r = r'$) and transitive ($\forall r, r', r'' \in \Sigma^{\omega}, r \preccurlyeq r' \wedge r' \preccurlyeq r'' \Rightarrow r \preccurlyeq r''$). The order is partial since some words (such as $a^2b\sigma$ and $ab^2\sigma$ presented above) are incomparable. □

Definition 3. *[Stutter equivalence]: a word r is* stutter equivalent *to r', denoted as $r \sim r'$ if and only if there exists a shorter word r'' such that $r'' \preccurlyeq r \wedge r'' \preccurlyeq r'$. This relation $\sim$ is an equivalence relation thus partitioning words of Σ^{ω} into equivalence classes.*
We denote $\hat{r}$ the equivalence class of a word r and denote $\underline{\hat{r}}$ the shortest word in that equivalence class.

For any given word $r = a_0^{n_0} a_1^{n_1} a_2^{n_2} \dots$ there is a shortest representative in $\hat{r}$ that is the word $\underline{\hat{r}} = a_0 a_1 a_2 \dots$ where no symbol is ever consecutively repeated more than once (until the ω for an ω-word). By definition all words that are comparable to $\underline{\hat{r}}$ are stutter equivalent to each other, since $\underline{\hat{r}}$ can play the role of r'' in the definition of stutter equivalence, giving us an equivalence relation: it is reflexive, symmetric and transitive.

For instance, with σ denoting a suffix, $\underline{\hat{r}} = ab\sigma$ would be the shortest representative of any word of $\hat{r}$ of the form $a^{n_0}b^{n_1}\sigma$. We can see by this definition that despite being incomparable, $a^2b\sigma \sim ab^2\sigma$ since $ab\sigma \preccurlyeq a^2b\sigma$ and $ab\sigma \preccurlyeq ab^2\sigma$.

2.2 Sensitivity of a language to the length of words

Definition 4 (Language). *: a language $\mathscr{L}$ over a finite alphabet Σ is a set of words over Σ, hence $\mathscr{L} \subseteq \Sigma^{\omega}$. We denote $\bar{\mathscr{L}} = \Sigma^{\omega} \setminus \mathscr{L}$ the complement of a language $\mathscr{L}$.*

In the literature, most studies that exploit a form of stuttering are focused on *stutter insensitive* languages [19, 24, 18, 9, 10]. In a stutter insensitive language $\mathscr{L}$, duplicating any letter (also called stuttering) or removing any duplicate letter from a word of $\mathscr{L}$ must produce another word of $\mathscr{L}$. In other words, all stutter equivalent words in a class $\hat{r}$ must be either in the language or outside of it. Let us introduce weaker variants of this property, original in this paper.

Definition 5. *[Shortening insensitive]: a language $\mathscr{L}$ is* shortening insensitive *if and only if for any word r it contains, all shorter words r' such that $r' \preccurlyeq r$ are also in $\mathscr{L}$.*

For instance, a shortening insensitive language $\mathscr{L}$ that contains the word $a^3b\sigma$ must also contain shorter words $a^2b\sigma$, and $ab\sigma$. If it contains $a^2b^2\sigma$ it also contains $a^2b\sigma$, $ab^2\sigma$ and $ab\sigma$.

Definition 6. *[Lengthening insensitive]: a language $\mathscr{L}$ is* lengthening insensitive *if and only if for any word r it contains, all longer words r' such that $r \preccurlyeq r'$ are also in $\mathscr{L}$.*

For instance, a lengthening insensitive language $\mathscr{L}$ that contains the word $a^2b\sigma$ must also contain all longer words $a^3b\sigma$, $a^2b^2\sigma$ ..., and more generally words of the form $a^nb^{n'}\sigma$ with $n \geq 2$ and $n' \geq 1$. If it contains $\underline{\hat{r}} = ab\sigma$ the shortest representative of an equivalence class, it contains all words in the stutter equivalence class.

While stutter insensitive languages have been heavily studied, there is to our knowledge no study on what reductions are possible if only one direction holds, i.e. the language is shortening or lengthening insensitive, but not both. A shortening insensitive language is essentially asking for something to happen *before* a certain deadline or stuttering "too much". A lengthening insensitive language is asking for something to happen *at the earliest* at a certain date or after having stuttered at least a certain number of times. Figure 1b represents these situations graphically.

2.3 Relationship to stutter insensitive logic

A language is both shortening and lengthening insensitive if and only if it is stutter insensitive (see Fig. 1b). This fact is already used in [16] to identify such stutter insensitive languages using only their automaton. Furthermore since stutter equivalent classes of runs are entirely inside or outside a stutter insensitive language, a language $\mathscr{L}$ is stutter insensitive if and only if the complement language $\bar{\mathscr{L}}$ is stutter insensitive.

However, if we look at sensitivity to length and how it interacts with the complement operation, we find a dual relationship where the complement of a shortening insensitive language is lengthening insensitive and vice versa.

Property 2. A language $\mathscr{L}$ is shortening insensitive if and only if the complement language $\bar{\mathscr{L}}$ is lengthening insensitive.

Proof. Let $\mathscr{L}$ be shortening insensitive. Let $r \in \bar{\mathscr{L}}$ be a word in the complement of $\mathscr{L}$. Any word r' such that $r \preccurlyeq r'$ must also belong to $\bar{\mathscr{L}}$, since if it belonged to the shortening insensitive $\mathscr{L}$, r would also belong to $\mathscr{L}$. Hence $\bar{\mathscr{L}}$ is lengthening insensitive. The converse implication can be proved using the same reasoning. □

If we look at Figure 1b, the dual effect of complement on the sensitivity of the language to length is apparent: if gray and white are switched we can see $\bar{\mathscr{L}}_b$ is lengthening insensitive and $\bar{\mathscr{L}}_c$ shortening insensitive.

2.4 When is visiting shorter words enough?

Definition 7. *[Reduction] Let I be a reduction function $\Sigma^\omega \mapsto \Sigma^\omega$ such that $\forall r, I(r) \preccurlyeq r$. The reduction by I of a language $\mathscr{L}$ is $Red_I(\mathscr{L}) = \{I(r) \mid r \in \mathscr{L}\}$.*

Note that the $\preccurlyeq$ partial order is not strict so that the image of a word may be the word itself, hence identity is a reduction function. In most cases however we expect the reduction function to map many words r of the original language to a single shorter word r' of the reduced language. Note that given any two reduction functions I and I', $Red_I(Red_{I'}(\mathscr{L}))$ is still a reduction of $\mathscr{L}$. Hence chaining reduction rules still produces a reduction. As we will discuss in Section 3.1 structural reductions of a specification such as Lipton's transaction reduction [15, 14] or Petri net agglomerations [1, 23] induce a reduction at the language level. In Fig. 1a fusing statements into a single atomic step in the program induces a reduction of the language.

Theorem 1 (Reduced Emptiness Checks). *Given two languages $\mathscr{L}$ and $\mathscr{L}'$,*

- *if $\mathscr{L}$ is shortening insensitive, then $\mathscr{L} \cap Red_I(\mathscr{L}') = \emptyset \Rightarrow \mathscr{L} \cap \mathscr{L}' = \emptyset$*
- *if $\mathscr{L}$ is lengthening insensitive, then $\mathscr{L} \cap Red_I(\mathscr{L}') \neq \emptyset \Rightarrow \mathscr{L} \cap \mathscr{L}' \neq \emptyset$.*

Proof. (Shortening insensitive $\mathscr{L}$)$\mathscr{L} \cap Red_I(\mathscr{L}') = \emptyset$ so there does not exists $r' \in \mathscr{L} \cap Red_I(\mathscr{L}')$. Because $\mathscr{L}$ is shortening insensitive, it is impossible that any run r with $r' \prec r$ belongs to $\mathscr{L} \cap \mathscr{L}'$. (Lengthening insensitive $\mathscr{L}$) At least one word r' is in $\mathscr{L}$ and $Red_I(\mathscr{L})$. Therefore the longer word r of $\mathscr{L}'$ that r' represents is also in $\mathscr{L}$ since the language is lengthening insensitive.

□

With this theorem original to this paper we now can build a semi-decision procedure that is able to prove *some* lengthening or disprove *some* shortening insensitive properties using a reduction of a system.

3 Application to Verification

We now introduce the more concrete setting of LTL verification to exploit the theoretical results on languages and their shortening/lengthening sensitivity developed in Section 2.

3.1 Kripke Structure

From the point of view of LTL verification with a state-based logic, executions of a system (also called *runs*) are seen as infinite words over the alphabet $\Sigma = 2^{AP}$, where AP is a set of atomic propositions that may be true or false in each state. So each symbol in a run gives the truth value of all of the atomic propositions in that state of the execution, and each time an action happens we progress in the run to the next symbol. Some actions of the system update the truth value of atomic propositions, but some actions can leave them unchanged which corresponds to stuttering.

Definition 8 (Kripke Structure Syntax). *Let* AP *designate a set of atomic propositions. A Kripke structure* $KS_{\mathrm{AP}} = \langle S, R, \lambda, s_0 \rangle$ *over* AP *is a tuple where* S *is the finite set of states,* $R \subseteq S \times S$ *is the transition relation,* $\lambda : S \mapsto 2^{\mathrm{AP}}$ *is the state labeling function, and* $s_0 \in S$ *is the initial state.*

Definition 9 (Kripke Structure Semantics). *The language* $\mathscr{L}(KS_{\mathrm{AP}})$ *of a Kripke structure* KS_{AP} *is defined over the alphabet* 2^{AP}*. It contains all runs of the form* $r = \lambda(s_0)\lambda(s_1)\lambda(s_2)\ldots$ *where* s_0 *is the initial state of* KS_{AP} *and* $\forall i \in \mathbb{N}$*, either* $(s_i, s_{i+1}) \in R$*, or if* s_i *is a deadlock state such that* $\forall s' \in S, (s_i, s') \notin R$ *then* $s_{i+1} = s_i$*.*

All system executions are considered maximal, so that they are represented by infinite runs. If the system can deadlock or terminate in some way, we can extend these finite words by an ω stutter on the last symbol of the run to obtain a run.

Subfigure (1) of Figure 1a depicts a program where each thread (α and β) has three reachable positions (we consider that each instruction is atomic). In this example we consider that the logic only observes two atomic propositions p (true when $x = 0$) and q (true when $y = 0$). The variable z is not observed.

Subfigure (2) of Figure 1a depicts the reachable states of this system as a Kripke structure. Actions of thread β (which do not modify the value of p or q) are horizontal while actions of thread α are vertical. While each thread has 3 reachable positions, the emission of the message by β must precede the reception by α so that some situations are unreachable. Based on Definition 9 that extends by an infinite stutter runs that end in a deadlock, we can compute the language $\mathscr{L}_A$ of this system. It consists in three parts: when thread β goes first $pq^3\ \bar{p}q\ \bar{p}\bar{q}^\omega$, with an interleaving $pq^2\ \bar{p}q^2\ \bar{p}\bar{q}^\omega$, and when thread α goes first $pq\ \bar{p}q^3\ \bar{p}\bar{q}^\omega$.

In subfigure (3) of Figure 1a, the actions "$z = 40; chan.send(z)$;"of thread β are fused into a single atomic operation. This is possible because action $z = 40$ of thread β is stuttering (it cannot affect either p or q) and is non-interfering with other events (it neither enables nor disables any event other than subsequent instruction "chan.send(z)"). The language of this smaller KS is a reduction of the language of the original system. It contains two runs: thread α goes first $pq\ \bar{p}q^2\ \bar{p}\bar{q}^\omega$ and thread β goes first $pq^2\ \bar{p}q\ \bar{p}\bar{q}^\omega$.

In subfigure (4) of Figure 1a, the already fused action "$z = 40; chan.send(z)$;"of thread β is further fused with the *chan.recv();* action of thread α. This leads to a smaller KS whose language is still a reduction of the original system now containing a single run: $pq\ \bar{p}q\ \bar{p}\bar{q}^\omega$. This simple example shows the power of structural reductions when they are applicable, with a drastic reduction of the initial language.

3.2 Automata theoretic verification

Let us consider the problem of model-checking of an ω-regular property φ (such as LTL) on a system using the automata-theoretic approach [25]. In this approach, we wish to answer the problem of language inclusion: do all runs of the system $\mathscr{L}(KS)$ belong to the language of the property $\mathscr{L}(\varphi)$? To do this, when the property φ is an omega-regular language (e.g. an LTL or PSL formula), we first negate the property $\neg\varphi$, then build a

(variant of) a Büchi automaton $A_{\neg\varphi}$ whose language[4] consists of runs that are not in the property language $\mathscr{L}(A_{\neg\varphi}) = \Sigma^\omega \setminus \mathscr{L}(\varphi)$. We then perform a synchronized product between this Büchi automaton and the Kripke structure KS corresponding to the system's state space $A_{\neg\varphi} \otimes KS$ (where $\otimes$ is defined to satisfy $\mathscr{L}(A \otimes B) = \mathscr{L}(A) \cap \mathscr{L}(B)$). Either the language of the product is empty $\mathscr{L}(A_{\neg\varphi} \otimes KS) = \emptyset$, and the property φ is thus true of this system, or the product is non empty, and from any run in the language of the product we can build a counter-example to the property.

We will consider in the rest of the paper that the shortening or lengthening insensitive language of definitions 5 and 6 is given as an omega-regular language or Büchi automaton typically obtained from the negation of an LTL property, and that the reduction of definition 7 is applied to a language that corresponds to all runs in a Kripke structure typically capturing the state space of a system.

LTL verification with reductions. With theorem 1, a shortening insensitive property shown to be true on the reduction (empty intersection with the language of the negation of the property) is also true of the original system. A lengthening insensitive property shown to be false on the reduction (non-empty intersection with the language of the negation of the property, hence counter-examples exist) is also false in the original system. Unfortunately, our procedure cannot prove using a reduction that a shortening insensitive property is false, or that a lengthening insensitive property is true. We offer a semi-decision procedure.

3.3 Detection of language sensitivity

We now present a strategy to decide if a given property expressed as a Büchi automaton is shortening insensitive, lengthening insensitive, or both.

This section relies heavily on the operations introduced and discussed at length in [16]. The authors define two *syntactic* transformations sl and cl of a transition-based generalized Büchi automaton (TGBA) A_φ that can be built from any LTL formula φ to represent its language $\mathscr{L}(\varphi) = \mathscr{L}(A_\varphi)$ [3]. TGBA are a variant of Büchi automata where the acceptance conditions are placed on edges rather than states of the automaton.

The cl closure operation *decreases stutter*, it adds to the language any word $r' \in \Sigma^\omega$ that is shorter than a word r in the language. Informally, the strategy consists in detecting when a sequence $q_1 \xrightarrow{a} q_2 \xrightarrow{a} q_3$ is possible and adding an edge $q_1 \xrightarrow{a} q_3$, hence its name cl for "closure". The sl self-loopization operation *increases stutter*, it adds to the language any run $r' \in \Sigma^\omega$ that is longer than a run r in the language. Informally, the strategy consists in adding a self-loop to any state labeled with all outgoing expressions so that we can always decide to repeat a letter rather than progress in the automaton, hence its name sl for "self-loop". More formally $\mathscr{L}(cl(A_\varphi)) = \{r' \mid \exists r \in \mathscr{L}(A_\varphi), r' \preccurlyeq r\}$ and $\mathscr{L}(sl(A_\varphi)) = \{r' \mid \exists r \in \mathscr{L}(A_\varphi), r \preccurlyeq r'\}$.

Using these operations [16] shows that there are several possible ways to test if an omega-regular language (encoded as a Büchi automaton) is stutter insensitive: essentially

[4] Because computing the complement $\bar{A}$ of an automaton A is exponential in the worst case, syntactically negating φ and producing an automaton $A_{\neg\varphi}$ is preferable when A is derived from e.g. an LTL formula.

applying either of the operations cl or sl should leave the language unchanged. This allows to recognize that a property is stutter insensitive even though it syntactically contains e.g. the neXt operator of LTL.

For instance A_φ is stutter insensitive if and only if $\mathscr{L}(sl(cl(A_\varphi)) \otimes A_{\neg\varphi}) = \emptyset$. The full test is thus simply reduced to a language emptiness check testing that both sl and cl operations leave the language of the automaton unchanged.

Indeed for stutter insensitive languages, all or none of the runs belonging to a given stutter equivalence class of runs $\hat{r}$ must belong to the language $\mathscr{L}(A_\varphi)$. In other words, if shortening or lengthening a run can make it switch from belonging to A_φ to belonging to $A_{\neg\varphi}$, the language is stutter sensitive. This is apparent on Figure 1b

We want weaker conditions here, but we can reuse the sl and cl operations developed for testing stutter insensitivity. Indeed for an automaton A encoding a shortening insensitive language, $\mathscr{L}(cl(A)) = \mathscr{L}(A)$ should hold. Conversely if A encodes a lengthening insensitive language, $\mathscr{L}(sl(A)) = \mathscr{L}(A)$ should hold. We express these tests as emptiness checks on a product in the following way.

Theorem 2 (Testing sensitivity). *Let A designate a Büchi automaton, and $\bar{A}$ designate its complement.*

$\mathscr{L}(cl(A) \otimes \bar{A}) = \emptyset$, if and only if A defines a shortening insensitive language.

$\mathscr{L}(sl(A) \otimes \bar{A}) = \emptyset$ if and only if A defines a lengthening insensitive language.

Proof. The expression $\mathscr{L}(cl(A) \otimes \bar{A}) = \emptyset$ is equivalent to $\mathscr{L}(cl(A)) = \mathscr{L}(A)$. The lengthening insensitive case is similar. □

Thanks to property 2, and in the spirit of [16] we could also test the complement of a language for the dual property if that is more efficient, i.e. $\mathscr{L}(sl(\bar{A}) \otimes A) = \emptyset$ if and only if A defines a shortening insensitive language and similarly $\mathscr{L}(cl(\bar{A}) \otimes A) = \emptyset$ iff A is lengthening insensitive. We did not really investigate these alternatives as the complexity of the test was already negligible in all of our experiments.

3.4 Agglomeration of events produces shorter runs

Among the possible strategies to reduce the complexity of analyzing a system are structural reductions. Depending on the input formalism the terminology used is different, but the main results remain stable.

In [15] transaction reduction consists in fusing two adjacent actions of a thread (or even across threads in recent versions such as [14]). The first action must not modify atomic properties and must be commutative with any action of other threads. Fusing these actions leads to shorter runs, where a stutter is lost. In the program of Fig. 1a, "z=40" is enabled from the initial state and must happen before "chan.send(z)", but it commutes with instructions of thread α and is not observable. Hence the language $\mathscr{L}_B$ built with an atomic assumption on "z=40;chan.send(z)" is indeed a reduction of $\mathscr{L}_A$.

Let us reason at the level of a Kripke structure. The goal of such reductions is to structurally detect the following situation in language $\mathscr{L}$: let $r = a_0^{n_0} a_1^{n_1} a_2^{n_2} \ldots$ designate a run (not necessarily in the language), there must exist two indexes i and j such that for any natural number k, $i \leq k \leq j$, $r_k = a_0^{n_0} \ldots a_i^{n_i} \ldots a_k^{n_k+1} \ldots a_j^{n_j} \ldots$ is in the language. In

other words, the set of runs described as : $\{r_k = a_0^{n_0} \dots a_i^{n_i} \dots a_k^{n_k+1} \dots a_j^{n_j} \dots \mid i \leq k \leq j\}$ must belong to the language. This corresponds to an event that does not impact the truth value of atomic propositions (it stutters) and can be freely commuted with any event that occurs between indexes i and j in the run. This event is simply constrained to occur at the earliest at index i in the run and at the latest at index j. In Fig 1a the event "z=40" can happen as early as in the initial state, and must occur before "chan.send(z)" and thus matches this definition.

Note that these runs are all stutter equivalent, but are incomparable by the shorter than relation (e.g. $aabc, abbc, abcc$ are incomparable). In this situation, a *reduction* can choose to only represent the run r instead of any of these runs. This run was not originally in the language in general, but it is indeed shorter than any of the r_k runs so it matches definition 7 for a reduction. Note that $\hat{r}$ does contain all these longer runs so that in a stutter insensitive context, examining r is enough to conclude for any of these runs. This is why usage of structural reductions is compatible with verification of a logic such as $LTL_{\setminus X}$ and has been proposed for that express purpose in the literature [10, 14].

Thus transaction reductions [15, 14] as well as both pre-agglomeration and post-agglomeration of Petri nets [20, 7, 10, 23] produce a system whose language is a reduction of the language of the original system.

For lack of space, in this paper we decided not to provide proofs that these structural transformation rules induce reductions at the language level. A formal definition involves a) introducing the syntax of a formalism and b) its semantics in terms of language, then c) defining the reduction rule, and d) proving its effect is a reduction at the language level. The exercise is not particularly difficult, and the definition of reduction rules mostly fall into the category above, where a non observable event that happens at the earliest at point a and at the latest at point b is abstracted from the trace.

Our experimental Section 4.2 uses the rules of [23] for (potentially partial) pre and post-agglomeration. That paper presents 22 structural reductions rules from which we selected the rules valid in the context of LTL verification. Only one rule preserving stutter insensitive LTL was not compatible with our approach since it does not produce a reduction at the language level: rule 3 "Redundant transitions" proposes that if two transitions t_1 and t_2 have the same combined effect as a transition t, and firing t_1 enables t_2, t can be discarded from the net. This reduces the number of edges in the underlying KS representing the state space, but does not affect reachability of states. However, it selects as representative a run involving both t_1 and t_2 that is longer than the one using t in the original net, it is thus not legitimate to use in our strategy (although it remains valid for $LTL_{\setminus X}$). Rules 14 "Pre agglomeration" and 15 "Post agglomeration" are the most powerful rules of [23] that we are able to apply in our context. They are known to preserve $LTL_{\setminus X}$ (but not full LTL) and their effect is a reduction at the language level, hence we *can* use them when dealing with shortening/lengthening insensitive formulae.

4 Experimentation

4.1 A Study of Properties

This section provides an empirical study of the applicability of the techniques presented in this paper to LTL properties found in the literature. To achieve this we explored several

LTL benchmarks [6, 21, 5, 11, 13]. Some work [6, 21] summarises the typical properties that users express in LTL. The formulae of this benchmark have been extracted directly from the literature. Dwyer et al. [5] proposes property specification patterns, expressed in several logics including LTL. These patterns have been extracted by analysing 447 formulae coming from real world projects. The RERS challenge [11] presents generated formulae inspired from real world reactive systems. The MCC [13] benchmark establishes a huge database of 45152 LTL formulae in the form of 1411 Petri net models coming from 114 origins with for each one 32 random LTL formulae. These formulae use up to 5 state-based atomic propositions, limit the nesting depth of temporal operators to 5 and are filtered in order to be non trivial. Since these formulae come with a concrete system we were able to use this benchmark to also provide performance results for our approach in Section 4.2. We retained 43989 model/formula pairs from this benchmark, the missing 1163 were rejected due to parse limitations of our tool when the model size is excessive ($> 10^7$ transitions). This set of roughly 2200 human-like formulae and 44k random ones lets us evaluate if the fragment of LTL that we consider is common in practice. Table 1 summarizes, for each benchmark, the number and percentage of formulae that are either stuttering insensitive, lengthening insensitive, or shortening insensitive. The sum of both shortening and lengthening formulae represents more than one third (and up to 60 percent) of the formulae of these benchmarks.

Concerning the polarity, although lengthening insensitive formulae seem to appear more frequently, most of these benchmarks actually contain each formula in both positive and negative forms (we retained only one) so that the summed percentage might be more relevant as a metric since lengthening insensitivity of φ is equivalent to shortening insensitivity of $\neg\varphi$. Analysis of the human-like Dwyer patterns [5] reveals that shortening/lengthening insensitive formulae mostly come from the patterns *precedence chain*, *response chain* and *constrained chain*. These properties specify causal relation between events, that are observable as causal relations between *observably different* states (that might be required to strictly follow each other), but this causality chain is not impacted by non observable events.

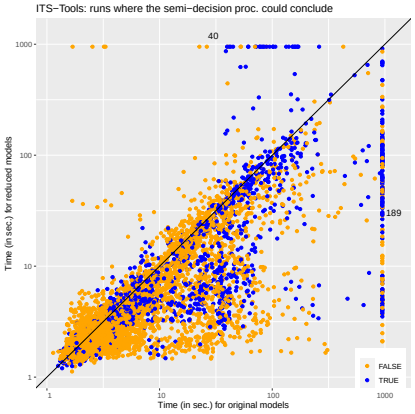
4.2 A Study of Performances

Benchmark Setup. Among the LTL benchmarks presented in Table 1, we opted for the MCC benchmark to evaluate the techniques presented in this paper. This benchmark seems relevant since (1) it contains both academic and industrial models, (2) it has a huge set of (random) formulae and (3) includes models so that we could measure the effect of the approach in a model-checking setting. The model-checking competition (MCC) is an annual event in its 10^{th} edition in 2021 where competing tools are evaluated on a large benchmark. We use the formulae and models from the latest 2021 edition of the contest, where Tapaal [4] was awarded the gold medal and ITS-Tools [22] was silver in the LTL category of the contest. We evaluate both of these tools in the following performance measures, showing that our strategy is agnostic to the back-end analysis engine. Our experimental setup consists in two steps.

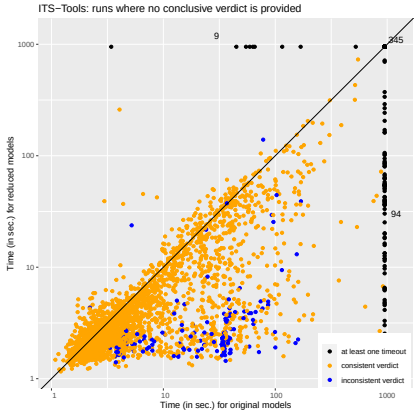
1. Parse the model and formula pair, and analyze the sensitivity of the formula. When the formula is shortening or lengthening insensitive (but not both) output two

Benchmark	Stutter Insens.	Length. Insens.	Short. Insens.	Others	Total
Spot [6, 5, 21]	63 (67%)	17 (18%)	11 (1%)	3 (3%)	94
Dwyer et al. [5]	32 (58%)	13 (23%)	9 (16%)	1 (1.81%)	55
RERS [11].	714 (35%)	777 (38%)	559 (28%)	0	2050
MCC [13]	24462 (56%)	6837 (14%)	5390 (12%)	7300 (16%)	43989

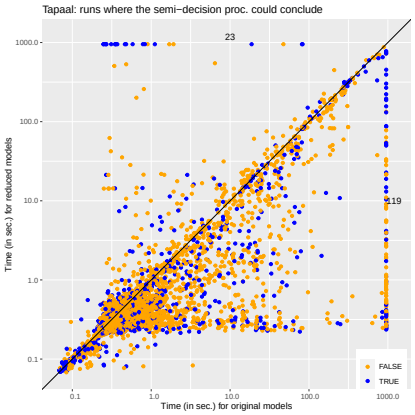
Table 1: Sensitivity to length of properties measured using several LTL benchmarks.



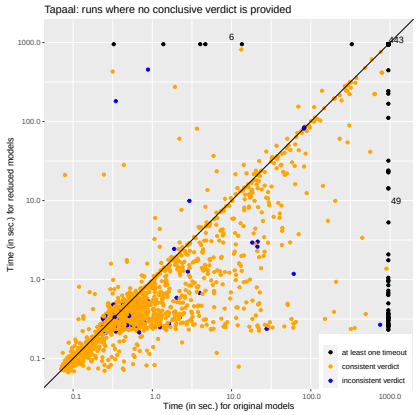
(a) ITS-tools on decidable instances



(b) ITS-tools on non decidable instances



(c) Tapaal on decidable instances



(d) Tapaal on non decidable instances

Table 2: Experiments on the MCC'2021 LTL benchmark using the two best tool of the MCC contest: Tapaal and ITS-tools. Figures (a) and (c) contain the cases where the verdict of the semi-decisions procedures is reliable, and distinguish cases where the output is True (empty product) and False (non empty product). (b) and (d) display the cases where the verdict is not reliable and distinguish cases where the output is inconsistent with the ground truth from cases where they agree.

model/formula pairs : reduced and original. The "original" version does also benefit from reduction rules, but we apply only rules that are compatible with full LTL. The "reduced" version additionally benefits from rules that are reductions at the language level, i.e. mainly pre and post agglomeration (but enacting these rules can cause further simplifications). The original and reduced model/formula pairs that result from this procedure are then exported in the same format the contest uses. This step was implemented within ITS-tools.
2. Run an MCC compatible tool on both the reduced and original versions of each model/formula pair and record the time performance and the verdict.

For the first step, using Spot [16], we detect that a formula is either shortening or lengthening insensitive for 99.81% of formulae in less than 1 second. After this analysis, we obtain 12 227 model/formula pairs where the formula is either shortening insensitive or lengthening insensitive (but not both). Among these pairs, in 3005 cases (24.6%) the model was resistant to the structural reduction rules we use. Since our strategy does not improve such cases, we retain the remaining 9 222 (75.4%) model/formula pairs in the performance plots of Figure 2. We measured that on average 34.19% of places and 32.69% of the transitions of the models were discarded by reduction rules with respect to the "original" model, though the spread is high as there are models that are almost fully reducible and some that are barely so. Application of reduction rules is in complexity related to the size of the structure of the net and takes less than 20 seconds to compute in 95.5% of the models. We are able to treat 9222 examples (21% of the original 43989 model/formula pairs of the MCC) using reductions. All these formulae until now could not be handled using reduction techniques.

For the second step, we measured the solution time for both reduced and original model/formula pairs using the two best tools of the MCC'2021 contest. A full tool using our strategy might optimistically first run on the reduced model/formula pair hoping for a definitive answer, but we recommend the use of a portfolio approach where the first reliable answer is kept. In these experiments we neutrally measured the time for taking a semi-decision on the reduced model vs. the time for taking a (complete) decision on the original model. We then classify the results into two sets, decidable instances are shown on the left of Fig. 2 and instances that are not decidable (by our procedure) are on the right. On "decidable instances" our semi-decision procedure could have concluded reliably because the formula is true and the property shortening insensitive or the formula is false and the property lengthening insensitive. Non decidable instances shown on the right are those where the verdict on the reduced model is not to be trusted (or both the original and reduced procedures timed out).

With this workflow we show that our approach is generic and can be easily implemented on top of any MCC compatible model-checking tool. All experiments were run with a 950 seconds timeout (close to 15 minutes, which is generous when the contest offers 1 hour for 16 properties). We used a heterogeneous cluster of machines with four cores allocated to each experiment, and ensured that experiments concerning reduced and original versions of a given model/formula are comparable.

Figure 2 presents the results of these experiments. The results are all presented as log-log scatter plots opposing a run on the original to a run on the reduced model/formula pair. Each dot represents an experiment on a model/formula pair; a dot below the diagonal

indicates that the reduced version was faster to solve, while a point above it indicates a case where the reduced model actually took longer to solve than the original (fortunately there are relatively few). Points that timeout for one (or both) of the approaches are plotted on the line at 950 seconds, we also indicate the number of points that are in this line (or corner) next to it.

The plots on the left (a) and (c) correspond to "decidable instances" while those on the right are not decidable by our procedure. The two plots on the top correspond to the performance of ITS-tools, while those on the bottom give the results with Tapaal. The general form of the results with both tools is quite similar confirming that our strategy is indeed responsible for the measured gains in performance and that they are reproducible. Reduced problems *are* generally easier to solve than the original. This gain is in the best case exponential as is visible through the existence of spread of points reaching out horizontally in this log-log space (particularly on the Tapaal plots).

The colors on the decidable instances reflect whether the verdict was true or false. For false properties a counter-example was found by both procedures interrupting the search, and while the search space of a reduced model is a priori smaller, heuristics and even luck can play a role in finding a counter-example early. True answers on the other hand generally require a full exploration of the state space so that the reductions should play a major role in reducing the complexity of model-checking. The existence of True answers where the reduction fails is surprising at first, but a smaller Kripke structure does not necessarily induce a smaller product as happens sometimes in this large benchmark (and in other reduction techniques such as stubborn sets [24]). On the other hand the points aligned to the right of the plots a) and c) (189 for ITS-tools and 119 for Tapaal) correspond to cases where our procedure improved these state of the art tools, allowing to reach a conclusion when the original method fails.

The plots on the right use orange to denote cases where the verdict on the reduced and original models were the same; on these points the procedures had comparable behaviors (either exploring a whole state space or exhibiting a counter-example). The blue color denotes points where the two procedures disagree, with several blue points above the diagonal reflecting cases where the reduced procedure explored the whole state space and thought the property was true while the original procedure found a counter-example (this is the worst case). Surprisingly, even though on these non decidable plots b) and d) our procedure should not be trusted, it mostly agrees (in 95% of the cases) with the decision reached on the original.

Out of the 9222 experiments in total, for ITS-tools 5901 runs reached a trusted decision (64 %), 2927 instances reached an untrusted verdict (32 %), and the reduced procedure timed out in 394 instances (4 %). Tapaal reached a trusted decision in 5866 instances (64 %), 2884 instances reached an untrusted verdict (31 %), and the reduced procedure timed out in 472 instances (5 %). On this benchmark of formulae we thus reached a trusted decision in almost two thirds of the cases using the reduced procedure.

5 Related Work

Partial order vs structural reductions. Partial order reduction (POR) [19, 24, 18, 9] is a very popular approach to combat state explosion for *stutter insensitive* formulae. These

approaches use diverse strategies (stubborn sets, ample sets, sleep sets. . .) to consider only a subset of events at each step of the model-checking while still ensuring that at least one representative of each stutter equivalent class of runs is explored. Because the preservation criterion is based on equivalence classes of runs, this family of approaches is limited only to the stutter insensitive fragment of LTL (see Fig.1b). However the structural reduction rules used in this paper are compatible and can be stacked with POR when the formula is stutter insensitive; this is the setting in which most structural reduction rules were originally defined.

Structural reductions in the literature. The structural reductions rules we used in the performance evaluation are defined on Petri nets where the literature on the subject is rich [1, 20, 7, 10, 2, 23]. However there are other formalism where similar reduction rules have been defined such as [17] using "atomic" blocks in Promela, transaction reductions for the widely encompassing intermediate language PINS of LTSmin [14], and even in the context of multi-threaded programs [8]. All these approaches are structural or syntactic, they are run prior to model-checking per se.

Non structural reductions in the literature. Other strategies have been proposed that instead of structurally reducing the system, dynamically build an abstraction of the Kripke structure where less observable stuttering occurs. These strategies build a *KS* whose language *is* a reduction of the language of the original *KS* (in the sense of Def. 7), that can then be presented to the emptiness check algorithm with the negation of the formula. They are thus also compatible with the approach proposed in this paper. Such strategies include the Covering Step Graph (CSG) construction of [26] where a "step" is performed (instead of firing a single event) that includes several independent transitions. The Symbolic Observation Graph of [12] is another example where states of the original *KS* are computed (using BDDs) and aggregated as long as the atomic proposition values do not evolve; in practice it exhibits to the emptiness check only shortest runs in each equivalence class hence it is a reduction.

6 Conclusion

To combat the state space explosion problem that LTL model-checking meets, structural reductions have been proposed that syntactically compact the model so that it exhibits less interleaving of non observable actions. Prior to this work, all of these approaches were limited to the stutter insensitive fragment of the logic. We bring a semi-decision procedure that widens the applicability of these strategies to formulae which are *shortening insensitive* or *lengthening insensitive*. The experimental evidence presented shows that the fragment of the logic covered by these new categories is quite useful in practice. An extensive measure using the models, formulae and the two best tools of the model-checking competition 2021 shows that our strategy can improve the decision power of state of the art tools, and confirm that in the best case an exponential speedup of the decision procedure can be attained. We also identified several other strategies that are compatible with our approach since they construct a reduced language. In further work we are investigating how non trusted counter-examples of the reduced model could be confirmed on the original model.

References

1. Berthelot, G.: Checking properties of nets using transformation. In: Applications and Theory in Petri Nets. LNCS, vol. 222, pp. 19–40. Springer (1985)
2. Berthomieu, B., Le Botlan, D., Dal Zilio, S.: Counting Petri net markings from reduction equations. International Journal on Software Tools for Technology Transfer (Apr 2019)
3. Couvreur, J.: On-the-fly verification of linear temporal logic. In: World Congress on Formal Methods. Lecture Notes in Computer Science, vol. 1708, pp. 253–271. Springer (1999)
4. David, A., Jacobsen, L., Jacobsen, M., Jørgensen, K.Y., Møller, M.H., Srba, J.: TAPAAL 2.0: Integrated development environment for timed-arc Petri nets. In: Flanagan, C., König, B. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 492–497. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
5. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Property specification patterns for finite-state verification. In: Ardis, M. (ed.) Proceedings of the 2nd Workshop on Formal Methods in Software Practice (FMSP'98). pp. 7–15. ACM Press (Mar 1998). https://doi.org/10.1145/298595.298598
6. Etessami, K., Holzmann, G.J.: Optimizing Büchi automata. In: Palamidessi, C. (ed.) Proceedings of the 11th International Conference on Concurrency Theory (Concur'00). LNCS, vol. 1877, pp. 153–167. Springer-Verlag, Pennsylvania, USA (2000)
7. Evangelista, S., Haddad, S., Pradat-Peyre, J.: Syntactical colored Petri nets reductions. In: ATVA. LNCS, vol. 3707, pp. 202–216. Springer (2005)
8. Flanagan, C., Qadeer, S.: A type and effect system for atomicity. In: PLDI. pp. 338–349. ACM (2003)
9. Godefroid, P., Wolper, P.: A partial approach to model checking. Inf. Comput. **110**(2), 305–326 (1994)
10. Haddad, S., Pradat-Peyre, J.: New efficient Petri nets reductions for parallel programs verification. Parallel Processing Letters **16**(1), 101–116 (2006)
11. Howar, F., Jasper, M., Mues, M., Schmidt, D., Steffen, B.: The rers challenge: towards controllable and scalable benchmark synthesis. International Journal on Software Tools for Technology Transfer pp. 1–14 (06 2021). https://doi.org/10.1007/s10009-021-00617-z
12. Klai, K., Poitrenaud, D.: MC-SOG: an LTL model checker based on symbolic observation graphs. In: van Hee, K.M., Valk, R. (eds.) Applications and Theory of Petri Nets, 29th International Conference, PETRI NETS 2008, Xi'an, China, June 23-27, 2008. Proceedings. LNCS, vol. 5062, pp. 288–306. Springer (2008)
13. Kordon, F., Bouvier, P., Garavel, H., Hillah, L.M., Hulin-Hubard, F., Amat., N., Amparore, E., Berthomieu, B., Biswal, S., Donatelli, D., Galla, F., , Dal Zilio, S., Jensen, P., He, C., Le Botlan, D., Li, S., , Srba, J., Thierry-Mieg, Y., Walner, A., Wolf, K.: Complete Results for the 2021 Edition of the Model Checking Contest. http://mcc.lip6.fr/2021/results.php (June 2021)
14. Laarman, A.: Stubborn transaction reduction. In: NFM. LNCS, vol. 10811, pp. 280–298. Springer (2018)
15. Lipton, R.J.: Reduction: A method of proving properties of parallel programs. Commun. ACM **18**(12), 717–721 (1975)
16. Michaud, T., Duret-Lutz, A.: Practical stutter-invariance checks for ω-regular languages. In: SPIN. LNCS, vol. 9232, pp. 84–101. Springer (2015)
17. Pajault, C., Pradat-Peyre, J., Rousseau, P.: Adapting Petri nets reductions to Promela specifications. In: FORTE. LNCS, vol. 5048, pp. 84–98. Springer (2008)
18. Peled, D.A.: Combining partial order reductions with on-the-fly model-checking. In: CAV. LNCS, vol. 818, pp. 377–390. Springer (1994)
19. Peled, D.A., Pratt, V.R., Holzmann, G.J. (eds.): Partial Order Methods in Verification, Proceedings of a DIMACS Workshop, 1996, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 29. DIMACS/AMS (1996)

20. Poitrenaud, D., Pradat-Peyre, J.: Pre- and post-agglomerations for LTL model checking. In: ICATPN. LNCS, vol. 1825, pp. 387–408. Springer (2000)
21. Somenzi, F., Bloem, R.: Efficient Büchi automata for LTL formulæ. In: Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00). LNCS, vol. 1855, pp. 247–263. Springer-Verlag, Chicago, Illinois, USA (2000)
22. Thierry-Mieg, Y.: Symbolic model-checking using ITS-tools. In: TACAS. LNCS, vol. 9035, pp. 231–237. Springer (2015)
23. Thierry-Mieg, Y.: Structural reductions revisited. In: Petri Nets. LNCS, vol. 12152, pp. 303–323. Springer (2020)
24. Valmari, A.: A stubborn attack on state explosion. In: CAV. LNCS, vol. 531, pp. 156–165. Springer (1990)
25. Vardi, M.Y.: Automata-theoretic model checking revisited. In: VMCAI. LNCS, vol. 4349, pp. 137–150. Springer (2007)
26. Vernadat, F., Michel, F.: Covering step graph preserving failure semantics. In: Azéma, P., Balbo, G. (eds.) Application and Theory of Petri Nets 1997, 18th International Conference, ICATPN '97, Toulouse, France, June 23-27, 1997, Proceedings. LNCS, vol. 1248, pp. 253–270. Springer (1997)